SÉRIE PENSAMENTO MATEMÁTICO @ CIÊNCIA COMPUTAÇÃO

Volume II: Da Computabilidade Formal às Máquinas Programáveis

João Bosco M. Sobral

EDIÇÃO DO AUTOR

João Bosco M. Sobral

Da Computabilidade Formal às Máquinas Programáveis

Série Pensamento Matemático @
Ciência da Computação
Edição do Autor

Laboratório e Grupo de Pesquisa
UFSC-CNPq

Revisão Técnica
Renato Bobsin Machado
Dr. em Ciências - UNICAMP
Programa de Pós-Graduação em Engenharia Elétrica e Computação - UNIOESTE

Universidade Federal de Santa Catarina
Centro Tecnológico
Departamento de Informática e Estatística
Laboratório DMC & NS
Projeto de Pesquisa PRPE 2013.1533

João Bosco M. Sobral

Da Computabilidade Formal às Máquinas Programáveis

Série Pensamento Matemático @
Ciência da Computação

1ª Edição

Florianópolis
Edição do Autor
2015

Universidade Federal de Santa Catarina
Centro Tecnológico
Departamento de Informática e Estatística
Laboratório DMC & NS

Projeto de Pesquisa PRPE 2013.1533

Nota - Muito trabalho foi empregado nesta edição. No entanto, podem ocorrer erros de digitação, impressão ou dúvida sobre os conceitos. Em qualquer das hipóteses, o autor e editor, solicita a comunicação no email *bosco.sobral@ufsc.br*, para que o mesmo possa encaminhar a correção da questão mencionada.

Catalogação na fonte pela Biblioteca Universitária
da
Universidade Federal de Santa Catarina

S677d Sobral, João Bosco M.
Da Computabilidade Formal às Máquinas Programáveis [recurso eletrônico] / João Bosco M. Sobral. - 1. ed. - Florianópolis : Edição do Autor, 2015.
297 p. : il., grafs., tabs.-(Série Pensamento Matemático. Ciência da Computação.)

Esta publicação foi organizada pelo Departamento de Informática e Estatística da Universidade Federal de Santa Catarina.

Inclui bibliografia e índice.
ISBN: 978-85-902995-3-0(v.2)

1. Computação - história. 2. Lógica matemática. 3. Ciência da Computação. I. Título. II. Série.

CDU: 519.687(091)

Agradecimentos

O autor é imensamente grato ao Departamento de Informática e Estatística pela oportunidade de trazer neste projeto de pesquisa, o fruto da sua formação matemática e da experiência de ensino de graduação e pós-graduação em Ciência da Computação, nos seus 38 anos na UFSC.

Embora, eu não tenha sido um professor do ensino em disciplinas de Matemática do INE, mas obtive todo o sentimento do elo entre a matemática e a lógica nas várias disciplinas onde atuei: ensino de programação, linguagens formais, redes de computadores, computação paralela e distribuida, sistemas distribuidos, métodos formais de especificação de sistemas, mobilidade em computação e segurança computacional.

Agradecimentos pessoais aos colegas do INE, Jerusa Marchi, Vânia Bogorny, Rafael Cancian, José Fletes, João Dovicchi, Fernando Cruz, Roberto Willrich, Juliana Eyng, Jean Martina, Sergio Peters, Patricia Plentz, Marcelo Menezes Reis, Carla Westphall e Carlos Westphall, com os quais em breves momentos, eu pudi falar do meu trabalho, mas que contribuiram direta ou indiretamente, motivando-me para a realização do mesmo. Também, a Lucas Guardalbem pela revisão e criação das capas, e a dedicação incondicional de Renato Bobsin Machado, incentivador e revisor deste projeto, a quem agradeço incondicionalmente.

A série de livros intitulada **Pensamento Matemático @ Ciência da Computação** é aqui iniciada, no INE-UFSC, aberta aos que lidam com matemática, lógica e ciência da computação, a qual não seria possivel ser publicada facilmente, sem o template ABNTeX2, a liberdade de edição do autor na Agência Brasileira do ISBN, o serviço de catalogação da BU-UFSC e a projeção do sistema Repositório Institucional da UFSC.

Lista de ilustrações

Lista de tabelas

Sumário

Prefácio $\mathcal{A}$

Podemos dizer que dentre as atividades intelectuais humanas, as principais são a ciência e a filosofia. Dentre estas, as consideradas mais elevadas e mais antigas são a matemática, a música, e a filosofia. A música e a matemática, aliás, se separaram em épocas relativamente recentes. A aritmética, a geometria, a trigonometria, a álgebra e outros ramos da matemática estão intrinsecamente ligados à noção de quantidade mas a natureza da matemática como ciência e o foco de seu estudo estão, também, vinculados à filosofia e à história da matemática.

A música, por exemplo, seja ela de Bach, Beethoven, Jazz, Rock ou Bossa Nova não tem muito significado fora de seu contexto histórico. Em alguns casos, temos que levar em consideração o contexto temporal e espacial. Por exemplo, podemos até gostar da música de Tchaikovisky mas, certamente não a percebemos como um russo do século XIX a perceberia. São questões vinculadas ao ethos cultural de tempo e origem. Assim é a matemática. Postulados, conceitos e teoremas estão em um contexto histórico em que o homem, em seu desejo de aprender foi estabelecendo as bases da matemática durante a construção de sua trajetória rumo à civilização.

As civilizações egípcias, babilônicas e, posteriormente, a grega clássica criaram os fundamentos da matemática que se estabeleceram para evoluírem até os dias de hoje. Durante sua evolução histórica a matemática passou de um nível de simplicidade aritmética para um nível de maior complexidade nas suas áreas de domínio. Ou seja, do simples ato de contar às integrais de contorno, aos diferenciais de Laplace, às equações de Riemann, à constante de Plank, aos princípios como o de Heisenberg, aos teoremas de Gödel ou mesmo aos operadores como os hamiltonianos da mecânica quântica. Cada problema, solução ou teorema estão ligados à história que deve ser conceitualmente compreendida em relação ao tempo e espaço.

Não temos como entender a computabilidade sem a sua perspectiva histórica. As provas de Alonso Church e Alan Turing vieram de conceitos anteriores como os conceitos de *recursividade* de **Cantor**, da lógica combinatória de **Schönfinkel** e do λ-Cálculo com **Haskell Curry**. **Hilbert**, no início do século XX, propôs 23 problemas a serem resolvidos no século. Em 1910 apresentou 10 deles na conferência de Paris, dos quais apenas 4 foram resolvidos. Assim, liga-se a história ao tempo espaço dos problemas e a história liga **Hilbert** com **von Neumann**, **Heisenberg**, **Dirac**, **Schrödinger**, **Niels Bohr** e **Albert Einstein**, mostrando como as principais teorias da física moderna foram criadas.

A perspectiva histórica também é importante para compreender como Einstein chegou à relatividade e a sua busca pela teoria geral do universo, ou como Turing usou a computabilidade para resolver o problema da máquina Enigma, ou, ainda, como Heisenberg chegou ao princípio da incerteza buscando pela posição do elétron. Como todos se relacionam? Se relacionam por meio da transformação histórica de conceitos matemáticos que vão se juntando como um enorme quebra-cabeça. Só a história pode mostrar que a matemática que estudamos hoje foi gerada por um processo construído no tempo da jornada da humanidade. Uma jornada de milhares de anos de teorias obtidas pela ciência e experimento. A história, na verdade, é quem molda teorias. A relatividade de Einstein nunca foi sua principal teoria. Um dos matemáticos da teoria quântica era von Neumann que propos o modelo da computação determinística. Vejam que paradoxo! Um dos cientistas da teoria quântica, nada determinística, propõe um modelo de arquitetura de processador para o computador como uma máquina de estado determinística.

Podemos dizer que a ciência é investigativa mas grande parte nasce da criatividade, da intuição e da interatividade humana, ou seja, do processo histórico. Portanto, há que se diferenciar entre o modelo investigativo da matemática e o modelo cognitivo. Sendo a matemática uma ciência com base em postulados toda validação de resultados da matemática como modelo investigativo, é corroborado por ela mesma como ciência cognitiva. Desta forma podemos dizer que o contexto histórico é fundamental para a compreensão do seu conteúdo. Certamente, a matemática atingiu, hoje, uma importância crucial em todas as atividades humanas. Seja na economia, na engenharia ou na saúde, todos dependem de métodos matemáticos. Cada um dos ramos da matemática agora se subdividem em áreas com especificidades como as especialidades da engenharia ou da medicina.

A matemática tem uma teia de relações com várias áreas do conhecimento. As relações da matemática com a astronomia, a física e outras ciências podem ser observadas desde há muito tempo, em livros como os de Newton, Kepler, Pascal, Descartes e outros mais modernos como Gauss, Hamilton, Boole, Cantor, Riemann, entre outros. Assim forma-se a base para a um livro de suma importância. Um livro que é fundamental para a compreensão, pelo leitor, de como surgiram as ideias matemáticas. Antes de deduzir e demonstrar os teoremas principais ou mais elaborados da matemática, é importante mostrar ao leitor as bases a partir das quais foram enunciados os problemas e como foi conduzida sua solução com o alicerce matemático existente. Mas o livro não para por aí. Trata-se de uma leitura muito agradável que nos transporta à visão de problemas e soluções matemáticas em seu tempo; na sua origem; ou na sua fonte. Organizado, preferencialmente numa linha temporal, o livro aborda as diversas teorias matemáticas de uma maneira interdependente, mostrando que os conceitos fundamentais se mantém e se aplicam às descobertas matemáticas cada vez mais complexas.

Evitando a grande dificuldade de expor e demonstrar as modernas teorias rela-

tivísticas, quânticas ou das cordas o livro compensa pela facilidade de expor as ideias sobre elas e como foram elaboradas, questionadas e resolvidas. Torna-se mais fácil entender e mais evidente de se constatar as relações conceituais do homem e sua obra no contexto temporal da evolução da matemática. É como compreender de que maneira a matemática vai se revelando a si mesma e por si mesma. É a história permitindo a formação do homem acostumado à discussão matemática e ao pensamento matemático, mais do que um usuário habituado a formulários e receitas como ferramentas de soluções de problemas. Resta-me apenas desejar uma boa leitura.

Florianópolis, Dezembro de 2015

João Cândido Dovicchi
Departamento de Informática e Estatística da UFSC

Prefácio $\mathcal{B}$

A computação é uma ciência moderna e muito nova, e cujo estágio atual de desenvolvimento já a inseriu no cotidiano das pessoas e já a transformou em ferramenta viabilizadora e indispensável a um grande conjunto de atividades humanas em todo o mundo. É a parte do que define nossa sociedade hoje. Tal estágio de desenvolvimento e esse impacto social em nível global seriam impensáveis há apenas um único século. O que tornou isso possível e como se deu tão importante processo?

Ao contrário do que os mais jovens hoje possam pensar, devido ao seu convívio intenso com sistemas computacionais, os computadores não existiram sempre. Pasmem, mas a humanidade já sobreviveu por algumas dezenas de milhares de anos sem computadores e sem telefones celulares, e esses equipamentos não surgiram sozinhos. A concepção da computação como ciência e sua evolução tecnológica foram viabilizadas através do tempo e de um grande conjunto de trabalhos independentes de pessoas inovadoras, de visionários e de cientistas, alguns famosos e muitos heróis desconhecidos, e de muitas áreas, desde a matemática até a tecelagem. Os trabalhos e contribuições de cada uma desses "gigantes", em seu devido contexto histórico e social, se intersectaram e se aperfeiçoaram numa rede histórica de acontecimentos e de inovações que só pode ser considerada fantástica e empolgante pelos entusiastas da área de computação e de informática, e por qualquer pessoa curiosa de forma geral.

Além de fantástica e empolgante, essa rede histórica de desenvolvimentos tecnológicos e científicos constitui também um conhecimento muito importante aos profissionais da área de computação, possibilitando a eles ter um entendimento mais holístico de sua área de atuação e uma visão mais clara "do quê", "de quem" e "de como" a ciência da computação chegou a ser aquilo que é hoje. Embora a computação como a conhecemos tenha menos de um século e muitos nomes conhecidos pelos profissionais da área de computação sejam de pessoas do século XX, algumas ainda vivas, outros gigantes por trás delas desenvolveram seus trabalhos há bem mais tempo e em outros contextos, e são pouco reconhecidos.

Parte desse problema se deve à falta de uma referência bibliográfica que agregue tais informações, relacionado as pessoas e seus trabalhos num certo contexto histórico com sua contribuição a uma área específica da computação, que pode ter acontecido décadas depois. Alguns livros citam alguns desses trabalhos, outros livros citam outros. De meu conhecimento, nenhum é razoavelmente completo. Em obras introdutórias à ciência da computação encontra-se sempre um histórico descrito num

grau de detalhamento compatível a um capítulo de livro, mas não muito mais que isso. Informações mais aprofundadas podem ser encontradas dispersas em diferentes livros específicos sobre determinada área da computação, mas são informações específicas daquela área. Como profissional da computação, eu, até agora, desconhecia uma obra que fizesse uma revisão ampla dos trabalhos desenvolvidos ao longo da história e relacionasse sua contribuição a diferentes áreas da ciência computação, num nível de detalhamento que não é superficial, mas que também não é massante e nem profundo ao ponto de almejar ser uma referência daquela área específica. Foi isso o que o autor deste livro fez.

Este livro traz uma rica coleção de informações sobre trabalhos de diferentes visionários e cientistas, em ordem cronológica, e descreve como esses trabalhos contribuíram para o desenvolvimento de trabalhos posteriores e de áreas e técnicas específicas da computação, como a computabilidade em si, funções, recursividade, circuitos lógicos digitais, linguagens e paradigmas de programação, complexidade de algoritmos, criptografia, comunicação de dados, grafos, máquinas de estados finitos, tipos de dados, entre várias outras. Os nomes dos cientistas, que geralmente são apenas citados nos livros e permanecem anônimos, aqui são transformados em rostos sempre que possível, de modo que o leitor possa realmente ver e identificar as pessoas que contribuíram com a história da computação. O autor também incluiu, ao longo dos capítulos, um grande conjunto de curiosidades e de imagens de equipamentos, de produtos e de lugares, e que tornam a leitura mais agradável, e também costuma fazer paralelos com os dias e tecnologias atuais, tornando a leitura e o aprendizado mais práticos.

Neste livro o leitor não encontrará apenas uma descrição superficial das áreas e técnicas associadas às contribuições históricas apresentadas. Ao apresentar uma área ou técnica, como recursividade, grafos, ou complexidade de algoritmos, o autor a descreve formalmente. Modelos matemáticos e teoremas são apresentados, e o tema é efetivamente abordado. Considero isso ótimo. Preenche uma lacuna que existia em outras obras. O nível de profundidade em que o autor apresenta as informações neste livro, as tornam, em minha opinião, acessíveis aos estudantes de graduação em computação e em áreas afins, como engenharias, e mesmo acessível a pessoas relativamente leigas, desde que sejam familiarizadas com algumas técnicas computacionais e com notação matemática, e que possuam raciocínio lógico em certo grau. Por outro lado, este livro também reúne em cada capítulo um conjunto de hyperlinks e referências a outras obras onde o leitor mais avançado (ou que busca informações mais profundas) pode continuar sua leitura. Portanto, considero que esse livro (volume 2) é uma boa referência aos curiosos e entusiastas da computação, sejam eles relativamente leigos ou mesmo já conhecedores de alguns aspectos da história da computação e de algumas áreas da computação. Estou convencido que os leitores interessados em computação terão uma ótima leitura.

Florianópolis, Dezembro de 2015

Rapael Luiz Cancian

Departamento de Informática e Estatística da UFSC

Do Autor

Benvindo ao Volume II, da série "Do Pensamento Matemático à Ciência da Computação". No Volume I foi abordado a história da Matemática e Lógica, para se chegar à Ciência da Computação. Os capítulos enfatizaram desde os primórdios da Matemática e da Lógica, até finalmente abordar quais são os sistemas formais da Ciência da Computação. Sem proporcionar uma apresentação rigorosa, o Volume I mostra informalmente e historicamente, as grandes ideias na Matemática e na Lógica que redundaram na aparição da Ciência da Computação. Neste Volume II, a apresentação é histórica, mas um pouco mais formal, começando em **Alonzo Church** e **Alan Turing**, passando pela teoria da computabilidade, mostrando a história da pesquisa da teoria da complexidade, um capítulo sobre a visão abstrata de dados e chegando à Computação Quântica. Todos os capítulos enfatizam, o lado humano dos cientistas da computação.

A partir da extensão alcançada pela abrangência do assunto em um só livro, o vasto conteúdo que se pode escrever sobre o assunto, fez com que se possa criar vários volumes. Assim, para o tempo de 2014 e 2015, um segundo volume também foi organizado. O volume II também foi organizado através das atividades do mesmo projeto de pesquisa INE-UFSC 2013.1533, realizado no Departamento de Informática e Estatística da UFSC, que comportará, então, primeiramente, estes dois volumes. Os seus conteúdos dizem respeito ao elo existente entre a Matemática, a Lógica e a Ciência da Computação e, destina-se a servir como material de apoio para o ensino de matemática para alunos em Ciência da Computação.

O conteúdo aqui exposto, tenta mostrar como a matemática contribuiu para o surgimento da Ciência da Computação, colocando do ponto de vista histórico os diversos acontecimentos que surgiram de mentes geniais e que, no passado, redundaram diretamente na aparição da Ciência da Computação. As páginas deste livro mostrarão que a Matemática proporcionou as ideias para as soluções dos problemas teóricos e práticos que fizeram surgir a Ciência da Computação. O tema do livro tem uma história bastante abrangente. Afinal, onde quer que olhemos a Ciência da Computação, a Matemática e a Lógica permeiam nos fundamentos desta ciência. Sem a Matemática e a Lógica como base, a Ciência da Computação não existiria.

Assim, os assuntos dos capítulos foram escolhidos para enfatizar como os conceitos da Matemática foram utilizados na Ciência da Computação. O livro deve ser utilizado como complemento de estudo relativo a outros livros de Matemática

ou Ciência da Computação teórica. O principal objetivo é mostrar as origens dos fundamentos da Ciência da Computação e de conceitos computacionais atuais. Os seus capitulos mostram para que serve a matemática, quando se pensa em Ciência da Computação. O livro focaliza a *matemática discreta*, de maior utilização para a Ciência da Computação, desde as ideias de **Church** e **Turing**, até os anos 80-90 do século XX, quando surgiu a ideia dos tipos abstratos de dados na computação. Mas finaliza lembrando que já temos um futuro promissor para a ciência da computação atual: a computação quântica.

Este livro tem a missão de ajudar alunos de graduação de Ciência da Computação, ou mesmo os de níveis mais avançados, a se conectarem com a matemática discreta da computação. Toda a evolução histórica deste segundo volume, é mostrada nos capítulos apropriados, mostrando a participação de matemáticos e lógicos famosos e importantes, como **Alonzo Church**,**Alan Turing**, **John von Neumann**, **Emil Post**, **Claude Shannon** e tantos outros nos primórdios da Ciência da Computação. O autor tenta enfatizar as ideias e os conceitos, sem entrar no mérito das demonstrações matemáticas, que estão em outros livros mais específicos sobre a teoria matemática da computação. Este Volume II apresenta o lado humano das mentes geniais, enaltecendo as realizações dos grandes matemáticos e lógicos, no seu contexto histórico, cujas ideias influiram na aparição da Ciência da Computação.

Embora o Volume I não seja uma exigência prévia e indispensável para o Volume II, é recomendado que o leitor que não conheça as raízes da Ciência da Computação, leia o Volume I, porque foram incluido tópicos que não são aqui cobertos e o leitor pode se habituar com a notação e terminologia matemática e lógica, usadas. Os tópicos agora abordados estão nos capítulos 1-15. Como no capítulo 1 do Volume I, começamos com aspectos hostóricos da Ciência da Computação para introduzir o restante deste livro.

Entre os princípios que nortearam o autor na apresentação do seu material no volume II, destacam-se os seguintes: (1) Relacionar as tendências da matemática aritmético-algébrica e da lógica, ambas formalistas, em direção à construção da Ciência da Computação. (3) Se a Matemática e a Lógica são uma grande aventura nas ideias, as suas histórias refletem alguns dos mais nobres pensamentos de inúmeras gerações. Como consequência, a Ciência da Computação também comporta ideias geniais de gerações mais recentes, mas sobretudo, numa evolução muito mais rápida, talvez pela grande contribuição dada pela Matemática, pela Lógica e pela aparição da microeletrônica. Assim, o autor vislumbrou, a princípio, a organização de dois volumes, dividindo em duas partes, baseando a exposição do surgimento da Ciência da Computação, em termos das personalidades desta ciência, as verdadeiras donas das ideias, as vezes narrando fatos e suas ideias geniais, em vez de tratar especificamente os assuntos. Os vários assuntos aqui abordados estão, de forma muito mais aprofundada, em outros livros mais específicos da Ciência da Computação teórica. Daí, o autor não se preocupar com demonstrações matemáticas que estão em outros livros, dando preferência aos conceitos chaves, tentados ser explicados de forma mais

explícita.

Por vezes, uma referência bibliográfica substitui uma análise histórica. A história da Ciência da Computação, como colocada aqui, começou nos tempos das calculadoras mecânicas, a pré-história tecnológica dos computadores. E em termos de lógica e matemática, começamos em **Alonzo Church** e **Alan Turing**, indo até **Claude Shannon**, mas acrescentando alguns capítulos para ressaltar os benefícios da matemática, sobre modelagen em grafos e a visão abstrata de dados em computação. A seleção do material foi baseada exclusivamente levando-se em conta os personagens da lógica e da matemática, e até na breve história das ideias que redundaram nos primeiros computadores . Nem sempre foi possivel consultar todas as fontes em primeira mão. Algumas vezes, foram ulilizadas fontes de segunda. O autor reconhece que é um bom princípio, conferir as afirmações tanto quanto possivel pelas fontes originais. Entretanto, no tempo disponível ao autor e com os recursos de que dispôs, as citações decorreram, de vários livros de sua própria biblioteca sobre matemática, lógica e computação. Além dos recursos próprios de bibliografia do autor, fiz uso extensivo de diversos sites de Internet, em especial, **www.wikipedia.org**, e alguns outros, pelo conteúdo interessante que apresentaram. Também, pelos recursos financeiros disponíveis, nestes dois volumes, o editor é o próprio autor, conforme as regras da Agência Brasileira do ISBN.

Florianópolis, Dezembro de 2015

João Bosco M. Sobral

CAPÍTULO 1

O Embrião da Ciência da Computação

Vimos no Volume I como a matemática e a lógica exerceram papel fundamental no surgimento da Ciência da Computação. A relevância da lógica se assemelha ao que o cálculo representou para a físicca e para a engenharia tradicional. A lógica proporciona as ferramentas intelectuais para se estudar inteligência artificial, engenharia de software, programação automática para prova de teoremas, teoria dos bancos, lógica de programação e teoria da computação. Do ponto de vista matemático, as ideias de **Dedekind**, **Gödel**, **Hebrand**, **Kleene** e **Rosser**, criaram as funções recursivas primitivas, direcionando os primeiros passos para a criação dos modelos computacionais dos anos 30 em direção à teoria da computabilidade.

A lógica foi incorporada na computabilidade de algoritmos executando as funções de (**Turing**) e, na década seguinte, foi transformada em circuitos elétricos de (**Shanonn**). A matemática da teoria dos grafos, surgida com Euler em 1736, pensada por **Turing**, vislumbrada por **Kleene** em 1951 e desenvolvida nos últimos anos de vida de **John von Neumann** , evoluiu proporcionando o modelo dos autômatos finitos, e as redes de **Carl Petri**. Tais redes foram iniciadas em 1939, tornando a computabilidade viável e verificável. Na década de 60, a teoria da computabilidade deu origem à teoria da complexidade e se formaram três áreas, hoje, tradicionalmente, centrais da teoria da computação: autômatos, computabilidade e complexidade, proporcionando as capacidades fundamentais dos computadores atuais.

1.1 Da Matemática à Ciência da Computação

Como mostrado no Volume I, um marco inicial na história da computação foi o *Entscheidungsproblem* (problema de decisão) de **David Hilbert**. Ele acreditava que todos os problemas poderiam ser resolvidos com algum procedimento efetivo, o qual consistia em encontrar um procedimento para se resolver um problema matemático. Em 1931, **Kurt Gödel**, com o teorema da incompletude , demonstrou que, como

propostos por **Hilbert**, certos problemas não tinham como ter solução nos sistemas formais axiomáticos como proposto por **Hilbert**. A classe de funções usada por **Gödel** foi a das *funções primitivas recursivas* definidas anteriormente por **Dedekind** em 1888 e depois, propostas por **Jacques Herbrand** . **Gödel** levou a discussão a frente, no sentido de identificar um formalismo para definir a noção de *procedimento efetivo*.

A ideia de *procedimentos recursivos* já era considerada valiosa, porque através de *procedimentos recursivos finitos* poderia-se chegar a conjuntos de resultados *infinitos*. No início dos anos 30, **Church** fazia seu trabalho pensando em funções. **Church** usou dois formalismos para estudar o problema efetivamente computável: λ-calculus (Church, 1936) e as funções recursivas (Kleene, 1936). **Church** definia procedimentos efetivos como "*caracterizações tão gerais da noção do efetivamente computável, quanto consistentes com o entendimento intuitivo usual*". A teoria da recursão foi, então, originada com o trabalho de **Kurt Gödel**, **Alonzo Church**, **Stephen Kleene**, **John Barkley Rosser** (1907-1989), **Alan Turing** e **Emil Post** nos anos 30.

Turing já imaginava o que poderia ser uma máquina programável. Em 1936 **Turing** propôs um formalismo para a representação de procedimentos efetivos. Esse foi o primeiro trabalho a identificar procedimentos para uma "máquina computacional autômata abstrata", com as noções intuitivas de efetividade. Desde então, diversas pesquisas foram desenvolvidas, outros formalismos foram propostos, com o intuito de definir um modelo computacional suficientemente genérico, capaz de implementar qualquer função computável, os quais possuem o mesmo poder computacional que as funções recursivas.

Surgiu, então, a Ciência da Computação , como o conhecimento sistematizado relativo à computação. A partir daí, **Turing** passou a imaginar a lógica computacional para o funcionamento dessa máquina abstrata, e proporcionar o que seria chamado de *procedimento efetivo*, a conceituação do que hoje chamamos *algoritmo* . Assim, se começou a indagar o que poderia ser uma máquina programável e de uso geral. Tal máquina era idealizada considerando um modelo computacional. Neste contexto surgiram os modelos computacionais da (a) máquina de Turing (1936); o sistema canônico de Post (1943); o modelo do algoritmo de Markov (1954); o modelo das máquinas de registradores (1963) e o modelo Random Acess Stored Programs (RASP) (1964). Um algoritmo é, então, definido como sendo um procedimento efetivo, que pode ser descrito usando qualquer destes formalismos equivalentes. Ou seja, sabe-se hoje que, qualquer destes formalismos permitia descrever todos os procedimentos possíveis que podem ser executados em um computador.

Como qualquer modelo científico, o computacional pode ser preciso em alguns aspectos e, possivelmente, impreciso em outros. Assim, surgiram vários modelos computacionais diferentes, e dependendo do que se desejasse focalizar, um ou outro é melhor para ser utilizado. Por exemplo, a pesquisa de **Stephen Kleene** era sobre a

teoria de algoritmos e teoria de funções recursivas, uma área que ele desenvolveu. Ele desenvolveu o campo da teoria da recursão juntamente com **Gödel**, **Church**, **Rosser**, **Turing**, **Post** e outros da época. O trabalho de **Kleene** na teoria da recursão ajudou a fornecer os fundamentos da Ciência da Computação teórica. Ao proporcionar métodos para determinar quais os problemas que são solúveis, o trabalho de **Kleene** levou ao estudo de funções que poderiam ser computadas. A partir dos desenhos de **Turing**, imaginando redes neurais nos anos 30, **Kleene**, no verão de 1951, descobriu uma caracterização importante de *autômatos finitos*, e seu trabalho tem sido muito influente para a Ciência da Computação teórica.

A respeito do trabalho de **Rosser**, **John Barkley Rosser** (1907-1989) foi um lógico estadunidense, estudante de **Church** contemporâneo de **Kleene** e **Turing** na Princeton University, conhecido pela sua participação no teorema *Church-Rosser* e no λ-calculus. Em 1936, ele provou o *Rosser's trick* , uma versão mais forte do teorema da incompletude de Gödel. Este método foi introduzido como um melhoramento da prova original de Gödel em 1931. **Rosser** também participou da teoria dos números primos tendo provado o *Rosser's theorem* em 1938, que diz que, se p_n é o n-ésimo número primo, então, para qualquer $n > 1$, tem-se $p_n > n.ln\ n$. O paradoxo de Kleene-Rosser mostrou que o λ-cálculo original era inconsistente, o que fez com que **Church** mudasse seu objetivo de ter um cálculo, passando a trabalhar com a parte da descrição de funções que havia definido para o λ-cálculo original Zach (2006).

O conceito de máquina então passa a ser a interpretação dos algoritmos de acordo com os dados fornecidos. Uma máquina seria capaz de interpretar um algoritmo, desde que possuisse uma interpretação para cada operação ou teste que constituia o algorimto. Uma *computação* seria um histórico do funcionamento da máquina para o algoritmo, considerando um valor inicial. E o conceito de *função computada* seria uma função (parcial), induzida a partir da máquina e dos dados, e definida sempre que, para um dado valor de entrada, existisse uma computação finita (a máquina pára). Algoritmos e máquinas são, então, tratados como entidades distintas, mas complementares e necessárias para a definição de computação Diverio e Menezes (2011).

O estudo da *teoria dos grafos* engatinhou do século XVIII (1736, quando surgiu a ideia por **Euler**) até o século XX, quando anos 40 e 50 começaram a surgir os primeiros computadores digitais. Daí em diante, o estudo e a aplicação da *teoria dos grafos* foi acelerado e praticamente tornou-se uma ferramenta para a solução de problemas, muito útil para a Ciência da Computação, como existem a *teoria dos autômatos* e a *teoria das redes de Petri*, iniciada em 1939, desenvolvida por **Carl Petri** (1926-2010), um matemático alemão e cientista da computação.

A partir de Kleene (1951), **John von Neumann** passou uma parte considerável dos últimos anos de sua vida trabalhando a **teoria de autômatos**, pensando em definições e propriedades de modelos matemáticos de computação. Envolvendo uma mistura de matemática pura e aplicada, bem como outras ciências, a teoria de

autômatos desempenha papel importante em diversas áreas aplicadas da Ciência da Computação. A teoria é uma maneira excelente de se começar a estudar a base da computação, já que as teorias da computabilidade e da complexidade exigem uma definição precisa de um computador.

Um dos modelos mais simples, baseados em grafos, são os autômatos finitos, que surgiu das ideias desenvolvidas por **Turing**, **Kleene** e **John von Neumann**. Os autômatos finitos são bons modelos para computadores com uma quantidade extremamente limitada de memória. O que se pode fazer com um computador de memória tão pequena? Nos dias atuais, interagimos com esses computadores o tempo todo, pois eles residem embarcados em vários dispositivos eletromecânicos atuais. Um exemplo é o de um controlador de uma porta automática. O projeto desses dispositivos requer que se tenha em mente a teoria, a terminologia e a metodologia dos autômatos finitos Sipser (2011).

Como contrapartidas probabilísticas para os autômatos finitos, existem as **cadeias de Markov** que são modelos úteis quando estamos interessados em tentar reconhecer padrões em dados. **Cadeias de Markov** podem ser utilizadas para reconhecimento de voz ou em outros tipos de reconhecimento óticos. São usadas para fazer previsão de mudança de preços no mercado financeiro Gallager (2014) e Coleman (1974).

1.2 A Necessidade das Linguagens

Se formos estudar a teoria dos autômatos finitos de uma perspectiva matemática, fazemos isso num nível abstrato, sem referência a qualquer aplicação específica. Mas, o que é importante na definição de um autômato é a utilização do alfabeto de uma *linguagem*. Autômatos devem prover a definição de uma computação e reconhecer linguagens, constituídas de cadeias de símbolos de um alfabeto em que se constrói a linguagem. Assim, foram criadas as , que são as que algum autômato as reconhecem. Mesmo do ponto de vista teórico, como autômatos são construídos, nota-se que esses autômatos precisam ser "alimentados"com cadeias de símbolos de algum alfabeto de alguma linguagem, como em Sudkamp (1988).

Das pequenas linguagens com alfabetos mínimos, a Ciência da Computação, na medida em que foram surgindo os grandes projetos de computadores, cada vez mais surgia a necessidade do homem interagir com a máquina. Dos códigos binários iniciais nascia tal necessidade da interação homem-máquina, mas imbuida da complexidade em se ter que programar um computador através de seu código nativo: a linguagem binária nativa contém somente dois símbolos no seu alfabeto, mas apresenta difícil entendimento do seu significado. Assim, nasceu a necessidade de se ter outros níveis de linguagens de comunicação com a máquina. Onde se precisa de comunicação entre duas partes, surge uma linguagem essencial à interação entre essas duas partes. Assim foram concebidas as linguagens de programação de computadores. Na medida em que o usuário do computador percebia sua dificuldade de interação, passava a trabalhar

com linguagens de nível mais alto de abstração. Entretanto, precisando traduzir da linguagem de nível mais alto, para a de nível mais baixo, já que é neste nível baixo de abstração que o computador foi construído para funcionar. Por isso, em 1952, **Grace Hopper** desenvolveu o primeiro tradutor de linguagem computacional, que veio a ser chamado de *compilador*.

Decorrente do conhecimento sobre linguagens naturais, que são mantidas por uma sintaxe e uma semântica, surgidas das formas convencionais com que os povos se comunicam entre si e entre povos, surgem as , alicerçadas por uma classificação criada por **Avram Noam Chomsky** (1928) que numerou quatro famílias de gramáticas (e linguagens) que compreendem uma hierarquia. **Chomsky** pesquisou vários tipos de procurando entender se poderiam ser capazes de capturar as propriedades-chave das línguas humanas e associá-las ao uso por usuários de computador.

Figura 1 – Noam Chomsky - Quatro famílias de linguagens e gramáticas.

Fonte: libraries.mit.edu.

A hierarquia de **Chomsky** divide as gramáticas formais em classes com poder expressivo crescente, por exemplo, cada classe sucessiva pode gerar um conjunto mais amplo de linguagens formais que a classe imediatamente anterior. Além de ser relevante em linguística, a hierarquia de **Chomsky** também tornou-se importante em Ciência da Computação, especialmente aplicada na teoria de autômatos e na construção de compiladores de linguagens.

Portanto, quando se pensa em linguagens e máquinas, acabamos por entrar no campo da teoria da computação teórica, para abordar os *métodos formais de descrever as linguagens* através de suas gramáticas (sintaxe). No que tange as semânticas de linguagens, a matemática e a lógica também contribuem com os métodos da semântica operacional (baseado em computações), método axiomático (baseado em regras de inferência do cálculo de uma lógica) e o método da semântica denotacional (baseado na criação de funções semânticas). A matemática e a lógica dos métodos semânticos de linguagens de computador, será assunto de outro volume previsto pelo autor.

1.3 Para os Nossos Propósitos ...

Estamos no século XX, nos anos 30. Neste volume II focalizaremos a essência da Ciência da Computação já existindo. Além deste capítulo inicial, começamos mostrando as várias calculadoras mecânicas do século XIX, que motivaram o surgimento no século XX das máquinas programáveis. Passamos da computação sem computador, abordando as funções de **Alonzo Church** nos início dos anos 30, e da ideia genial de **Alan Turing** definido uma maquina abstrata e algoritmo, para o caminho conduzindo à teoria da computabilidade. Enfatizamos a contribuição de **Emil Post** sobre a computabilidade e a lógica proposicional. Um capítulo está organizado para as funções computáveis μ-recursivas, e chegamos a **Claude Shannon**, trasformando a álgebra de uma lógica em circuitos elétricos. Focalizamos o legado de **John von Neumann**. Surgem os primeiros projetos de computadores digitais. Da *teoria da computabilidade* dos anos 30, surgiu a *teoria da complexidade*, do final dos anos 50 ao início dos anos 70 (o período mai produtivo). A *teoria dos grafos*, a partir dos anos 50, toma impulso e contribui com vários modelos de computação: os autômatos e as redes de Petri que estendem autômatos.

E dos paradoxos da teoria dos conjuntos de Cantor, surge a *teoria dos tipos* de **Bertrand Russell** e **Martin Löf**, e, posteriormente, os tipos abstratos de dados.

1.4 Bibliografia e Fonte de Consulta

Michael Sipser (2011) - Introdução à Teoria da Computação, 2 Ed. Cengage-Learning, 2011.

Teoria dos grafos - ⟨https://pt.wikipedia.org/wiki/Teoria_dos_grafos⟩

Languages and Machines: an Introduction to the Theory of Computer Science - Thomas Sudkamp, 1988.

Teoria da Computação: Máquinas Universais e Computabilidade - Tiaraju Diverio and Paulo Blauth Menezes, Editora Bookman, 2011.

1.5 Referências e Leitura Recomendada

Noan Chomsky - ⟨https://pt.wikipedia.org/wiki/Noam_Chomsky#Hierarquia_de_Chomsky⟩

Rosser, J. B. “The nth Prime is Greater than n.ln n”. Proceedings of the London Mathematical Society 45, 21-44, 1938.

Teoria da Computabilidade - ⟨https://pt.wikipedia.org/wiki/Teoria_da_computabilidade⟩

Richard Bird - Programs and Machines: An Introduction to the Theory of Computation, 1976.

Richard Zach - Kurt Gödel and Computability Theory, Research supported by the Social Sciences and Humanities Research Council of Canada, 2006.

CAPÍTULO 2

Calculadoras Mecânicas - a Pré-História dos Computadores

Este capítulo destaca o surgimento das primeiras máquinas de calcular, inclindo a máquina Antikythera (Anticythère) dos gregos no século II a.C., a máquina de **Schickard**, passando por **Blaise Pascal** , **Leibniz** , **Charles Thomas** , **Charles Babbage** , **W. T. Odhner** e **Baldwin**. Aborda-se a primeira idéia de computador de uso geral, a máquina analítica de **Charles Babbage**, as ideias de**Ada Lovelace** e a máquina de **Willian S. Burroughs**, este último o fundador da *Burroughs Corporation* no final do século XIX. Muitas máquinas Burroughs foram vendidas no Brasil. No capítulo também é abordada a ideia de Jacquard que programou um *tear* com cartões perfurados e que deu origem a **Herman Hollerith** para uma máquina de perfurar cartões, fundando a IBM (*The International Business Machines*), na segunda década do século XX. Cartões perfurados eram o meio pelo qual usuários de computadores programavam suas tarefas nos anos 1960-1970. Estes personagens fizeram a pré-história dos computadores de hoje.

2.1 A Máquina Antikythera (Anticythère)

Em Galvao (2007) é contada brevemente a história da máquina Antikythera (Anticythère). Como fizeram em tantos ramos do conhecimento, os gregos começaram a ideia de construir máquinas de computar. Construída no Século II a.C. o mecanismo da Antikythera (Anticythère) era um computador astronômico, mecânico, com muitas engrenagens, que permitia calcular as posições do sol e da lua, prever eclipses. Este artefato que se acredita tratar-se de um antigo mecanismo para auxílio à navegação. Infelizmente essa tecnologia relativamente avançada para a época foi perdida. Parte desta história só começou a ser esclarecida em 1901, quando um pescador encontrou por acaso um antigo naufrágio perto da Ilha de Antikythera (Anticythère), na Grécia. Entre outras coisas, foram descobertos os restos do que ficou conhecido como a máquina de Antikythera (Anticythère). Os restos do artefato foram resgatados, juntamente com várias estátuas e outros objetos, por mergulhadores, à

profundidade de aproximadamente 43 metros na costa da ilha grega de Antikythera (Anticythère), entre a ilha de Cítera e a de Creta. O artefato parecia um relógio, mas isso era pouco provável porque se acreditava que relógios mecânicos só passaram a ser usados amplamente muito mais tarde.

O mecanismo original descoberto está exposto na coleção de bronze do Museu Arqueológico Nacional de Atenas, acompanhado de uma réplica. Outra réplica está exposta no Museu Americano do Computador em Bozeman (Montana), nos Estados Unidos da América. Essa antiga máquina grega e muitos outros mecanismos de cálculo inventados até o seculo XVII tinham a limitação de que cada um servia somente para realizar uma tarefa. Eram mecanismos dedicados a operações específicas. O leitor pode também conhecer o dispositivo Antikythera (Anticythère) em Bolter (1984).

Figura 2 – O mecanismo de Anticythère

Fonte: observatorio.ufmg.br.

Passaram-se muito séculos antes do aparecimento de dispositivos mecânicos com sofisticação comparável. Como projetar uma máquina mais flexível o suficiente para realizar outros tipos de tarefas? A resposta mesmo, só começou a ser obtida a partir do século XVII, como contado no que segue.

2.2 A Mecânica das Calculadoras

Aproximadamente em 62 d.C., **Herao Hierao Heron Hieron Alexandria** (10 d.C-80 d.C), mais conhecido por **Heron de Alexandria** descreve duas idéias: (1) **a ligação de rodas dentadas de maneira a realizar a operação de vai um**, e (2) **a utilização de cilindros rotatórios com pinos e cordas para controlar**

sequências de ações de outros mecanismos. Heron de Alexandria foi um sábio matemático e mecânico grego. É de sua autoria um tratado chamado *Métrica*, que versa sobre a medição de figuras simples de planos sólidos, com prova das fórmulas envolvidas no processo. Tratava da divisão das figuras planas e sólidas e contém a fórmula de Herão (embora esta talvez tenha sido descoberta por Arquimedes) para o cálculo da área de um triângulo e um método (já antecipado pelos babilônios) de aproximação a uma raiz quadrada de números não quadrados.

Sua mecânica foi preservada pelos árabes e anuncia a regra do paralelogramo para a composição de velocidades. Determina os centros simples de gravidade e discute as engrenagens pelas quais uma pequena força pode ser usada para levantar grandes pesos.

Figura 3 – Heron - Inventor, matemático, engenheiro e escritor grego, que realizou excelentes trabalhos em Mecânica.

Fonte: www.dec.ufcg.edu.br.

Ele escreveu um manual de *poliorcética* (o termo é utilizado na arquitetura militar, como a arte de construir e aplicar máquinas bélicas para bater muros e expurgar fortalezas), que foi usado com uma das fontes por um autor bizantino anônimo, para escrever o livro *Parangelmata Poliorcetica* (Instruções para a Guerra de Cerco).

Ficou conhecido por inventar um mecanismo para provar a pressão do ar sobre os corpos, que ficou para a história como o primeiro motor a vapor documentado, a *eolípila* (a máquina térmica de Heron, um aparelh para medir a força do vapor). E

seu trabalho contribuiu valiosamente para o surgimento dos motores à propulsão que temos hoje, movendo veículos terrestres. A máquina à vapor deu origem aos sistemas de propulsão que ao longo dos tempos históricos nos trouxeram o que temos hoje.

2.3 Logaritmos e os Primeiros Dispositivos Mecânicos de Cálculo

John Napier (1550-1617), na Figura 4, foi um matemático, físico, astrônomo, astrólogo e teólogo escocês. É conhecido como o descodificador do logaritmo natural e por ter popularizado o ponto decimal. Sua mais notável realização foi a descoberta dos logaritmos, artifício que simplificou os cálculos aritméticos e assentou as bases para a formulação de princípios fundamentais da análise combinatória. Mas também gastou grande parte de sua vida inventando instrumentos para ajudar no cálculo aritmético, principalmente para o uso de sua primeira tabela de logaritmos. Na decodificação dos logaritmos naturais, **Napier** usou uma constante que, embora não a tenha descrito, foi a primeira referência ao notável número "e", descrito quase 100 anos depois por **Leonhard Euler**, e que se tornou conhecido como número de Euler ou número de Napier, a base dos logaritmos neperianos (logaritmos naturais). O número "e" Maor (2003), Brasil (2007) não foi inventado, mas descoberto como fazendo parte da natureza. A discussão de Napier sobre logaritmos aparece em *Mirifici logarithmorum canonis* descrito em 1614. Dois anos mais tarde (1616), uma tradução em inglês do texto original em latim de Napier foi publicado. No prefácio do livro, **Napier** explica seu pensamento por trás de sua grande descoberta.

Figura 4 – John Napier - O criador dos logaritmos naturais.

Fonte: www.thocp.net/biographies/napier$_j$*ohn.html*.

No início do século XVII, inventou um dispositivo chamado *Ossos de Napier* que são tabelas de multiplicação gravadas em bastão, permitindo multiplicar e dividir de forma automática, o que evitava a memorização da tabuada, e que trouxe grande auxílio ao uso de logaritmos, em execução de operações aritméticas como multiplicações

e divisões longas. Idealizou também um calculador com cartões que permitia a realização de multiplicações, que recebeu o nome de Estruturas de Napier.

A partir dos logaritmos de **Napier** surgiu uma outra grande invenção, desenvolvida pelo brilhante matemático **Willian Oughtred** (1575-1660), um matemático inglês, e tornada pública em 1630: a régua de cálculo. Ver **Oughtred** na Figura 5. Oughtred foi autor de *Clavis mathematicae*, em que recapitulou todos os conhecimentos da sua época referentes à álgebra e à aritmética. Introduziu o símbolo "×" para a multiplicação e atribui-se-lhe também as réguas de cálculo**Oughtred** foi criador da régua de cálculo, que ganhou sua forma atual por volta do ano de 1650 (de uma régua que se move entre dois outros blocos fixos). Tendo sido esquecida por duzentos anos, tornou-se no século XX o grande símbolo de avanço tecnológico, com uso bastante difundido, até ser definitivamente substituída pelas calculadoras eletrônicas.

Figura 5 – Willian Oughtred - O criador da régua de cálculo em 1622.

Fonte: pt.wikipedia.org.

Foi inventada pelo padre inglês William Oughtred, em 1622, baseando-se na tábua de logaritmos que fôra criada por John Napier pouco antes, em 1614. A régua de cálculo era um aparato de cálculo que se baseiava na sobreposição de escalas logarítmicas. Os cálculos eram realizados através de uma técnica mecânica analógica que permitia a elaboração dos cálculos por meio de guias deslizantes graduadas, ou seja, réguas logarítmicas que deslizam umas sobre as outras, e os valores mostrados em suas escalas são relacionados através da ligação por um cursor dotado de linhas estrategicamente dispostas, que têm a função de correlacionar as diversas escalas da

régua de cálculo.

Figura 6 – Uma típica régua de cálculo circular.

Fonte: https://pt.wikipedia.org/wiki/Régua_de_cálculo.

Apesar de todas elas se parecerem, existiam muitas variações de tipo de régua, quanto a sua aplicação, diferença esta que fica por conta das escalas presentes na régua de cálculo. Além das diferentes disponibilidades de escalas, elas também podiam ser circulares.

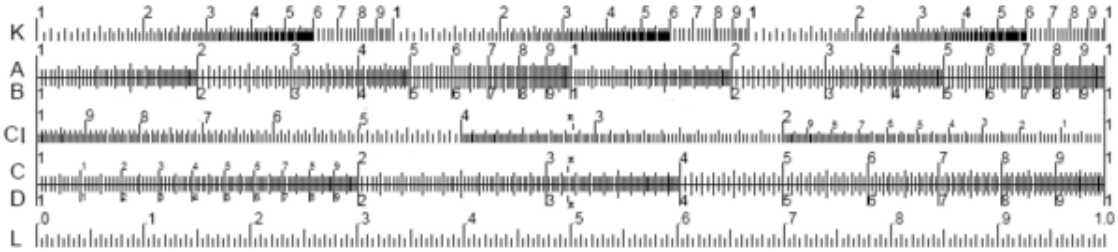


Figura 7 – Uma régua de cálculo do tipo mais convencional usada nos anos 1960, com as escalas mais comuns.

Fonte: Fonte: https://pt.wikipedia.org/wiki/Régua_de_cálculo.

Na prática, cada tipo de régua se destinava a uma aplicação específica, em função de suas escalas e de seu tipo, mas no mínimo as operações básicas eram todas realizáveis.

Em geral, operações de adição/subtração feitas a mão (com lápis e papel) são extremamente mais simples que todas as demais operações. São nas outras operações que as réguas de cálculos entram para facilitar o trabalho, e elas fazem isso convertendo para uma multiplicação numa soma, ou uma divisão para uma simples subtração. Isso é feito levando-se em conta as seguintes propriedades logarítmicas:

$$log(A \times B) = log\ A + log\ B \quad \text{e} \quad log(A/B) = log\ A - log\ B$$

Como as escalas da régua são logarítmicas quando se localiza na régua os ponto

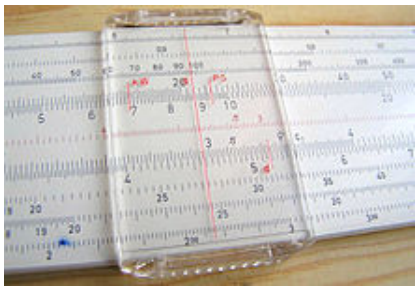


Figura 8 – O cursor de uma régua de cálculo nos anos 1960.

Fonte: https://pt.wikipedia.org/wiki/Régua_de_cálculo.

A e B, na verdade estamos localizando a distância logarítmica em que este ponto está contando do começo da régua. Quando se somam estas duas distâncias iremos obter na prática uma distância que é a distância do valor da multiplicação dos dois valores (como a primeira expressão acima prova). Se subtrairmos estas distâncias, então estaríamos dividindo um valor pelo outro.

Apesar da semelhança com uma régua, é uma régua com propriedades logaritmicas. A régua de cálculo é um dispositivo que não tem nada a ver com medição de pequenas distâncias ou traçagem de retas. A régua de cálculo é a mãe das calculadoras eletrônicas modernas, porque trabalha com logaritmos (até mesmo porque os engenheiros que criaram as calculadoras eletrônicas provavelmente fizeram isso usando réguas de cálculo nas suas funções iniciais), tendo sido largamente usada até a década de 1960 (qualquer aluno de escola técnica, três séculos e meio depois de inventada, no século XX, nos anos 60, ainda possuia uma régua de cálculo), quando então a versão eletrônica foi largamente difundida, porque superou a régua de cálculo e foi muito bem aceita, em função de sua simplicidade e precisão.

2.4 Schickard em 1623

A pré-história dos computadores remonta ao alemão **Wilhelm Schickard** (1592-1635), professor de astronomia na Universidade de Tübingen. Era contemporâneo e amigo de **Johannes Kepler** (1571-1630) foi um astrônomo e matemático alemão, considerado figura-chave da revolução científica do século XVII, de **Blaise Pascal** (1623-1662) e **Leibniz**. Mas, bem antes de **Pascal** e **Leibniz**, **Schickard** é considerado como o primeiro a construir, em 1623, uma **máquina de calcular mecânica** (utilizada por **Kepler**), capaz de realizar as 4 operações básicas com números de seis dígitos e indicar um *overflow* através do toque de um sino. Assim, a primeira máquina de verdade foi construída por **Wilhelm Schickard**, sendo capaz de somar, subtrair, multiplicar e dividir.

Durante muitos anos nada se soube sobre essa máquina, sendo que mais recentemente foi encontrada alguma documentação sobre ela. Foram encontradas algumas cartas de **Schickard**, enviadas a seu a amigo Kepler em 1624, acompanhadas de

vários esboços, onde explica o desenho e o funcionamento de uma máquina que havia construído e que que chamou de *relógio calculador*. Explicava que havia mandado construir um exemplar da máquina para ele, mas que fora destruída em um misterioso incêndio noturno ocorrido em sua casa, juntamente com alguns outros pertences. Os esboços do desenho estiveram perdidos até o século XIX. Por isso, atribui-se a **Blaise Pascal** (1623-1662)a construção da primeira máquina calculadora, que fazia apenas somas e subtrações.

Figura 9 – Schickard em 1623 - O pioneiro a construir uma máquina de calcular mecânica.

Fonte: history-computer.com.

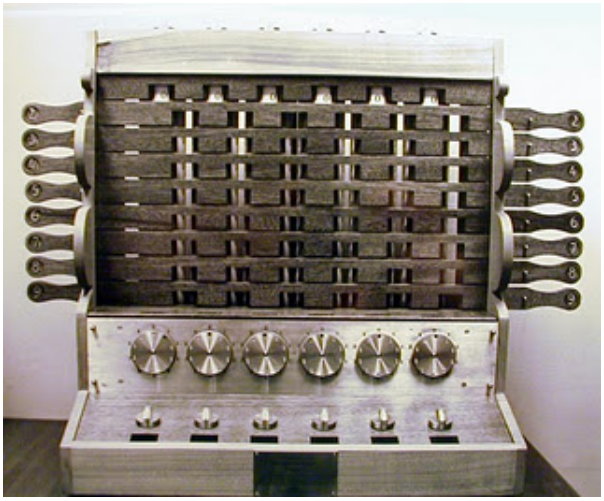


Figura 10 – Schickard - Essa sim, é a primeira máquina de calcular.

Fonte: computermuseum.li.

2.5 Blaise Pascal em 1642

Embora a teoria dos logarítmos de **Napier** passasse a ter aplicação permanente, logo foram substituídos pela régua de cálculo e outros tipos de calculadoras - especi-

almente por um pioneiro aparelho mecânico criado por Blaise Pascal, brilhante sábio francês. Filho de um coletor regional de impostos, Blaise Pascal tinha somente 19 anos quando começou a construir sua calculadora, em 1642, inspirado nos enfadonhos cálculos do trabalho de seu pai. Ao morrer, com 39 anos, havia passado para a História como grande matemático, físico, escritor e filósofo.

A máquina de **Pascal** foi criada com objetivo de ajudar o pai de Pascal a computar os impostos em Rouen, França. A máquina de Pascal, a *Pascaline*, era uma caixa com rodas e engrenagens da qual ele construiu mais de cinqüenta versões ao longo de uma década. O operador introduzia os algarismos a serem somados, "discando-os" , numa série de rodas dentadas, com algarismos de zero a nove, impressos de modo que os números a serem somados ficassem expostos num mostrador. Cada roda representava uma determinada coluna decimal - unidades, dezenas, centenas, e assim por diante. Uma roda, ao completar um giro, avançava em um dígito a roda à sua esquerda, de ordem decimal mais alta. A máquina também executava outras operações por meio de um sistema de adições repetitivas. Enfim, a máquina de Pascal só podia somar e subtrair. A máquina de somar de **Blaise Pascal** adiciona ou subtrai quando as rodas dentadas se engrenavam, ao serem giradas. Um giro leva um total superior a 9 para a coluna à esquerda. O resultado aparece no mostrador: os números da extrema direita para a adição e os da direita para a subtração.

Figura 11 – Pascal - Contribuiu para a Matemática, Física e a Filosofia de Matemática.

Fonte: brasilescola.uol.com.br.

Figura 12 – La Pascaline - A primeira calculadora mecânica do mundo, planejada por Blaise Pascal em 1642.

Fonte: hestoriadopc.wordpress.com.

2.6 Leibniz em 1672

O mais sério inconveniente da Pascaline era seu método convolutivo em aspiral, de executar quaisquer outros tipos de cálculo além da simples adição.

Trinta anos mais tarde, em 1672, durante uma temporada em Paris, o matemático **Leibniz** tentou melhorar a máquina de Pascal com uma "calculadora de passos" que tinha a capacidade de multiplicar e dividir começou a estudar com o matemático e astrônomo holandês **Christian Huygens**. A experiência estimulou-o a procurar descobrir um método mecânico de aliviar as intermináveis tarefas de cálculo dos astrônomos. Astrônomos perdiam horas e horas, em trabalhos de cálculo que poderiam, confiavelmente, ficar a cargo de qualquer pessoa, caso se usassem máquinas de calcular. No ano seguinte, ficou pronta sua calculadora mecânica, que se distinguia por possuir três elementos significativos. A porção aditiva era, essencialmente, idêntica à da *Pascaline*, mas **Leibniz** incluiu um componente móvel (precursor do carro móvel das calculadoras de mesa posteriores) e uma manivela manual, que ficava ao lado e acionava uma roda dentada - ou, nas versões posteriores, com cilindros dentro da máquina.

Esse mecanismo funcionava (máquina 1 de Leibniz), com o componente móvel, para acelerar as adições repetitivas envolvidas nas operações de multiplicação e divisão. A própria repetição tornava-se automatizada. A primeira máquina que efetuava facilmente soma, subtração, multiplicação e divisão foi inventada por **Leibniz** em 1673. O projeto *Pascaline* de Pascal foi bastante aprimorado por **Leibniz**, originando a primeira máquina de **Leibniz**.

A Figura 14 mostra a segunda máquina calculadora de Leibniz. Uma terceira calculadora foi criada por Leibniz. Ver na Figura 15.

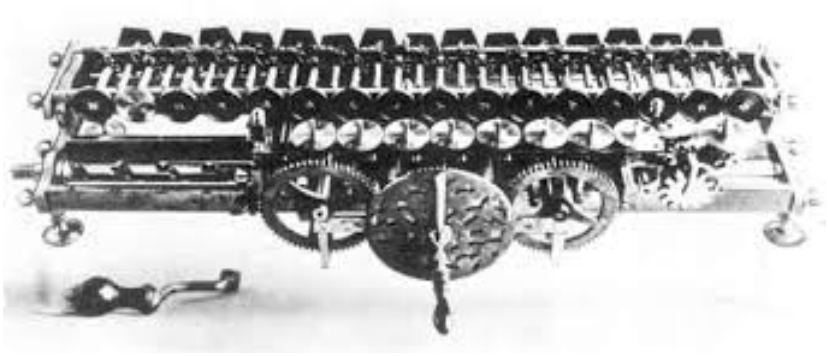


Figura 13 – A primeira máquina de calcular de Leibniz.

Fonte: prof-edigleyalexandre.com.

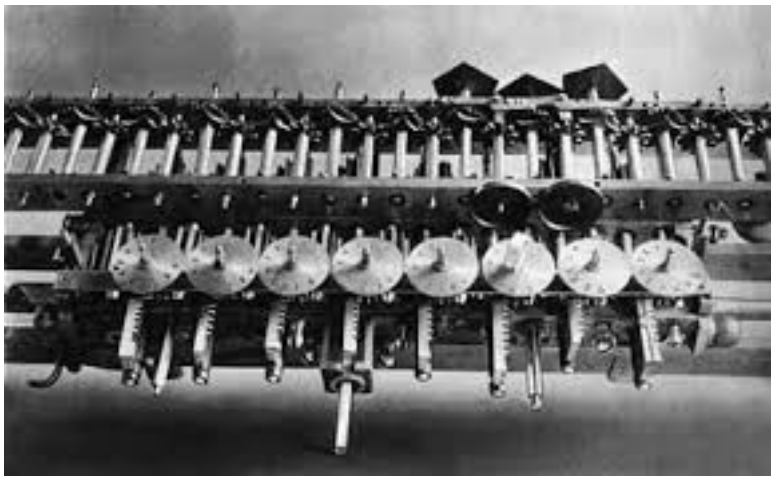


Figura 14 – A segunda máquina de calcular de Leibniz.

Fonte: guia.heu.mom.br.

Ao contrário de **Pascal**, **Leibniz** tinha habilidades teóricas e práticas, **Leibniz** era um teórico, sem colaboradores práticos, e não conseguiu produzir máquinas com versões operacionais confiáveis de seu mecanismo. Contudo, seu conceito principal, conhecido como roda de **Leibniz**, influenciaria os projetos até a época de **Charles Babbage**.

2.7 Joseph Jacquard em 1801 e a Ideia dos Cartões Perfurados

Joseph-Marie Jacquard (1752-1834), um mecânico francês, foi o inventor do *tear mecânico*. Curiosamente, ele era de um ramo que não tinha nada a ver com números e calculadoras. Mas, a tecelagem. Filho de tecelões - e, ele mesmo, um aprendiz têxtil desde os dez anos de idade, **Jacquard** sentiu-se incomodado com a monótona tarefa que lhe fora confiada na adolescência: alimentar os teares com novelos de linhas coloridas para formar os desenhos no pano que estava sendo fiado. Como toda a operação era manual, a tarefa de **Jacquard** era interminável: a cada segundo, ele tinha que mudar o novelo, seguindo as determinações do contratante. Com o tempo,

Figura 15 – A terceira máquina de calcular de Leibniz.

Fonte: guia.heu.mom.br.

Jacquard foi percebendo que as mudanças eram sempre sequenciais. Jacquard foi o inventor do tear mecânico. Ele inventou um processo simples: através de **cartões perfurados**, onde o contratante poderia registrar, ponto a ponto, a receita para a confecção de um tecido. Esse tear era programado por uma **série de cartões perfurados**, cada um deles controlando um único movimento da lançadeira e capaz de ler os cartões e executar as operações na sequencia programada. A primeira demonstração prática do sistema aconteceu na virada do século XIX, em 1801. Os mesmos cartões perfurados de Jacquard, que mudaram a rotina da industria têxtil. Em 1804, Jacquard construiu um tear inteiramente automatizado, que podia fazer desenhos muito mais complicados. Poucos anos depois, uma decisiva influência no ramo da computação. E, praticamente sem alterações, continuam a ser aplicados ainda até hoje.

2.8 Charles Thomas em 1820 - A Primeira Calculadora Comercial

Charles Xavier Thomas, conhecido como **Thomas de Colmar**, (1785-1870) foi um matemático e inventor francês, mais conhecido por projetar e patentear uma das primeiras calculadoras, o *Arithmomètre*, em 1820, a primeira calculadora que podia somar, subtrair e multiplicar, permitindo também dividir com alguma ajuda do operador. Ele a projetou, basicamente, usando o desenho de **Leibniz**. **Thomas** patenteou na França em 1820 e manufaturou de 1851 a 1915. Era uma máquina, ocupando todo o tampo de uma escrivaninha, media 70 cm de comprimento por 18 cm de largura e 10 cm de altura. Foi um passo importante para o avanço das calculadoras e primeiros computadores. A busca por uma solução: 1820-1851. A criação pela indústria: 1851-1887. O auge de mercado desta calculadora: 1887-1915. A calculadora de Thomas evoluiu até 1915, como está na Figura 20.

Figura 16 – Jacquard - A primeira ideia dos cartões perfurados.

Fonte: pt.wikipedia.org/wiki/Joseph_Marie_Jacquard.

Figura 17 – Jacquard - O tear de Jacquard no Musée des Arts et Métiers.

Fonte: pt.wikipedia.org/wiki/Joseph_Marie_Jacquard.

Figura 18 – Charles Thomas - Máquina comercial capaz de efetuar as quatro operações aritméticas básicas: a *Arithmometer*.

Fonte: https://en.wikipedia.org/wiki/Arithmometer.

Figura 19 – Arithmometer - A calculadora mecânica de Charles Thomas.

Fonte: https://en.wikipedia.org/wiki/Arithmometer.

Usando princípios de calculadoras mecânicas anteriores, o *Arithmomètre*, Figura 20, era mais confiável e foi produzida por 90 anos pelo próprio inventor e seus descendentes. Cerca de 5000 exemplares foram fabricados, a maioria entre 1851 e 1914. O *Arithmomètre* foi a primeira máquina de calcular que obteve sucesso comercial. Tinha reputação de confiabilidade e robustez e é considerada a primeira máquina de escritório, precursora das máquinas mecânicas e eletrônicas do século seguinte.

Figura 20 – Arithmometer em 1848 - máquina comercial para efetuar as quatro operações aritméticas básicas.

Fonte: http://history-computer.com/MechanicalCalculators/19thCentury/Colmar.html.

2.9 Charles Babbage e Ada Lovelace - De 1822 a 1843

Dentre todos os pensadores e inventores que acrescentaram algo ao desenvolvimento da computação, o único que quase chegou a criar, efetivamente, um computador no sentido da palavra, foi um inglês chamado **Charles Babbage** (1791 - 1871), um matemático inglês e inventor que na University of Cambridge, lecionou matemática (1828-1839).

Em 1822, Babbage descreveu, num artigo científico, uma máquina que poderia computar e imprimir extensas tabelas científicas. **Babbage** inventou a *máquina diferencial*, como na Figura 22, pois estava preocupado com os erros contidos nas tabelas matemáticas de sua época, assim, construiu um modelo para calcular tabelas de funções (logaritmos, funções trigonométricas, e outras) sem a intervenção de um operador humano. Ao operador caberia somente iniciar a cadeia de operações e, a seguir, a máquina fazia os cálculos, bastante repetitivos, terminando totalmente a

Figura 21 – Charles Babbage: o inventor da máquina analítica - um computador mecânico.

Fonte: Os Inovadores, p.18.

tabela prevista. Em 1832 - **Babbage** e **Joseph Clement** produzem uma porção da *Máquina de Diferenças*.

A ideia de **Jacquard**, os **cartões perfurados**, onde o contratante poderia registrar, ponto a ponto, a receita para a confecção de um tecido, foi desenvovida por **Charlie Babage**, através de sua segunda máquina: a **máquina analítica**, como na Figura 22. Em torno de 1833, **Babbage** resolveu deixar de lado seus planos da Máquina de Diferencial. O insucesso, porém, não o impediu de desenvolver idéias para construir uma máquina ainda mais ambiciosa, e entre 1834-1835, **Babbage** troca o enfoque de seus trabalhos para projetar a Máquina Analítica.

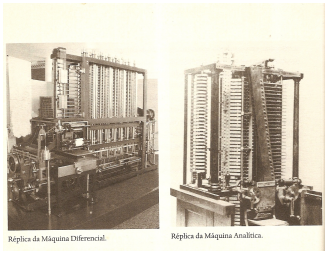


Figura 22 – Babage - Uma réplica de parte do calculador diferencial e da analítica.

Fonte: Os Inovadores, p.34.

A **Máquina Analítica**, ao contrário de sua predecessora, foi concebida não apenas para solucionar um tipo de problema matemático, mas para executar uma ampla gama de tarefas de cálculo, de acordo com instruções fornecidas por seu operador. Seria uma máquina de natureza a mais geral possível - com as ideias visando o primeiro computador programável para todos os fins. Seguindo **Jacquard**, as instruções seriam introduzidas por meio de cartões perfurados.

Babbage foi ajudado por **Augusta Ada Byron**, condessa de Lovelace, sua contemporânea, que dizia:

Podemos dizer mais convenientemente que a Máquina Analítica tecia padrões algébricos, assim como o tear de Jacquard tecia flores e folhas.

escreveu **Ada Lovelace**, uma das poucas pessoas que compreenderam o funcionamento da máquina e vislumbrou seu imenso potencial de aplicação. Esta máquina, já continha muitas ideias básicas usadas em qualquer computador automático moderno: podia acumular, fazer cálculos, podia ser controlada, mas nunca foi acabada. Mas dado que as operações tinham de ser totalmente mecânicas, somente com o advento da eletrônica atual, as ideias de **Babbage** poderiam tornar-se práticas.

Babbage é considerado com um dos pioneiros das máquinas de computação. Seu invento, porém, exigia técnicas bastante avançadas e caras na época, e nunca foi construído. A *máquina analítica*, deu um passo importante na história dos computadores, como o projeto de um computador mecânico de uso geral. Foi descrito pela primeira vez em 1834. **Babbage**, em torno de 1840 planejou, mas foi incapaz para construir uma máquina programável não-binária, mas decimal. **Babbage** continuou a trabalhar no projeto até sua morte, em 1871. Por causa de questões técnicas, financeiras, políticas e legais, a máquina nunca foi realmente construída. Computadores de uso geral, logicamente comparáveis ao engenho analítico, só iriam surgir, de forma totalmente independente da pesquisa de **Charles Babbage**, cerca de 100 anos mais tarde.

Alguns acreditam que as limitações tecnológicas da época constituíam-se num obstáculo adicional para a construção da máquina. Em todo caso, a máquina seria enorme e extremamente cara. Mais recentemente, entre 1985 e 1991, o Museu de Ciência de Londres construiu outra de suas invenções inacabadas, a máquina diferencial, usando apenas técnicas disponíveis na época de Babbage. **Charles Babbage** era matemático, mas na sua era tentou projetar um máquina de computar mecânica.

Ironicamente, a **Máquina de Diferenças** teve um destino um pouco melhor. Embora o próprio **Babbage** nunca mais voltasse a ela, um inventor e tradutor sueco chamado **Pehr Georg Scheutz** leu a respeito do dispositivo e construiu uma versão modificada, em 1854.

Augusta Ada King (Ada Lovelace) (1815-1852) foi uma matemática inglesa e hoje é principalmente reconhecida por ter escrito o primeiro procedimento para ser processado por uma máquina, a máquina analítica de **Charles Babbage**. Além de publicar uma coleção de notas sobre a máquina analítica, durante o período que esteve envolvida com o projeto de **Babbage**, ela desenvolveu os procedimentos que permitiriam a máquina computar os valores de funções matemáticas.

Figura 23 – A precursora de programação da máquina analítica

Fonte: Os Inovadores, p.18.

Na juventude seus talentos matemáticos levaram-a a uma relação de trabalho e de amizade com o colega matemático britânico **Charles Babbage** e, em particular, o trabalho de **Babbage** sobre a máquina Analítica .

Em 1842, **Charles Babbage** foi convidado a ministrar um seminário na Universidade de Turim sobre sua máquina analítica. A palestra de **Babbage** foi publicada num artigo em francês pelo engenheiro militar italiano **Luigi Federico Menabrea** e esta transcrição foi posteriormente publicada na Bibliothèque Universelle de Genève, em 1842. **Babbage** pediu a **Ada** para traduzir o artigo para o inglês, adicionando depois a tradução com as anotações que ela mesma havia feito. Ada levou grande parte do ano nesta tarefa. E entre 1842 e 1843, ela traduziu esse artigo sobre a máquina analítica, e complementou com um conjunto de sua própria autoria, que ela chamou de *Anotações*. Essas notas, que são mais extensas que o artigo original, foram então publicados no *The Ladies' Diary* e no *Memorial Científico de Taylor*.

Essas notas, contém um procedimento criado para ser processado por máquinas, o que muitos consideram ser o primeiro "programa" de computador. Ela também desenvolveu uma visão sobre a capacidade dos computadores de irem além do mero cálculo ou processamento de números, enquanto outros, incluindo o próprio **Babbage**, focavam apenas nessas capacidades. Sua mentalidade a levou a fazer perguntas sobre a Máquina Analítica (como mostrado em suas notas) e a examinar como os indivíduos e a sociedade se relacionavam com a tecnologia como uma ferramenta de colaboração.

As notas de **Ada** foram classificadas alfabeticamente de A a G. Na nota G ela descreve um procedimento para a máquina analítica computar a *Sequência de Bernoulli*. Este procedimento é, hoje, considerado *o primeiro algoritmo especificamente criado para ser implementado num computador*, e **Ada** é, por esta razão, recorrentemente citada como a primeira pessoa programadora da história da computação. No entanto, a máquina não foi construída durante o tempo de vida da Condessa de Lovelace.

2.10 A Odhner Arithmometer em 1873

O *Odhner Arithmometer* era uma calculadora muito bem sucedido inventado na Rússia em 1873 por W. T. Odhner, um imigrante sueco. De 1892 a meados do século XX, as empresas independentes foram criadas em todo o mundo para fabricar clones de *Odhner* e, na década de 1960, milhões delas já haviam sido vendidas, tornando-se um dos tipos mais bem sucedidos de calculadora mecânica já projetado.
Odhner pensou em sua máquina em 1871 durante a reparação de uma "Thomas's Arithmometer" (que foi a única calculadora mecânica em produção na época) e decidiu substituir a pesada e volumoso cilindro de **Leibniz**, por um disco mais leve menor. É por isso que as duas máquinas compartilham o mesmo nome. **Odhner**

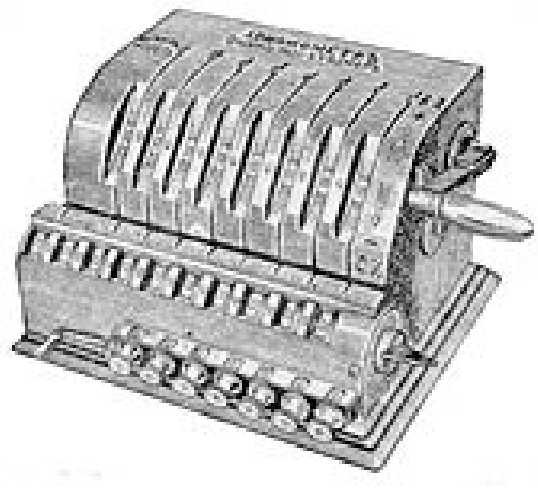


Figura 24 – O projeto original da Odhner-Arithmometer.

Fonte: www.en.wikipedia.org.

desenvolveu a primeira versão de sua calculadora mecânica em 1873, na oficina de **Odhner** em São Petersburgo. Em 1876, ele concordou em construir 14 máquinas para **Ludvig Nobel**, o seu empregador, no que ele as entregou em 1877. Ele patenteou sua máquina original em vários países entre 1878-1879 e uma versão melhorada dela em 1890. A produção em série começou com esta máquina melhorada em 1890.

Em 1891, Odhner abriu uma filial de sua fábrica na Alemanha. Infelizmente, ele teve que vendê-lo em 1892 para *Grimme, Natalis & Co.* por causa da dificuldade de ter duas instalações de fabricação distantes. *Grimme, Natalis & Co.* iniciou a produção em Braunschweig e vendeu as suas máquinas com a marca *Brunsviga* (Brunsviga é o nome latino da cidade de Braunschweig); eles se tornaram muito bem sucedidos por conta própria.

Após a morte de Odhner, em 1905, seus filhos continuaram a produção e cerca de 23.000 calculadoras foram feitas até que a fábrica foi nacionalizada durante a Revolução Russa de 1917, e foi forçado a encerrar em 1918. Isso fez com que a *Brunsviga arithmometer*, com seu início 1892, fosse o tipo de calculadora mais duradouro de Odhner na produção. Mesmo sendo a máquina muito popular, a produção só durou 30 anos.

No final de 1917, a família **Odhner** voltou para a Suécia e reiniciou a produção de sua calculadora sob o nome original **Odhner**. Em 1924, o governo russo mudou a unidade de produção para Moscou e comercializou a calculadora sob o nome de *Felix Arithmometer* que fou fabricada até a década de 1970.

Em 1950, com milhões de clones fabricados, a *Arithmometer* de *Odhner* foi uma das mais populares tipos de calculadora mecânica já feito. O número de máquinas produzidas aumentou constantemente até que a aparição das calculadoras eletrônicas

no início de 1970. Por exemplo, a produção de um desses clones, o *Arithmometer Felix* da Rússia, atingiu em 1969 em torno de 300 mil máquinas fabricadas.

A *Arithmometer de Odhner* foi copiada, fabricada e vendida por muitas outras empresas em todo o mundo. Na Alemanha existiu a *Thales*, a *Triumphator*, a *Walther* e a *Brunsviga*. Na Inglaterra havia *Britannic* e *Muldivo*. Na Suécia, a *Multo* e a *Original Odhner*. Na Rússia, a Arithmometer Felix e no Japão, *Tiger* e *Busicom*, que ficou famosa porque a Intel criou o primeiro microprocessador, o Intel 4004, ao projetar em uma de suas calculadoras eletrônicas em 1970.

2.11 A Baldwin Calculadora

Em 1875, **Frank S. Baldwin** de St. Louis, USA, patenteou uma máquina de calcular. Ele fabricou algumas dessas máquinas, mas não teve tanto sucesso em suas vendas. **Baldwin** passou a trabalhar com uma série de outras patentes.

Figura 25 – Frank S. Baldwin - Sem sucesso nas vendas de sua máquina de calcular.

Fonte: www.en.wikipedia.org.

Em 1900, ele patenteou a *Baldwin Computing Engine*, uma máquina pela qual multiplicação ou divisão foi realizada por um toque para cada dígito. Em 1901, ele se mudou para Newark, New Jersey, onde ele projetou uma máquina melhor. Ele obteve uma patente no ano seguinte, **Baldwin** passou a inventar outras máquinas de calcular, principalmente os fabricados pela *Monroe Calculating Machine Company*. A calculadora de **Baldwin** foi comercializada entre 1903-1907. Em 1908, ele foi premiado com uma patente sobre o *Baldwin Record Calculator*, que combinava uma impressora com a calculadora. Em 1911, uma parceria com **Jay R. Monroe**, da *Western Electric Company* em Nova York para criar a *Monroe Calculadora Company*.

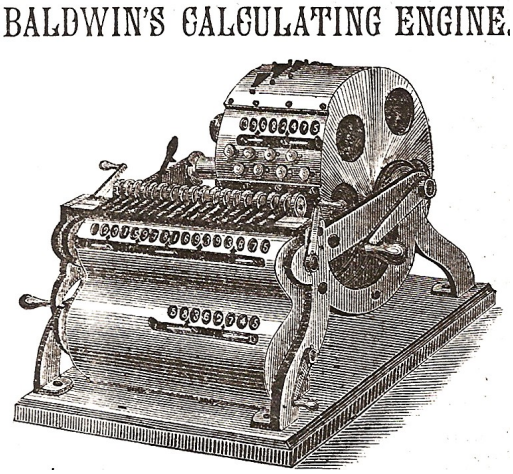


Figura 26 – Frank S. Baldwin - A sua máquina de calcular de 1875.

Fonte: retroplayerbrazil.wordpress.com.

Figura 27 – William Burroughs - Apresentou a primeira máquina de calcular com teclado.

Fonte: en.wikipedia.org.

2.12 A Calculadora Burroughs em 1890

Em 1886, a **Burroughs Corporation** foi fundada em 1886 sob o nome "**American Arithmometer Company**"em Saint Louis, Missouri, USA. **William S. Burroughs** (1857-1898) patenteou uma máquina de calcular em 1888. E em 1890 apresentou a sua primeira máquina com teclado.

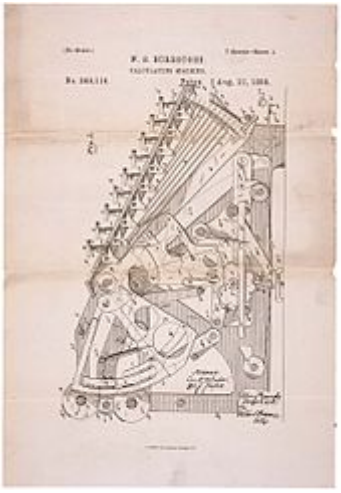


Figura 28 – Patente no. 388,116 da "Burroughs calculating machine" em 1888.

Fonte: https://en.wikipedia.org/wiki/Burroughs_Corporation.

A empresa mudou-se para Detroit in 1904 e mudou seu nome para **Burroughs Adding Machine Company**. A *Burroughs Adding Machine Company* evoluiu para produzir máquinas eletrônicas de faturamento e mainframes e, posteriormente, fundiu-se com a *Sperry* para formar a *Unisys*.

2.13 Fim do Século XIX - Hollerith e sua Máquina de Perfurar Cartões

Também muito importante para a história dos computadores, entre 1884 e 1890, o americano **Herman Hollerith** (1860-1929), um funcionário do *United States Census Bureau* - Escritório de Recenseamento dos E.U.A - um estatístico americano, que trabalhou no recenseamento americano de 1890, foi o responsável por uma grande mudança na maneira de se processar os dados do censo da época. Ele inventou uma máquina capaz de processar dados, baseada na separação de cartões perfurados de oito colunas, conforme o código BCD (Binary Coded Decimal). Cada cartão era usado um para cada pessoa. Pelos seus furos, cada posição dos furos representava uma condição (profissão, idade, escolaridade, entre outros dados de interesse do censo).

A máquina de **Hollerith** foi utilizada no censo de 1890, reduzindo o tempo de processamento de dados. Os dados do censo de 1880 (o anterior), manualmente

Figura 29 – Burroughs - A primeira calculadora com teclado.

Fonte: https://en.wikipedia.org/wiki/Burroughs_Corporation.

Figura 30 – Burroughs - O modelo desktop de 1910.

Fonte: https://en.wikipedia.org/wiki/Burroughs_Corporation.

Figura 31 – Hollerith - A mudança na maneira de se processar os dados.

Fonte: pt.wikipedia.org.

processados, levaram 7 anos e meio para serem compilados. Os do censo de 1890 foram processados em 2 anos e meio, com a ajuda de uma máquina de perfurar cartões e máquinas de tabular e ordenar, criadas por Hollerith e sua equipe. Ela foi também pioneira ao utilizar a eletricidade na separação, contagem e tabulação dos cartões.

As informações sobre os indivíduos eram armazenadas por meio de perfurações em locais específicos do cartão. Nas máquinas de tabular, um pino passava pelo furo e chegava a uma jarra de mercúrio, fechando um circuito elétrico e causando um incremento de 1 em um contador mecânico.

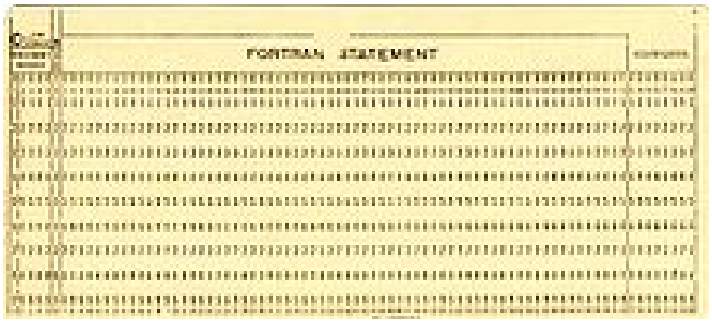


Figura 32 – O cartão perfurado aperfeiçoado por Hollerith.

Fonte: wikiwand.com.

Assim, a primeira aplicação prática da programação havia surgido. E os cartões perfurados passaram a ser o meio de incluir comandos e dados nas máquinas.

Hollerith baseou-se na idéia de **Charles Babbage**, e aperfeiçoou os cartões perfurados (os utilizados por Jacquard)e inventou máquinas para manipulá-los (ou seja,

tanto a máquina para perfurar cartões, como máquinas de tabular e ordenar).

Mais tarde, **Hollerith** fundou uma companhia para produzir máquinas de tabulação. Anos depois, em 1924, essa companhia veio a se chamar *International Business Machines* - IBM, como é hoje conhecida. A partir dái, a IBM foi pioneira em muitas invenções que influiram nas atividades da humanidade.

Importante é notar que durante a década de 50 até meados dos anos 70, os cartões perfurados foram utilizados como o principal modo de entrada e armazenamento de dados. E, dependente da aplicação, são usados até os dias de hoje.

2.14 O Advento das Máquinas Programáveis

Como visto, a invenção da mais moderna máquina de calcular mecânica data do final do século XIX. Logo surgiram diversos fabricantes, como a alemã *Brunsviga* e as americanas *Burroughs*, e posteriormente, a *American Adding Machine*.

Todas essas máquinas, porém, estavam longe de ser um computador de uso geral, pois não eram programáveis. Isto quer dizer que a entrada era feita apenas de números, mas não de instruções, a respeito do que fazer com os números de entrada na máquina.

Como projetar uma máquina suficientemente flexível para realizar qualquer tipo de cálculo? A resposta só foi obtida no século XX, originada na engenhosidade mecânica mostrada neste capítulo e como resultado da criatividade teórica dos tempos de **Leibniz**, **Boole**, **Cantor**, **Frege**, **Hilbert**, **Gödel**, **Alonzo Church**, **Alan Turing**, **Shannon**, **John von Neumann** e tantos outros. A ideia era criar uma máquina capaz de receber informações precisas sobre a tarefa a ser desempenhada. Somente com o esforço de guerra empreendido durante a Segunda Guerra Mundial que a ideia de computadores programáveis se tornou realidade. **Konrad Zuse** (1910-1995) foi um engenheiro alemão e um pioneiro dos computadores. O seu maior feito foi o projeto do primeiro computador de programa controlado por fita perfurada, o Z3, com **Helmut Schreyer** em 1941, que se considera ser o primeiro computador programável, o primeiro computador *eletromecânico*, constituído de relés (utilizava a tecnologia das centrais telefônicas alemãs da época), que efetuava cálculos e exibia os resultados em fita perfurada. Depois deste, surgiram outros projetos de computadores importantes, na Inglaterra e nos Estados Unidos (ver no capitulo 9), e a partir daí, a grande inovação foi passar das máquinas eletromecânicas para as máquinas programáveis com a utilização da eletrônica nos computadores. Assim, ao invés de usarem engrenagens e discos mecânicos para codificar e processar informações, usavam *válvulas* a vácuo.

Em 1947, com a invenção do *transistor* pelos engenheiros **William Shockley**,

John Bardeen e **Walter Brattain**, todos do Bell Labs, foi dado o passo fundamental para o lançamento em 1962 da *Anita MkVIII*, a primeira calculadora eletrônica de mesa. A primeira calculadora eletrônica portátil está comemorando 53 anos (no tempo deste livro). Seu criador foi o engenheiro **Jack St. Clair Kilby** (1923-2005), ganhador do Nobel de Física de 2000 pela invenção do circuito integrado em 1958, ao lado de **Bob Noyce** (1927-1990), um dos fundadores da Intel em 1968.

2.15 Bibliografia e Fonte de Consulta

Trabalho citado por Bolter, que descreve o dispositivo Antikythera (Anticythère), na Scientific American, junho de 1959, pgs. 60-67.

John Napier - ⟨http://www.johnnapier.com/⟩

John Napier - ⟨http://www-history.mcs.st-and.ac.uk/Biographies/Napier.html⟩

Régua de cálculo - ⟨https://pt.wikipedia.org/wiki/Regua_de_calculo⟩

Calculadora de Schickard - ⟨https://en.wikipedia.org/wiki/Wilhelm_Schickard⟩

Calculadora de Pascal - ⟨http://www.di.ufpb.br/raimundo/Revolucao_dos_Compu\penalty\@M\hskip\z@skip\discretionary{-}{}{}\penalty\@M\hskip\z@skiptadores/Histpage2.htm⟩

Calculadora de Leibniz - ⟨http://www.di.ufpb.br/raimundo/Revolucao_dos_Compu\penalty\@M\hskip\z@skip\discretionary{-}{}{}\penalty\@M\hskip\z@skiptadores/Histpage17.htm⟩

Joseph Jacquard - ⟨http://www.di.ufpb.br/raimundo/Revolucao_dos_Computadores/Histpage3.htm⟩

Charles X. Thomas - ⟨https://pt.wikipedia.org/wiki/Charles_Xavier_Thomas,http://www.arithmometre.org/⟩

Arithmometer - ⟨http://history-computer.com/MechanicalCalculators/19thCentury/Colmar.html⟩

Charlie Babbage e Ada Lovelace - ⟨http://www.di.ufpb.br/raimundo/Revolucao_dos_Computadores/Histpage4.htm⟩

Ada Lovelace - *Os Inovadores uma biografia da revolução digital*, Walter Isaacson. Companhia da Letras, 2014.

T. W. Odhner - ⟨https://en.wikipedia.org/wiki/Odhner_Arithmometer⟩

William S. Burroughs - ⟨https://en.wikipedia.org/wiki/William_S._Burroughs⟩

Hollerith - ⟨https://pt.wikipedia.org/wiki/Herman_Hollerith⟩

Primeira calculadora eletrônica - http://gilbertomelo.com.br/primeira-calculadora-eletronica completa-40-anos/

Ernesto F. Galvão (2007) - O que é Computação Quântica, Vieira & lent Casa Editorial. Rio de Janeiro.

2.16 Referências - Leitura Recomendada

Máquina Antikythera (Anticythère) - https://pt.wikipedia.org/wiki/Maquina_de_Anticitera

George C. Chase - History of Mechanical Computing Machinery. Volume 2, Number 3 (en inglés). IEEE Annals of the History of Computing, (July de 1980)

Ifrah, Georges (2001). The Universal History of Computing (en inglés). John Wiley & Sons, Inc. ISBN 0-471-39671-0.

Marguin, Jean (1994). Histoire des instruments et machines à calculer, trois siècles de mécanique pensante 1642-1942 (en frances). Hermann. ISBN 978-2-7056-6166-3.

Frank S. Baldwin, Calculating-Machine, U.S.A Patent 706375, August 5, 1902.

Frank S. Baldwin Inventor, Dies at 86. Originator of the Calculating Machine, the Anemometer and Many Other Devices. The New York Times. April 9, 1925.

Advertisement, The Spectator, vol. 76 #3 (Feb 1, 1904), p. 66.

Stephen Johnson, Making the Arithmometer Count, Bulletin of the Scientific Instrument Society, No. 52, 1997, 12-21.

Rapport fait par M. Francoueur, Bulletin de la Société d'Encouragement pour l'Industrie Nationale, 21, 1822, pp. 33-36.

Charles-Xavier Thomas, 1420, 18 Novembre 1820. This is the number of Thomas's 1820 French patent.

Hoyau, Description d'une machine à calculer nommée Arithmomètre. Bulletin de la Société d'Encouragement pour l'Industrie Nationale, 21, 1822, pp. 355-365.

Resumo histórico - ⟨https://www.inf.pucrs.br/~gustavo/disciplinas/iec/material/apres_hist_portugues.pdf⟩

Chase G.C.: History of Mechanical Computing Machinery, Vol. 2, Number 3, July 1980, page 204, IEEE Annals of the History of Computing.

The Origins of Digital Computers: Selected Papers, Texts and Monographs in Computer Science, 1975. Spring-Verlag.

CAPÍTULO 3

Alonzo Church - As Funções Computáveis sem Computação Concreto

Alonzo Church (1903-1995), na University of Princeton e depois na University of Califórnia, foi orientador de vários pesquisadores de renome, entre outros, **Martin Davis** , **Stephen Kleene** , **Michael Rabin** , **Dana Scott** e **Alan Turing** . **Church** foi um matemático e lógico estadunidense. Atuou principalmente nas áreas de lógica matemática, teoria da recursão e teoria da computação. Entre suas melhores contribuições, está o λ-Cálculo, um sistema matemático formal que investiga funções e aplicação de funções tidas, posteriormente por Turing, como funções computáveis. Funções computáveis são os objetos básicos de estudo na teoria da computabilidade. Desde **Herbrand-Gödel** , que funções recursivas já eram usadas para discutir a computabilidade, no sentido da avaliação da função, mesmo sem se referir a nenhum modelo de computação concreto, como, por exemplo, a máquina de Turing. **Church** também contribuiu para uma tese, que veio a ficar conhecida como **Tese de Church-Turing**.

As primeiras publicações de **Church** sobre o λ-Cálculo foram igualmente voltadas para os problemas fundamentais em matemática: o objetivo declarado de **Church** era desenvolver uma nova axiomatização de lógica que evitasse os paradoxos, mas de uma forma diferente da teoria de tipos de **Russell**, ou a teoria dos conjuntos axiomática. Embora nós podemos pensar, agora, sobre o λ-Cálculo (simples) como um formalismo para expressar funções computáveis, **Church**, originalmente, não o concebeu dessa forma. Para ele, o sistema que evoluiu para o λ-Cálculo foi um formalismo lógico que, ele esperava, ser capaz de uma formalização da matemática, livre de contradição. Infelizmente, o sistema original de **Church** provou ser inconsistente (Kleene e Rosser, 1935). A prova de **Kleene** e **Rosser** , de que era inconsistente fez uso essencial do método de codificação usado por **Gödel**, introduzido em (Gödel, 1931). O desenvolvimento de **Kleene** da aritmética (1935) e da representatividade

Figura 33 – Alonzo Church na University of Princeton: o orientador de Turing.

Fonte: quotationof.com.

das funções recursivas dentro do λ-Cálculo foi motivada, em parte, pelo objetivo de reproduzir o resultado da incompletude de **Gödel** no contexto do λ-Cálculo, e seu importante teorema da forma normal também contou com a codificação de **Gödel**. Foi neste contexto, em direção às investigações metamatemáticas do λ-Cálculo, ao longo do que **Gödel** pensava (1931), que a noção de λ-*definibilidade* alcançou um lugar de destaque no trabalho de **Church**, **Kleene** e **Rosser**.

Os resultados positivos obtidos por **Kleene**, no sentido de que um grande número de funções recursivas poderia ser formalizado no λ-Cálculo, levou **Church** a formular o que agora veio a ser conhecido como **Tese de Church**, isto é, que toda função efetivamente computável é λ-definível. Novamente foi **Gödel**, no tempo que estava em Princeton (1934), quem conduziu **Church** e seus alunos a ter uma visão mais ampla. Seu ceticismo sobre a **Tese de Church** quando formulada pela primeira vez, a respeito de λ-*definibilidade*, e sua proposta de que recursividade geral poderia ser uma melhor candidata para uma caracterização precisa da computabilidade efetiva, levou **Kleene** a mostrar que as duas noções são co-extensivos: toda função λ-*definível* é geral recursiva e, contrariamente (Kleene, 1936b).

3.1 Sistema de Church e a Incompletude de Gödel

Como em Zach (2006), nos anos 1929-1931, **Church** desenvolveu uma formulação alternativa de lógica (Church, 1932, 1933), que ele esperava servir como uma nova fundamentação da matemática, o que evitaria os paradoxos. **Church** ministrou um curso sobre lógica no outono de 1931, onde **Kleene**, então um estudante de pós-graduação, tomou nota. Durante esse tempo, **Church** e **Kleene** foram introduzidos primeiramente ao trabalho de **Gödel** sobre a incompletude. A ocasião foi uma palestra de **John von Neumann** sobre o trabalho de **Gödel**. **Church** e **Kleene**

imediatamente estudaram o artigo em detalhe. Na época, ainda não era claro, como, em geral, os resultados de **Gödel** eram. **Church** acreditava que a incompletude do sistema P de **Gödel** (a formulação "type-theoretical" de alta ordem da aritmética de **Peano**) baseiava-se essencialmente em alguma característica da teoria dos tipos, e que o resultado de **Gödel** não seria aplicável ao sistema próprio de **Church**. No entanto, parece que o problema se tornou uma questão premente para **Church**, para determinar em que medida os resultados e métodos de **Gödel** poderiam ser levados a cabo no seu sistema. Ele colocou **Kleene** para trabalhar na tarefa de obter a aritmética de **Peano** no sistema. **Kleene** conseguiu realizar isso no primeiro semestre de 1932. Tratava-se, em particular, mostrar que várias "number-theoretic functions" eram λ-definível. Em julho de 1932, **Gödel** escreveu a **Church**, perguntando se o sistema de **Church** poderia ser provado consistente em relação ao *Principia Mathematica* de **Russell** e **Whitehead**. **Church** era cético quanto à utilidade de tal prova relativa à consistência. Ele respondeu:

Na verdade, a única evidência para ser livre de contradição, relativo à Principia Mathematica é a evidência empírica decorrente do fato de que o sistema tem sido usado por algum tempo, muitas de suas consequências têm sido vistas, e ninguém encontrou uma contradição. Se meu sistema for realmente livre de contradição, então, uma quantidade igual de trabalho em derivar as suas consequências devem fornecer um peso igual de evidência empírica para a sua liberdade de contradição. [. . .] Mas, continua a ser escassamente possível que uma prova da liberdade de contradição para o meu sistema possa ser encontrada, um tanto na linha sugerida por ***Hilbert****. Eu, na verdade, tenho feito várias tentativas frustradas de fazer isso. Dr. von Neumann chamou minha atenção para o seu artigo do último outono, intitulado "Uber formal unentscheidbare sätze der Principia Mathematica". Eu tenho sido incapaz, contudo, que suas conclusões no 4 deste artigo, aplica-se ao meu sistema. Possivelmente, seu argumento possa ser modificado, de modo a torná-lo aplicável ao meu sistema, mas eu não tenho sido capaz para encontrar uma tal modificação de seu argumento.* (Church para Gödel, July 27, 1932) Godel (2003).

A seção 4 de Godel (1931), que menciona **Church**, é a seção em que **Gödel** esboçou o segundo teorema da incompletude. Desde que **Gödel** não forneceu uma prova completa do teorema, de fato, a primeira prova completa não apareceu até **Hilbert** e **Bernays** (1939) Hilbert e Bernays (1939). **Church** foi certamente justificado, sem dúvida, que o resultado era aplicável ao seu sistema. Deixou-se em aberto a questão, se **Church** acreditou, naquele momento, que a construção do primeiro teorema da incompletude passava por seu sistema.

Kleene e **Rosser** (1935) mostraram que o sistema de **Church** era inconsistente. O fragmento do sistema de **Church** com os axiomas lógicos removidos é comprovadamente consistente. Esse fragmento é o λ-Cálculo simples (não tipado), que pode ser visto em Barendregt (1997), para o impacto do λ-Cálculo em Ciência da Computação, e em Seldin (2006) para uma história do λ-Cálculo.

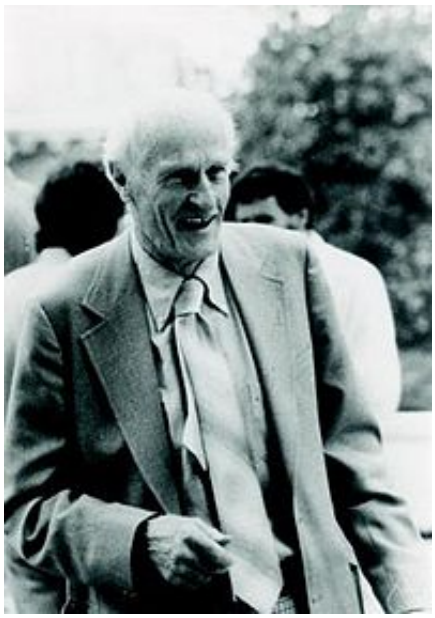


Figura 34 – Stephen Kleene - A classe das funções computáveis recursivas.

Fonte: en.wikipedia.org.

Church, então, acabou por estar correto. Conforme teorema da incompletude de **Gödel** , este não se aplica ao seu sistema, porque o teorema somente é aplicável em sistemas formais consistentes. No intuito de se obter este resultado, e muitos dos resultados positivos devido à **Kleene**, se proporcionaram os fundamentos para a indecidibilidade dos resultados de **Church**, e um ano depois, os métodos de **Gödel** foram de importância crucial, porque motivou uma determinada linha de investigação e porque **Kleene**, **Rosser** e **Church** foram capazes de construir sobre eles.

Os métodos introduzidos em Gödel (1931) e utilizados por **Kleene** e **Rosser** para mostrar que o sistema de Church era inconsistente, e também prevaleceram na solução negativa do problema de decisão de **Church**. **Church** (1935, 1936b) (ver referências no final do capítulo), primeiro demostrou que dada expressão do λ-Cálculo com forma normal, esta é não recursiva.

Neste mesmo artigo, **Church** afirmou o que é hoje conhecido como **Tese de Church**, por exemplo, que as funções recursivas gerais (e, portanto, as λ-definíveis) também são exatamente as que são efetivamente "computáveis". O teorema e a tese combinam-se para produzir o resultado, de que ter uma forma normal não é uma propriedade efetivamente decidable. A gênese de *Tese de Church* serão apresentados no a próxima seção. Aqui, é salientado apenas que o resultado em si, e com ela a solução negativa do problema de decisão para a lógica de primeira ordem (Church, 1936a) fez uso essencial do trabalho de **Gödel**. O próprio **Kleene** (1987) enfatiza a importância de **Gödel** (1931) no trabalho que ele e **Rosser** realizaram em suas contribuições seminais para a teoria de recursão no início de 1930.

Gödel (1931) teve uma influência duradoura sobre os pioneiros da recursão teoria e o desenvolvimento do λ-Cálculo. **Gödel** teve uma importância pessoal direta na formação da Tese de **Church**. Ele visitou Princeton em no ano letivo de 1933/34 e deu uma série de palestras entre fevereiro e maio de 1934, contando inclusive com a presença **Church**, **Kleene**, e **Rosser**. Quanto ao trabalho de **Kleene** na definição de várias "number-theoretical functions" (funções definidas em $\mathbb{N} \times \ldots \times \mathbb{N} \longrightarrow \mathbb{N}$) no λ-Cálculo (1935), primeiro **Church** solicitou a apresentar uma versão provisória da tese no final de 1933 ou início de 1934, sob a forma: "cada função efetivamente calculável é λ-definível". Mas, **Gödel** expressava ceticismo sobre a tese.

Para uma discussão histórica mais detalhada sobre a origem da Tese de Church e a influência de **Gödel**, ver Davis (1982) e Sieg (1997).

3.2 Resumindo a Computabilidade das Funções Lambda

O λ-Cálculo foi apresentado por **Alonzo Church** na década de 1930 como parte da investigação dos fundamentos da matemática Church (1932) e Hindley (2006). O sistema original de **Church** foi demonstrado ser logicamente inconsistente em 1935, quando **Stephen Kleene** e **J. Barkley Rosser** desenvolveram o paradoxo **Kleene-Rosser**.

Em seguida, em 1936, **Church** isolou e publicou apenas a parte computacional das funções, e que depois ficou conhecida como o λ-Cálculo não-tipado **Church** Church (1936). Em 1940, ele também apresentou uma versão computacionalmente mais fraca, mas com um sistema lógico consistente, conhecido como λ-Cálculo simplesmente tipado Church (1940).

No que se refere a teoria da computabilidade, na Ciência da Computação, o λ-Cálculo, é um sistema formal que estuda **funções recursivas computáveis**. O λ-Cálculo é baseado em uma notação para funções, com o intuito de capturar os aspectos mais básicos da maneira pela qual operadores ou funções poderiam ser combinados para formar outros operadores.

Por se um **cálculo** , é um **sistema formal** , e assim, possui uma **linguagem** que no caso é extremamente simples, consistindo em somente algumas construções sintáticas e de uma semântica simples. A parte relevante do λ-Cálculo para computação ficou conhecida como λ-Cálculo **não-tipado** (sem a consideração de tipos de dados). Mas, trata-se de uma linguagem expressiva, a qual é suficientemente poderosa para expressar todos os programas funcionais e, por conseguinte, todas as funções computáveis. O λ-Cálculo serviu de base para a criação de linguagens de programação funcionais de alto nível, que surgiram nos anos 50. Isto significa que a partir do λ-Cálculo pode-se implementar qualquer linguagem funcional através da implementação de um compilador desta para o λ-Cálculo.

No mesmo ano, 1936, **Alonzo Church** havia chegado à mesma conclusão de **Turing** , de forma totalmente independente, e por um caminho diferente traçado por ele.

Assim, **Turing** e **Church** trabalharam a terceira questão levantada por **Hilbert**. Inicia-se uma nova era, onde *problemas não solucionados* se confundem com *problemas não solucionáveis* , e não há um procedimento efetivo que permita distinguir um caso do outro.

Partindo em busca do que seria um procedimento efetivo ou mecânico, procurado por **Hilbert** (o que pode ser feito seguindo-se diretamente um conjunto de regra), surgiram a sistematização e desenvolvimento das "funções recursivas". O λ-Cálculo de **Alonzo Church**, tornou pública a possibilidade da definição bem elaborada de procedimento efetivo.

Aplicado ao *Cálculo dos Predicados* criado por **Frege** em 1879, um teorema devotado a **Church** consiste fundamentalmente na demonstração de que não existe procedimento efetivo capaz de enumerar as expressões não válidas, de maneira que fica excluído *a priori* todo procedimento de decisão para as expressões do *Cálculo de Predicados* . O teorema ficou conhecido como o *Teorema de Church* , e o problema da validade para linguagens de primeira ordem é indecidível.

Church estava interessado no **problema da indecidibilidade** de **Hilbert** (*Entscheidungsproblem*). **Church** tinha alcançado resultados, empregando o conceito de λ-**definibilidade** (ao invés do computável como definido por **Turing**), mostrando assim que λ-**definibilidade** é equivalente ao conceito de *recursividade* de **Gödel-Herbrand**.

O λ-Cálculo, como sistema elaborado por **Church** para ajudar a fundamentar a matemática era **inconsistente**, mas a parte do λ-Cálculo que tratava de **funções recursivas** estava correta e teve sucesso. Usando sua teoria, **Church** propôs uma formalização da noção de "**efetivamente computável**" , através do conceito de λ-**definibilidade**. Desta forma mostrou depois, que λ-*definibilidade* viria a ser equivalente à maquina de Turing.

Esses resultados levaram **Sthephen Kleene** (1952) a unir os dois nomes "**Church's Thesis**" e "*Turing's Thesis*". Atualmente, essas hipóteses são consideradas como uma única hipótese, a **Tese de Church-Turing** , que afirma que **qualquer função que é computável por um algoritmo é uma função computável**. Anteriormente, **Gödel** também argumentou em favor dessa tese por volta de 1946.

O trabalho de **Church** e **Turing** fundamentalmente liga, como será visto no capítulo 4, as máquinas de Turing com a ideia de uma máquina computável que viria, posteriormente, ser chamada de *computador*. Os limites das máquinas de Turing, de acordo com a **Tese de Church-Turing**, também descreve os limites de todos os computadores.

Posteriormente, o λ-Cálculo influenciou na criação das linguagens de programação, principalmente as primeiras linguagens funcionais, como em LISP (List Processing).

3.3 Bibliografia e Fonte de Consulta

Biografia de Alonzo Church - ⟨http://www-history.mcs.st-and.ac.uk/Mathematicians/Church.html⟩

Richard Zach - Kurt Gödel and Computability Theory, Research supported by the Social Sciences and Humanities Research Council of Canada.

Alan Turing - ⟨https://pt.wikipedia.org/wiki/Alan_Turing⟩

Máquina de Turing -
Turing Completude - ⟨https://pt.wikipedia.org/wiki/Turing_completude⟩

Biografia de Turing - ⟨http://www-history.mcs.st-and.ac.uk/Biographies/Turing.html⟩

Stephen Kleene - ⟨https://pt.wikipedia.org/wiki/Stephen_Kleene⟩

Michael Sipser - Introdução à Teoria da Computação. Cengage Learning. 2011.

Thomas A. Sudkamp - Languages and Machines: an introdution to theory of computer science. Addison Wesley. 1988.

Walter A. Carnielli e Richard L. Epstein - Computabilidade, Funções Computáveis, Lógica e os Fundamentos da Matemática. Ed. UNESP. 2006.

Richard L. Epstein e Walter A. Carnielli - Computability, Computable Functions, Logic and The Foundations of Mathematics. Wadsworth & Brooks/Cole. Mathematics Series. 1989.

Stephen Kleene - ⟨https://pt.wikipedia.org/wiki/Stephen_Kleene⟩

Marvin Minsky - ⟨https://en.wikipedia.org/wiki/Marvin_Minsky⟩

3.4 Referências - Leitura Recomendada

Richard Zach - Kurt Gödel and Computability Theory. Research supported by the Social Sciences and Humanities Research Council of Canada. Department of Philosophy University of Calgary. Calgary, AB T2N 1N4, Canada. (não foi possível obter o ano desta publicação) - ⟨http://people.ucalgary.ca/~rzach/static/cie-zach.pdf⟩,

acessado em 12 de Outubro de 2015.

Martin Davis, Emil L. Post : His life and work, in M. Davis (ed.), Solvability, provability, definability : the collected works of Emil L Post (Boston, MA, 1994), xi-xxviii.

Davis, Martin. 1982. Why G¨odel didnt have Churchs thesis. Information and Control 54: 324.

Church, Alonzo. 1932. A set of postulates for the foundation of logic. Annals of Mathematics 33: 346366.

Church, Alonzo. 1933. A set of postulates for the foundation of logic (second paper). Annals of Mathematics 34: 839864.

Church, Alonzo. 1935. An unsolvable problem of elementary number theory. Bulletin of the American Mathematical Society 41: 332333.

Church, Alonzo. 1936a. A note on the Entscheidungsproblem. Journal of Symbolic Logic 1: 4041.

Church, Alonzo. 1936b. An unsolvable problem of elementary number theory. American Journal of Mathematics 58: 345363.

Davis, Martin. 1982. Why Gödel didnt have Church's thesis. Information and Control 54: 324

Hilbert, David and Paul Bernays. 1939. Grundlagen der Mathematik, vol. 2. Berlin: Springer.

Kleene, Stephen C. 1935. A theory of positive integers in formal logic. American Journal of Mathematics 57: 153173, 219244.

Kleene, Stephen C. 1936a. General recursive functions of natural numbers. Mathematische Annalen 112: 727742.

Kleene, Stephen C. 1936b. λ-definability and recursiveness. Duke Mathematical Journal 2: 340353.

Kleene, Stephen C. 1952. Introduction to Metamathematics. Amsterdam: NorthHolland.

Kleene, Stephen C. 1981. Origins of recursive function theory. Annals of the History of Computing 3: 5267.

Kleene, Stephen C. 1987. Gödels impression on students of logic in the 1931s. In Gödel Remembered, eds. Paul Weingartner and Leopold Schmetterer, 4964. Bibliopolis.

Kleene, Stephen C. and J. Barkley Rosser. 1935. The inconsistency of certain formal logics. Annals of Mathematics 36: 630636.

Sieg, Wilfried. 1997. Step by recursive step: Churchs analysis of effective computability. Bulletin of Symbolic Logic 3: 154180.

Sieg, Wilfried. 2005. Only two letters: The correspondence between Herbrand and Gödel. Bulletin of Symbolic Logic 11: 172184.

Sieg, Wilfried. 2006. Gödel on computability. Philosophia Mathematica 14. Forthcoming.

CAPÍTULO 4

Turing - A Computação sem Computadores

A Pré-história dos computadores atuais foi abordada no capítulo 2 e remonta aos tempos das primeiras máquinas calculadoras mecânicas, de **Schickard** (1623), **Blaise Pascal** (1642), **Leibniz** (1673) e outros. Embora não tendo o objetivo de calcular, **Joseph-Marie Jacquard** em 1801, inventou o tear mecânico automatizado e programado através de cartões perfurados que conduziu mais tarde à ideia de se usar numa máquina de calcular programável.

A ideia de **Jacquard** foi desenvovida por **Charlie Babbage**, através de sua máquina analítica (1834). Essa máquina, já continha muitas ideias básicas usadas em qualquer computador automático moderno: podia acumular, fazer cálculos, podia ser controlada, porém nunca foi foi concluída, possivelmente pelas limitações teconológicas da época. **Babbage**, em torno de 1840, até planejou uma máquina programável não-binária, mas decimal.

A máquina de **Babage** deu um passo importante na história do que viria a ser um computador, como o projeto de uma máquina mecânica de uso geral. Contemporânea de **Babbage**, o talento matemático de **Ada Lovelace** (1843) , colaborando na máquina de **Babbage** , faz surgir um procedimento criado para ser processado pela máquina, o que muitos consideram ser o primeiro "programa" de computador .

Outras ideias se sucederam como **Frank S. Baldwin** em 1885, a máquina de **William Seward Burroughs** em 1886, e **Herman Hollerith** , entre 1884-1890, que baseou-se na idéia de **Charles Babbage** e aperfeiçoou a ideia dos cartões perfurados de **Jacquard**. Das ideias da computação mecânica no século XIX surgiram as primeiras concepções tecnológicas do que poderia ser um de computador. Entretanto:

Todas essas máquinas estavam longe de ser um computador de uso geral, pois não eram programáveis. Isto quer dizer que a entrada era feita apenas de números, mas não de instruções, a respeito do que fazer com os números de entrada na máquina.

No século XX, na década de 30, **Alonzo Church** e **Alan Turing**, iniciaram a criação da **Ciência da Computação** , tendo a ideia de criar máquinas programáveis nos anos 30, antes mesmo, das ideias dos circuitos lógicos de Shannon, do projeto lógico de **John von Neumann** e antecedendo ao advento dos vários projetos de computadores nas décadas de 40 e 50.

Alonzo Church (1903-1995), na University of Princeton e depois na University of Califórnia, foi orientador de vários pesquisadores de renome, entre esses, **Alan Turing**. Foi um matemático e lógico estadunidense, atuou na área da lógica matemática e contribuiu para a **teoria da computação**, desenvolvendo a **teoria da recursão** . O trabalho de **Church** com **Turing** será focalizado no capítulo 3.

4.1 Alan Turing

Alan Mathison Turing nasceu em 23 de junho de 1912 em Londres. A revolução do computador começou efetivamente a realizar-se na primavera de 1935 na Inglaterra, quando o estudante do King's College, Cambridge, **Alan Turing**, durante um curso ministrado pelo matemático **Max Neumann**, tomou conhecimento do trabalho de **Hilbert**. Turing como aluno de mestrado em Kings College Cambridge, Reino Unido , aceitou o desafio; ele tinha sido estimulado pelas palestras do lógico **M. H. A. Newman** e aprendeu nestas, o trabalho de **Gödel** e o *Entscheidungsproblem* (problema de decisão no idioma alemão). **Newman** usou a palavra "mecânico". Para a pergunta "o que era um processo mecânico?" **Turing** deu a resposta característica: "algo que pode ser feito por uma máquina" e ele começou a se empenhar na tarefa de analisar a noção geral de uma máquina de computação .

O trabalho de **Hilbert** ao findar o século XIX e iniciar o século XX, o teorema da incompletude de **Gödel** (1931) e seus demais resultados, sensibilizaram **Alan Turing**, o então estudante de mestrado em *Cambridge*.
É atribuído a **David Hilbert**, a ideia de "**problema de decisão**". Na conferencia apontada no capítulo 14 do Volume I, **Hilbert** colocou questões bastante precisas. Primeiro, se a matemática era **completa** ...? Segundo, se a matemática era **consistente** ...? E por terceiro, se a matemática era **decidível** ...? Com isto, ele queria dizer:

"*existiria um método definitivo que poderia, em princípio, ser aplicado à qualquer asserção, e que garantiria a resposta correta da decisão, nos casos em que a asserção fosse verdadeira*?"

Depois de concluir o mestrado em King's College (1935) e receber o *Smith's Prize* em 1936 com um trabalho sobre a Teoria das Probabilidades, **Turing** se enveredou pela área promissora da **Ciência da Computação**.

De setembro de 1936 a julho de 1938, **Turing** realizou seu doutorado em Princeton, Nova Jersey, sob a orientação de **Alonzo Church**. Lá, **Turing** conheceu

Figura 35 – Alan Turing nos anos 30.

Fonte: www.historytoday.com.

John von Neumann em Princeton. Durante este período, **Turing** também estudou *criptologia* . **Turing** construiu uma máquina de cifras baseada em um multiplicador binário construído utilizando relés eletromagnéticos . Neste período, a possibilidade de uma guerra contra a Alemanha estava se desenhando e já era bastante concreta.

Em 1936, em Princeton (USA), **Alan Turing**, ainda jovem, definiu um **modelo abstrato de uma máquina lógica**, construída mentalmente, um modelo abstrato de um computador, que se restringia apenas aos aspectos lógicos do seu funcionamento (memória, estados e transições) e não a sua implementação física, possível para resolver questões tais como o problema de decisão proposto por **Hilbert** no seu problema número 10, e que foi chamada a **máquina de Turing**. Em 1936, antes da existência de computadores, **Alan Turing** propôs o termo **computável**. **Turing**, definiu em seu trabalho, a máquina teórica, que ele chamou de "máquina de computar".

Robin Oliver Gandy (1919-1995) foi um aluno e ajudante de **Alan Turing**, tendo sido orientado por **Turing** durante o seu doutorado na Universidade de Cambridge, onde trabalharam juntos. Como também seu amigo ao longo da vida, traça a linhagem da noção de "máquina de calcular" até a de **Babbage** (em cerca de 1834) e propõe a "Tese de Babbage" :

Que o desenvolvimento e análise de operações são agora capazes de serem executados por máquinas.

A análise de **Robin Gandy** , da máquina analítica de Babbage , descreve as seguintes cinco operações:

1. As funções aritméticas $+$, $-$, ., onde $-$ indica subtração apropriada $x - y = 0$, se $y \geq x$.

2. Qualquer seqüência de operações é uma operação.
3. Iteração de uma operação (repetir n vezes uma operação P).
4. Iteração condicional (repetir n vezes uma operação P condicional sobre o sucesso do teste T).
5. Transferência condicional (i.e. a instrução condicional *goto*).

Robin Gandy afirmava que as funções que podem ser calculadas por (1), (2), e (4) são precisamente as que são Turing computáveis . Ele cita outras propostas de máquinas de calcular universais incluídas as de **Percy Ludgate** (1909), **Leonardo Torres y Quevedo** (1914), **Maurice d'Ocagne** (1922), **Louis Couffignal** (1933), **Vannevar Bush** (1936), **Howard Aiken** (1937). No entanto:

> ... A ênfase está na programação de uma sequência fixa iterável de operações aritméticas. A importância fundamental da iteração condicional e transferência condicional para uma teoria geral da máquinas de calcular não é reconhecida ...

A preocupação era saber o que, efetivamente, tal computação poderia fazer. As respostas vieram sob a forma teórica, de sua máquina abstrata, que possuia uma fita de tamanho infinito e um cabeçote para leitura e escrita, movendo-se pela fita como na Figura 36. A máquina de Turing possibilitava calcular qualquer número e função, de acordo com instruções apropriadas. Devido ao seu caráter infinito, tal máquina não podia ser construída, mas tal modelo podia simular a computação de qualquer *procedimento efetivo* - sequências finita de passos para completar tarefas, chamada, posteriormente, de *algoritmo* - que ele imaginava, poder ser executado em um computador real, a ser construído posteriormente. Isto é, a palavra algoritmo corresponde a procedimentos computáveis.

Muito embora, procedimentos efetivos tenham tido uma longa história na matemática, a noção em si de algoritmo não foi precisamente definida até o século XX. Antes disso, os matemáticos tinham uma noção intuitiva do que eram algoritmos, e se baseavam naquela noção quando os usavam e descreviam. Mas, aquela noção intuitiva era insuficiente para se chegar a um entendimento mais profundo do algoritmos. A história de **Hilbert** com seu problema 10 relata como a definição precisa de algoritmo foi crucial para esse problema matemático lançado por **Hilbert** em 1900. O décimo problema de **Hilbert** era conceber um processo (como Hilbert chamava) que testasse se um polinômio tinha raizes inteiras Sipser (2011).

Até então, os matemáticos da época tinham o conceito intuitivo do que fosse um algoritmo, e como sabemos hoje, nenhum algoritmo existe para a tarefa do problema 10 de **Hilbert**. Era impossível para os matemáticos da época chegarem a esta conclusão, com o conceito intuitivo de algoritmo. Este conceito intuitivo podia ser adequado para se prover algoritmos direcionados determinadas tarefas, mas era inútil para provar que nenhum algoritmo existia para uma tarefa específica. Deste modo, o problema 10 de **Hilbert** teve que esperar por essa definição.

4.2 Conceito Informal de Procedimento Efetivo

Os historiadores da palavra **algoritmo** encontraram a origem do termo no sobrenome, *Al-Khwarizmi* , do matemático do século IX, cujas obras foram traduzidas no ocidente no século XII. Uma dessas obras recebeu o nome *Algorithmi de numero indorum*, sobre os algoritmos usando o sistema de numeração decimal (hindu-arábico). Outros autores, entretanto, defendem a origem da palavra em *Al-goreten* (raiz - conceito que se pode aplicar aos cálculos).

O dicionário *Vollständiges Mathematisches Lexicon* (Leipzig, 1747) refere a palavra "Algorithmus" . Nesta designação está combinado as noções de quatro cálculos aritméticos, nomeadamente a adição, multiplicação, subtração e divisão. Como uma invenção de **Leibniz**, o termo "*algorithmus infinitesimalis*" foi utilizado para significar; "maneiras de calcular com quantidades infinitésimas".

Desde a época de *Al-Khwarizmi* (830), de onde surgiu a palavra **algoritmo** , passando pelos tempos de **Charles Babage** e **Ada Lovelace** (1840), quando surgiu a ideia de se programar computadores, que baseou-se no conceito intuitivo do que vem a ser um *algoritmo*: "os passos necessários para realizar uma tarefa".

Como imaginado na época de **Alan Turing**, **procedimento efetivo** correspondia a ideia de capturar a noção informal de uma especificação do que, hoje, chamamos um *programa* de computador, este podendo ser colocado numa máquina que iria executá-lo.

Foi pensado como uma **descrição finita** , **não ambígua**, de um **conjunto de operações**, onde cada operação deveria ser efetiva, no sentido de que existiria um procedimento efetivanente mecânico para computá-las. Tal descrição deveria incluir o **procedimento de decisão** da ordem de execução das operações, que também deveria ser mecanicamente computável.

A ideia era incluir todas as descrições possíveis que poderiam ser consideradas como um procedimento efetivo. Tal definição, implicitamente, exige a possibilidade da construção de um máquina capaz de executar as operações na ordem pré-estabelecida no procedimento. Com isso, o procedimento precisava ser interpretado por um usuário da possível máquina, e isto implicava que deveria existir uma *linguagem* usada para codificar a descrição do procedimento.

4.3 Formalização do Conceito de Algoritmo

Em meados dos anos 30, **Alan Turing** e **Alonzo Church** chamavam tal sequência de regras, de procedimento efetivo, como algo que se podia excutar mecanicamente. Partindo em busca do que seria um procedimento efetivo ou mecânico (o que pode ser feito, seguindo-se diretamente um algoritmo ou conjunto de regras), surgiram a

sistematização e desenvolvimento das "funções recursivas" . O λ-Cálculo de **Alonzo Church**, tornou pública a possibilidade da definição bem elaborada de um *procedimento efetivo recursivo*. A grande vantagem da recursão está na possibilidade de usar um procedimento efetivo *finito* para definir, analisar ou produzir um estoque potencialmente *infinito* de sentenças ou outros dados. **Gödel** introduziu a noção de função recursiva geral, baseada sobre uma sugestão de Herbrand (1931). Quando **Gödel** visitou Princeton no ano letivo de 1933/34, ele deu uma série de palestras lá entre fevereiro e maio de 1934, que contou com a presença **Church**, **Kleene**, e **Rosser**. **Gödel** definiu as funções recursivas gerais (1934) como aquelas que poderiam ser avaliadas (computadas) usando um conjunto específico de regras de substituição, a partir de um conjunto de equações de definição, e para as quais o resultado da avaliação (computação) era unicamente determinado.

Um *procedimento efetivo* era o procedimento para uma classe de problemas, sendo um método para o qual, cada passo podia ser descrito como uma operação mecânica, e que, se executadas rigorosamente resultava em:

- sempre dar alguma resposta, em vez de nunca dar nenhuma resposta;
- sempre dar a resposta certa e nunca dar uma resposta errada;
- sempre efetuada num número finito de passos, em vez de um número infinito;
- funcionar para todas as instâncias de problemas da classe .

Uma característica essencial de um *procedimento efetivo* é que ele não requer qualquer engenhosidade de qualquer pessoa ou máquina para executá-lo.

Pode-se requerer, também, que um procedimento efetivo quando aplicado a um problema fora de uma classe de problemas em questão, para a qual é eficaz, possa ser interrompido sem resultado ou continuar indefinidamente sem parar, mas não deve retornar um resultado como se fosse a resposta ao problema.

O conceito de um algoritmo foi formalizado, pela **máquina de Alan Turing** e pelo λ-Cálculo de **Alonzo Church**, que formaram as primeiras bases da Ciência da Computação . A formalização do que era chamado de procedimento efetivo, veio nos artigos de 1936 de **Church** e **Turing**. **Church** usou um sistema notacional denominado λ-Cálculo para definir funções e executá-las como algoritmos. Por outro lado, **Alan Turing** conceituou algoritmo através de sua ideia de máquina abstrata e, essas duas definições foram demonstradas equivalentes. E essa conexão entre a **noção informal de algoritmo dos matemáticos** e a **definição precisa** veio a ser chamada a **Tese de Church-Turing**.

A **Tese de Church-Turing** - **Noção intuitiva de algoritmos** (matemáticos) é igual a **algoritmos de máquina de Turing** (computação).

A **Tese de Church-Turing** provê a definição de algoritmo necessário para resolver o décimo problema de **Hilbert**. Esse décimo problema era "*conceber um processo que testasse se um polinômio tinha uma raiz inteira*". **Hilbert** não usou o termo *algoritmo*, mas sim, um *processo* com o qual o teste pudesse ser determinado por um número finito de operações. O que **Hilbert** queria é que algum algoritmo fosse construído, assumindo que o mesmo já exisita e precisa ser construído.

Entretanto, em 1970, **Yuri V. Matijacevic** em *Enumerable Sets are Diophantine* Matijacevic (1970), baseado notrabalho de **Martin Davis** , **Hilary Putnam** e **Julia Robinson** , mostrou que *nenhum algoritmo existe* para se testar se um polinômio tem raízes inteiras. Na realidade, sabe-se hoje que existem raízes nos números complexos. Existem técnicas que formam a base para se provar que esse e outros problemas são algoritmicamente insolúveis, como desenvolvido em Sipser (2011). Um esboço desta prova pode ser vista em Sipser (2011), Cap.3 p. 163.

Podemos continuar falando de máquinas de Turing , mas o foco mais interessante, aqui, é abordar *algoritmos*.

4.4 O Que Seria um Procedimento Computável

Um procedimento efetivo, como afirmado acima, seria um **algoritmo** , que quando executado com qualquer entrada, sempre deveria terminar. Foi imaginado, que embora um procedimento efetivo fosse sempre finito, ele poderia descrever uma tarefa (processo) que nunca terminasse, ou seja um *loop* infinito .

O termo **algoritmo** tem sido muitas vezes utilizado no mesmo sentido que procedimento efetivo acima. Entretanto, para os nossos propósitos, restringimos o conceito de algoritmo para um procedimento efetivo que nunca entra em *loop*.

Uma máquina de Turing serve como um modelo preciso para a definição de algoritmo e, assim podemos padronizar uma forma pela qual se pode descrever **algoritmos de máquinas de Turing**. Logo, um *algoritmo* pode ser considerado uma sequência de operações que podem ser simuladas por uma máquina de Turing.

Agora, a noção intuitiva de **procedimentos efetivos**, usada pelos matemáticos, é igual a **algoritmos de máquina de Turing**, ou seja, a mesma coisa que **procedimentos computáveis**.

Sobre a **computabilidade** , pensada na época, ocorria a **Turing** algumas questões:

- O que é um **problema computável**?
- Quais são os **problemas computáveis**?
- O que pode ser efetuado pela máquina de computar?

- Existem problemas **que não podem ser resolvidos** pelo computador?

Estas questões fizeram surgir a **Teoria da Computabilidade** . Esta teoria, também chamada de **Teoria da Recursão** , é um ramo da lógica matemática que foi originado na década de 30 com o estudo das funções computáveis de Turing. A **Teoria da Recursão** foi originada com o trabalho de **Kurt Godel**, **Alonzo Church**, **Alan Turing**, **Stephen Kleene** e **Emil Post** nos anos 30.

Na Teoria da Computabilidade, um *modelo de computação* é a definição de um conjunto de operações que poderiam ser executadas numa computação. Somente assumindo certo *modelo de computação*, é possível discutir as limitações de algoritmos ou computadores e, também analisar os recursos computacionais requeridos, como **tempo de execução** e **espaço de armazenamento** para um algoritmo. Isto fez surgir a Teoria da Complexidade , que é hoje tratado numa disciplina sobre Análise de Algoritmos .

Entretanto, o estudo da **computabilidade** mostrou dois resultados negativos com relação ao computador teórico. Pensava **Turing**:

- Nem tudo poderia ser resolvido com o intermédio da máquina de computar (**Cantor**,**Gödel**).
- É impossível apontar com precisão a classe dos problemas computáveis.
- Sabemos que existem problemas não resolvíveis através dos computadores atuais, mas não sabemos exatamente quais são.

4.5 Formalmente Definindo uma Máquina de Turing

Como em Sudkamp (1988), formalmente, uma **máquina de Turing** padrão é uma tupla de 5 componentes, $M = (Q, \Sigma, \Gamma, \delta, q_0)$, onde Q é um conjunto finito de estados, Γ é um conjunto finito chamado alfabeto da fita, Γ contém um símbolo especial ϵ que representa a cadeia vazia, Σ^* é um subconjunto de $\Gamma - \{\epsilon\}$ chamado o alfabeto de entrada (o alfabeto todo Σ inclui a cadeia vazia ϵ), δ é uma função parcial de $Q \times \Gamma$ para $Q \times \Gamma \times \{L, R\}$, e $q_0 \in Q$ é um estado distinguido chamado *estado inicial*.

Um L indica um movimento de uma posição da fita à esquerda, e R uma posição à direita. A fita de uma máquina de Turing se estende indefinidamente em uma direção. Suponha de estender para à direita da fita. A fita é numerada da posição 0, em diante, sobre os números naturais.

Uma *computação* começa com a cabeçote da fita na posição 0, no estado q_0 na posição mais à esquerda (limite da fita è esquerda). A entrada, um símbolo de Σ^* é escrita no início da fita na posição 1. A posição 0 e o restante da fita são assumidos ser o caracter branco (ϵ). O alfabeto da fita provê símbolos adicionais que podem ser

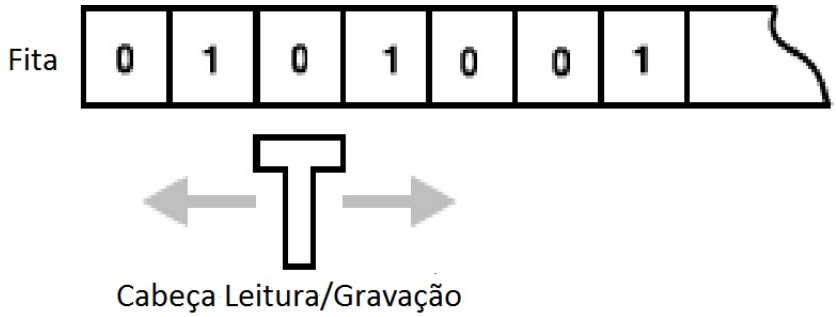


Figura 36 – A máquina de Turing em 1936.

Fonte: pt.wikipedia.org.

usados durante uma computação.

Uma transição consiste de três ações: mudança de estado, escrever um símbolo sobre uma posição da fita, e movimentar a cabeçote da fita. A direção do movimento $\{L, R\}$ é indicado no terceiro componente da transição.

A configuração de máquina, posição 2, contendo um símbolo x, designando um estado q_i e a transição $\delta(q_j, y, L)$ combinam para produzir a nova configuração, o símbolo y escrito substituindo x (y sobreescreve x) e a cabeça da fita é posicionada à esquerda, ou seja, na posição 1.

A habilidade da máquina se mover em ambas as direções e processar brancos (equivalentes à cadeia vazia ϵ) introduz a possibilidade de uma computação continuar indefinidamente.

Mas, a máquina de Turing **pára** quando ela encontra um estado, um par de símbolos (q, x) para o qual nenhuma transição é definida (lembre que δ é uma função parcial de $Q \times \Gamma$, e neste caso nenhum valor y existe em $Q \times \Gamma \times \{L, R\}$. Quando dizemos que uma computação pára, isto significa que essa termina em uma situação **normal**. Uma transição da posição zero pode especificar um movimento da cabeça da fita à esquerda do limite da fita. Quando isto ocorre, a computação é dita **terminar anormalmente**.

Como foi observado, a máquina de Turing pode ser interpretada como um **algoritmo**. Efetivamente toda ação de uma máquina algorítmica , como o computador pode ser considerada, se resume a calcular o valor de uma **função** com determinados argumentos. Este fato é interessante, pois dá uma maneira de se medir a capacidade computacional de uma máquina. Uma máquina que compute mais funções que outra seria mais poderosa.

4.6 Utilidades das Máquinas de Turing

Utilidades de máquinas de Turing são apontadas em Schechter (2015):

1. Estudar os limites do que pode ser resolvido algoritmicamente (**funções são computáveis**).
2. Mostrar que existem (muitos) problemas sem solução algorítmica .
3. Estudar os requisitos de tempo e espaço (memória) necessários para resolver algoritmicamente um dado problema.
4. Construção de uma hierarquia de complexidade para os problemas, que deu origem a Teoria da Complexidade Computacional .

Também para tratar:

- Problemas Decidíveis × Indecidíveis.
- Teoria de Complexidade de Algoritmos.
- Hierarquia de Classes de Problemas (P, NP,...)

Como curiosidade, o leitor pode ver implementações de uma máquina de Turing encontradas em:

(a) Veja em ⟨http://aturingmachine.com/⟩ uma implementação de uma máquina reconhecivel como uma méquina de Turing no estilo clássico, como na Figura 37. Uma demonstração em video é mostrada no site *The Turing Machine in the classic style*.

(b) Em ⟨http://legoofdoom.blogspot.com⟩ uma outra demonstração em video pode ser vista.

Figura 37 – Uma implementação reconhecível como a máquina de Turing.

Fonte: http://legoofdoom.blogspot.com.

4.7 Critérios Gerais para Algoritmos

Se precisamos ter uma noção geral de algoritmo e, se quisermos mostrar que para alguns problemas, não existe solução algorítmica, alguns critérios gerais que propõem definições para procedimentos computáveis, devem ser conhecidos. Para este tópico, o leitor pode consultar Carnielli e Epstein (2005), Cap.7, sobre computabilidade, p.96-100, sobre os tópicos:

1. Critérios de Mal'cev , de algoritmos e funções recursivas.
2. Critérios de Hermes sobre Enumeração, Decidibilidade e Computabilidade: o conceito de algoritmo, algoritmos como procedimentos gerais, realização de algoritmos e a enumeração de Gödel.

4.8 Problemas de Decisão

Na **teoria da computabilidade** e na **teoria da complexidade computacional** um **problema de decisão** é uma questão sobre um **sistema formal** com uma resposta do tipo *sim-ou-não*. Os **problemas de decisão** são problemas de decidir se um determinado elemento de algum universo pertence ou não a um determinado conjunto. Ou equivalentemente, se satisfaz uma determinada propriedade. Métodos usados para resolver problemas de decisão são chamados de **procedimentos efetivos** ou **algoritmos**. **Se existir um algoritmo** que receba como entrada um elemento x e retorne como saída '*sim*', caso x pertença a um conjunto C, ou '*não*', caso contrário, então diz-se que o problema de decisão para o conjunto C é **decidível**. **Se não existir um algoritmo** capaz de fazer essa avaliação, então diz-se que o problema de decisão para o conjunto C é **indecidível**.

Por exemplo:

- **Problema 1**: Um exemplo clássico de problema de decisão é o conjunto dos números primos . É possível decidir, efetivamente, se um número natural é primo testando cada número possível. Embora métodos muito mais eficientes para testar a primalidade de um número sejam conhecidos, a existência de qualquer método é suficiente para estabelecer a decidibilidade. Assim, este é um problema decidível.
- **Problema 2**: Verificar se uma determinada cadeia de caracteres pertence ou não a um conjunto de cadeias de caracteres de uma linguagem formal formal. O conjunto contém exatamente as cadeias para as quais a resposta a pergunta é '*sim*'. Voltando ao exemplo dos números primos proposto anteriormente, pode-se vê-lo também como a linguagem de todas as cadeias no alfabeto 0, 1, 2, ..., 9 tal que o inteiro correspondente à cadeia é um número primo.
- **Problema 3**: "Dados dois números x e y, y é divisível por x?" Este é outro problema de decisão. Ou ainda: "Dado um número inteiro x, x é um número

primo?". A resposta para esses problemas pode ser '*sim*' ou '*não*', e depende dos valores que as variáveis assumem em cada instância do problema. Para a seguinte instância do segundo problema, "7 é um número primo?" a resposta é *sim*, já para a instância "8 é um número primo?" a resposta é *não*.

Problemas de decisão estão intimamente ligados com **problemas de função**, que podem ter **respostas mais complexas** do que um simples "**sim**" ou "**não**". Um **típico prblema de função** é: "dado dois números x e y, para qual x, temos que y é divisível por x?". Esses tipos de problema estão relacionados também com os **problemas de otimização** , os quais se preocupam em achar a melhor solução para um problema particular.

O **campo da complexidade computacional** classifica **problemas de decisão decidíveis** pelo **grau de dificuldade** em resolvê-los. Dificuldade, nesse sentido, é descrito em termos de esforço computacional necessário para o algoritmo mais eficiente para um determinado problema.

O **campo da teoria da recursão**, entretanto, classifica problemas através do **grau de insolubilidade** da Teoria da Computação e da Complexidade Computacional (no inglês, Turing degree), a qual é uma medida da **não-computabilidade** inerente a cada solução.

Problemas de decisão **indecidíveis** importantes incluem o chamado **problema da parada**. Ou seja, um algoritmo para um determinado problema, pára sua execução ou não?

4.9 Problema da Parada

O mais famoso dos problemas indecidíveis está relacionado com as próprias propridades da máquina de **Turing**. Este problema pode ser formulado como segue:

> "**dada uma máquina de Turing M com alfabeto de entrada Σ e uma cadeia** (string) $w \in \Sigma^*$, **a computação de M com a entrada w, pára**?".

Nos capítulos sobre "Decidibilidade" (*Decidability*), de qualquer livro de Teoria da Computação, é mostrado que, não existe nenhum algoritmo que decida o "problema da parada" . Isto significa que **o problema da parada é indecidível**. Este é um resultado fundamental na teoria da Ciência da Computação.

No que segue, consideraremos o conjunto de todos os procedimentos efetivos (algoritmos com tempo de execução finito) em uma dada linguagem (PROGS). E assim, podemos tirar conclusões importantes a respeito da existência de **funções não-computáveis** e **problemas indecidíveis**.

4.10 Função Computável

Uma função f é dita **computável** (ou efetivamente computável), se existir um procedimento efetivo P que a computa.

No caso de uma função computável não ser total, a definição acima não fica clara. Veja a seguinte questão: o que o procedimento efetivo deve fazer, quando a função é submetida a um elemento de E, que está fora do domínio de f?

Neste caso, é preciso estabelecer uma definição mais precisa. Assim, a seguinte definição contemplará a questão colocada.

Definição (**Função Computável**)

Uma função f é dita ser computável, se existir um procedimento efetivo P tal que:

- $\forall\, x \in E = dom(f)$, P termina e gera como saída $f(x)$.
- se $x \notin E = dom(f)$, P não termina, ou seja P entra em um *loop* infinito.

Com estas explicações, o leitor pode pensar nas seguintes questões:

1. Como se pode provar que uma função é computável?
2. Como se pode provar que uma função é não-computável?
3. Qual o papel do *loop* infinito na computação de um procedimento efetivo?
4. Será que é sempre possivel construir uma linguagem que garanta que os procedimentos efetivos quando executados nunca entrem em loop infinito?
5. Será uma tal linguagem preservarão o mesmo poder de computação de linguagens reais como, por exemplo, C, C++ ou Java?
6. Será que toda função é computável por algum procedimento efetivo?

4.11 Funções Não-Computáveis

A partir dos resultados de **Cantor**,**Kurt Gödel**, **Alan Turing** e **Church**, pode-se dizer que existem funções para as quais não existe uma sequência de passos que determinem o seu valor com base nos seus argumentos.

Dizendo-se de outra maneira, não existem algoritmos para a solução de determinadas funções. São as chamadas **funções não computáveis**. Isto significa que para tais funções não há nem haverá capacidade computacional suficiente para resolvê-las, com os computadores ainda atuais. Logo, descobrir as fronteiras entre funções computáveis e não computáveis seria equivalente a descobrir os limites do computador em geral.

A tese de **Church-Turing** representava um importante passo nesse sentido. Eles perceberam que o poder computacional das máquinas de Turing se resumia a qualquer processo algorítmico, ou seja, todas as funções computáveis que pudessem ser descritas por algoritmos. Isto foi a contribuição dada pelo trabalho de **Turing** e **Church**. As **funções computáveis** são as mesmas funções **Turing-computáveis**. A importância disso estava na possibilidade de se verificar o alcance e limites do computador que se imaginava poder existir.

Um célebre teorema de **Alonzo Church** demonstrado em 1936, diz que:

não pode existir um procedimento geral de decisão para todas as expressões do Cálculo de Predicados de primeira ordem, ainda que exista tal procedimento para classes especiais de expressões de tal cálculo.

Foi então que o **conceito de um algoritmo** foi **formalizado em 1936** pela **Máquina de Turing** de **Alan Turing** e pelo λ-Cálculo de **Alonzo Church** numa abordagem usando funções, que formaram as primeiras bases da Ciência da Computação.

4.12 Porque Computabilidade é Importante?

Ao depararmos com um problema computacional , a princípio de difícil solução, as seguintes questões devem ser levantadas.

- Será que este problema possui solução para todas as entradas? Ou seja, é o problema **decidível**?
- Pode-se construir um *algoritmo* que termine em tempo finito para o problema, num tempo de execução eficiente?

4.13 Nos Dias Atuais ...

- Todos os problemas que podem ser computados numa máquina de Turing, são de alguma forma computados pelas máquinas que conhecemos nos dias de hoje.
- Embora ainda não exista uma máquina concreta com maior poder computacional do que o proposto pela máquina de Turing, a ideia da computação quântica tem indicado que existem chances de que não conhecemos todo o poder das

máquinas. Mas, se imaginarmos o **computador quântico** , possivelmente isto venha a ser verdadeiro.

4.14 Turing, a Segunda Guerra Mundial e a Criptanálise

De volta a Cambridge, **Turing** tentou construir uma máquina para calcular a função *Zeta* de **Riemann** (seu objetivo era encontrar soluções fora do que já era conhecido). A partir de setembro de 1938, **Turing** começou a trabalhar para a divisão do Governo Britânico responsável pela quebra de códigos . Em setembro de 1939, após o Reino Unido declarar guerra à Alemanha, **Turing** se apresentou em *Bletchley Park*, o centro das operações de criptanálise durante a guerra, localizado em Bletchley (64 km ao norte de Londres).

Figura 38 – Bletchley Park - o centro das operações de criptoanálise.

Fonte: www.bletchleypark.org.uk.

A máquina *Enigma* era uma máquina desenvolvida pelos alemães para codificar suas instruções militares. A cifra chamada *Lorenz*, implementada pela *Enigma* era considerada inicialmente inquebrável. Entretanto, três matemáticos Poloneses conseguiram quebrar a cifra da máquina Enigma em um modo muito particular de operação.

A partir daí, as ideias de **Turing** permitiram generalizar este método de maneira que qualquer mensagem cifrada com a *Enigma* pudesse ser decifrada. **Turing**, então, projetou uma máquina para automatizar o processo de decifrar as mensagens e surgiu a máquina *Bombe* . **Turing** precisou desenvolver métodos estatísticos sofisticados para poder realizar esta tarefa.

Durante a Segunda Guerra Mundial, mais próximo ao fim da guerra, a equipe de *Bletchley Park* desenvolveu a máquina *Colossus* (1944), um computador inglês projetado pela equipe liderada por **Alan Turing**. A inovação no Colossus foi o uso da eletrônica. O Colossus usava válvulas à vacuo. Uma válvula era um pequeno bulbo de vidro com eletrodos dentro. Seu principal objetivo era fazer a criptoanálise de

códigos ultra-secretos utilizados pelos nazistas, crifrados com a máquina *Lorenz SZ* 40/42 . Utilizando símbolos perfurados em fitas de papel, o equipamento processava a uma velocidade de 25 mil caracteres por segundo, para decodificar a cifra *Lorenz*, uma cifra utilizada pelo alto comando alemão.

Figura 39 – Máquina Enigma - versão da Marinha, exposta em Bletchley Park.

Fonte: en.wikipedia.org.

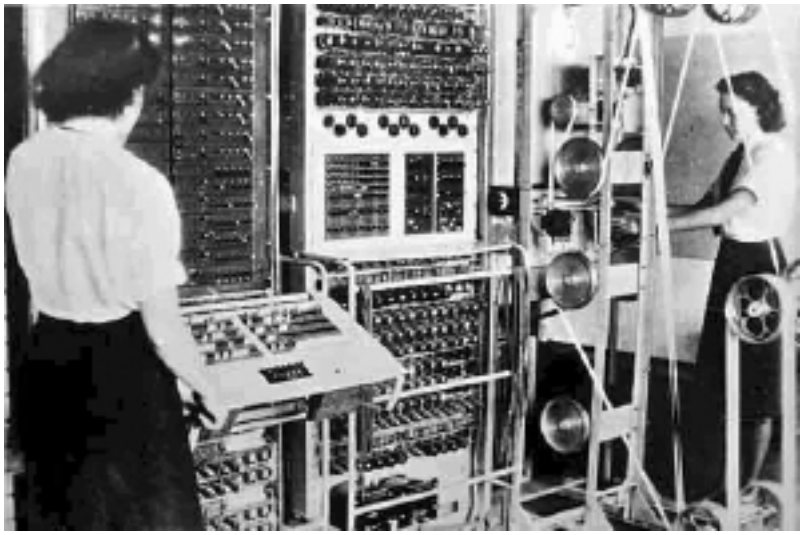


Figura 40 – Colossus - decodificava a cifra Lorenz.

Fonte: en.wikipedia.org.

Os modelos estatísticos de **Turing** também foram fundamentais para a quebra da cifra *Lorenz*. O *Colossus* foi a primeira aplicação com uso em larga escala de circuitos eletrônicos digitais, possivelmente beneficiado com a importante contribuição de Shannon.

Também neste período, **Turing** estava se dedicando à construção de uma máquina

para a criptografar sinal de voz. Ele desenvolveu um protótipo, chamado *Delilah* , mas que não foi utilizada. **Turing** recebeu a Ordem do Império Britânico (OBE) por sua contribuição durante a guerra. Essa contribuição permaneceu em segredo até depois de sua morte em 1954.

4.15 Concretização da Máquina de Turing Universal

- Máquina de Turing = Algoritmo.
- Máquina de Turing Universal (MTU) = Computador Programável.
- MTU era uma máquina capaz de realizar qualquer tarefa algorítmica, desde que o conjunto correto de instruções fosse armazenado nela.
- Ao final da 2a Guerra Mundial, **Turing** estava de posse de três ideias fundamentais:
    1. Seu próprio modelo de Máquina de Turing Universal de 1936.
    2. A velocidade e confiabilidade da tecnologia eletrônica (conforme visto no Colossus).
    3. A ineficiência de construir diferentes máquinas para diferentes propósitos.

Turing concluiu que era o momento apropriado para construir uma versão concreta de sua Máquina Universal, isto é, construir um *computador programável*, com memória interna, onde tanto instruções quanto conjuntos de dados , fossem armazenados com a mesma representação, de tal forma que o computador fosse capaz de executar sobre qualquer conjunto de dados, qualquer tarefa descrita corretamente pelas instruções.

Nenhuma das máquinas desenvolvidas até o final da Guerra puderam ser consideradas como "computadores", que atendessem a todos os requisitos acima. Algumas são máquinas de uso particular (como o Colossus, que tem como única função decifrar mensagens codificadas com a cifra *Lorenz*). Outras eram máquinas de uso geral, mas sem a capacidade de armazenamento interno das instruções.

Mais tarde, em 1945, em Manchester, na Inglaterra, **Turing** desenvolveria as suas ideias para construir os primeiros projetos de computadores no Reino Unido, como: Bombe, Colossus e o Delilah. A história de vários projetos dos primeiros computadores pode ser vista em ⟨www.cryptomuseum.com⟩.

Em 1945, **John von Neumann** , que conhecia o trabalho de **Turing** de 1936, publicou o *Report on the EDVAC*, descrevendo (de forma incompleta) o projeto para um computador com armazenamento interno de programas. A competição americana pelo desenvolvimento do computador foi positiva para **Turing** no primeiro momento. O *National Physics Laboratory* (*NPL*) contratou **Turing** para elaborar um projeto de um computador com armazenamento interno de programas.

Em fevereiro de 1946, **Turing** apresentou ao *NPL* um relatório técnico detalhado do projeto para o *ACE* (*Automatic Computing Engine*). O *ACE* foi o primeiro computador projetado no Reino Unido, uma realização de **Alan Turing** em 1946. O *ACE* previa o uso de uma linguagem de programação rudimentar para a escrita dos programas. A ideia de **Turing**: no relatório do *ACE*, **Turing** propôs até a possibilidade de que usuários remotos utilizassem o **ACE** através de uma conexão telefônica.

Turing deu importância em seu projeto ao tamanho e à velocidade de acesso à memória interna do computador. O segredo das atividades de **Turing** durante a Guerra fez com que as pessoas considerassem que ele não tinha experiência suficiente para o projeto e que o projeto não era realmente factível. Sem conseguir construir o *ACE*, **Turing** retornou para Cambridge para um ano sabático. As Universidades de Cambridge e Manchester entraram na corrida pela construção do computador e colocaram seus projetos em funcionamento antes do *ACE*. Uma versão simplificada do *ACE*, o *Pilot Model ACE*, acabou sendo construída no início da década de 1950, depois da saída de **Turing** do *National Physics Laboratory*.

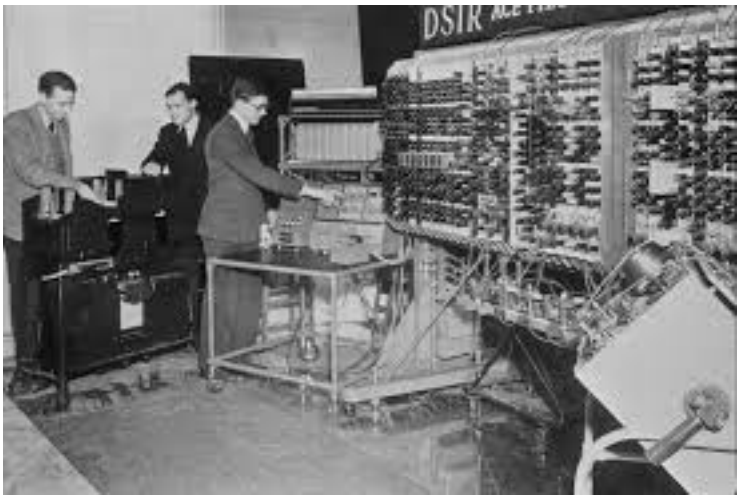


Figura 41 – ACE - o primeiro computador projetado no Reino Unido.

Fonte: www.pinterest.com.

4.16 Turing e o Surgimento das Redes Neurais

Em 1947, durante seu ano sabático em Cambridge, **Turing** se voltou para a questão de "cérebros artificiais". Esses "cérebros" deveriam ser capazes de ser treinados para a realização de tarefas. **Turing** defendia a ideia de que um sistema mecânico suficientemente complexo poderia exibir habilidades de aprendizado . Esta pesquisa foi submetida para o NPL como um relatório interno e nunca foi publicada durante sua vida. **Turing** descrevia estruturas muito semelhantes ao que hoje conhecemos como redes neurais . Ele descrevia graficamente, numa forma similar ao que conhecemos

hoje, por autômatos . Possivelmente, tenha tido as primeiras ideias de um autômato, porque a máquina de Turin se comporta como um autômato. O que motivou **John von Neumann** a desenvolver a *Teoria dos Autômatos*. Um diagrama de uma rede neural presente no relatório de textbfTuring é mostrado na Figura 42.

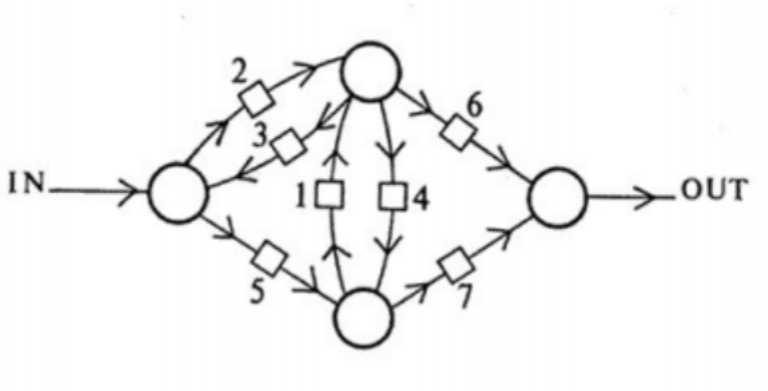


Figura 42 – Diagrama de uma Rede Neural presente no Relatório de Turing.

Fonte: streebgreebling.blogspot.com.

4.17 Turing e a Computação Científica

No final de 1947, no artigo *Rounding-off Errors in Matrix Processes*, **Turing** inventou a decomposição LU de matrizes, que ele chamou de decomposição triangular . Nesta decomposição, uma matriz é vista como o produto de duas outras matrizes caracterizadas, uma pelos seus elementos inferiores e da diagonal principal, e outra pelos seus elementos superiores e da diagonal. O método tornou-se útil para resolução de sistemas lineares, cálculo de inverso de matrizes e cálculo de determinantes .

No artigo, **Turing** se preocupou com questões a respeito da complexidade dos cálculos propostos por ele (descritas em função do número de operações de adição e multiplicação necessárias em função do tamanho da matriz de entrada) e a respeito do condicionamento das matrizes, sugerindo abordagens para evitar o acúmulo de erros devido ao processamento de matrizes mal-condicionadas.

4.18 Turing e o Surgimento da Inteligência Artificial

Em 1948, **Turing** demitiu-se do NPL e aceitou uma posição na Universidade de Manchester. Lá, ele esteve em contato com a equipe que fabricou o computador *Ferranti Mark 1* , o primeiro computador de uso geral disponível comercialmente. **Turing** contribuiu na elaboração do manual de uso deste computador. Durante este período, **Turing** continuou refletindo a respeito dos "cérebros artificiais", colocando a seguinte pergunta:

Quando podemos considerar que um sistema artificial tem realmente inteligência?

Turing discute essas ideias no artigo *Computing Machinery and Intelligence* de 1950. Neste artigo, ele propõe um possível teste (batizado por ele de *Jogo da Imitação*) que pode ser utilizado para responder a questão acima. Este teste, posteriormente, ficou conhecido como **Teste de Turing** . E o teste virou filme *O Jogo da Imitação*, lançado nos cinemas do Reino Unido e dos EUA em novembro de 2014, e posteriormetne nos cinemas do Brasil (Janeiro de 2015), e no qual o ator *Traz Benedict Cumberbatch* faz o papel de **Turing**. O filme venceu o *Oscar* de melhor roteiro adaptado, com o roteiro de *Graham Moore* baseado no livro *Alan Turing: the Enigma* de Andrew Hodges.

Exemplificando o Teste de Turing:

1) Suponha que uma loja de comércio eletrônico utilize um serviço de atendimento *online* através de um *chat*. 2) Suponha que, em princípio, os clientes são atendidos por funcionários humanos, mas que, caso não haja funcionãrios disponíveis no momento, o cliente é então atendido por um software (*chatter bot*). 3) Dizemos que este software utilizado no serviço de atendimento aos clientes passa no *Teste de Turing* se, em geral, após finalizarem seu atendimento, os clientes são incapazes de responder com segurança maior do que a de um palpite aleatório, se foram atendidas por um humano ou pelo software. O teste *CAPTCHA*, presente em diversos sites, pode ser pensado como um *Teste de Turing* reverso.

4.19 Mais sobre a Tese de Church-Turing

Os resultados fundamentais que os pesquisadores obtiveram estabeleceram, posteriormente, a **computabilidade** de **Turing**, como uma formalização correta da ideia informal do que seria um *algoritmo* ou *procedimento efetivo*.

Como explicado em Sudkamp (1988), a **Tese de Church-Turing** afirma que todo problema de decisão que se pode resolver, pode ser transformado em um problema equivalente de máquina de Turing . Isto pode ser colocado como:

Tese de Church-Turing para Problemas de Decisão - Existe um procedimento efetivo para resolver um problema de decisão P se, e somente se, existe uma máquina de Turing aceitando uma **linguagem recursiva** que resolve o problema. Neste caso, uma solução para P requer a computação retornar uma resposta para toda instância (entrada) do problema P.

Tese de Church-Turing estendida para Problemas de Decisão - Um problema de decisão P é parcialmente resolvido se, e somente se, existe uma máquina de Turing que aceita precisamente os elementos de P cujas respostas é "sim". Nesta

caso, uma solução parcial para P é uma resposta, não necessariamente completa, isto é, o procedimento efetivo retorna uma resposta afirmativa para todo p pertencente a P cuja resposta é "sim". Entretanto, se a resposta é negativa, o procedimento pode retornar "não" ou falhar para produzir uma resposta.

Assim, como uma solução para um problema de decisão pode ser formulado aceitando uma **linguagem recursiva**, uma solução parcial para um problema de decisão é equivalente a questão de aceitar uma **linguagem recursivamente enumerável** . Neste caso, temos um problema de decisão **parcialmente decidível**.

4.20 Resumindo a Tese de Church-Turing

Tese de Church-Turing: **um problema é decidível se, e somente se, ele é decidível por uma máquina de Turing**.

A tese pode ser refutada pela descoberta de um modelo de computação mais poderoso do que as máquinas de Turing. Argumentos a favor da tese:

- (a) Máquinas de Turing não podem ter seu poder computacional aumentado.
- (b) A computabilidade do λ-Cálculo de Alonzo Church.
- (c) A computabilidade das Funções μ-recursivas.
- (d) Máquinas de registradores.
- (e) A tese é comumente aceita atualmente.
- (f) Curiosidade: mesmo os algoritmos quânticos já existentes, não refutam a Tese de Church-Turing.

Após o final de seu doutorado, **Turing** retornou a Cambridge. Na mesma época, na University of Michigan, **Claude E. Shannon**, a partir da sua tese de mestrado em 1937, *A Symbolic Analysis of Relay and Switching Circuits* SHANONN (1937), foi publicado na edição de 1938 da *Transactions of the American Institute of Electrical Engineers*, onde fixava o formalismo lógico e os circuitos elétricos. Ver no capítulo 8.

4.21 Abordagens para Formalizar Computabilidade

Neste ponto, mencionamos os formalismos para noção de computabilidade. Aqui está um breve *survey* de diferentes abordagens:

1. *Representabilidade em um sistema formal* (Church, 1933; Gödel and Herbrand, 1934) - Nesta abordagem, sistema formal da aritmética (axiomas e regras de prova) é considerado e uma função é declarada computável, se para todo m e n, tal que

$f(m) = n$, podemos provar $f(m) = n$ no sistema. Ver na Parte III de Epstein (1989).

2. λ-Cálculo (Church, 1936) - Esta é a abordagem mais próxima de um sistema formal. **Church** tomou um simples alfabeto e linguagem, junto com a noção de derivabilidade (dedução) e ainda parece mais simples. Tudo é reduzido a manipulação de símbolos e $f(m) = n$, se podemos derivar esta no sistema. Ver em Rosser (1984).

3. *Descrições Aritméticas* (**Kleene**, 1936) - Uma classe de funções que inclui + e . é fechada sob algumas regras simples, como a definição de indução . Isto acarreta a classe das funções μ-recursivas 6, que é o sistema formal mais simples para trabalhar. Ver em Epstein (1989) e Sudkamp (1988). **Stephen Cole Kleene** (1909-1994) foi um matemático estadunidense. Um dos estudantes mais notórios de **Alonzo Church**, **Kleene** é reconhecido junto com **Alan Turing**, **Emil Post** e outros como um fundador da ramificação da lógica matemática conhecida por *teoria da computabilidade*. Seu trabalho fundamentou o estudo das funções computáveis. Diversos conceitos matemáticos têm seu nome, como a hierarquia de **Kleene**, a álgebra de **Kleene**, o fecho de **Kleene**, o teorema da recursão de **Kleene** e o teorema do ponto fixo de **Kleene** (usado quando se estuda semântica denotacional de linguagens de programação). Ele também é o inventor das **expressões regulares** estudadas nas disciplinas de Linguagens Formais .

O trabalho de **Kleene** na teoria da recursão , ajudou a fornecer os fundamentos da Ciência da Computação teórica. Ao proporcionar métodos para determinar quais os problemas que são solúveis, o trabalho de **Kleene** levou ao estudo de funções que podem ser computadas. No verão de 1951, na RAND Corporation, ele descobriu uma caracterização importante de autômatos finitos . Seu relatório para a RAND foi muito influente para a Ciência da Computação teórica.

A formulação de **Kleene** de função computável é um dos mais úteis, e seu trabalho anterior em funções λ, desempenhou um papel importante no apoio à Tese de Church, verificando que as funções λ coincidem com as funções intuitivamente computáveis de **Turing**.

A partir de 1930, **Kleene** mais do que qualquer outro matemático, desenvolveu as noções de computabilidade e a ideia de *processo efetivo* (como Hilbert denominava) em todas as suas formas, tanto abstratas e concretas, tanto matemáticas, quanto filosóficas. Ele tendia a lançar as bases de uma área e, em seguida, passar para a próxima, já que cada uma que se sucedia, florescia em uma área importante de investigação no seu objetivo.

Kleene desenvolveu um diversificado leque de tópicos sobre computabilidade: a hierarquia aritmética, graus de computabilidade, ordinais computáveis, autômatos finitos e conjuntos regulares com enormes consequências para a Ciência da Computação, com consequências para a filosofia e para a prova da correção de programas em Ciência da Computação.

O livros mais conhecidos de **Kleene** são *Introdução à metamatemática* (1952) e *Lógica Matemática* (1967). **Kleene** escreve no primeiro deles:

"*O objetivo deste livro é fornecer uma introdução ligada aos temas da lógica matemática e funções recursivas em particular, e para as investigações mais recentes fundamentais em geral.*"

4. *Descrições como máquinas* - Existiram algumas tentativas (a maioria antes do advento dos computadores). Cada uma tentou formalizar a noção intuitiva, por dar uma descrição de toda máquina possível: (a) *Máquinas de Turing*, Turing (1936), ver em Sudkamp (1988), Epstein (1989). (b) *Markov Algorithms*. (c) *Máquinas de Registros Ilimitados* (Shepherdson and Sturgis, 1963). Este modelo é uma idealização de computadores com tempo ilimitado, memória ilimitada e que fucione sem erros.

O que estas formalizações tem em comum é que elas são puramente sintáticas. São métodos para manipular símbolos.

Importante aqui é saber sobre a diferença entre os conceitos de **computabilidade** e **computação** .

- **computabilidade** (*computability*) = é um conceito *intuitivo*, mas *semântico*.
- **computação** (*computation*) = é um conceito *formal*, mas *puramente sintático*.

Os trabalhos de **Church** e **Turing** fundamentalmente estão ligados à construção dos computadores. Os limites das máquinas de Turing, de acordo com a tese de Church-Turing, também descreve os limites de todos os computadores. **Church** tinha alcançado resultados importantes, empregando o conceito de **efetivamente computável** ao invés do **computável** por uma máquina de **Turing**. O conceito de **efetivamente computável** é equivalente ao **conceito de recursividade de Gödel-Herbrand**.

No sentido de formalizar matematicamente o conceito de **computabilidade**, como apontado em Epstein (1989), nunca podemos nos livrar completamente do aspecto semântico deste conceito. O processo de computação é uma noção linguística (pressupondo que nossa noção de linguagem é suficientemente geral); o que se pode fazer é **delimitar** uma classe daquelas funções (consideradas como objetos matemáticos abstratos), para a qual exista um correspondente objeto linguístico (um processo de computação), ou seja um algoritmo.

4.22 Problemas de Decisão - Decidibilidade

Uma máquina de Turing pode ser vista como um modelo de computador de propósito geral, em que se pode definir a noção de algoritmo em termos de máquinas de Turing,

por meio da Tese de Church-Turing. No entanto, existindo um algoritmo(s) para se resolver algum problema, podemos pensar a investigar o poder de algoritmos para resolver problemas. Existem certos problemas que podem ser resolvidos algoritmicamente, enquanto outros não. Tais problemas são os chamados problemas de decisão : uma questão sobre um conjunto com uma resposta do tipo sim-ou-não. Desta forma podemos classificar os problemas: **decidíveis** , **parcialmente decidíveis** e **indecidíveis** . Neste ponto, o mais interessante é explorar os limites da solubilidade de algoritmos.

Um problema é **decídivel**, se existe um **algoritmo** que dado um elemento potencial de um conjunto, o algoritmo responde 'sim' ou 'não', de acordo se o elemento está no conjunto ou não.

Um problema é **parcialmente decidível** (o mesmo que *recursivamente enumerável*), se existe um **algoritmo** que reconhece elementos do conjunto, por dizer 'sim', mas que pode *não terminar* para elementos fora do conjunto.

Computadores parecem ser tão poderosos que até podemos acreditar que todos os problemas são solúveis por computador. Entretanto, existem problemas específicos que são *algoritmicamente insolúveis*. Na realidade, computadores são limitados e, por isso, mesmo alguns problemas comuns que pessoas precisam resolver acontecem de ser computacionalmente insolúveis. Assim, podemos dizer que um problema é indecidível. Neste caso, um algorimo dado para resolver o problema, não tem como decidir sobre o problema.

4.23 Indecidibilidade

Neste ponto, é interessante mostrar o primeiro problema que é indecidível, como provado na Teoria da Computação: o problema de se determinar se uma máquina de Turing aceita uma dada cadeia de entrada. Chamamo-lo de P_{MT}.

$P_{MT} = \{\langle M, w\rangle\}$ tal que M é uma MT e M aceita w.

Antes de tudo, pode-se observar que P_{MT} é **Turing-reconhecível**.

Mas, prova-se como um teorema clássico da Teoria da Computação, que P_{MT} é indecidível.

Vejamos o exemplo de Sipser (2011). Seja uma máquina de Turing U que reconhece P_{MT}.

U = Sobre a entrada $\langle M, w\rangle$, onde M é uma MT e w é uma cadeia. A entrada de máquinas de Turing é sempre uma cadeia de símbolos.

1. Simule M sobre a entrada w.

2. Se M em algum momento entra no seu estado de aceitação, *aceite* w; Se M em algum momento entra em seu estado de rejeição, *rejeite*.

Esta MT entra em *loop* sobre a entrada $\langle M, w\rangle$, se M entra em *loop* sobre w e, é por isso que esta máquina MT não decide sobre o problema P_{MT}. Se o algoritmo tivesse alguma forma de determinar que M não iria parar sobre w, ele poderia dizer*rejeite* w. P_{MT} é conhecido como o problema da parada. Pode-se demonstrar que um algoritmo não tem como fazer essa determinação. Portanto, P_{MT} é indecidível. Também podemos enunciar: "o problema da parada é indecidível".

Um programa de computador é essencialmente um algoritmo que diz ao computador os passos específicos e em que ordem eles devem ser executados. A implementação de algoritmos pode ser feita para um computador e, diferentes algoritmos podem realizar a mesma tarefa usando um conjunto diferenciado de instruções em mais ou menos tempo, espaço ou esforço do que outros. Tal diferença pode ser reflexo da complexidade computacional aplicada, que depende dos dados fornecidos ou de estruturas de dados adequadas ao algoritmo.

Alguns autores restringem a definição de algoritmo para procedimentos que eventualmente terminam. **Marvin Lee Minsky** (1927) é um cientista cognitivo estadunidense, e sua principal área de atuação são estudos cognitivos no campo da Inteligência Artificial. **Minski** é co-fundador do Laboratório de Inteligência Artificial do Instituto de Tecnologia de Massachusetts (MIT) e autor sobre o tema e suas implicações filosóficas. Sua principal contribuição para a Ciência da Computação são as redes neurais, que estabeleceu uma sub-área da Inteligência Artificial , que já haviam sido imaginadas nos tempos de **Turing**.

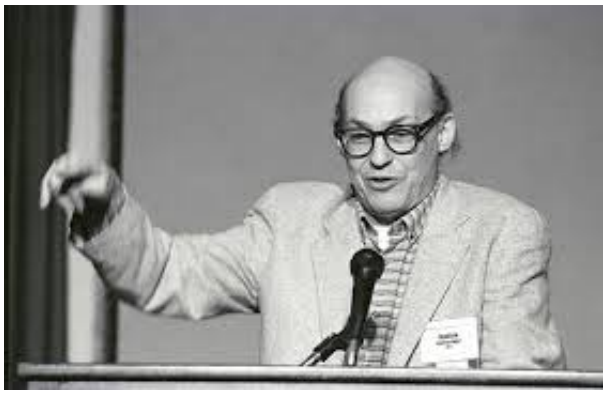


Figura 43 – Marvin Minsky - O desenvolvedor das redes neurais imaginadas por Turing.

Fonte: www.edn.com.

Minsky constatou, que se o tamanho de um procedimento não é conhecido de antemão, tentar descobri-lo é um problema indecidível, já que o procedimento pode ser executado infinitamente, de forma que nunca se terá a resposta. **Alan Turing** provou em 1936 que não existe máquina de Turing para realizar tal análise para

todos os casos, logo não há algoritmo para realizar tal tarefa para todos os casos. Tal condição é conhecida como o **problema da parada**.

4.24 Concluindo os Tempos de Turing

A encarnação moderna da Ciência da Computação foi prenunciada pelo matemático **Alan Turing**. Ele desenvolveu em detalhes uma noção abstrata do que se poderia agora chamar de computador programável, um modelo de computação conhecido como máquina de Turing. Esse cientista demonstrou a existência da máquina de Turing Universal que pode ser usada para simular qualquer outra máquina de Turing. Ademais, ele afirmou que a máquina de Turing Universal captura completamente o significado de se realizar uma tarefa por meios algorítmicos. Isto é, se existe um algoritmo equivalente para uma máquina de Turing Universal, então, esse algoritmo pode ser realizado em qualquer peça de hardware, que realiza exatamente a mesma tarefa que o algoritmo original. Esta declaração, conhecida como a tese de Church-Turing, asserta a equivalência entre o conceito de que classes de algoritmos podem ser executados em algum dispositivo físico com o rigor matemático de uma máquina de Turing Universal. A larga aceitação desta tese levou ao desenvolvimento de uma rica *teoria de Ciência da Computação*.

4.25 Bibliografia e Fonte de Consulta

Luis Menasché Schechter - A Vida e o Legado de Alan Turing para a Ciência, Seminários Apresentados na UFRJ e no IMPA Departamento de Ciência da Computação / UFRJ, ⟨http://www.dcc.ufrj.br/~luisms/turing/Seminarios.pdf⟩, site atualizado em março de 2015. Acessado 23 de Abril de 2015.

Newman, M. H. A.. (1955-01-01). Alan Mathison Turing. 1912-1954. Biographical Memoirs of Fellows of the Royal Society 1 (0). DOI:10.1098/rsbm.1955.0019.

Alan Turing - ⟨https://pt.wikipedia.org/wiki/Alan_Turing⟩, Página visitada em 13 Maio 2015.

The Alan Turing Internet Scrapbook - http://www.turing.org.uk/scrapbook/early.html

Thomas A. Sudkamp - Languages and Machines - An Introduction to the Theory of Computer Science, 1988.

Homer, Steven; Selman, Alan L.. In: Steven. Computability and Complexity Theory. [S.l.: s.n.], 2001. p. 35. ISBN 0-387-95055-9, Página visitada em 13 Maio 2015.

4.26 Referências - Leitura Recomendada

Turing, A.M. (1936), On Computable Numbers, with an Application to the Entscheidungsproblem, Proceedings of the London Mathematical Society, 2 42: 230-65, 1937, doi:10.1112/plms/s2-42.1.230

Turing, A.M. (1938), On Computable Numbers, with an Application to the Entscheidungsproblem: A correction, Proceedings of the London Mathematical Society, 2 43 (6): 544-6, 1937, doi:10.1112/plms/s2-43.6.544

COPELAND, B. J. (Ed.), 2005. Alan Turing's Automatic Computing Engine. OUP, Oxford. ISBN 0-19-856593-3.

CARPENTER, B. E., DORAN R. W., 1986. A. M. Turing's ACE Report of 1946 and Other Papers. MIT Press, Cambridge.

YATES, David M., 1997. Turing's Legacy: A History of Computing at the National Physical Laboratory, 1945-1995. Science Museum, Londres.

WILKINSON, J. H., 1980. Turing's Work at the National Physical Laboratory and the Construction of Pilot ACE, DEUCE and ACE. In N. Metropolis, J. Howlett, G.

Martin Davis - The Universal Computer: The Road from Leibniz to Turing, 2011.

Charles Petzold - The Annotated Turing: A Guided Tour Through Alan Turing's Historic Paper on Computability and the Turing Machine 1st Edition.

CAPÍTULO 5

A Computabilidade de Emil Post

Emil Leon Post (1897-1954) , nascido no Império Russo (atual Polônia) e falecido em Nova York, Estados Unidos, era um especialista em Lógica Matemática. Após a graduação como bacharel, **Post** iniciou a pós-graduação na Columbia University. Um evento significativo para a carreira de **Post** foi a publicação do *Principia Mathematica* de **Bertrand Russell** e **Alfred North Whitehead**. O primeiro volume do *Principia Mathematica* foi publicado em 1910, o segundo em 1912, e o terceiro em 1913. Quando **Post** iniciou seus estudos de graduação, o projeto estava em pleno desenvolvimento, e **Post** participou de um seminário em Columbia sobre o *Principia Mathematica*. **Post** foi agraciado com o grau de Mestre em 1918 e de Ph.D. em 1920. Sua tese de doutorado foi sobre lógica matemática.

Figura 44 – Post - O criador do método da tabela-verdade na Lógica Proposicional.

Fonte: www.pucsp.br.

5.1 A Conhecida Tese de Doutorado de Post

A tese de doutorado de **Post** é utilizada até os dias atuais, na qual ele provou a *completude* e a *consistência* do **Cálculo Proposicional** (o Cálculo da Lógica Proposicional), descrito em *Principia Mathematica* através da introdução do *método da tabela-verdade* . Ele atribuiu este método a seu professor em Columbia, **C. J. Keyser**. Ele então generalizou este seu método, que era baseado nos dois valores "verdadeiro" e "falso", para um método que tinha um número finito arbitrário de valores verdadeiros. O final, e talvez a mais marcante inovação de **Post**, foi a introdução em sua tese do conceito de *modelo* para sistemas de lógica, como, por exemplo, os sistemas de inferência baseados em um processo finito de manipulação de símbolos. Assim, o sistema lógico proposto por **Post** produz, na terminologia atual, um conjunto de palavras recursivelmente enumerável em um alfabeto finito . A tese de **Post** marcou o nascimento da *Teoria da Prova* .

Depois de seu doutoramento, **Post** foi para a Universidade de Princeton (a quarta mais rica do planeta), onde permaneceu por uma ano como supervisor assistente. Neste tempo deve ter trabalhado com **Alonzo Church**, na Teoria da Computabilidade, e ter definido sua máquina de Post , que veio a generalizar a máquina de Turing .

5.2 Post e os Resultados de Gödel

Na década de 1920, **Post** provou resultados semelhantes aos que **Gödel**, **Church** e **Turing** descobriram mais tarde, mas ele não os publicou. O motivo foi que **Post** sentia que uma análise completa era necessário para ganhar aceitação. Ele escreveu:

- A corretude deste resultado é claramente, inteiramente, dependente da confiabilidade da análise levando à generalização acima ... é fundamentalmente fraco em sua confiança na lógica do Principia Mathematica ... para a generalidade completa, uma análise completa teria de ser dada de todas as maneiras possíveis em que a mente humana poderia configurar processos finitos para a gerar sequências.
(tradução do autor)

Quando **Gödel** publicou seus teoremas da incompletude em 1931, **Post** percebeu que tinha esperado muito tempo para publicar o que ele tinha provado e que agora todo o crédito iria para **Gödel**. Em um cartão postal escrito a **Gödel** em 1938, logo depois que eles se encontraram pela primeira vez, **Post** escreveu:

... - Por quinze anos eu carreguei o pensamento de surpreender o mundo matemático com minhas idéias não ortodoxas (heterodoxas) e encontrar o homem, o principal responsável pelo desaparecimento desse sonho, ... Desde que você parecia interessado na minha maneira de chegar a estes novos desenvolvimentos, talvez Church pode mostrar-lhe uma longa carta que escrevi a ele sobre minhas ideias. Quanto a quaisquer reclamações que eu poderia fazer, talvez, o melhor que eu posso dizer é que

eu teria provado o teorema de Gödel em 1921 - e eu teria sido Gödel. (tradução do autor)

5.3 Concorrendo com Turing

Como apontado em Torres (2000), em 1936, antes do aparecimento dos computadores, **Alan Turing** e **Emil Post** publicaram, respectivamente, os artigos:

A. M. Turing. *On Computable Numbers with an Application to the Entscheidungs-Problem.* Proc. London Math. Soc., 2, 1936, pp. 230265.

E. L. Post. *Finite Combinatory Processes Formulation I.* Journal of Symbolic Logic, 1, 1936, pp. 103105.

Estes dois trabalhos propuseram, de modo independente, o conceito rigoroso (matemático) de *algoritmo*, conceito este tão importante que fornece, de forma abstrata, os traços fundamentais dos atuais computadores eletrônicos . O computador abstrato (ou matemático) proposto por **Post**, é mais simples que o de **Turing**, no que respeita instruções elementares e meios de armazenamento. Esta simplicidade tem um custo: os algoritmos do computador matemático de **Post** exigem, em geral, mais memória e maior quantidade de passos do que os correspondentes algoritmos de **Turing** . Este fato explica, em certa medida, porque as ideias de **Turing** são constantemente usadas em Ciência da Computação , e porque elas são tão comuns na literatura de investigação e de divulgação científica, em detrimento das de **Post**. Este capítulo é dedicado ao conceito de "computador matemático de Post". Um bom exercício para iniciantes na teoria da computabilidade é, usando uma notação adequada, que permita de um modo completamente natural e intuitivo, usar o computador de Post como um meta-computador.

Emil Post, em 1936, ele propôs o que agora é conhecido como uma **máquina Post**, uma espécie de *autômato* que antecede a noção de um programa que **von Neumann** veio a estudar em 1946.

Em 1941, ele escreveu:

... pensamento matemático é, e deve ser, essencialmente criativa ...

mas ele disse que há limitações e **lógica simbólica** é: -

... o indiscutível meio para revelar e desenvolver essas limitações.

5.4 A Máquina de Post

Na teoria da computabilidade, uma máquina de **Post**, assim denominada em honra a **Emil Leon Post**, é um autômato determinístico , baseado na estrutura de dados do tipo **fila** com um símbolo auxiliar. **Post** publicou este modelo computacional em 1943, como uma forma simples de sistema canônico de **Post**. Resumindo, é uma máquina de estados finita cuja fila (para entrada e saída) tem um tamanho ilimitado, tal que em cada transição a máquina lê o símbolo da cabeça da fila, remove um número fixo de símbolos da cabeça, e ao fim concatena uma palavra-símbolo pré-definida ao símbolo removido. O que parece ter sido a ideia original de como os discos rígidos atuais funcionam.

Uma máquina de **Post** é uma tripla $M = (\Sigma, D, \#)$, onde Σ é um alfabeto finito de símbolos, um dos quais é um símbolo especial de parada. Cadeias finitas (possivelmente vazias) em Σ são chamadas de palavras. D é um conjunto de regras de produção, atribuindo uma palavra $D(x)$ (chamada de regra de produção) para cada símbolo em Σ. A produção (diga-se D(H)) atribuída ao símbolo de parada é vista abaixo e não possui influência nas computações, mas por conveniência usa-se $D(H) = H$ (H de *halt*). E $\#$ é um inteiro positivo auxiliar, chamado de *número de deleção*.

5.4.1 Componentes elementares de um diagrama de fluxo

Os componentes elementares para representar uma máquina de Post podem ser colocados na construção de um fluxograma, considerando-se as construções gráficas elementares de um fluxograma tradicional, como: retângulos curvos, retângulos comuns, losângulos e setas ligando esas construçoes gráficas.

- (a) **Partida**: Existe somente uma instrução de início em um programa.
- (b) **Parada**: Existem duas alternativas de instruções de parada em um programa, uma de **aceitação** e outra de **rejeição**.
- (c) **Desvio** (ou leitura com teste): $\mathbf{X} \longleftarrow \mathbf{ler}(\mathbf{X})$
  - Denota o comando que lê o símbolo mais à esquerda da palavra armazenada em **X**, retirando o primeiro símbolo.
  - É uma instrução composta de uma leitura do símbolo à esquerda (início da fila), excluindo-o da fila e desviando o fluxo do programa de acordo com o símbolo lido;
  - Deve ser prevista a possibilidade de **X** conter a **palavra vazia**.
  - Se o cardinal de Σ é n, então existem $(n+2)$ arestas de desvios condicionais , pois se deve incluir as possibilidades $\#$ e ϵ.
  - **Atribuição X $\longleftarrow$ Xs**.

– É uma instrução de concatenação, gravando o símbolo indicado (pertencente a $\Sigma \cup \{\#\}$) à direita da palavra armazenada na variável **X** (fim da fila).

– A operação de **atribuição** é representada num retângulo de fluxograma contendo $X \leftarrow Xs$, supondo que $s \in \Sigma \cup \{\#\}$.

5.5 Turing-completude de Máquinas de Post

Para cada $\# > 1$, o conjunto de máquinas de Post-# é Turing completude; isto é, para cada $\# > 1$, e deste modo para qualquer **máquina de Turing** T existe uma **máquina de Post**-# que a simula T. Em particular, uma máquina de Post-2 pode ser construída para simular a máquina de Turing Universal, como foi feito por Wang (1963) e por Cocke & Minsky (1964).

Contrariamente, uma máquina de Turing pode ser mostrada ser universal, provando-se que ela pode simular uma classe de Turing-completa de *Máquinas de Post-#*. Por exemplo, **Rogozhin** (1996) provou a universalidade da classe das *máquinas de Post-2* com o alfabeto a1, ..., an, H e as correspondentes produções ananW1, ..., ananWn-1, anan, H, onde as Wk são palavras não vazias; ele então provou a universalidade de uma áquina de Turing muito pequena (4 estados, 6 símbolos) mostrando que ela pode simular a classe das máquinas de Post.

5.6 Benefícios de Post

O conceito de computador abstracto de Post é simples. As suas potencialidades são vastas: o computador de **Post** permite formular tudo aquilo que é passível de ser computado (vide [5, 6]). Podemos estabelecer uma analogia com o computador eletrônico, no qual, e não obstante a sua complexidade, a máquina de Post pode representá-lo. Os conceitos matemáticos de programação, algoritmo e computador universal, podem ser até introduzidos em cusos para alunos do ensino fundamental.

O próprio homem parece ser um exemplo de tal situação, em que a complexidade de sua matéria pode ser abstraída da simplicidade de um modelo de memória artificial. Sabe-se que o cérebro humano é composto por um número muito grande de neurônios , dez milhares de milh ees, ou mais, de acordo com algumas estimativas. E os estados "ativo" ou "inativo" de cada neurônio (0 ou 1, na linguagem do computador de **Post** mostrado neste capítulo, formam a base dos processos do pensamento humano e que é responsável pelo nosso comportamento.

Um simulador da máquina de **Post** pode ser encontrado em ⟨http://hdl.handle.net/10183/99567⟩.

5.7 Bibliografia e Fonte de Consulta

Máquina de Post - https://pt.wikipedia.org/wiki/Máquina_de_Post

D. F. M. Torres. Simulando o computador abstracto de Post - http://www.mat.ua.pt/-delfim/simula-post.htm.

V. Uspensky. A Máquina de Post, Mir, 1985.

D. Wood. Theory of Computation, Harper & Row Publishers, New York, 1987.

M. Davis, Emil L. Post : His life and work, in M. Davis (ed.), Solvability, provability, definability : the collected works of Emil L Post (Boston, MA, 1994), xi-xxviii.

Zohar Manna - Mathematical Theory of Computation. McGraw-Hill. 1974.

Biografia de Emil Post - ⟨http://www-history.mcs.st-and.ac.uk/Biographies/Post.html⟩

Máquina de Post (2000) - O Computador Matemático de Post, Delfim Fernando Marado Torres, Departamento de Matemática, Universidade de Aveiro, Portugal. Em ⟨http://arquivoescolar.org/bitstream/arquivo-e/84/1/Post.pdf⟩, acessado em 22 de agosto de 2015.

5.8 Referências - Leitura Recomendada

Wang, H.: Tag Systems and Lag Systems, Math. Annalen 152, 65-74, 1963.

Cocke, J., and Minsky, M.: Universality of Tag Systems with P=2, J. Assoc. Comput. Mach. 11, 15-20, 1964.

Zohar Manna - Mathematical Theory of Computation. McGraw-Hill. 1974.

M Davis, Emil L. Post : His life and work, in M Davis (ed.), Solvability, provability, definability : the collected works of Emil L Post (Boston, MA, 1994), xi-xxviii.

Rogozhin, Yu.: Small Universal Turing Machines, Theoret. Comput. Sci. 168, 215-240, 1996.

DIVERIO, Tiarajú Asmuz; MENEZES, Paulo Blauth. Teoria da Computação: Máquinas Universais e Computabilidade. 2ª ed. Porto Alegre: Sagra Luzzatto, 2000. 205 p.; p. 67; 103-105; ISBN 85-241-0593-3.

CAPÍTULO 6

Funções Recursivas Computáveis

O Desenvolvimento de funções recursivas vem desde suas origens no fim do século XIX, quando recursão foi primeiro usada como um método de definir funções aritméticas simples, até o meado dos anos 30 no século XX, quando a classe de funções recursivas gerais foi introduzida por **Herbrand-Gödel**, formalizada por **Kleene** e usada por **Church** no seu λ-Cálculo. *Máquinas de Turing* proporcionam uma abordagem para computabilidade: as operações elementares de uma máquina de Turing definem um procedimento efetivo para computar os valores de uma função. Agora, podemos pensar na computabilidade apresentada a partir do ponto de vista das funções. Ao contrário do que focalizar operações elementares, os objetos fundamentais são, agora, as próprias funções. Duas famílias de funções, as **funções recursivas primitivas** e as **funções μ-recursivas** são aqui mostradas. A computabilidade das funções recursivas primitivas e das funções μ-recursivas pode ser demonstrada através de um método efetivo para gerar os valores dessas funções. Neste sentido, os argumentos de uma função são referidos como entrada para a função, e a **avaliação de uma função** como uma **computação**.

6.1 Funções Recursivas Primitivas

Os anos 30 foram muito férteis de ideias matemáticas sobre a computação de funções recursivas (avaliação). **Herbrand**, **Gödel**, **Church**, **Kleene**, **Rosser**, pensavam na recursividade de funções. Enquanto a mais convincente definição de procedimento mecânico era dado por meio da ideia das máquinas de Turing, o conceito equivalente de funções recursivas primitivas apareceu historicamente, completando as extensões das definições recursivas simples de adição e multiplicação.

Definição (**Funções Recursivas Primitivas**)

Uma **função** é **recursiva primitiva** se ela pode ser construída a partir do **zero**, uma **função sucessor** e **funções-projeção** por um número finito de aplicações de composição e recursão primitiva.

Tabela 1 – Funções Aritméticas Recursivas Primitivas.

Descrição	**Função**	**Definição**
adição	$add(x+y)$ $x+y$	$add(x,0)=x$ $add(x,y+1)=add(x,y)+1$
multiplicação	$mult(x,y)$ $x.y$	$mult(x,0)=0$ $mult(x,y+1)=mult(x,y)+x$
predecessor	$pred(y)$	$pred(0)=0$ $pred(y+1)=y$
subtração	$sub(x,y)$ $x-y$	$sub(x,0)=x$ $sub(x,y+1)=pred(sub(x,y))$
exponenciação	$exp(x,y)$ x^y	$exp(x,0)=1$ $exp(x,y+1)=exp(x,y).x$

Fonte: Languages and Machines, Thomas A. Sudkamp, Cap. 13, p.304, 1988.

Uma **função projeção** P_i^k é definida para todos números naturais i,k, tal que $1 \leq i \leq k$. $P_i^k \triangleq f(x_1,x_2,...,x_k)=x_i$.

Algumas **Funções Recursivas Primitivas**: **Funções Aritméticas Recursivas Primitivas** , vistas na tabela 1 e as **Funções-Predicado Recursivas Primitivas** mostradas na tabela 2.
A operação fundamental da divisão, *div*, não é um função total. A função $div(x,y)$ retorna o quociente, o inteiro parte de x/y, quando o argumento divisor y é diferente de zero. A função é indefinida quando y é zero. Visto que todas as função recursivas primitivas são totais, segue que *div* não é uma função recursiva primitiva Sudkamp (1988).

Um **predicado** é uma função que exibe a **veracidade** ou a **falsidade** de uma proposição ou sentença, uma fórmula bem-formada da **Lógica dos Predicados**.

Um *predicado recursivo primitivo* é uma função recursiva primitiva que toma valores no conjunto $\{0,1\}$, os quais podem ser interpretados como **falso** ou **verdadeiro**.

Na tabela 2 seguinte, o predicado *sinal* especifica o sinal do argumento e a função *sg* indica se o argumento é positivo.

Predicados são funções recursivas primitivas mostrados na tabela 2.

O valor retornado por um predicado p designa se a entrada (argumento) satisfaz a propriedade representada por p. Por exemplo, se p é o predicado $p(n)$ definido como

Tabela 2 – Funções-Predicado Recursivas Primitivas.

Descrição	**Predicado**	**Definição**
sinal	sg(x)	sg(0) = 0 sg(y+1) = 1
sinal complement	cosg(x)	cosg(0) = 1 cosg(y+1) = 0
igual a	eq(x,y)	cosg(lt(x,y) + gt(x,y))
menor que	lt(x,y)	sg(y¿x)
maior que	gt(x,y)	sg(x¿y)
não igual	ne(x,y)	cosg(eq(x,y))

Fonte: Languages and Machines, Thomas A. Sudkamp, Cap. 13, p. 305, 1988.

$n < 3$, o valor retornado pode ser **verdade** se n for 0, 1 ou 2, mas para $n \geq 3$, o valor retornado será sempre **falso**.

Em termos da computabilidade dessas funções recursivas primitivas, existe um teorema que garante:

Teorema (**Computabilidade das funções recursivas primitivas**)

Toda função recursiva primitiva é Turing-computável.

Uma demonstração deste teorema pode ser vista em Sudkamp (1988), Cap. 13, p. 303.

6.2 Funções Mu-Recursivas

O enfoque de computabilidade formal de **Kleene** em 1936, sobre *descrições aritméticas* é baseado na generalização da noção de definição por indução . Uma classe de funções que inclui as operações de "+" e "." é fechada sob algumas regras simples, por definição por indução. Isto produz a classe de funções *μ-recursivas* que é um sistema bem aceito para se trabalhar matematicamente. Este sistema está desenvolvido em Carnielli (2009), nos capítulos 10, 13-15.

A notação μ:

Genericamente, para valores fixados $x_1, ..., X_n$, a notação $\mu\ z[p(x_1, ..., X_n, z)]$ é definida ser o número natural z, **mínimo**, tal que $p(x_1, ..., x_n, z)$ é verdadeiro. No

exemplo $p(n) = n < 3$, z corresponde ao valor $n = 0$ e podemos escrever $\mu\, z[p(x_1, z)] = \mu\, z[p(n, z)]$, onde $z = 0$.

A notação pode ser lida como o mínimo z satisfazendo $p(x_1, ..., X_n, z)$. Esta construção é chamada **minimalização** de p e $\mu\, z$ é chamado μ-operador.

Definição (Família de Funções μ-Recursivas)

A família de funções μ-recursivas é definida como segue:

1. As funções **sucessor**, **zero** e **projeção** são μ-recursivas.
2. Se h é uma função μ-recursiva de n variáveis e $g_1, ..., g_n$ são funções μ-recursivas de k variáveis, então $f = h \circ (g_1, ..., g_n)$ é μ-recursiva.
3. Se g e h são funções μ-recursivas de n e $n + 2$ variáveis, então a função f definida de g e h por recursão primitiva é μ-recursiva.
4. Se $p(x_1, ..., x_n, y)$ é um predicado μ-recursivo total, então $f = \mu\, z[p(x_1, ..., X_n, z)]$ é μ-recursiva.
5. Uma função é μ-recursiva, se e somente se pode ser obtida de (1) por um número finito de aplicações das regras em (2), (3) e (4).

As condições (1), (2) e (3) implicam que todas as funções recursivas primitivas são μ-recursivas.

Uma função é **Turing-computável** , se e somente se, existe uma máquina de Turing que computa ela, e estabelecendo uma relação entre μ-recursivo e **Turing-computável**, existe um teorema que diz:

Teorema (**Computabilidade das funções μ-recursivas**)

Toda função μ-recursiva é **Turing-computável**.

O leitor pode encontrar a prova deste teorema em Sudkamp (1988), Cap.13, p. 323.

Em lógica matemática e na ciência computacional, as funções μ-recursivas constituem uma *classe de funções parciais* de números naturais para números naturais que são *computáveis*.

De fato, na teoria da computabilidade é mostrado que as funções μ-recursivas são precisamente as que podem ser computadas por máquinas de **Turing**. As funções μ-recursivas são intimamente relacionadas às **funções recursivas primitivas** e sua definição indutiva se baseia nestas funções recursivas primitivas. No entanto, nem toda função μ-recursiva é uma função primitiva recursiva. Um exemplo muito

conhecido é a função de **Ackermann**.

Na teoria da computabilidade, a **Função de Ackermann** , de nome dado por **Wilhelm Ackermann** é um dos exemplos mais simples, de uma *função computável* que não é uma *função recursiva primitiva*. Todas as *funções recursivas primitivas* são *totais* e computáveis, mas a **Função de Ackermann** mostra que *nem toda função total-computável é recursiva primitiva*.

Depois que **Ackermann** publicou sua função, que continha três inteiros positivos como argumentos, vários autores a modificaram para atender a várias finalidades. Então, a função de **Ackermann** atual pode ser referenciada como uma de suas variantes da função original. Uma das versões mais comuns, a **função de Ackermann-Péter** (com dois argumentos), é definida a seguir para os inteiros não negativos m e n:

$$A(m,n) = \begin{cases} n+1, & \text{se} \quad m = 0 \\ A(m-1,1), & \text{se} \quad m > 0, n = 0 \\ A(m-1, A(m, n-1)), & \text{se} \quad m, n > 0 \end{cases}$$

A função é caracterizada, seu valor cresce rapidamente, até mesmo para pequenas entradas. Por exemplo, $A(4,2)$ resulta em um inteiro com 19729 dígitos.

Agora, podemos afirmar, sem prova, o seguinte teorema que compara a taxa de crescimento da função de Arckermann, com aquela das funções recursivas primitivas.

Teorema (**Comparando taxa de crescimento de funções**)

Para toda função recursiva primitiva de uma variável, f, existe algum $i \in \mathbb{N}$, tal que $f(i) < A(i,i)$, onde A é a função de Arckermann.

O leitor pode encontrar mais sobre a função de Arckermann em Sudkamp (1988), Cap. 13, p. 320.

6.3 Funções Parciais Computáveis

As funções recursivas primitivas foram definidas como uma família de funções computáveis intuitivamente. Nós temos estabelecido que todas as funções recursivas primitivas são totais. Mas, contrariamente, será que todas as funções totais computáveis são recursivas primitivas? Além disso, devemos restringir nossa análise de computabilidade a funções totais? Agora, podemos nesta seção apresentarmos argumentos para uma resposta negativa a ambas as questões.

Teorema (**Subconjunto de funções numéricas teóricas**)

O conjunto de *funções recursivas primitivas* é um subconjunto próprio do con-

junto de *funções numéricas teóricas* totais computáveis.

Uma função numérica teórica é uma função da forma $\mathbb{N} \times \mathbb{N} \times \mathbb{N} \ldots \times \mathbb{N} \rightarrow \mathbb{N}$, onde o domínio consiste de n-tuplas de números naturais ou de números naturais, somente. A demonstração, o leitor encontrará em Sudkamp (1988), Cap. 13, p. 319.

6.4 A Tese de Church-Turing revisada

A tese de **Church-Turing** como já mencionada no capítulo 4, afirmou que todo problema de decisão efetivamente resolvível (decidível), admite uma solução por máquina de Turing. Subsequentemente, pode-se projetar máquinas de Turing para computar *funções numéricas teóricas*. Nesta sua forma funcional, a tese de **Church-Turing** associa a computação efetiva de funções com a computabilidade de **Turing**. Assim, a Tese de Church-Turing pode ser enunciada:

Tese de Church-Turing: Uma função parcial é computável se, e somente se, ela é μ-recursiva.

O leitor encontrará mais informação deste resultado em Sudkamp (1988), Cap. 13, seção 13.8, p. 329.

6.5 Contribuições

A Teoria da Computabilidade foi responsável pela definição formal de **computação** e **computabilidade**, e pela prova da existência de problemas insolúveis computacionalmente .

Mais tarde, também foi possível a construção e formalização do conceito de linguagem de computador , sobretudo *linguagem de programação* , uma ferramenta para a expressão precisa da lógica computacional de programação, suficiente para ser representada em um nível de abstração voltada ao usuário do computador.

Para outros campos científicos e para a sociedade de forma geral, o surgimento da Ciência da Computação forneceu suporte para a Revolução Digital, dando origem a Era da Informação com **Claude E. Shannon**. Ver no capítulo 8.

Como exemplo, importante, de uso da Ciência da Computação, a **computação científica** é uma área da computação que permite o avanço de estudos em grandes projetos para a humanidade, como por exemplo, o mapeamento do *genoma* humano, objetivo do *Projeto Genoma Humano*.

6.6 Bibliografia e Fonte de Consulta

Languages and Machines: an Introdution to the Theory of Computer Science - Thomas A. Sudkamp. 1988.

Michael Sipser - Introdução à Teoria da Computação. 2011.

6.7 Referências - Leitura Recomendada

C. Rota, (Eds.), A History of Computing in the Twentieth Century, Academic Press, Nova York, 1980.

LAVINGTON, Simon H. - Early British Computers: The Story of Vintage Computers and The People Who Built Them. Manchester University Press, 1980.

Rod Adams - An Early History of Recursive Functions and Computability, 2011.

CAPÍTULO 7

O Legado de von Neumann

Contemporâneo e aluno de **Hilbert** , **John von Neumann** (1903-1957) foi um matemático Húngaro de origem Judaica, naturalizado Estadunidense. Com apenas 22 anos de idade (1925), publicou 5 artigos que influiram enormente o mundo acadêmico. Três deles criticando a física quântica , um outro sobre a Teoria dos Jogos e o quinto sobre Ciência da Computação .

John von Neumann auxiliou **Hilbert** em Göttingen (Alemanha) durante 1926-1927. Nesta época **von Neumann** tinha alcançado o *status* de celebridade. **Von Neumann** realizou seu doutorado em matemática na Universidade de Budapeste, também em 1926, com uma tese sobre teoria dos conjuntos . Ele publicou uma definição de números ordinais e a definição é a usada hoje. Por seus vinte e poucos anos, a fama de von **Neumann** tinha se espalhado em todo o mundo na comunidade matemática . Em conferências acadêmicas, ele iria encontrar-se apontado como um jovem gênio.

7.1 von Neumann e os alicerces teóricos da computação

Dos cinco (5) artigos de **von Neumann** que influenciaram enormente o mundo acadêmico, o último teve grande repercussão para a Ciência da Computação. O trabalho abordou o relacionamento dos sistemas formais lógicos e os limites da matemática .**John von Neumann** demonstrou a necessidade de se provar a *consistência* da matemática (tudo o que é provado pela matemática seria verdadeiro). Este era um passo importante e problemático, tendo em vista o estabelecimento dos alicerces teóricos para **Ciência da Computação**. Uma visão que ninguém tinha, na época.

7.2 John von Neumann na Lógica

No início do século XX, a teoria dos conjuntos ainda não tinha sido formalizada e estava em crise devido ao paradoxo de **Bertrand Russell**, e a axiomatização da matemática , sobre o modelo dos *Elementos de Euclides* , estava a atingir novos níveis de

Figura 45 – John von Neumann em 1928.

Fonte: Arquivo pessoal de Nicholas A. Vonneuman. Ver http://www.ime.usp.br/ yoshi/opus.html.

rigor, particularmente na aritmética e na geometria . **Ernst Zermelo** e **Abraham Fraenkel** resolveram parcialmente este problema, formulando princípios que permitiam a construção de todos os conjuntos usados na matemática, mas não excluíam a possibilidade de existirem conjuntos que pertencessem a eles mesmos. Na sua tese de doutoramento, apresentada em 1925, **von Neumann** demonstrou como era possível excluir esta possibilidade de duas maneiras complementares: a *noção de classe* e o axioma da fundação (um dos axiomas da teoria dos conjuntos de **Zermelo-Frankel**).

Uma aproximação ao problema foi efetuado por mieo do uso da noção de classe: define-se como conjunto uma classe que pertence a outras classes, enquanto uma classe própria é uma classe que não pertence a nenhuma outra classe. De acordo com os axiomas da teoria de **Zermelo-Frankel**, não é possível a construção de um conjunto que contenha todos os conjuntos que não pertencem a si mesmos. Pelo contrário, usando a noção de *classe*, a classe de todos os conjuntos que não pertencem a si mesmos pode ser construída, não sendo, no entanto, um conjunto, mas sim uma classe própria. Por exemplo, o substantivo "substantivo" pertence à classe dos substantivos, por contraste ao substantivo "adjetivo", que não pertence a esta classe.

A outra aproximação ao problema é conseguida pelo *axioma da fundação* , que diz que todo o conjunto pode ser construído a partir da base, numa sucessão ordenada de passos, de tal modo que se um conjunto pertence a outro, então o primeiro tem necessariamente de vir antes do segundo na sucessão (o que exclui a possibilidade de um conjunto pertencer a si mesmo). Para demonstrar que este axioma não

estava em contradição com os outros, **von Neumann** criou um novo método de demonstração que se tornou numa ferramenta fundamental na teoria dos conjuntos, o método dos modelos interiores .

Desta maneira, o sistema axiomático da teoria dos conjuntos tornou-se completamente satisfatório, e a pergunta que pairava era se esta axiomática era ou não definitiva, e se estava ou não sujeita a melhoria. A resposta a esta questão surgiu em Setembro de 1930, no Congresso de Matemática de Köningsberg, no qual **Gödel** anunciou o seu primeiro teorema da incompletude (os sistemas axiomáticos usuais são incompletos, uma vez que não podem provar todas as verdades que sejam expressas na sua linguagem). Menos de um mês depois, **von Neumann** informou **Gödel** de uma consequência do seu teorema: os sistemas axiomáticos usuais são incapazes de demonstrar a sua própria consistência. Contudo, **Gödel** já o tinha concluído de modo independente, pelo que este resultado é o chamado segundo teorema de Gödel, sem referência a **von Neumann**.

Von Neumann tinha uma grande admiração por **Gödel**, e era frequente elogiá-lo de maneira entusiástica:

"*O feito de* ***Kurt Gödel*** *na lógica moderna é singular e monumental na verdade é mais do que um monumento, é um marco que se manterá visível longe no espaço e no tempo. ... O tema da lógica tem certamente mudado por completo a sua natureza e possibilidades com o feito de Gödel.*".

E quando lhe perguntaram porque não se referia ao trabalho de **Frank Plumpton Ramsey** (1903-1930), um matemático britânico que fez importantes contribuições para a matemática, filosofia e economia, que seria conhecido para alguém que se interessasse pelo campo da lógica, respondeu que depois de **Gödel** ter publicado os seus artigos sobre a indecidibilidade e incompletude da lógica, não tinha lido mais nenhum artigo sobre lógica simbólica . Noutra altura, numa entrevista intitulada "The Mathematician", disse, a respeito do trabalho de **Gödel**:

"*Isto aconteceu durante a nossa vida, e eu sei como os meus valores sobre a verdade matemática absoluta mudaram, de maneira humilhantemente fácil, durante este acontecimento, e como eles mudaram três vezes em sucessão!*"

Devido a este fascínio por **Gödel** e pelo seu trabalho, afigura-se bastante natural que tenha escolhido a lógica para tese de doutoramento. Tal como **Ulam** diz:

"*No seu trabalho de juventude, estava preocupado não só com a lógica matemática e a axiomatização da teoria dos conjuntos, mas, simultaneamente, com a substância da teoria dos conjuntos, obtendo resultados na teoria da medida e na teoria das variáveis reais*".

7.3 John von Neumann na Física

John von Neumann tinha uma capacidade de explicar ideias complexas na Física. Em 1929, **Osvald Veblen** convidou **von Neumann** para Princeton (USA) para uma palestra sobre *quantum theory*. Entre 1930 e 1933, **von Neumann** lecionou em Princeton, mas este não era um dos seus pontos fortes. Mesmo assim, ele se tornou um dos seis professores de matemática, juntamente com (**J. W. Alexander**, **A. Einstein**, **M. Morse**, **O. Veblen**, **John von Neumann** e **H. Weyl**), em 1933, no recém-fundado Instituto de Estudos Avançados de Princeton, uma posição que manteve durante o resto da vida dele.

Em 1933, **von Neumann** tornou-se co-editor da revista *Annals of Mathematics* (⟨annals.math.princeton.edu/⟩) e, dois anos mais tarde, tornou-se co-editor de *Compositio Mathematica* (⟨www.compositio.nl/compositio.html⟩). Ele manteve, ambas, as editorias até sua morte.

Em 1935, **von Neumann** convidou **Stanislaw Marcin Ulam** (1909-1984), um matemático Polonês para visitar o Instituto de Estudos Avançados de Princeton por alguns meses. **Ulan** foi quem resolveu o problema de iniciar a fusão da bomba de hidrogênio, resume o trabalho de **von Neumann**:

Em sua obra, ele estava preocupado não só com a lógica matemática e a axiomática da teoria dos conjuntos , mas, simultaneamente, com a própria substância da teoria dos conjuntos, obtendo resultados interessantes na teoria da medida e a teoria das variáveis reais . Foi nesse período também que ele começou seu trabalho clássico sobre teoria quântica , a base matemática da teoria da medição e a nova mecânica estatística . (Mecânica estatística modela um sistema em termos do comportamento médio do grande número de átomos e moléculas que constituem o sistema.)

Seu texto *Mathematische Grundlagen der Quantenmechanik* (1932) construiu uma estrutura sólida para a nova mecânica quântica. **Van Hove** escreveu:

*A mecânica quântica foi muito afortunado certamente atrair, nos primeiros anos após a sua descoberta, em 1925, o interesse de um gênio matemático do quilate de **von Neumann**. Como resultado, a estrutura matemática da teoria foi desenvolvida e os aspectos das suas formas inteiramente novas regras de interpretação foram analisados por um único homem em dois anos (1927-1929).*

Álgebras de operadores lineares limitados em um espaço de **Hilbert**, fechado na topologia operador fraco, foram introduzidos por **von Neumann** em 1929, em um artigo em *Mathematische Annalen*. Seu interesse pela teoria ergódica , as representações do grupo e mecânica quântica contribuiu significativamente para a realização de **von Neumann** que uma teoria de álgebras de operadores foi a próxima etapa importante no desenvolvimento desta área da matemática .

Tais álgebras de operadores foram chamados "anéis de operadores" por **von Neumann**. **J. Dixmier**, em 1957, chamou-lhes álgebras de **von Neumann** em sua monografia sobre Álgebras de Operadores no espaço de Hilbert (álgebras de von Neumann).

Na segunda metade da década de 1930 e início dos anos 1940, **von Neumann**, trabalhando com seu colaborador **F. J. Murray**, lançou as bases para o estudo das álgebras de **von Neumann** em uma série de artigos. Em 1938, a *American Mathematical Society* agraciou-o com o *Bôcher Prize* a **John von Neumann**.

Em meados dos anos 30, **von Neumamm** era fascinado pelo problema da turbulência hidrodinâmica . Foi então que ele se tornou ciente dos mistérios subjacentes, objeto das equações diferenciais parciais não-lineares. Seu trabalho, desde os primórdios da Segunda Guerra Mundial, diz respeito a um estudo das equações da hidrodinâmica e da teoria dos choques . Os fenômenos descritos por estas equações não-lineares são desconcertantes analiticamente e desafiam a percepção ainda qualitativa pelos métodos atuais. Esse trabalho matemático parecia-lhe a maneira mais promissora para obter uma sensação para o comportamento de tais sistemas. Isso o levou a estudar novas possibilidades de computação em máquinas eletrônicas .

No entanto, **von Neumann** participou de grande variedade de diferentes estudos científicos. **John von Neumann**, gradualmente, expandiu seu trabalho na teoria dos jogos , e com **Oskar Morgenstern** (co-autor), ele escreveu o texto clássico *Teoria de Jogos e Comportamento Econômico* (1944).

Durante e após a Segunda Guerra Mundial, **von Neumann** trabalhou como consultor para as forças armadas. Suas contribuições valiosas incluiu uma proposta do método de implosão para a interposição de combustível nuclear para explosão e sua participação no desenvolvimento da bomba de hidrogênio . A partir de 1940 ele era um membro do Comitê Científico Consultivo dos *Ballistic Research Laboratories* em *Aberdeen Proving Ground*, em Maryland (USA). Ele era um membro da *Navy Bureau of Ordnance* (1941-1955), e um consultor do *Los Alamos Scientific Laboratory* de 1943 a 1955. De 1950 a 1955, ele era um membro do projeto das armas especiais das forças armadas em Washington.

7.4 Honras a von Neumann

Uma variedade de honras foram atribuídas a **von Neumann**. O cientista foi eleito para diversas academias científicas, incluindo a Academia Nacional de Ciências Exactas (Lima, Peru), Academia Nazionale dei Lincei (Rome, Italy), American Academy of Arts and Sciences (USA), American Philosophical Society (USA), Instituto Lombardo di Scienze e Lettere (Milan, Italy), National Academy of Sciences (USA) and Royal Netherlands Academy of Sciences and Letters (Amsterdam, The Netherlands). **John von Neumann** foi um cientista que trabalhou em várias áreas da matemática,

tanto pura como aplicada. O leitor pode conhecer mais sobre **von Neumann** em ⟨http://www-history.mcs.st-and.ac.uk/Biographies/Von_Neumann.html⟩.

John von Neumann foi um dos pioneiros da Ciência da Computação que fazem contribuições significativas para o desenvolvimento do projeto lógico do computador digital .

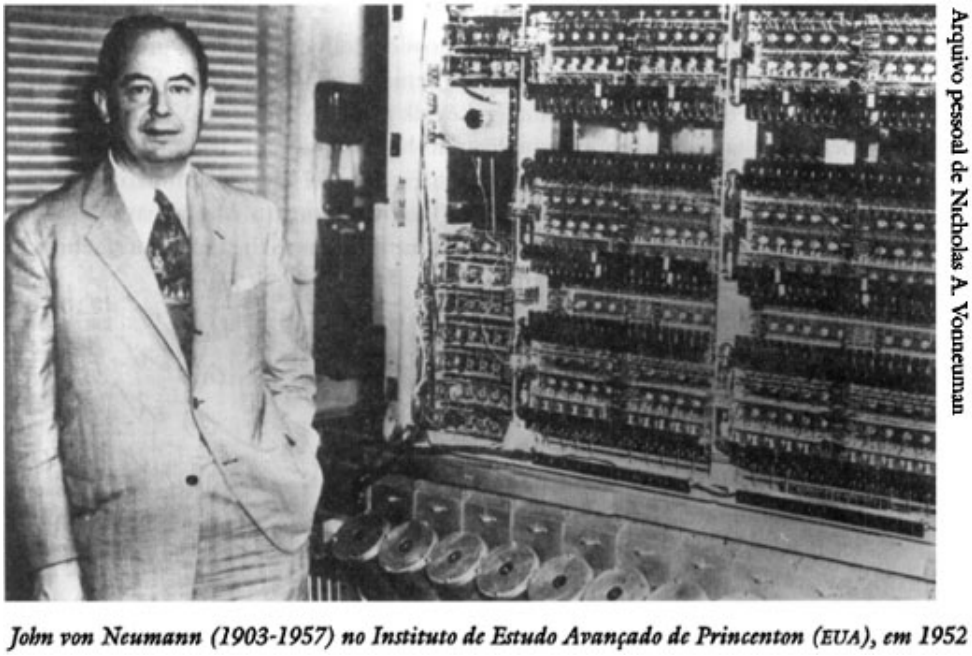


Figura 46 – John von Neumann - no Advanced Research Institute em Princeton (USA) em 1952.

Fonte: Arquivo pessoal de Nicholas A. Vonneuman. Ver http://www.ime.usp.br/ yoshi/opus.html.

Claude E. Shannon, seu contemporâneo, escreveu sobre ele:

Von Neumann passou uma parte considerável dos últimos anos de sua vida trabalhando a Teoria dos Autômatos. Esta representou para ele uma síntese de seu interesse na lógica, na teoria da prova e seu trabalho mais tarde, durante a Segunda Guerra Mundial, e depois, em computadores eletrônicos de grande porte. Envolvendo uma mistura de matemática pura e aplicada, bem como outras ciências. A teoria de autômatos era um campo ideal para o intelecto de ***von Neumann****, que trouxe para ele outras linhas de investigação.*

7.5 A Teoria dos Autômatos

Na Ciência da Computação teórica, **Teoria dos Autômatos** é o estudo de máquinas abstratas ou autômatos, e os problemas computacionais que podem ser resolvidos usando essas máquinas. Autômato vem do grego, que significa uma ação sem influência externa, ou seja, automático.

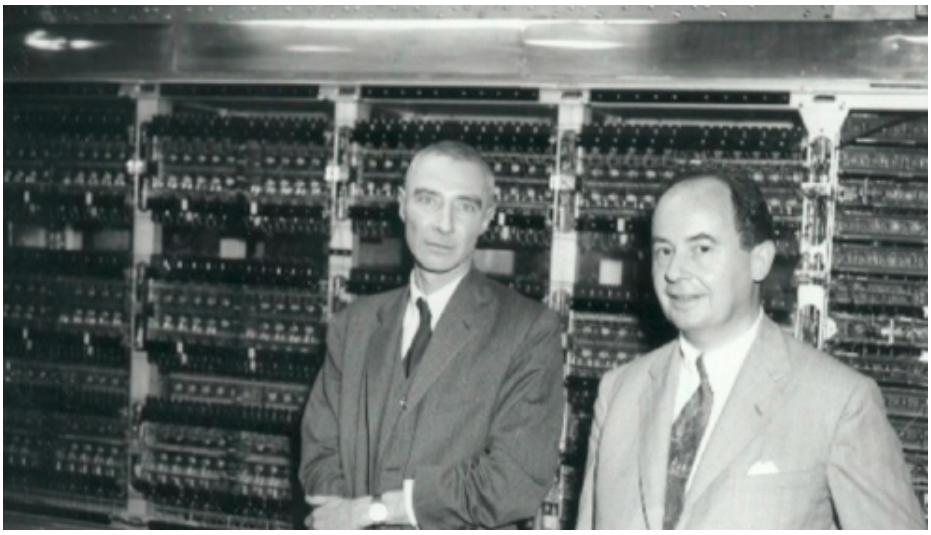


Figura 47 – Jonh Von Neumann e Openheimer no Projeto Manhattan.

Fonte: http://www.jornaldoempreendedor.com.br/destaques/conheca-o-verdadeiro-dr-fantastico-john-von-neumann/.

Um autômato é uma máquina de estados finitos , consistindo de estados (representados graficamente por círculos), e transições (representado a mudança de estados). Quando o autômato recebe um símbolo de entrada, ele faz uma transição (ou salto) para outro estado, de acordo com sua função de transição (que tem como entradas o estado atual e o símbolo recente).

Autômatos desempenham um papel importante em teoria da computação , e em muitos métodos formais para especificação de sistemas de computação , funcionam como um *modelo-base*, o qual é descrito de alguma forma, caraterizando o método formal, na linguagem em que esse for descrito. Ele desenvolveu a teoria de autômatos e defendeu a ideia do *bit* (royalties para Claude Shannon, contemporâneo de von Naumann), como uma medida da memória do computador.

As questões a seguir são relacionadas à *teoria dos autômatos*:

- Qual classe de linguagens formais é reconhecível por algum tipo de autômato? (linguagens reconhecíveis).
- Quão expressivo é um tipo de autômato em termos de reconhecer classe de linguagens formais? E, quanto ao poder relativo da expressividade de uma linguagem?

A *teoria dos autômatos* também estuda se existe algum procedimento efetivo ou não, para resolver problemas semelhantes à seguinte lista:

- Um autômato aceita alguma palavra de entrada?

- É possível transformar um dado autômato não-determinístico em um autômato determinístico sem mudar a linguagem reconhecível?

- Para uma dada linguagem formal, qual é o menor autômato que a reconhece?

A teoria dos autômatos também está profundamente relacionada à teoria das linguagens formais . Um autômato é uma representação finita de uma linguagem formal que pode ser um conjunto infinito . Autômatos são frequentemente classificados pela classe das linguagens formais que são capazes de reconhecer.

O escopo de trabalho de **John von Neumamm** foi marcado por várias manifestações de interesses diversificados e multidisciplinares, sempre ultrapassando as fronteiras da matemática, e levando-as a territórios desconhecidos. O seu escopo inclui várias áreas da matemática pura , matemática aplicada, física , meteorologia , economia e **computação** .

Como, por exemplo, no caso do computador IAS 1952, **von Neumann** introduziu o sistema binário , os programas armazenados na memória e criou os fluxogramas , isto é, fazendo a separação do projeto da lógica , do projeto da engenharia , descrito por ele, anteriormente, em um relatório sobre projeto e construção do EDVAC (1944-1951), computador com programa armazenado na memória, resultante principalmente da colaboração de **John von Neumann** e outros colaboradores, e incorporada pela primeira vez no IAS 1952 (1946-1952), projeto e construção do computador do Instituto de Estudos Avançados (IAS) de Princeton por **John von Neumann**.

Para uma grande parte dos praticantes da Computação, o nome de **von Neumann** está geralmente associado à idéia de arquitetura de **von Neumann**, ou seja, à estrutura, hoje considerada clássica, de computadores digitais com programa armazenado na própria memória . **John von Neumann** teve contribuições importantes nas áreas de arquitetura de computadores , princípios de programação , análise de algoritmos , análise numérica , computação científica , teoria dos autômatos , redes neurais , tolerância a falhas , sendo o verdadeiro fundador de algumas delas. A obra e o legado de **John von Neumann** pode ser encontrada, também em ⟨http://www.ime.usp.br/~yoshi/opus.html⟩.

Se alguém mudou o seu mundo enquanto vivo, **von Neumann** é o candidato ao número 1. Ele está na base do pensamento moderno, na Física, na Matemática, na Lógica, na Economia, na Teoria dos Jogos. Virtualmente todos os computadores hoje, de super-computadores de milhões de dólares, até pequenos *chips* para celulares e brinquedos, todos tem uma coisa em comum: eles todos são "máquinas de Von Neumann" , com variações de uma arquitetura básica de computação que **John Von neumann**, construiu sobre o trabalho de **Alan Turing** feito nos anos 40. Mas durante a vida deste matemático Húngaro que teve um dedo em tudo, desde a

física quântica até nas políticas dos EUA durante a guerra fria, a máquina de **Von Neumann** foi apenas uma de suas menores realizações.

7.6 Bibliografia e Fonte de Consulta

A Obra e o Legado de John von Neumann - Instituto de Matemática e Estatística, USP - http://www.ime.usp.br/ yoshi/opus.html

John von Neumann - http://www-history.mcs.st-and.ac.uk/Biographies/Von_Neuman.html

Teoria dos Autômatos - https://pt.wikipedia.org/wiki/Teoria_dos_automatos

Tomasz Kowaltowski - A OBRA E O LEGADO DE JOHN VON NEUMANN, Von Neumann: suas contribuições à Computação - http://www.scielo.br/pdf/ea/-v10n26/v10n26a22.pdf

John von Neumann - https://pt.wikipedia.org/wiki/John_von_Neumann

O Mundo Fantástico de von Neumann - http://www.jornaldoempreendedor.com.br/destaques/conheca-o-verdadeiro-dr-fantastico-john-von-neumann/

7.7 Referências - Leitura Recomendada

Por Nicholas A. von Neumann - O Legado Filosófico de John von Neumann, Instituto de Matemática e Estatística, Instituto de Estudos Avançados da USP, http://www.-ime.usp.br/ yoshi/opus.html

Biografia de John von Neumann - http://www-history.mcs.st-and.ac.uk/Biographies/-Von_Neumann.html

Imre Simon, Von Neumann, o Cientista e a Figura Humana.

Ruy Exel, Von Neumann e a Teoria de Álgebras de Operadores.

Chaim Samuel Hönig, O Legado Científico de John von Neumann: Teoria da Medida e outras Contribuições.

Tomasz Kowaltowski, John von Neumann: suas Contribuições à Computação.

Antonio Divino Moura, John von Neumann e a Previsão Numérica de Tempo e Clima.

Walter F. Wreszinsk, Algumas Contribuições de John von Neumann à Física Matemática.

CAPÍTULO 8

Shannon - Da Álgebra de Boole à Matemática da Comunicação

Depois de **Alan Turing** e **Alonzo Church**, surge no cenário da história da Ciência da Computação e da teoria da informação , **Claude Elwood Shannon** (1916-2001). **Claude Shannon** revolucionou o mundo através do desenvolvimento de teoria da informação, o que abriu caminho para a comunicação digital . Neste capítulo é abordado o legado de **Shanonn**, um engenheiro eletrônico, matemático e criptógrafo, que nos deu os princípios do computador digital que temos hoje, e é conhecido como o criador da teoria matemática da comunicação (BIOGRAPHICAL MEMOIRS).

8.1 O que é informação

A resposta à pergunta "*O que é informação?* não é única Capurro (2003). Existem muitas repostas possíveis. Além disso, associadas à questão "O que é informação?" existem outras questões, tais como: Qual é o significado do conteúdo informacional?; O que é ciência da informação?; Para que serve a informação?; Para que serve a ciência da informação? Capurro (1991). Assim, deve-se ficar atento para que a discussão do conceito de informação , juntamente com a identificação da necessidade de interpretação da informação - ou o conteúdo informacional, não leve a uma confusão entre o que é informação, qual é o significado da informação e qual é o papel da ciência da informação Matheus (2005).

Pense em **símbolos**. O leitor verá que podemos juntar símbolos do alfabeto de uma linguagem , e formar **cadeias de símbolos** da linguagem. Uma cadeia de símbolos é um conceito síntático. Com cadeias de símbolos podemos formar **dados**. Um dado é uma seqüência de símbolos , é algo totalmente sintático. Pense no significado dos dados, e então encontramos, na semântica, o significado das coisas. O leitor verá que dados não envolve semântica. Os **dados** são percebidos através dos sentidos e acabam por gerar a informação necessária para produzir o conhecimento . Existe uma diferença entre dados e informação. **Informação** é um conceito primitivo. A

informação é um fenômeno que confere significado ou sentido às coisas, já que através de códigos e de conjuntos de dados , forma os modelos do pensamento humano . A informação é um conjunto organizado de dados, que constitui uma mensagem sobre algum fato, um determinado fenômeno ou certo evento. A informação permite resolver problemas e tomar decisões, tendo em conta que o seu uso racional é a base do conhecimento . Para os estudiosos da ciência da informação, o mais natural é que uma mensagem seja eivada de significados.

8.2 A Ciência e a Teoria da Informação

Uma definição clássica da **Ciência da Informação** diz que essa ciência tem como objeto a produção, seleção, organização, interpretação, armazenamento, recuperação, disseminação, transformação e uso da informação (AMERICAN DOCUMENTATION, 1968) (CAPURRO, 2003).

Entretanto, é bastante comum encontrar, na área da **Ciência da Informação**, a indicação da importância da **Teoria Matemática da Comunicação** de **Shannon** e **Weaver**, apresentada em 1948 e publicada em 1949, a qual mencionamos a seguir, como um prenúncio inauguradora do campo. Essa teoria é normalmente conhecida como **Teoria da Informação** e tal denominação é que essa teoria, pela primeira vez enunciou um conceito científico de **informação** Araújo (2009).

A **Teoria da Informação** ou **Teoria Matemática da Comunicação** é um ramo da teoria da probabilidade e da matemática estatística que lida com sistemas de comunicação , transmissão de dados , criptografia , codificação , teoria do ruído , correção de erros e compressão de dados .

Antes de **Shannon**, a **Teoria da Informação** foi o resultado de trabalhos que começaram em 1910, com as pesquisas do matemático russo **Andrei A. Markov** sobre a teoria das cadeias de símbolos na literatura, prosseguiram com as hipóteses do norte-americano **Ralph V.L. Hartley** em 1927, que propõe a primeira medida precisa de informação associada à emissão de símbolos, o ancestral do bit (*digit binary*) e da linguagem de numeração binária (MATTELART, 1999).

A teoria de **Shannon** também foi precedida pelas pesquisas teóricas de **Harry Theodor Nyquist** (1889-1976), como mencionamos no que segue. Este foi um teórico da informação, Sueco, que se estableceu Estadunidense em 1907. Ele freqüentou a Universidade de Dakota do Norte, Grand Forks, 1912-1915 e recebeu o B.S. e M.S. graus em Engenharia Elétrica em 1914 e 1915, respectivamente. Ele frequentou a Universidade de Yale, em New Haven, 1915-1917, e foi premiado com o grau de Ph.D. em 1917.

Entre de 1917-1934, **Nyquist** foi contratado pela *American Telephone and Telegraph Company* no *Department of Development and Research Transmission*, onde

ele estava preocupado com os estudos sobre telégrafo imagem e transmissão de voz . De 1934 a 1954, ele esteve com a *Bell Telephone Laboratories, Inc.*, onde ele continuou no trabalho de engenharia de comunicações , especialmente em engenharia de transporte e engenharia de sistemas . No momento da sua aposentadoria da Bell Telephone Laboratories em 1954, **Nyquist** foi diretor adjunto de estudos de sistemas.

Figura 48 – Nyquist - Pesquisa teóricas sobre a Teoria da Informação.

Fonte: e.wikipedia.org.

Nyquist foi contemporâneo do matemático britânico **Alan Turing**, que, na época concebia, a partir de 1936, o esquema de uma máquina capaz de tratar essa informação.

Durante seus 37 anos de serviço na *Bell Systems*, ele recebeu 138 patentes nos EUA e publicou doze artigos técnicos. Seu trabalho variou de ruído térmico à transmissão do sinal . O teorema de amostragem de **Nyquist** postula que a taxa de amostragem deve ser, pelo menos, duas vezes a frequência mais elevada na amostra, a fim de reconstruir o sinal.

Sua explicação matemática do ruído térmico também manteve seu nome intimamente ligado com o fenômeno. Seu trabalho lançou as bases para a teoria da informação moderna e transmissão de dados , a invenção do sistema de transmissão de bandas de frequência, agora amplamente utilizado na radiodifusão televisiva , e o diagrama de **Nyquist** , bem conhecido para determinar a estabilidade dos *feedback systems*.

Após sua aposentadoria, **Nyquist** foi contratado como engenheiro consultor em tempo parcial em matéria de comunicação no **Department of Defense** dos Estados Unidos, na *Stavid Engineering Inc.*, e na *WL Maxson Corporation*.

Antes de sua morte em 1976, **Nyquist** recebeu muitas honras por seu excelente trabalho em comunicações . Ele foi a quarta pessoa a receber a medalha da *National Academy of Engineer's Founder*, em reconhecimento a suas muitas contribuições fundamentais para a engenharia . Em 1960, ele recebeu a medalha do *IRE Medal of Honor*, pela suas contribuições fundamentais para um conhecimento quantitativo de ruído térmico , transmissão de dados e feedback negativo. **Nyquist** também foi premiado com o *Stuart Ballantine*, medalha do *Franklin Institute* em 1960, e o prêmio *Mervin J. Kelly* em 1961.

No que segue estão resumidos os mais importantes resultados de **Nyquist**.

- Em um canal livre de ruídos, a única limitação imposta à taxa de transmissão de dados será devida à largura de banda do canal.
- A formulação para esta limitação é devida à **Nyquist** e estabelece que, dada uma largura de banda B, a maior taxa de sinal que poderá ser suportada por esta largura de banda será 2B.
- Em um sinal binário, a taxa de dados que pode ser suportada por B Hz será 2B bps.
- Um canal de voz de BW igual a 3100 Hz está sendo utilizado via MODEM para transmitir dados digitais. A capacidade do canal será, então, igual a 2B = 6200 bps.
- O Teorema de Nyquist é de extrema importância no desenvolvimento de codificadores de sinais analógicos para digitais porque estabelece o critério adequado para amostragem dos sinais.
- Nyquist provou que, se um sinal arbitrário é transmitido através de um canal de largura de banda B Hz, o sinal resultante da filtragem poderá ser completamente reconstruído pelo receptor através da amostragem do sinal transmitido, a uma freqüência igual a, no mínimo 2B vezes por segundo.
- Esta freqüência, denominada Freqüência de Nyquist , é a freqüência de amostragem requerida para a reconstrução adequada do sinal.

8.3 Vannevar Bush

Vannevar Bush (1890-1974) foi um engenheiro Estadunidense, um cientista notável que, entre outras, ficou conhecido pela ideia do Memex, sistema visto com um conceito pioneiro, o primeiro sistema de consulta de informações, a empregar o conceito de hipertexto, precursor da World Wide Web.

Os primeiros computadores foram máquinas analógicas. Eram lentas e constituídas de dispositivos mecânicos de alta complexidade, mas funcionavam. Uma destas pioneiras foi o "Analisador Diferencial", concebido e montado no final dos anos vinte do século passado por **Vannevar Bush**.

O Analisador Diferencial era uma máquina capaz de resolver equações diferenciais de segunda ordem. Na verdade, um dos primeiros computadores da era moderna, uma máquina analógica fabricada usando componentes mecânicos e elétricos baseados em relés.

Vannevar Bush, Figura 49, concebeu e fabricou sua máquina no MIT, o Instituto de Tecnologia de Massachusetts, onde **Shannon**, formado em engenharia pela Universidade de Michigan, iniciou seu mestrado em engenharia em 1936. **Vannevar Bush** foi o orientador de **Shannon**, e portanto, natural que o interesse de ambos estivesse em torno do Analisador Diferencial. Mas, um detalhe em particular chamou a atenção de **Shannon**: justamente a utilização de relés na fabricação da máquina.

Figura 49 – Vannevar Bush - 1940-1944.

Fonte: en.wikipedia.org.

Um relé é um interruptor acionado eletricamente. Sua concepção é muito simples: dois contatos elétricos, um dos quais contém um imã, estão separados por uma mola. Quando se aplica uma tensão elétrica a um terceiro contato, uma corrente elétrica atravessa uma bobina que induz um campo magnético que por sua vez atrai o imã e fecha os dois primeiros contatos permitindo que uma corrente elétrica flua entre eles.

Isto faz-nos lembrar alguma coisa relativa aos *transistores*. Pois, exceto pelo tamanho e uso do magnetismo para seu acionamento, um relé atua exatamente da mesma forma que um transistor quando usado como chaveador de corrente, aplicando-se uma tensão ao terminal denominado *Base*, que permite que uma corrente elétrica flua entre seus terminais *Emissor* e *Coletor* de um *transistor*.

O transístor de silício e germânio foi inventado nos Laboratórios da *Bell Telephone* por **John Bardeen** e **Walter Houser Brattain** em 1947 e, inicialmente, demonstrado em 23 de Dezembro de 1948 por **John Bardeen**. **Walter Houser Brattain** e **William Bradford Shockley** foram laureados com o Nobel de Física em 1956. Eles pretendiam fabricar um transistor de efeito de campo (FET) idealizado por **Julius Edgar Lilienfeld** antes de 1925, mas acabaram por descobrir uma amplificação da corrente no ponto de contato do transistor. Isto evoluiu posteriormente para converter-se no transistor de junção bipolar (BJT). O objetivo do projeto era criar um dispositivo compacto e barato para substituir as **válvulas termoiônicas** usadas nos sistemas telefônicos da época.

A grande vantagem dos **transistores** em relação às **válvulas** foi demonstrada

em 1958, quando **Jack Kilby**, da Texas Instruments, desenvolveu o primeiro circuito integrado, consistindo de um transistor, três resistores e um capacitor, implementando um oscilador simples. A partir daí, via-se a possibilidade de criação de circuitos mais complexos, utilizando integração de componentes, como visualizado na Figura 50.

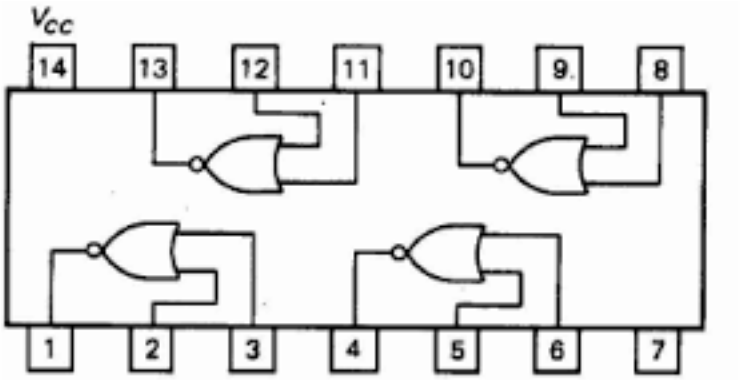


Figura 50 – Um desenho de um circuito integrado.

Fonte: Google, Imagens de Circuitos Integrados.

Isto marcou uma transição na história dos transistores, que deixaram de ser vistos como substitutos das válvulas e passaram a ser encarados como dispositivos que possibilitam a criação de circuitos complexos, integrados.

8.4 A contribuição de Shannon

Shannon nasceu em Petoskey, Michigan. Os primeiros 16 anos de **Shannon** foram em Gaylord, Michigan, onde ele frequentou o ensino público, graduando-se no Gaylord High School em 1932. **Shannon** mostrou uma inclinação para coisas mecânicas, seus melhores talentos eram para a e matemática. Em 1932 **Shannon** começou a cursar a Universidade de Michigan, formando-se em 1936 em duas graduações de bacharelado em engenharia elétrica e matemática. Posteriormente, começou seus estudos de pós-graduação no Instituto de Tecnologia de Massachusetts (MIT), onde trabalhou com o analisador diferencial de **Vannevar Bush**.

Naquela época, o sistema de numeração decimal estava na mente dos engenheiros. **Shannon**, além de provar a possibilidade de se construir um computador totalmente eletrônico, foi o primeiro a entender que os respectivos circuitos ficavam muito mais simples (e mais baratos) com o abandono do sistema decimal em favor do sistema binário.

A razão era simples: diferentemente do sistema numérico decimal, o sistema numérico binário, exprime qualquer número usando apenas dois algarismos, o zero e o um. Ora, um relé assume apenas dois estados: fechado (quando permite que uma corrente

elétrica flua entre seus dois terminais) e aberto (quando não deixa passar corrente). Então é possível correlacionar um destes estados (em geral o primeiro) com o algarismo "um" e o outro com o algarismo "zero". E com isto, usando uma quantidade suficiente de relés, exprimir qualquer número. Com isto, é muito mais fácil conceber computadores que usem internamente o sistema numérico de base dois.

Shannon foi o primeiro a identificar este conceito, e ainda percebeu ainda que todas as operações internas de processamento de dados poderiam ser efetuadas usando combinações de relés que poderiam não apenas tomar decisões baseadas na comparação de valores como também executar as operações matemáticas elementares que, por sua vez, poderiam ser combinadas para executar operações mais complexas. Em outras palavras: os dois estados dos relés poderiam ser usados não apenas para representarem os algarismos "um" e "zero" do sistema numérico de base dois como também os valores "Verdadeiro" e "Falso" da lógica digital. Portanto, se com relés interligados se poderia representar números, efetuar operações matemáticas com estes números e tomar decisões baseadas nos resultados destas operações, seria possível conceber um computador cujos componentes ativos fossem exclusivamente relés: um computador digital.

Figura 51 – Shannon: o precursor da Teoria da Comunicação de Dados.

Fonte: https://pt.wikipedia.org/wiki/Claude$_S$*hannon*.

O que faltava para **Shannon** era uma ferramenta teórica para formular e analisar os circuitos baseados nestes componentes, circuitos que obedeciam à lógica digital. Esta ferramenta existia e já estava pronta há quase um século, por **George Boole** , a espera de alguém a descobrisse e desse a ela uma utilização prática.

Ao estudar os complexos circuitos do analisador diferencial, **Shannon** observou que os conceitos de **George Boole**, inventor da álgebra booleana, poderia ser útil para várias coisas. Um documento elaborado a partir da sua tese de mestrado em 1937, *A Symbolic Analysis of Relay and Switching Circuits* SHANONN (1937), foi publicado na edição de 1938 da *Transactions of the American Institute of Electrical*

Tabela 3 – Equivalência entre Álgebra Booleana e as Portas Lógicas.

Álgebra Booleana	**Portas Lógicas**
$A \wedge B$	porta A *e* B
$A \vee B$	porta A *ou* B
$\neg A$	porta *não*
$\neg(A \wedge B)$	não (A *ou* B)
$\neg(A \vee B)$	não (A *e* B)

Fonte: Google, Imagens de Portas Lógicas.

Tabela 4 – Portas Lógica E em valores Binários.

A	B	A e B	Lâmpada
1	0	0	apagada (F)
1	1	1	acesa (V)
0	0	0	apagada (F)
0	1	0	apagada (F)

Fonte: Google, Imagens de Portas Lógicas.

Tabela 5 – Portas Lógica OU em valores Binários.

A	B	A e B	Lâmpada
1	0	1	acesa (V)
1	1	1	acesa (V)
0	0	0	apagada (F)
0	1	1	acesa (V)

Fonte: Google Images - Portas Lógicas.

Engineers, onde fixava o formalismo lógico e os circuitos elétricos, como tabelas apresentadas.

Neste trabalho, **Shannon** provou que a álgebra booleana e a aritmética binária poderiam ser utilizadas para simplificar o arranjo dos e então utilizados em comutadores para roteamento em redes telefônicas. Expandindo o conceito, ele também mostrou que deveria ser possível a utilização de arranjos de relés para resolver problemas de álgebra booleana. A exploração dessa propriedade de interruptores elétricos criou a lógica e os conceitos mais básico dos computadores digitais.

O trabalho de **Shannon** tornou-se o principal na área de circuitos digitais quando se

Tabela 6 – Portas Lógica $N\tilde{A}O$ em valores Binários

A	B	Lâmpada
1	0	apagada (F)
0	1	acesa (V)

Fonte: Google Images - Portas Lógicas.

BLOCOS LÓGICOS BÁSICOS

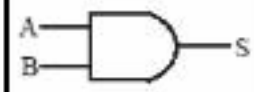
PORTA	Símbolo Usual	Tabela da Verdade	Função Lógica	Expressão
E **AND**		A B S 0 0 0 0 1 0 1 0 0 1 1 1	**Função E:** Assume 1 quando todas as variáveis forem 1 e 0 nos outros casos.	S=A.B
OU **OR**		A B S 0 0 0 0 1 1 1 0 1 1 1 1	**Função E:** Assume 0 quando todas as variáveis forem 0 e 1 nos outros casos.	S=A+B
NÃO **NOT**		A S 0 1 1 0	**Função NÃO:** Inverte a variável aplicada à sua entrada.	$S=\overline{A}$
NE **NAND**		A B S 0 0 1 0 1 1 1 0 1 1 1 0	**Função NE:** Inverso da função E.	$S=\overline{(A.B)}$
NOU **NOR**		A B S 0 0 1 0 1 0 1 0 0 1 1 0	**Função NOU:** Inverso da função OU.	$S=\overline{(A+B)}$
OU Exclusivo		A B S 0 0 0 0 1 1 1 0 1 1 1 0	**Função OU Exclusivo:** Assume 1 quando as variáveis assumirem valores diferentes entre si.	S=A⊕B $S=\overline{A}.B+A.\overline{B}$
Coincidência		A B S 0 0 1 0 1 0 1 0 0 1 1 1	**Função Coincidência:** Assume 1 quando houver coincidência entre os valores das variáveis.	S=A⊙B $S=\overline{A}.\overline{B}+A.B$

Figura 52 – Tabela das Portas Lógicas Básicas.

Fonte: Google Images - Tabela das Portas Lógicas Básicas.

tornou amplamente conhecido entre a comunidade de engenharia elétrica durante e após a segunda guerra mundial. O trabalho teórico rigoroso de **Shannon** substituiu completamente os métodos *ad hoc* que haviam prevalecido anteriormente.

Figura 53 – Shannon e a matemática no Bell Labs em 1955.

Fonte: www.corp.att.com.

Claude Shannon havia divisado a possibilidade de conceber um computador digital baseado inteiramente no uso de chaveadores de corrente (relés, no caso) não apenas para representar números no sistema binário, ou de base dois, como também para efetuar operações matemáticas e lógicas com estes números. O que lhe faltava era uma ferramenta teórica para conceber e analisar os circuitos montados com os relés que constituiriam o computador.

Foi então que ocorreu a **Shannon** recorrer ao trabalho de **George Boole** *An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities* publicado quase um século antes.

Este trabalho que estabeleceu as bases de uma nova álgebra, a Álgebra Booleana, que quando foi publicado não se tinha a menor ideia de sua possível utilidade nem se descortinara qualquer aplicação prática na engenharia. **Shannon** não apenas percebeu que aquela era a ferramenta de que carecia como também devotou seu mestrado a demonstrar que ela podia ser usada para interpretar, conceber e analisar os circuitos de relés que, por sua vez, combinados, poderiam materializar as expressões da Álgebra Booleana. E, usando as propriedades binárias dos relés (ou de qualquer outro chaveador elétrico de corrente, como os transístores inventados dez anos mais tarde) para executar funções lógicas, estabeleceu os conceitos básicos que norteiam o projeto de todos os computadores digitais.

Por "ensinar" um rato elétrico para encontrar o seu caminho através de um labirinto, **Shannon** ajudou a estimular os pesquisadores no Bell Labs, a pensar em

novas maneiras de usar os poderes lógicos de computadores para outras operações que não fossem o cálculo numérico (Figura 54).

Figura 54 – Shannon e o experimento do rato elétrico num labirinto.

Fonte: tecnologia.terra.com.br / www.kidscodecs.com.

8.5 História da Palavra BIT

A palavra *bit* foi utilizada pela primeira vez na década de 30, surpreendentemente, para designar partes de informação (bits of information, em inglês). Porém, a definição de *bit* como ficou muito conhecida até hoje foi empregada em 1948 pelo engenheiro **Claude Shannon**. Naquele ano, **Shannon** elaborou o artigo *A Mathematical Theory of Communication* e usou a palavra para designar o dígito binário.

Um *bit* - abreviação de Binary Digit (digito binário) - é exatamente isso: uma combinação de dois dígitos que se junta com outros dígitos do mesmo tipo para construir a informação completa. Bits também são utilizados para a classificação de cores de uma imagem. Por exemplo: uma imagem monocromática tem 1 *bit* em cada ponto, enquanto uma imagem de 8 bits suporta até 256 cores.

Bit é a e a menor unidade de informação de um computador. Um *bit* tem somente um valor (que pode ser 0 ou 1). Vários *bits* combinam entre si e dão origem a outras unidades, como *bytes*, *megabytes*, *gigabytes* e *terabytes* atuais.

Todas as informações processadas por um computador são medidas e codificadas em *bits*. Tamanhos de arquivos são medidos em *bits*, taxas de transferência são medidas em *bit*, informações na linguagem do usuário são convertidas em *bits* para serem preocessados por computador.

Certamente você já ouviu falar em sistemas de 32 *bits* ou 64 *bits*. Este número

indica a capacidade que o computador tem de processar a quantidade de *bits* indicada de uma só vez. Também pode significar o número de *bits* utilizados para representar dados num barramento de endereços de memória.

Foi **Claude Shannon** quem cunhou o termo *bit* para se referir à menor quantidade de informação que pode ser transmitida.

8.6 Shannon e a Teoria Matemática da Comunicação

Segundo **Shannon**, o objetivo da comunicação seria reproduzir num ponto, de forma exata, uma mensagem selecionada em outro ponto. Porém, toda transmissão de informação poderia chegar acarretada de interrupções e ruídos. **Shannon** e **Warren Weaver** apresentaram um modelo de comunicação conforme a Figura 55.

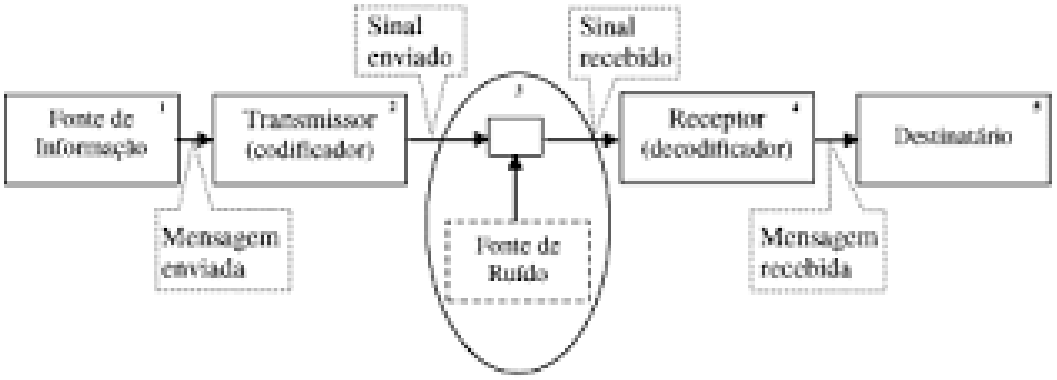


Figura 55 – Modelo de Comunicação Shannon.

Fonte: SHANNON (1948).

Mas sua maior contribuição para a ciência das telecomunicações foi seu trabalho *A Mathematical Theory of Communication*, publicado em 1948, e considerado a maior contribuição do século para a teoria das telecomunicações. Neste trabalho, **Shannon** estabeleceu o conceito de "quantidade de comunicação" e demonstrou que a capacidade de transmissão de informações de um canal de comunicação, seja ele de que tipo for (desde os fios metálicos dos velhos telégrafos até as modernas fibras óticas) é limitada por fatores que nada têm a ver com a natureza do canal mas simplesmente com o logaritmo da relação sinal/ruído somada à unidade, multiplicado pela frequência da transmissão, de acordo com a relação mostrada na Figura 56. Esse teorema é conhecido como "Fórmula de Shannon" (onde C_{max} é a taxa máxima de transmissão do canal em bits/s, B é a frequência da transmissão em *hertz*, e S e N são, respectivamente, as potências do sinal e do ruído em *watts*).

Os aspectos matemáticos da informação foram, evidentemente, mais amplamente abraçados pela Ciência da Computação do que pela Ciência da Informação. O acesso à Internet via linha telefônica, através de uma conexão usando *modem*, é dado pela Fórmula de **Shannon** que explica o limite máximo que se dispõe em uma conexão

$$\mathbf{C_{max} = B \,.\, \log_2(1 + \frac{S}{N})}$$

Figura 56 – A Fórmula de Shannon - a quantidade de informação num canal.

Fonte: www.techtudo.com.br.

em alta taxa (banda larga).

Seus trabalhos sobre criptografia são considerados de suma importância e foram suas pesquisas conduzidas no Bell Labs que permitiram o uso de sistemas comutados usando relés para conexão telefônica, que até então eram feitas manualmente por telefonistas.

Shannon desenvolveu e publicou diversos programas para jogar xadrez e em 1980 montou um computador especialmente para este fim, que ganhou o Campeonato Internacional de Xadrez para computadores daquele ano.

Warren Weaver (1894-1978) foi um matemático Estadunidense, co-autor do livro *The Mathematical Theory of Communication*, publicado em 1949 juntamente com **Claude Shannon**. Por sua forma acessível também a não especialistas, este livro popularizou os conceitos de um artigo científico de **Shannon** publicado no ano anterior, intitulado *A Mathematical Theory of Communication.*

Figura 57 – Warren Weaver - *The Mathematical Theory of Communication.*

Fonte: blog.comshalom.org.

Vamos colocar as coisas em sua devida perspectiva. A tese de **Shannon** foi publicada em 1938, quando ele tinha apenas 22 anos e uma década antes da invenção dos transistores. Por sua vez, o trabalho de **Boole** jazia semiesquecido por quase um século nas prateleiras das bibliotecas universitárias, sem que ninguém tivesse

encontrado qualquer utilidade prática para ele.

A possibilidade de usar chaveadores de corrente (no caso, relés), não apenas para representar números no sistema binário, como também para materializar circuitos que emulassem as equações da lógica digital jamais tinha sido divisada.

Segundo **Robert Gallager**, seu colega no MIT, "Até hoje (1938), ninguém tinha sequer se aproximado dessa ideia. Foi um avanço que, sem ele, teria demorado muito a ser conseguido. Juntar estas peças esparsas do conhecimento humano e, com elas, gerar uma teoria que possibilitou o desenvolvimento dos computadores digitais e revolucionou a sociedade a partir de então foi uma façanha de gênio".

Consta que **Victor Shestakov**, da Universidade de Moscou, propôs uma teoria de comutadores elétricos baseada na álgebra booleana em 1935, dois anos antes de **Shannon** iniciar sua tese de mestrado, mas sua primeira publicação sobre o tema data de 1941. Três anos depois da publicação da tese de **Shannon**.

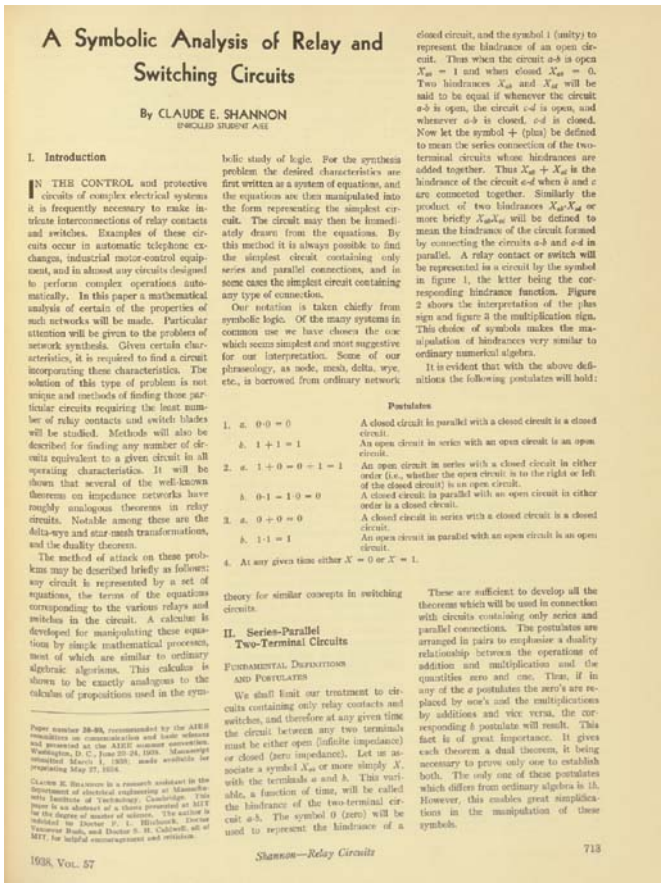
Não é à toa que a tese de **Shannon** *A Symbolic Analysis of Relay and Switching Circuits* foi considerada por **Howard Gardner**, da Universidade de Harvard, como, possivelmente a mais importante tese de mestrado do século XX. Tanto assim que mereceu, dois anos após sua publicação, o destacado *Alfred Noble American Institute of American Engineers Award*.

A Figura 58 mostra uma foto da primeira página da tese de **Shannon** na edição 57 do *Transactions of the American Institute of Electrical Engineers* de 1938.

O progresso científico e tecnológico ocorre assim, aos saltos, promovido por mentes geniais, cada uma deles se baseando no trabalho dos que os antecederam. Mas se prestarmos atenção, perceberemos que o laço entre Boole e Shannon é quase mágico. Os dois trabalhos se encaixam tão perfeitamente que parece que Boole criou sua álgebra tendo em mente sua utilização por Shannon quase um século mais tarde. E que este desenvolveu sua tese para dar uma utilização prática à Álgebra de Boole. Mas o fato é que, embora tenha tido a percepção de sua importância ao afirmar que gostaria de ser conhecido pela posteridade devido a ela, quando desenvolveu sua Álgebra Boole sequer poderia sonhar na utilização prática que **Shannon** deu a ela. E quando **Shannon** começou suas pesquisas sobre o uso de relés sequer lembrava da existência de uma ferramenta desenvolvida há quase um século como se o tivesse sido especificamente para basear suas conclusões.

8.7 Shannon e a Teoria Matemática da Comunicação

Shannon começou a desenvolver uma descrição da informação transmitida num canal de comunicação entre duas partes, dando continuidade a um ramo de estudos conhecido como Teoria da Informação, iniciada em 1910.



A Symbolic Analysis of Relay and Switching Circuits

By CLAUDE E. SHANNON
ENROLLED STUDENT AIEE

I. Introduction

IN THE CONTROL and protective circuits of complex electrical systems it is frequently necessary to make intricate interconnections of relay contacts and switches. Examples of these circuits occur in automatic telephone exchanges, industrial motor-control equipment, and in almost any circuits designed to perform complex operations automatically. In this paper a mathematical analysis of certain of the properties of such networks will be made. Particular attention will be given to the problem of network synthesis. Given certain characteristics, it is required to find a circuit incorporating these characteristics. The solution of this type of problem is not unique and methods of finding those particular circuits requiring the least number of relay contacts and switch blades will be studied. Methods will also be described for finding any number of circuits equivalent to a given circuit in all operating characteristics. It will be shown that several of the well-known theorems on impedance networks have roughly analogous theorems in relay circuits. Notable among these are the delta-wye and star-mesh transformations, and the duality theorem.

The method of attack on these problems may be described briefly as follows: any circuit is represented by a set of equations, the terms of the equations corresponding to the various relays and switches in the circuit. A calculus is developed for manipulating these equations by simple mathematical processes, most of which are similar to ordinary algebraic algorisms. This calculus is shown to be exactly analogous to the calculus of propositions used in the symbolic study of logic. For the synthesis problem the desired characteristics are first written as a system of equations, and the equations are then manipulated into the form representing the simplest circuit. The circuit may then be immediately drawn from the equations. By this method it is always possible to find the simplest circuit containing only series and parallel connections, and in some cases the simplest circuit containing any type of connection.

Our notation is taken chiefly from symbolic logic. Of the many systems in common use we have chosen the one which seems simplest and most suggestive for our interpretation. Some of our phraseology, as node, mesh, delta, wye, etc., is borrowed from ordinary network theory for similar concepts in switching circuits.

II. Series-Parallel Two-Terminal Circuits

Fundamental Definitions and Postulates

We shall limit our treatment to circuits containing only relay contacts and switches, and therefore at any given time the circuit between any two terminals must be either open (infinite impedance) or closed (zero impedance). Let us associate a symbol X_{ab} or more simply X, with the terminals a and b. This variable, a function of time, will be called the hindrance of the two-terminal circuit a-b. The symbol 0 (zero) will be used to represent the hindrance of a closed circuit, and the symbol 1 (unity) to represent the hindrance of an open circuit. Thus when the circuit a-b is open $X_{ab} = 1$ and when closed $X_{ab} = 0$. Two hindrances X_{ab} and X_{cd} will be said to be equal if whenever the circuit a-b is open, the circuit c-d is open, and whenever a-b is closed, c-d is closed. Now let the symbol + (plus) be defined to mean the series connection of the two-terminal circuits whose hindrances are added together. Thus $X_{ab} + X_{cd}$ is the hindrance of the circuit a-d when b and c are connected together. Similarly the product of two hindrances $X_{ab} \cdot X_{cd}$ or more briefly $X_{ab}X_{cd}$ will be defined to mean the hindrance of the circuit formed by connecting the circuits a-b and c-d in parallel. A relay contact or switch will be represented in a circuit by the symbol in figure 1, the letter being the corresponding hindrance function. Figure 2 shows the interpretation of the plus sign and figure 3 the multiplication sign. This choice of symbols makes the manipulation of hindrances very similar to ordinary numerical algebra.

It is evident that with the above definitions the following postulates will hold:

Postulates

1. a. $0 \cdot 0 = 0$ — A closed circuit in parallel with a closed circuit is a closed circuit.
   b. $1 + 1 = 1$ — An open circuit in series with an open circuit is an open circuit.
2. a. $1 + 0 = 0 + 1 = 1$ — An open circuit in series with a closed circuit in either order (i.e., whether the open circuit is to the right or left of the closed circuit) is an open circuit.
   b. $0 \cdot 1 = 1 \cdot 0 = 0$ — A closed circuit in parallel with an open circuit in either order is a closed circuit.
3. a. $0 + 0 = 0$ — A closed circuit in series with a closed circuit is a closed circuit.
   b. $1 \cdot 1 = 1$ — An open circuit in parallel with an open circuit is an open circuit.
4. At any given time either $X = 0$ or $X = 1$.

These are sufficient to develop all the theorems which will be used in connection with circuits containing only series and parallel connections. The postulates are arranged in pairs to emphasize a duality relationship between the operations of addition and multiplication and the quantities zero and one. Thus, if in any of the a postulates the zero's are replaced by one's and the multiplications by additions and vice versa, the corresponding b postulate will result. This fact is of great importance. It gives each theorem a dual theorem, it being necessary to prove only one to establish both. The only one of these postulates which differs from ordinary algebra is 1b. However, this enables great simplifications in the manipulation of these symbols.

Paper number 38-80, recommended by the AIEE committees on communication and basic sciences and presented at the AIEE summer convention, Washington, D. C., June 23-27, 1938. Manuscript submitted March 1, 1938; made available for preprinting May 27, 1938.

Claude E. Shannon is a research assistant in the Department of electrical engineering at Massachusetts Institute of Technology, Cambridge. This paper is an abstract of a thesis presented at MIT for the degree of master of science. The author is indebted to Doctor F. L. Hitchcock, Doctor Vannevar Bush, and Doctor S. H. Caldwell, all of MIT, for helpful encouragement and criticism.

Figura 58 – Claude E. Shannon - A tese de mestrado de Shannon.

Fonte: www.techtudo.com.br.

Em 1940, **Shannon** se tornou pesquisador do Instituto nacional de Estudos Avançados em Princeton, Nova Jersey. Em Princeton, **Shannon** teve a oportunidade de discutir suas idéias com cientistas e matemáticos influentes como **Hermann Weyl** e **John von Neumann**, além de um encontro ocasional com **Albert Einstein**. **Shannon** trabalhou livremente em todas as áreas, e começou a moldar as idéias que se tornariam a teoria da informação GUIZZO (2003).

Em 1948, publicou o importante artigo científico intitulado **A Mathematical Theory of Communication** enfocando o problema de qual é a melhor forma para codificar a informação que um emissor queira transmitir para um receptor. Neste artigo, trabalhando inclusive com as ferramentas teóricas utilizadas por **Norbert Wiener**, **Claude Shannon** propôs com sucesso uma medida de informação, própria para medir incerteza sobre espaços desordenados (mais tarde complementada por **Ronald Fisher**, que criou uma medida alternativa de informação apropriada para medir incerteza sobre espaços ordenados). **Shannon** é famoso por ter fundado a **teoria da informação**.

Em 1949, em co-autoria com o também matemático Estadunidense **Warren Weaver** (1894-1978), publicou o livro **The Mathematical Theory of Communication** (Teoria Matemática da Comunicação) SHANON-WARREN (1949) contendo reimpressões do seu artigo científico de 1948 de forma acessível também a não-especialistas - isto popularizou seus conceitos.

Shannon também contribuiu para o campo da criptoanálise durante a segunda guerra mundial. Outro trabalho notável, publicado em 1949, é a *Communication Theory of Secrecy Systems*, uma versão do seu trabalho em tempo de guerra sobre a teoria matemática de criptografia, no qual ele provou que todas as cifras teoricamente inquebráveis deve ter os mesmos requisitos que a cifragem *One-Time Pad* - uma cifra criada do ponto de vista da criptografia perfeita KAHN (1996).

Deu ainda importantes contribuições na área da Inteligência Artificial.Uma contribuição fundamental da teoria da informação, para o processamento de linguagem natural e lingüística computacional, foi ainda estabelecida em 1951, em seu artigo *Previsão e Entropia de Impresso Inglês*, mostrando limites superior e inferior da entropia - uma medida de aleatoriedade - nas estatísticas da língua inglesa - proporcionando uma base estatística para análise da linguagem.

8.8 Shannon e a Cibernética

A palavra **cibernética** deriva de um termo grego que significa **piloto**. **Cibernética** é o estudo dos autocontroles encontrados em sistemas estáveis, sejam eles mecânicos, elétricos ou biológicos.

Entre 1946 e 1953, **Claude Shannon** também contribuiu para a consolidação da *teoria cibernética* junto com outros cientistas renomados no grupo reunido sob o nome de **Macy Conferences**, contribuindo para a consolidação da teoria cibernética junto com outros cientistas renomados, como **Norberto Wiener** e **John von Neumann** , entre outros.

Norberto Wiener (1894-1964) (Figura 59) foi um matemático Estadunidense, conhecido como o fundador da cibernética. A contribuição de **Wiener** para a Ciência da Computação veio mais tarde.

Foi **Wiener** quem visualizou que a informação, como uma quantidade, era tão importante quanto a energia ou a matéria. O fio de cobre, por exemplo, pode ser estudado pela energia que ele é capaz de transmitir, ou pela informação que pode comunicar. A revolução trazida pelo computador é em parte baseada nessa idéia. A contribuição de **Wiener** não foi uma simples peça de hardware, mas a criação de um ambiente intelectual em que **computadores** e **autômatos** (criação de **John von Neumann**) pudessem ser desenvolvidos. **Wiener** percebeu que para os computadores serem desenvolvidos, teriam que se assemelhar à habilidade dos seres humanos

no controle de suas próprias atividades.

A **Shannon**, também é creditado a introdução da *teoria da amostragem*, que se preocupa com o que representa um sinal de tempo contínuo a partir de um conjunto (uniforme) discreto de amostras. Essa teoria foi essencial para permitir a passagem das telecomunicações dos sistemas analógicos para as comunicações dos sistemas digitais no ano de 1960 e posteriores.

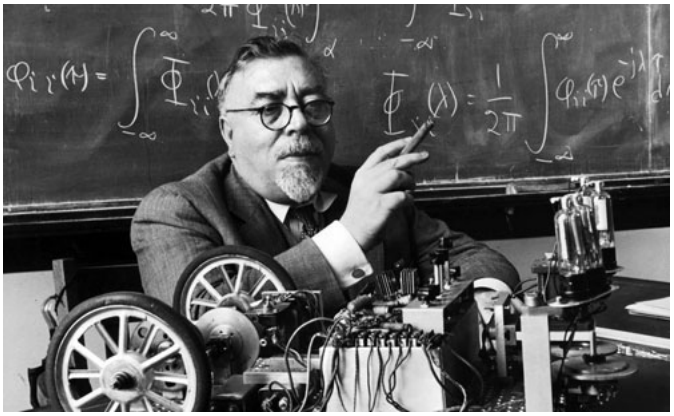


Figura 59 – Norbert Wiener - o criador da Cibernética.

Fonte: history-computer.com.

A perspectiva introduzida por **Shannon**, com a *teoria da comunicação* é que toda a revolução digital começou com **Claude Shanonn**. Ele formou a base da revolução digital, que ocorreu nas décadas seguintes, visto que, cada dispositivo que contém um microprocessador ou microcontrolador é um descendente conceitual das publicações de **Shannon**.

8.9 Bibliografia e Fonte de Consulta

Informação - (http://conceito.de/informacao)

Harry Nyquist - http://ethw.org/Harry_Nyquist

Harry Nyquist Papers - http://webapp.und.edu/dept/library/Collections/og1176.html

Resultados de Nyquist - http://www.feng.pucrs.br/d̃ecastro/TPI/TPI_Cap3_parte2.pdf

Armand e Michèle Mattelart. História das Teorias da Comunicação. Título original: Histories des Théories de la Communication. 1995,1997. Éditions la Découverte et Syros. Paris. ISBN 2-7071-2469-9.

Biographical Memoirs (Shanonn) - doi:10.1098/rsbm.2009.0015, http://rsbm.royalsocietypublishing.org/content/roybiogmem/55/257, acessado em 05 de Julho de 2015.

1916: Remembering Claude Shannon - http://www.corp.att.com/attlabs/reputation/-timeline/16shannon.html

Shannon, Claude Elwood (1916-2001) - http://scienceworld.wolfram.com/biography/-Shannon.html, acessado em 05 de Julho de 2015.

Robert B. Ash. Information Theory. New York: Interscience, 1965. ISBN 0-470-03445-9. New York: Dover 1990. ISBN 0-486-66521-6, p. v

Yeung, R. W. (2008). The Science of Information. Information Theory and Network Coding. pp. 1-01. doi:10.1007/978-0-387-79234-7_1. ISBN 978-0-387-79233-0.

Shannon, Claude E. (July 1948). A Mathematical Theory of Communication. Bell System Technical Journal 27 (3): 379-423. doi:10.1002/j.1538-7305.1948.tb01338.x.

Shannon, Claude E. (October 1948). A Mathematical Theory of Communication. Bell System Technical Journal 27 (4): 623-656. doi:10.1002/j.1538-7305.1948.tb00917.x.

Claude E. Shannon, Warren Weaver. The Mathematical Theory of Communication. Univ of Illinois Press, 1949. ISBN 0-252-72548-4

Warren Weaver - https://pt.wikipedia.org/wiki/Warren_Weaver

8.10 Referências - Leitura Recomendada

Griffith, B. C. Ed. (1980): Key papers in information science. New York: Knowledge Industry Publ.

CLAUDE ELWOOD SHANNON, Collected Papers, Edited by N.J.A Sloane and Aaron D. Wyner, IEEE press, ISBN 0-7803-0434-9

Shannon, Claude E.; Weaver, Warren. The Mathematical Theory of Communication (em inglês). Illinois: Illini Books, 1949. 117 p. Library of Congress Catalog Card nº 49-11922.

Robert Price (1982). Claude E. Shannon, an oral history IEEE Global History Network IEEE, Em http://www.ieeeghn.org/wiki/index.php/Oral-History:Claude_E.-_Shannon , Visitado em 05 de Julho de 2015.

Claude Shannon - A Symbolic Analysis of Relay and Switching Circuits. Massachusetts Institute of Technology, August 10, 1937.

Shannon, Claude (1949). Communication Theory of Secrecy Systems. Bell Sys-

tem Technical Journal 28 (4): 656-715.

Erico Marui Guizzo, *The Essential Message: Claude Shannon and the Making of Information Theory*, (M.S. Thesis, Massachusetts Institute of Technology, Dept. of Humanities, Program in Writing and Humanistic Studies, 2003), 14.

Poundstone, William (2005): Fortune's Formula : The Untold Story of the Scientific Betting System That Beat the Casinos and Wall Street - http://www.amazon.com/gp/-reader/0809046377

David A. Mindell, *Between Human and Machine: Feedback, Control, and Computing Before Cybernetics*, (Baltimore: Johns Hopkins University Press), 2004, pp. 319-320. ISBN 0-8018-8057-2.

David Kahn, *The Codebreakers*, rev. ed., (New York: Simon and Schuster), 1996, pp. 743-751. ISBN 0-684-83130-9.

MORIN, Edgar. A física da informação. O método: 1. A natureza da natureza. 2.ed. Mira-Sintra: Europa-América, ©1977. p. 276-289.

SHANNON, Claude E. Collected papers. New York: IEEE, 1993.

SHANNON, Claude E.; WEAVER, Warren. A teoria matemática da comunicação. 11. ed. São Paulo: DIFEL, 1975.

SIMON, Judith. Interdisciplinary knowledge creation: using wikis in science. In: BUDIN, Gerhard; SWERTZ, Christian; MITGUTSCH, Konstantin (Eds.). Knowledge organization for a global learning society: proceedings of the Ninth International ISKO Conference, 4-7 July 2006, Vienna, Austria. Wurzburg: Ergon-Verlag, 2006. p. 123-130.

FLORIDI, Luciano. Semantic conceptions of information. In: ZALTA, Edward N. (Ed.). The Stanford Encyclopedia of Philosophy (Spring 2013 Edition).

SHANNON, Claude E. A mathematical theory of communication. The Bell System Technical Journal, v. 27, jul./oct., p. 379-423, 623-656, 1948.

⟨http://www.capurro.de/enancib_p.htm⟩

CAPÍTULO 9

Breve História dos Primeiros Computadores

Os grandes cientistas da Ciência da Computação avançaram no tempo, construindo a ciência que levou à construção do computador digital e o desenvolvimento desta ciência. A evolução da Ciência da Computação, ocorreu promovida pelas mentes privilegiadas de cientistas, cada uma deles se baseando no trabalho dos que os antecederam. Se prestarmos atenção, perceberemos que o laço entre certos gênios, mesmo separados entre séculos, é algo admirável. A matemática e a lógica conduziram à computabilidade e aos modelos de computação que temos hoje. As ideias da matemática e da lógica, algumas que estão aí a séculos, impulsionaram o desenvolvimento inicial da Ciência da Computação, até chegarmos aos primeiros computadores como são apresentados neste capítulo.

9.1 As Grandes Inovações dos Computadores

Seguindo a história das grandes inovações, no século XIX surgiram diversas máquinas, mas sem poderem ser programadas. A criação em 1822, da máquina diferencial, por **Charles Babbage**, que poderia computar e imprimir extensas tabelas científicas, construindo um modelo para calcular tabelas de funções (logaritmos, funções trigonométricas, e outras) sem a intervenção de um operador humano. Em 1832 - **Babbage** e **Joseph Clement** produzem uma parte da *Máquina de Diferenças*. A ideia de **Jacquard**, os **cartões perfurados**, onde o contratante poderia registrar, ponto a ponto, a receita para a confecção de um tecido, foi desenvovida por **Charles Babage**, através de sua segunda máquina: a **máquina analítica**.

Em torno de 1833, **Babbage** resolveu deixar de lado seus planos da Máquina de Diferencial. O insucesso, porém, não o impediu de desenvolver idéias para construir uma máquina ainda mais ambiciosa, e entre 1834-1835, **Babbage** troca o enfoque de seus trabalhos para projetar a sua *Máquina Analítica*. Neste tempo surge **Ada Lovelace** em 1843, publicando *Notes* sobre a Máquina Analítica de **Babbage**. Embora

Babbage tenha dispendido muito de sua vida e de sua fortuna tentando construir sua "máquina analítica", ele jamais conseguiu por o seu projeto em funcionamento porque era simplesmente um modelo matemático e a tecnologia da época não era capaz de produzir rodas, engrenagens, dentes e outras partes mecânicas para a alta precisão que necessitava. Desnecessário se dizer que a máquina analítica não teve um sistema operacional.

Em 1854, **George Boole** aproximou a lógica de uma nova maneira, reduzindo-a a um , incorporando a lógica à matemática. Ele ressaltou a analogia entre os símbolos algébricos e aqueles que representam as formas lógicas. Ele começou a álgebra da lógica booleana, que depois veio a ter aplicações na construção computador digital atual. Em 1890, o censo nos USA é tabulado com as máquinas de cartões perfurados de **Hermann Hollerith**.

9.1.1 Os Computadores Analógicos

No século XX, na linha do tempo dos computadores analógicos, o mecanismo de Antikythera (Anticythère) a ser vista no capítulo 2 é o computador analógico mais antigo conhecido. Foi projetado para calcular posições astronômicas. Foi descoberto em 1901 na ilha grega de Antikythera (Anticythère), entre Kythera e Creta, e foi datado para cerca de 100 a.C. Depois, **Al-Biruni** inventou o primeiro calendário-computador movido mecanicamente, cerca de 1000 d.C.

A régua de cálculo era um computador analógico operado manualmente para fazer multiplicações e divisões, inventada entre 1620-1630, logo após a publicação do conceito de logaritmo. Nos tempos de aluno na antiga Escola Técnica Nacional no Rio de Janeiro, o autor usava régua de cálculo nas tarefas escolares.

Em 1931, **Vannevar Bush** inventa o *Analisador Diferencial*, um computador analógico-mecânico. Da primeira ideia da Máquina Diferencial de **Charles Babbage** em 1822, o analisador diferencial evolui nas mãos de **Vannevar Bush** entre as décadas de 1920 e 1930. Era um computador analógico mecânico projetado para resolver equações diferenciais por integração, usando mecanismos de engrenagem para realizar a integração de funções. As miras de bombas da segunda guerra mundial (1939-1945) usavam computadores analógicos mecânicos.

O computador MONIAC (*Monetary National Income Analogue Computer*), também conhecida como *Phillips Computer* ou o *Financephalograph*, foi criado em 1949 pelo economista da Nova Zelândia **William Phillips** para modelar os processos econômicos nacionais do Reino Unido, enquanto **Phillips** era um estudante na London School of Economics (LSE). O MONIAC era um computador analógico, que usou a lógica fluídica para modelar o funcionamento de uma economia. O nome MONIAC pode ter sido sugerida por uma associação de dinheiro e ENIAC, um computador digital eletrônico construído no início da década de 1950.

Heathkit EC-1 era um outro computador analógico educacional construído pela *Heath Company*, EUA, por volta de 1960. Computadores analógicos foram utilizados no Brasil, nos ambientes acadêmicos de universidades e o autor lembra, quando estudante de mestrado na UFRJ-COPPE-Programa de Engenharia de Sistemas em meados dos anos 70, que havia um computador analógico usado pelos pesquisadores da área de sistema de controle deste programa acadêmico.

Figura 60 – MONIAC - um computador analógico para modelar o funcionamento de uma economia.

Fonte: pt.wikipedia.org/.

9.2 Os Cientistas da Computação em 1937

No século XX, **Alan Turing** (1912-1954), em meados dos 30, enquanto imaginava a sua máquina abstrata, pensando a computação sem computador, formalizava o conceito de algoritmo como essência da Ciência da Computação. Em 1937, **Turing** publica *On Computable Numbers*, descrevendo um computador universal com a ideia de vir a existir uma máquina computadora, porém programável. Na época começava a ser esboçada a ideia de uma máquina baseada em estados que eram alterados por operações de um algoritmo submetido à máquina. Ele esboçou o que veio a ser chamado de autômato (automático), desenvolvida posteriormente por **John von Neumann** (1903-1957). Também em 1937, **Claude Shannon** descreve como circuitos chaveadores (interruptores) podiam resolver experiências de álgebra booleana.

George Stibitz (1904-1995) foi um engenheiro eletrônico e inventor Estadunidense. Trabalhou no *Bell Labs*, sendo que os seus trabalhos mais conhecidos foram realizados nas décadas de 1930 e 1940 e eram sobre circuitos digitais baseados em lógica booleana, usando *relays* electromecânicos como comutadores. No ano de 1937, **George Stibitz** constrói no *Bell Telephone Laboratories*, a primeira calculadora

binária. No ano de 1940, ainda no *Bell Telephone Laboratories*, ele demonstrou o *Complex Number Calculator* que deve ter sido o primeiro computador digital. E também foi criado o primeiro terminal de computador.

Figura 61 – George Stibitz - A primeira calculadora usando o sistema binário.

Fonte: alumnoebn.blogspot.com.

Figura 62 – George Stibitz (1937) - Sistemas digitais usando a lógica booleana.

Fonte: www.plimbi.com.

Em 1937, **Howard Aiken**, descobrindo partes da Máquina Diferencial de **Charles Babbage**, propõe a construção de um grande computador digital em Harvard (USA).

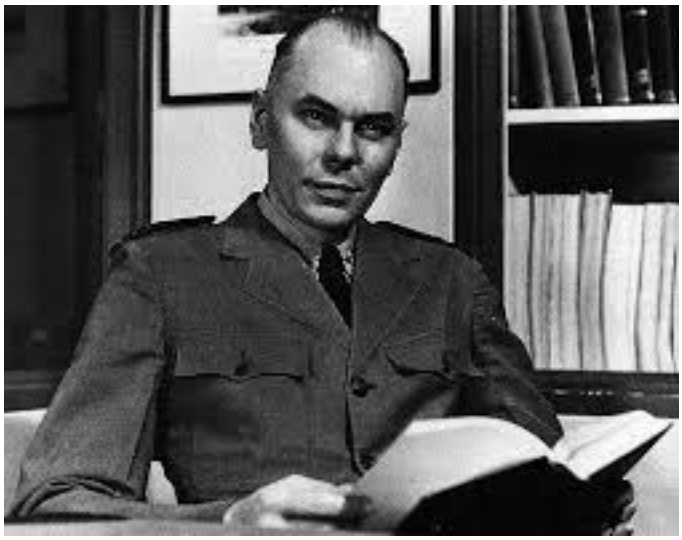


Figura 63 – Aiken - Propôs um projeto para a construção de um computador digital em Harvard.

Fonte: www.computerhistory.org.

9.3 Surge a HP - Hewlett e Packard

Em 1938, numa garagem em Palo Alto (USA), **William Hewlett** e **David Packard**, dois estudantes da Stanford University, criaram uma empresa, sendo o seu primeiro produto que um oscilador de áudio, um instrumento muito usado por engenheiros de som para fazer testes. Um de seus primeiros clientes foi a **Walt Disney Studios** que adquiriu, na época, 8 destes osciladores para desenvolver e testar o som para o filme de animação "Fantasia", de 1943.

Figura 64 – Hewlett e Packard - o Surgimento da HP.

Fonte: www.flatstanley.com.

9.4 O Primeiro Computador Digital

John Vincent Atanasoff (1903-1995) foi um matemático Estadunidense de origem Búlgara. **John Atanasoff** junto com o seu aluno **Clifford Berry** foram os

verdadeiros pioneiros dos computadores modernos. Em 1937, **Atanasoff** já reunia conceitos para serem usados na criação de um computador eletrônico. Em 1939, **John Atanasoff** completa seu modelo de computador eletrônico com tambores mecânicos para armazenamento em memória (as ideias de **Heron** de 62 d.C. são aplicadas). **Clifford Edward Berry** (1918-1963) foi um cientista da computação estadunidense. Em parceria com **John Atanasoff** criou, em 1939, o primeiro computador eletrônico, o *AtanasoffBerry Computer*, o ABC Computer.

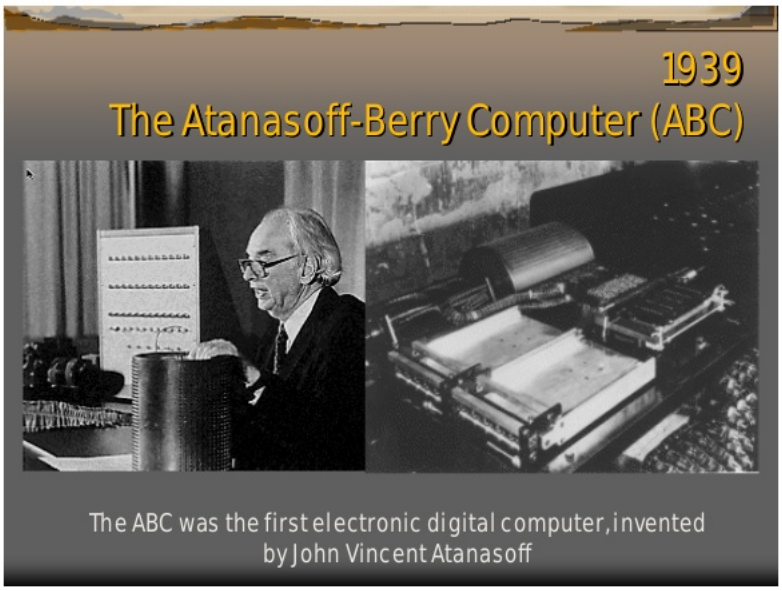


Figura 65 – Atanasoff - Conceitos para o computador eletrônico.

Fonte: www.slideshare.net.

O computador tinha válvulas eletrônicas, números binários, capacitores e 1 quilômetro de fios. Foi desenvolvido como um calculador eletrônico binário destinado a resolver sistemas de equações lineares. A memória era construída com dois tambores magnéticos e a sincronização dos ciclos era efetuada por um relógio mecânico. Seu sistema de armazenamento de resultados intermediários era um sistema de escrita/leitura de cartões perfurados, o que deixava o sistema não-confiável. O projeto foi abandonado em 1942 após o rompimento de **Atanasoff** com a **Iowa State College**.

Também em 1939, **Alan Turing** chega em *Bletchley Park* para trabalhar no deciframento de códigos alemães (ver no capítulo 4).

Em 1941, o engenheiro alemão **Konrard Zuse** completa o **Z3**, um computador digital programável eletromecânico totalmente funcional. **Konrad Zuse** (1910-1995), no periodo de 1935-1940, levara a ideia de máquina programável de **Turing** à frente.

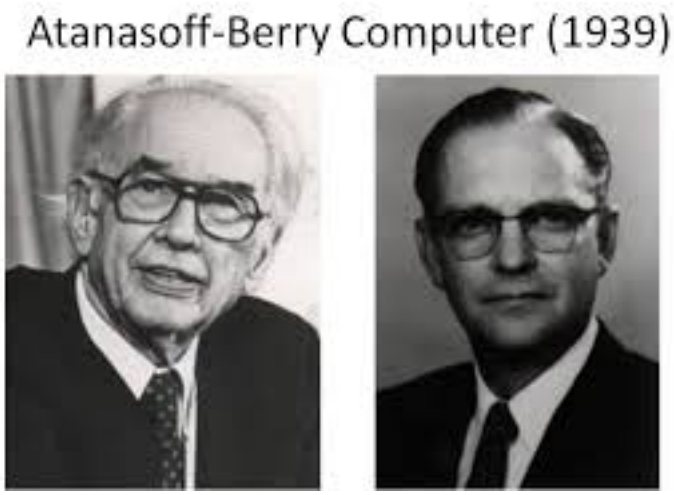


Figura 66 – John Atanasoff e Clifford Berry - o primeiro computador eletrônico.

Fonte: prezi.com.

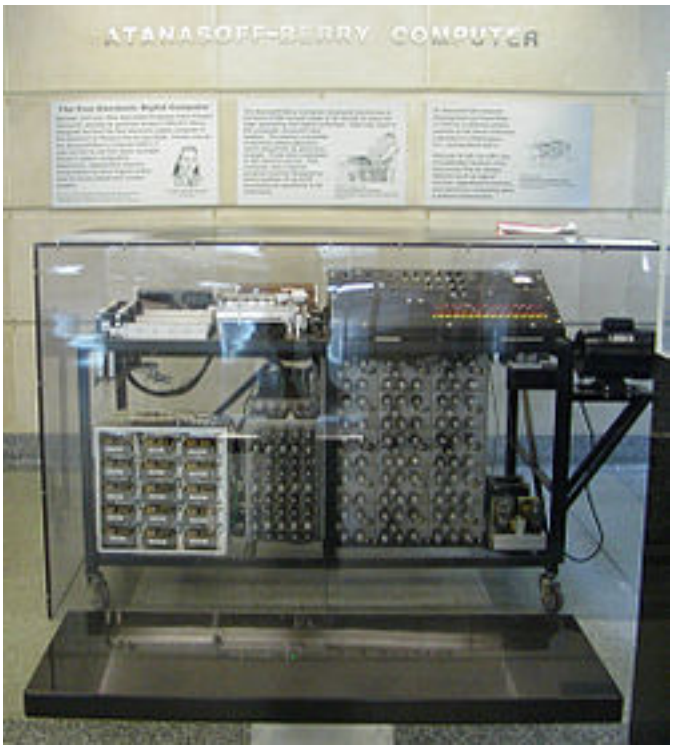


Figura 67 – John Atanasoff e Clifford Berry - O primeiro computador a usar válvulas termiônicas.

Fonte: mobile.gadzetomania.pl.

A sua terceira versão de máquina, o Z3, foi finalizado em 1941. Ela era baseada em relés telefônicos e funcionou satisfatoriamente. A troca do sistema decimal, mais difícil de implementar (utilizado no projeto de **Charles Babbage**) pelo simples sistema binário tornou a máquina de **Zuse** mais fácil de construir e potencialmente mais confiável, com a tecnologia disponível naquele tempo. O Z3 passou a ser o primeiro computador programável funcionando. Em vários aspectos ele era muito semelhante às máquinas modernas, sendo pioneiro em vários avanços, como o uso de aritmética binária e números de ponto flutuante.

Os programas eram armazenados no Z3 em filmes perfurados. Desvios condicionais não existiam, mas o Z3 ainda era um computador universal (ignorando sua limitação no seu espaço de armazenamento físico). Em duas patentes de 1937, **Konrad Zuse** antecipou que as instruções da máquina poderiam ser armazenadas no mesmo espaço de armazenamento utilizado para os dados, a primeira idéia do que viria a ser conhecida como a *arquitetura de* ***John Von Neumann*** e que seria implementada no projeto do EDSAC britânico (1949). **Zuse** ainda projetou a primeira linguagem de alto nível, em 1945, chamada de *Plankalkül*.

Figura 68 – Konrad Zuse - O primeiro computador programável com sistema binário e arquitetura de von Neumann.

Fonte: www.dsc.ufcg.edu.br.

9.5 O Projeto ENIAC

Também em 1941, **John William Mauchly** (1907-1980) visita **Atanasoff** em Iowa (USA) e assiste a uma demonstração do computador de *Atanasoff*. **John Mauchly** foi um físico Estadunidense, que junto com **John Presper Eckert** (1919-1995) e engenheiros da Universidade da Pensilvânia (USA), em parceria com o governo federal dos Estados Unidos, construíram o primeiro computador eletrônico, conhecido como (*Electronic Numerical Integrator and Computer*).

.

Figura 69 – Mauchly e Eckert - O primeiro computador eletrônico, conhecido como ENIAC.

Fonte: images.google.com.

Figura 70 – ENIAC - O primeiro projeto de computador eletrônico.

Fonte: www.gettyimages.pt.

Em 1942, **John Atanasoff** completa um computador parcialmente funcional com trezentas válvulas termiônicas para servir à marinha dos USA.

9.6 O Projeto Colossus

Em 1943, o *Colossus*, um computador à válvula feito para decifrar códigos alemães na segunda guerra mundial, fica pronto em **Bletchley Park**.

Figura 71 – Válvula Termiônica - Os primeiros componentes eletrônicos dos primeiros computadores.

Fonte: Imagens Google.

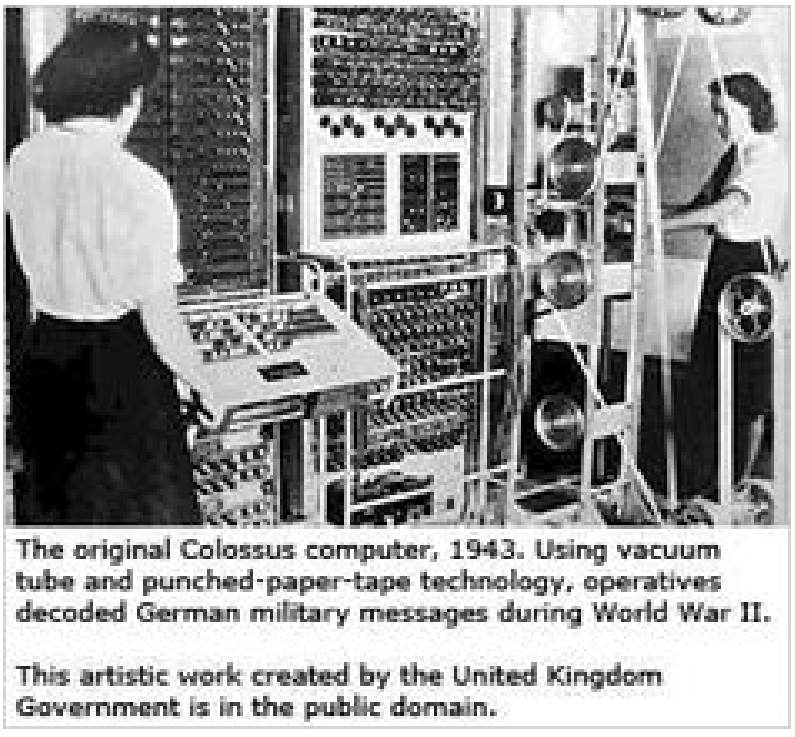


Figura 72 – Colossus - O decodificador dos códigos alemães na II Guerra Mundial.

Fonte: Imagens Google.

9.7 O Projeto Harvard Mark I

Em 1944, o ASCC (*Automatic Sequence Controlled Calculator*), chamado de *Mark I* pela Universidade de Harvard, entra operação. Este foi o primeiro computador eletromecânico automático de larga escala desenvolvido nos USA. O *Mark* I, mas este era apenas eletromecânico. Ele foi construído num projeto da Universidade de Harvard em conjunto com a IBM. Neste mesmo ano (1944) **John von Neumann** entra para Universidade da Pensilvânia para trabalhar no projeto do ENIAC (*Electronic Numerical Integrator and Computer*), que foi o primeiro computador digital eletrônico americano de grande escala.

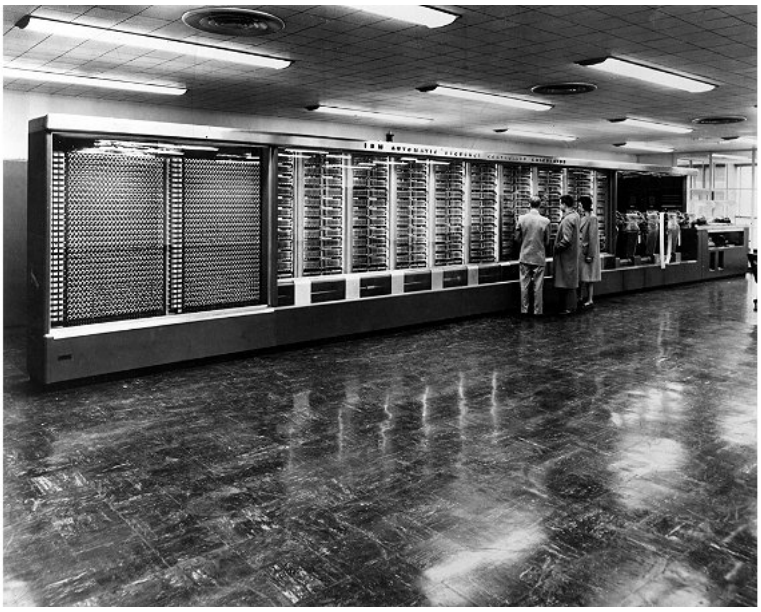


Figura 73 – Harvard Mark I - O computador de Harvard. O primeiro computador digital eletrônico de grande escala.

Fonte: www.computerhistory.org.

9.8 O Projeto EDVAC

Em 1945, o EDVAC (*Electronic Discrete Variable Automatic Computer*), diferentemente de seu predecessor ENIAC, foi um dos primeiros computadores eletrônicos que utilizava o sistema binário e possuía arquitetura de **von Neumann**. A partir do "*First Draft of Report on the EDVAC*", **John von Neumann** descreve este computador com armazenamento de programas na memória. Seis programadoras do ENIAC são enviadas a Aberdeen a fim de realizar treinamento.

Neste mesmo ano, Vannevar Bush **Vannevar Bush** publica "*As We May Think*", texto em que descreve a ideia de um computador pessoal. **Bush** publica "*Science the Endless Frontier*", propondo financiamento governamental para pesquisa acadêmica e industrial. A esta altura do tempo, o ENIAC já está em pleno funcionamento.

Figura 74 – EDVAC - Sistema binário e arquitetura de armazenamento de programas na memória.

Fonte: www.youtube.com.

9.9 Os Primeiros Sistemas Operacionais

Após os esforços sem sucesso de **Babbage**, pouco progresso se teve na construção de computadores digitais até a Segunda Guerra Mundial. Em torno de 1940, **Howard Aiken** em Harvard, **John Von Neumann** no Advanced Research Institute em Princeton, **John Presper Eckert** e **William Mauchley** na Universidade de Pennsylvania e **Konrad Zuse** na Alemanha, entre outros, tiveram sucesso na construção de máquinas computadores usando válvulas. Essas máquinas eram enormes, ocupando salas completas, com dezenas de milhares de válvulas, porém eram muito mais lentas do que os mais simples computadores pessoais de hoje.

Naqueles tempos primitivos, um pequeno grupo de pessoas construiu, programou, operou e deu manutenção a cada máquina. Toda a programação era feita em linguagem de máquina, sempre se conectando fios com *plugs* em painéis para controlar as funções básicas da máquina. As linguagens de programação não eram conhecidas (nem a linguagem Assembly). Nem se ouvia falar em sistemas operacionais. O modo usual de operação consistia no programador elaborar o programa numa folha e então ir à sala da máquina, inserir os *plugs* nos painéis do computador e gastar as próximas horas apelando que nenhuma das 20.000 ou mais válvulas não se queimasse durante a execução do programa. Na verdade, todos os problemas eram inerentemente sobre cálculos numéricos tais como gerações de tabelas de números.

9.10 A Evolução da Eletrônica I

Em 1947, a área da eletrônica dá um salto das válvulas para o transistor. Este último é inventado nos Laboratórios Bell (USA) por **William Bradford Shockley**, (1910-1989) **Walter Houser Brattain** (1902-1987) e **John Bardeen** (1908-1991). O transístor (como no português europeu) ou transistor (como no português brasileiro)

é um componente eletrônico que começou a popularizar-se na década de 1950, tendo sido o principal responsável pela revolução da eletrônica na década de 1960. São utilizados principalmente como amplificadores e interruptores de sinais elétricos, além de retificadores elétricos em um circuito, podendo ter variadas funções. O termo provém do inglês *transfer resistor* (resistor/resistência de transferência), como era conhecido pelos seus inventores.

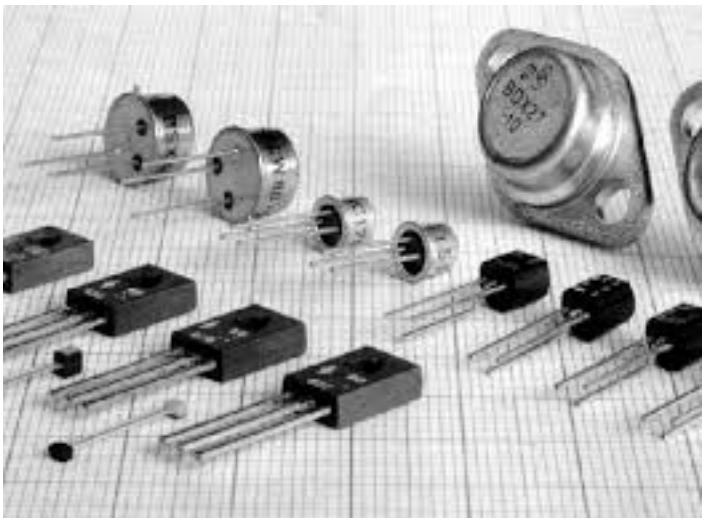


Figura 75 – O transistor - Os componentes eletrônicos que substituiram as válvulas e diminuiram o tamanho dos computadores.

Fonte: Imagens Google.

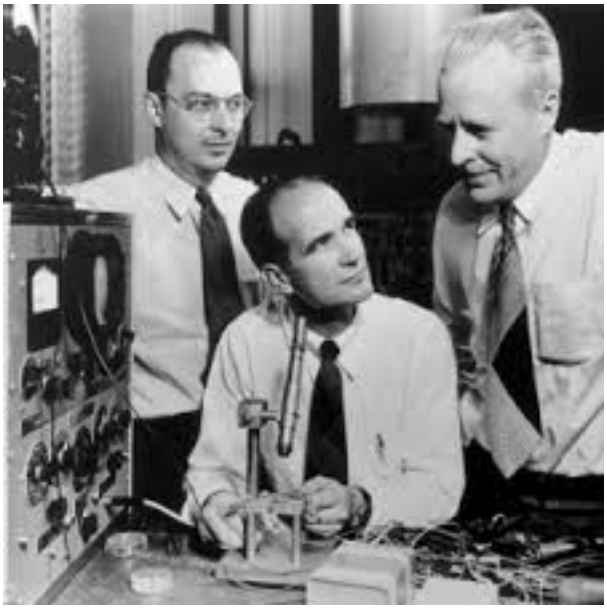


Figura 76 – John Bardeen, Walter Houser Brattain e William Bradford Shockley.

Fonte: www.biography.com.

9.11 O Manchester Mark I

Entre 1948 e 1949, o *Manchester Mark I* foi um dos primeiros computadores eletrônicos desenvolvidos. Foi construído pela *University of de Manchester*. Também era chamado de MADM (Manchester Automatic Digital Machine). Ele foi desenvolvido a partir do SSEM ("The Baby Machine"), pelo **Prof. F. C. Williams** e pelo **Prof. Tom Kilburn** (nesse caso o seu prótotipo - O *Manchester MARK I* teve importância histórica devido ao pioneirismo no uso de um tipo de registrador de índice em sua arquitetura, além de ter sido a plataforma na qual a *Autocode*, uma das primeiras linguagens de programação de alto nível, foi desenvolvida.

O Manchester Mark I servia para calcular operações aritméticas, dispondo ainda de sub-rotinas integradas que calculavam funções logarítmicas e trigonométricas; mesmo assim o Manchester Mark I era um calculador lento demorando 3 a 5 segundos para efetuar uma multiplicação, mas era totalmente automático e podia realizar cálculos extensos sem intervenção humana.

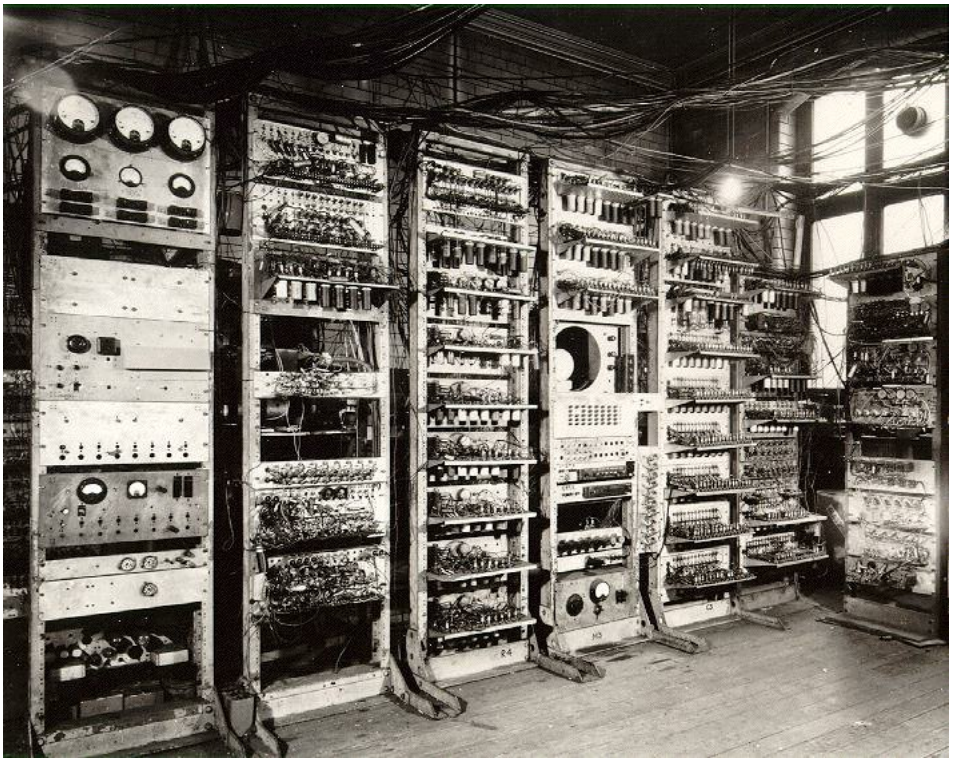


Figura 77 – O Manchester Mark I - Um dos primeiros computadores eletrônicos desenvolvidos, construído pela University of de Manchester.

Fonte: piano.dsi.uminho.pt.

9.12 Na Década de 50

E em 1950, **Turing** publica um artigo que descreve um teste para inteligência artificial. Como o leitor poderá saber, no capítulo 4 descreve-se que **Alan Turing**, em sua época, iniciou as primeiras pesquisas na área da Inteligência Artificial para a Ciência da Computação.

Em 1952, trabalhando no projeto do computador Harvard Mark I, **Grace Murray Hopper** (1906-1992) desenvolveu a ideia do primeiro compilador. **Grace Hopper**, na Figura 78, foi uma analista de sistemas da Marinha dos Estados Unidos nas décadas de 1940 e 1950. Foi ela quem criou a linguagem de programação *Flow-Matic*, hoje extinta. Esta linguagem serviu como base para a criação da linguagem COBOL, mantida no mercado corporativo pela IBM, nos anos 60-70, para aplicações comerciais.

Figura 78 – Grace Hopper - a criadora do primeiro compilador de linguagem de programação.

Fonte: www.shorpy.com.

O primeiro *bug* da história - A palavra *bug* (inseto em inglês) é empregada atualmente para designar um defeito, geralmente de software. Mas sua utilização com este sentido remonta a esta época. Conta a história que um dia o computador apresentou defeito, e ao serem investigadas as causas, verificou-se que um inseto havia prejudicado seu funcionamento. A foto abaixo, supostamente, indica a presença do primeiro *bug*.

Em 1952, **von Neumann** completa um outro projeto de computador mais moderno (para a época) no Instituto de Estudos Avançados na University of Princeton, o

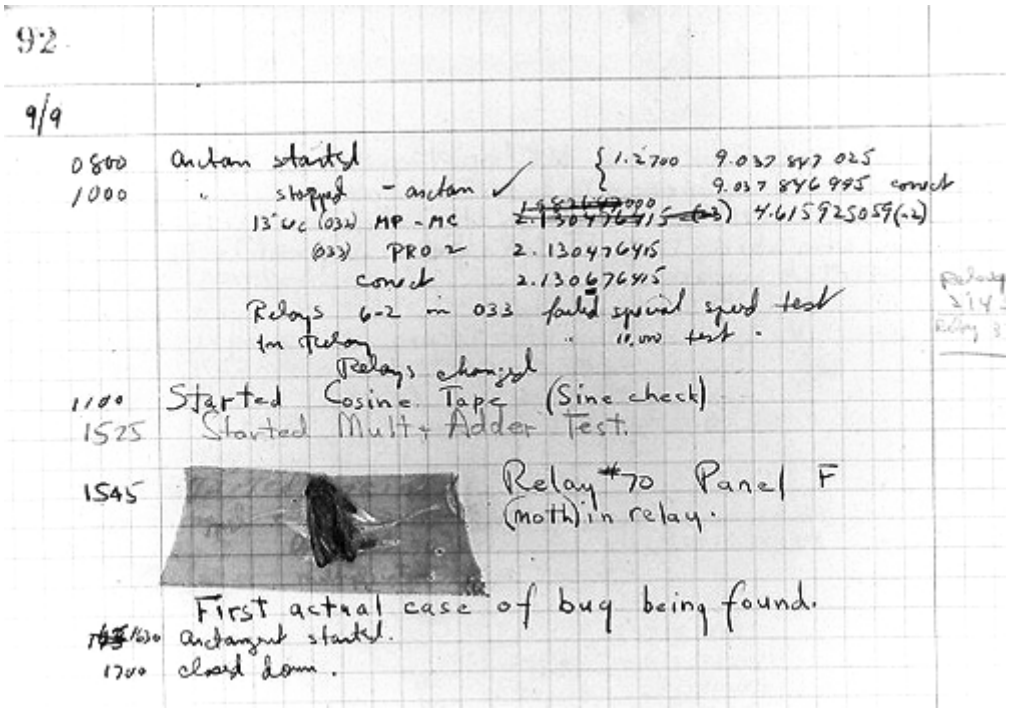


Figura 79 – O primeiro bug documentado no relatório da foto.

Fonte: ⟨http://producao.virtual.ufpb.br/books/camyle/introducao-a-computacao-livro/livro/livro.chunked/ch01s02.html⟩.

UNIVAC. Com o UNIVAC foi realizada a previsão da eleição de Eisenhouwer nos USA.

Figura 80 – UNIVAC - O primeiro computador comercial nos USA.

Fonte: pt.wikipedia.org.

O UNIVAC I (de *UNIVersal Automatic Computer* - Computador Automático Uni-

versal) foi o primeiro computador comercial fabricado e comercializado nos Estados Unidos. Era programado ajustando-se cerca de 6.000 chaves e conectando-se cabos a um painel. Foi projetado por **J. Presper Eckert** e **John Mauchly**, os inventores do ENIAC, para uma empresa fundada por ambos, a *Eckert-Mauchly Computer Corporation*. Porém só ficou pronto após esta ser adquirida pela *Remington* e criar a divisão UNIVAC.

O primeiro UNIVAC foi entregue ao escritório do censo dos Estados Unidos em 1951, mas demorou para começar a funcionar, então o primeiro que entrou em operação foi o segundo a ser fabricado, para o Pentágono, a sede do departamento de Defesa dos USA.

O UNIVAC usava 5.200 válvulas, pesava 13 toneladas e consumia 125 kW para fazer 1905 operações por segundo, com um *clock* de 2.25MHz. O sistema completo ocupava mais de $35m^2$ de espaço no piso. Sua memória de mil palavras era armazenada num dispositivo chamado *delay line memory*, construído com mercúrio e cristais piezoelétricos. A entrada e saída de informações eram realizadas por uma fita metálica de 1/2 polegada de largura e 400m de comprimento. Normalmente acompanhados de um dispositivo impressor chamado Uniprinter, que, sozinho, consumia 14 kW.

Algumas unidades estiveram em serviço por muitos anos. A primeira unidade funcionou até 1963. Duas unidades da própria Remington funcionaram até 1968 e outra unidade, de uma companhia de seguros do Tennessee, até 1970, com mais de treze anos de serviço.

9.13 Os Primórdios da Computação no Brasil

A presente seção resume o que está em Cardi e Barreto (2002). Um marco importante na História da Computação ocorreu na década de 30, onde **Helmut Theodor Schreyer** (1912-1984), engenheiro eletricista alemão falecido no Rio de Janeiro, Brasil, havia auxiliado **Konrad Zuse** no projeto de construção do primeiro computador constituído por componentes mecânicos e eletromecânicos.

Em 1934, foi construído o Z1, máquina programável com relés a trabalhar sob o controle de um programa em fita perfurada, que possuía um teclado onde era introduzido os problemas, e o resultado faiscava num quadro com muitas lâmpadas. O Z1 foi modificado originando o Z2 que codificava as instruções perfurando uma série de orifícios em filmes usados de 35 mm. A partir deste momento, **Schreyer** e **Zuse** passaram a trabalhar separadamente. **Schreyer** iniciou então uma linha de projetos baseados em válvulas eletrônicas, que ele mesmo projetou e que foram fabricadas pela Telefunken.

Schreyer viveu no Brasil, Rio de Janeiro, e em sua chegada foi recebido pelo Dr. Sauer, Professor de Máquinas Elétricas da Escola Técnica do Exército (ETE),

hoje Instituto Militar de Engenharia (IME) que fica situado na Praia Vermelha, Rio de Janeiro. Na época, estavam inaugurando o Curso de Engenharia Eletrônica na então Escola Técnica do Exército dirigida pelo **General Dubois**, que foi junto com o **Almirante Álvaro Alberto**, **Lelio Gama**, um matemático brasileiro, diretor do Observatório Nacional por muitos anos, conhecido por seus trabalhos em magnetismo terrestre. Todos, membros fundadores do *Conselho Nacional de Pesquisas* (o atual CNPq) criado em 1951. Este general empregou **Schreyer** fixando-o no Brasil.

Durante sua vida em terras brasileiras, **Schreyer** prestou outros importantes serviços à sociedade, tendo trabalhado nos Correios e Telégrafos da época e sido professor da PUC (Pontifícia Universidade Católica), antiga Escola de Engenharia depois transformada em Centro Tecnológico. Em 1960 participou da direção do projeto de fim de curso de eletrônica, que consistiu em um Computador, sendo a nosso conhecimento o primeiro computador projetado e construído no Brasil. Em 1952, **Schreyer** publicou no Rio de Janeiro, pela editora da ETE, um livro sobre "Computadores Eletrônicos Digitais", onde apresentou o projeto dos circuitos básicos usados em um computador digital. Em reconhecimento aos feitos de **Schreyer**, há no Laboratório de Técnica Digitais do IME, uma placa atida, com um resumo biográfico do pesquisador.

No Brasil, na década de 50, os computadores eram raridade curiosa, quase inacessível. Assim, a computação no Brasil iniciou-se, no decorrer do mandato de Juscelino Kubitschek (1956-1961), que possuía uma filosofia de governo baseada no desenvolvimento econômico planejado e destinada a tirar o país do atraso. Neste tempo, os computadores não ficaram de fora da revolução de modernidade, esta fase inicial da informática no Brasil, foi caracterizada pela importação de tecnologia de países com capitalismo avançado.

Em 1958, o economista Roberto de Oliveira Campos, secretário-geral do Conselho de Desenvolvimento Nacional, por sugestão do Capitão de Corveta Geraldo Maia (na época recém pós-graduado em engenharia eletrônica nos Estados Unidos), sugeriu e o Governo autorizou, a criação de um "Grupo de Trabalho" com a finalidade de analisar a utilização de computadores eletrônicos nos cálculos orçamentários e no controle da distribuição das verbas governamentais. Este grupo apresentou um relatório, em janeiro de 1959, que sugeria a criação de centros de processamento de dados e o desenvolvimento de recursos humanos, além da criação, na área de atividade do Conselho Nacional de Desenvolvimento, de um Grupo Executivo mais assíduo.

Assim, foi criado pelo Decreto nº. 45.832, de 20 de abril de 1959, no Conselho de Desenvolvimento, o GEACE - Grupo Executivo para Aplicação de Computadores Eletrônicos, com a finalidade de incentivar a instalação de Centros de Processamento de Dados, assim como a montagem e fabricação de computadores e seus componentes; orientar a instalação de um Centro de Processamento de Dados a ser criado em órgão oficial adequado; e promover intercâmbio e troca de informações com entidades estrangeiras congêneres. Com a criação do GEACE, principiaram-se os processos de

importação de computadores, tais como: o UNIVAC 1103 para o IBGE Instituto Brasileiro de Geografia e Estatística, e o computador Gamma da Bull Machines, para a empresa Listas Telefônicas Brasileiras.

Conscientes da necessidade do país em utilizar computador para auxiliar as pesquisas, o GEACE e o CNPq deram início ao processo de importação do computador, B-205 da Burroughs. Para a aquisição do equipamento, criaram um tipo de consórcio de entidades e empresas para dividir as despesas, pois se depararam com inúmeras adversidades tais como: custo altíssimo da máquina (equivalente a 400 mil dólares), energia instável e pessoas não qualificadas. Participaram desse processo, o CNPq - Conselho Nacional de Pesquisas que colaborou com grande parte do capital para a compra; o Ministério da Guerra com a colaboração técnica; a Comissão Nacional de Energia Nuclear, a Companhia Siderúrgica Nacional e a PUC-RJ que cedeu uma sala para instalação do computador.

Para o processo de negociação com os americanos foi composta uma comissão com mais de 10 pessoas, das quais apenas o Professor **Major Haroldo Correa de Mattos**, da então Escola Técnica do Exército, viajou aos Estados Unidos para participar na escolha da configuração da máquina a ser adquirida. Major Mattos era engenheiro eletricista, pósgraduado nos Estados Unidos em duas Universidades, e já tinha alguma experiência no assunto. Registre-se que Major Mattos foi também Ministro das Comunicações.

O B-205 era um computador completamente diferente dos que conhecemos hoje, pois ocupava uma sala inteira. O chamado "cérebro eletrônico" era um Burroughs Datatron 205, da primeira geração de computadores à válvulas (ele possuía cerca de 4.600), efetuava uma adição em 0,1 milissegundos e a memória era uma espécie de tambor magnético com capacidade a cerca de 20K bytes. A entrada de dados era feita através de cartões e fitas perfuradas, além de teclado manual. Os dados eram armazenados em fitas magnéticas. A programação era efetuada em linguagem de máquina absoluta, não possuía sistema operacional, sistema de arquivos, processador de linguagem ou qualquer outro software de apoio. Trabalhava apenas em ponto fixo e em consequência tinha uma lâmpada "overflow" que significava ter de começar fazendo nova escolha de escalas para as variáveis (coisas que desapareceram com os progressos dos programas). A "impressora" era um tipo de máquina de escrever *flexowriter* com velocidade de dez caracteres por segundo. Posteriormente foi agregada uma tabuladora IBM 407, que expandiu a velocidade de impressão para 100 linhas por minuto.

Finalmente, em 1960, foi inaugurado o primeiro computador da América Latina em Universidades e o primeiro do Brasil, no recém-criado Centro de Processamento de Dados da PUC-RJ. O equipamento teve o mérito de mostrar aos estudantes, entre outras coisas, novas técnicas de cálculos científicos para aplicação em várias áreas de engenharia e pesquisa. A direção do Centro de Processamento de Dados foi entregue ao Prof. Pierre Lucie do Departamento de Física da PUC/RJ. Ele era

imigrado da França após a guerra de 40-45 tendo sido membro da resistência francesa contra a ditadura dos alemães. Muito atarefado com suas pesquisas e com a reforma do ensino da Física, passou a chefia para o **Prof. Helio Drago Romano**, Major ex-professor do IME, popular na época por seu conhecimento em síntese de circuitos para telecomunicações. Foi ele quem efetivamente colocou o Centro de Processamento de Dados em regime de funcionamento, criando uma série de cursos sobre computação e correlatos. Depois de algum tempo, foi para a cidade de Campinas/SP integrar a equipe que fundou a UNICAMP em Campinas.

O desenvolvimento da computação no Brasil começou a se estabilizar a partir do desenvolvimento de computadores no País. Os primeiros protótipos surgiram nas universidades nacionais como projeto de conclusão dos cursos de Graduação em engenharia. Foi com base nestes projetos que o desenvolvimento tecnológico do país alavancou. A formação de pessoal qualificado ficou a cargo da Escola Técnica do Exército, atual IME (Instituto Militar de Engenharia), por ter muitos professores com pós-graduação nos Estados Unidos e França. No curso de eletrônica, do IME, iniciou-se o projeto de computadores, efetivado a partir de 1958 (parte analógica), combinando com o projeto de fim de curso da turma de 1960. Criaram um computador que além da parte digital incluía circuitos analógicos capazes de simular, em tempo real, sistema de equações diferenciais e com isto resolver problemas complexos. Hoje, esta parte de circuitos analógicos seria implementada por programas de simulação tais como ACSL (Analog Computer Simulation Language) ou o clássico CSMP (Continuous System Modeling Program).

O projeto contou com a orientação dos professores do curso de eletrônica: **Antônio Maria Meira Chaves** (Chefe do Curso de Eletrônica); **Antônio José Duffles Amarante**; **Danilo Marcondes**; **Rubens T. Carrilho**; **Werther Aristides Vervloet** e **Dr. Helmut Theodore Schreyer**. E assim, em dezembro de 1960, os alunos **José Augusto Mariz de Mendonça**, **Jorge Muniz Barreto**, **Herbert Baptista Fiuza**, **Edison Dytz**, **Mário de Moura Alencastro** e **Walter Mario Lace**, apresentaram dois computadores (um analógico e outro digital). Os dois primeiros alunos (Mariz e Barreto) desenvolveram o digital e os outros quatro (Fiuza, Dytz, Alencastro e Lace), o analógico. Após a defesa do projeto a máquina foi desmontada e transformada em placas para o estudo da Arquitetura de Computador, peça usada até os anos 70 no Laboratório de Circuitos Digitais. Este projeto cumpriu sua finalidade e a História da Computação no Brasil ganhou um novo marco.

No ano de 1961, os alunos do ITA- Instituto Tecnológico da Aeronáutica, Alfred Volkmer, Andras Gyorgy Vasarhelyi, Fernando Vieira de Souza e José Ellis Ripper Filho, entusiasmados com uma visita que haviam feito à Cie. de Machines Bull na França, onde vislumbraram detalhadamente as etapas do projeto e fabricação de computadores, apresentaram como trabalho de conclusão de curso, juntamente com a Escola Politécnica da Universidade de São Paulo (USP) e a Pontifícia Universidade Católica do Rio de Janeiro (PUC/Rio), um equipamento didático que mostrava como a informação se processava dentro do computador. Esta máquina denominada ITA I,

batizada como foi construída com transistores discretos, usando soquetes de válvulas para demonstração e uso em laboratório. Tinha dois metros de largura por um metro e meio de altura.

Em julho de 1972, o Laboratório de Sistemas Digitais (LSD) do Departamento de Engenharia da Eletricidade da Escola Politécnica da Universidade de São Paulo inaugurou outro projeto. O trabalho foi 100% desenvolvido com recursos da Escola Politécnica, graças ao apoio do Diretor Prof. **Osvaldo Fadigas Fontes Torres**, sem ajuda de instituições externas, ainda um pouco céticas em relação à viabilidade do projeto. O projeto alcançou com êxito seus objetivos, que eram de formar jovens com a habilidade de projetar e implementar computadores. O computador era composto de 450 pastilhas de circuitos integrados, contendo cerca de três mil blocos lógicos, distribuídos em 45 placas de circuito impresso e cinco mil pinos interligados segundo a técnica *wire-wrap*. A memória principal tinha capacidade para 4.096 palavras de oito bits.

Do início da década de 60 até o começo dos anos 70, era enorme a insatisfação com a situação brasileira no setor tecnológico. Nessa época, todos os computadores no país era importados. No mesmo período, a Marinha comprou fragatas inglesas comandadas por computador. O almirantado espantou-se com o alto preço dos computadores em embarcações de combate com artilharia eletrônica. Um grupo de oficiais conseguiram que parte do equipamento passasse a ser fabricado por empresas brasileiras, reivindicando a criação de uma indústria de eletrônica digital.

Em 18 de julho de 1974, a E.E. Eletrônica, o BNDE e a inglesa Ferranti associaram-se para formar a COBRA - Computadores e Sistemas Brasileiros Ltda, empresa cuja história se liga estreitamente à política de informática no Brasil, foi a primeira empresa a desenvolver, produzir e comercializar tecnologia genuinamente brasileira na área de informática.

Através de parcerias com a inglesa Ferranti e a americana Sycor Inc. a COBRA acumulou conhecimento técnico-industrial. Do início, quando tudo era novo e precisava ser desvendado, passou-se rapidamente ao desenvolvimento de tecnologia e geração de seus próprios produtos.

Nessa época, quem atuavam como professores de cursos que visavam à criação da competência para desenvolver computadores, eram engenheiros de empresas estrangeiras: **Glen Langdon** (IBM), **Jim Rudolph** (HP) e dentre outros. Segundo o Prof. Antônio Hélio Guerra Vieira (USP), responsável pelo projeto, apesar de o projeto ser ambicioso, o computador era pequeno (porte do PDP8 Digital), com arquitetura clássica, CPU e administração de alguns periféricos. O GTE (Grupo de Trabalho Especial) encomendou um protótipo de computador ao Laboratório de Sistemas Digitais da USP (que faria o hardware) e ao Departamento de Informática da PUC do Rio de Janeiro (que faria o software), que foi entregue em 1975. Tratava-se de um protótipo industrial mais compacto, seguindo os recursos da época, mais

fácil de montar e com componentes periféricos, batizado de G-10, segundo Antonio Vieira, o G-10 tinha as características de um protótipo. Possuía documentação com desenhos e especificações, software e sistema operacional desenvolvido pela PUC-RJ. Em setembro de 1977, no VII Secomu, realizado em Florianópolis/SC, sob influências da CAPRE (Comissão de Coordenação das Atividades de Processamento Eletrônico) e da FINEP (Financiadora de Estudos e Projetos) que estava financiando o projeto, os executivos da empresa COBRA (Computadores Brasileiros) comprometeram-se a tocar o projeto do G-10 de forma mais efetiva. A máquina foi reprojetada, passando a ser designada de G-11. Multiusuário, mas ainda sem o efetivo comprometimento da COBRA na sua industrialização. Porém, quando houve a decisão da COBRA em assegurar o projeto, a máquina foi novamente reprojetada, originando a linha COBRA 500.

Percebendo a impossibilidade de competir com as gigantes estrangeiras na produção de equipamentos de grande porte, a indústria nacional procurava um espaço que permitisse seu desenvolvimento e auto-suficiência. A escolha do setor de mínis e micros prendia-se a uma razão muito forte. Ao contrário dos grandes computadores, o componente eletrônico principal desses equipamentos eram os chips, facilmente comprados no exterior.

O COBRA 530, lançado no início da década de 80, foi o primeiro computador totalmente projetado, desenvolvido e industrializado no Brasil. Nessa época foram lançados os modelos da mesma linha do C-530, como o C-520, C-540, C-480 e C-580, até chegar a linha X. Também foram lançados os primeiros microcomputadores de 8 bits - o COBRA 300, COBRA 305 e o COBRA 210, além de terminais remotos. Nessa fase, uma série de sistemas operacionais como o SOM, SOD, SPM e SOX (compatível com o Unix), e várias linguagens como LPS, LTD, Cobol e Mumps foram criadas. Em 1987, a COBRA havia lançado o XPC, o seu compatível PC-XT.

Em 1984, a primeira lei sobre Informática no Brasil, a Lei Federal nº 7.232/84, estabeleceu a reserva de mercado para este ramo de atividade, induzindo fortemente o investimento do Governo e Setor Privado na formação e especialização de recursos humanos voltados à transferência e absorção de tecnologia em montagem microeletrônica, arquiteturas de hardware, desenvolvimento de software básico e de suporte, entre outros. A ideia de instituir uma reserva de mercado para fabricantes nacionais de produtos de informática começou a tomar forma na primeira metade da década de 1970, durante a vigência do Regime Militar. A justificativa é que, protegidas da concorrência com as multinacionais do setor (IBM, Burroughs, HP, Olivetti etc), os fabricantes brasileiros poderiam desenvolver uma tecnologia genuinamente nacional e estariam plenamente aptos para competir em pé de igualdade com suas concorrentes estrangeiras quando a reserva de mercado terminasse.

Embora não se possa negar a realização de grandes investimentos internos, a Política Nacional de Informática, então, em vigor acabou por engessar o desenvolvimento econômico do país e chegou a favorecer a pirataria de hardware e software, com o

surgimento de diversas empresas nacionais que oficialmente fabricavam equipamentos ou desenvolviam sistemas copiados de projetos estrangeiros, principalmente de origem norte-americana.

A única empresa estrangeira que pareceu ter obtido autorização do governo brasileiro para comercializar calculadoras eletônicas no país, nessa época, foi a Hewlett-Packard, com seu modelo HP85B. A única restrição colocada pelo Governo foi a de que a máquina só poderia ser negociada para aplicações técnico-científicas, mas não para fins comerciais.

Na segunda metade da década de 80, o controle de preços e o aumento das despesas com os sucessivos planos econômicos descapitalizaram as empresas. Além disso, com o fim da reserva de mercado da informática, as gigantes mundiais do setor de informática puderam se firmar no Brasil. Foi um período em que muitas empresas nacionais sucumbiram. A COBRA buscou novos caminhos e tornou-se integradora de soluções tecnológicas e prestadora de serviços.

O fim da reserva de mercado, pela Lei Federal nº 8.248/91, incrementou o livre acesso da mão-de-obra especializada a recursos laboratoriais de ponta, já consolidados, testados e aprovados em economia de escala mundial e condicionou o investimento em novos projetos como contrapartida das empresas que se beneficiavam de incentivos fiscais concedidos ao desenvolvimento de produtos ou serviços com valor nacional agregado.

No início da década de 90, a COBRA se afinou à tendência mundial de parcerias, dentre as quais a Sun Microsystems, IBM, Cisco Systems, Microsoft, Oracle e SCO. Por essa época, o Banco do Brasil passou a acionista majoritário da COBRA e no final dos anos 90, entrou firme e forte no mercado de serviços para a área bancária, fabricando terminais de auto-atendimento. Enfim, a tentativa da reserva mercado dos governos militares da época, acabou por ser relativamente frustrante.

9.14 Outros Acontecimentos Marcantes

Em 1957, **Robert Noyce**, **Gordon Moore** e outros fundam a *Fairchild Semiconductor*. Em 1958, é anunciada a criação da *Advanced Research Projects Agency* - ARPA. **Jack Kilby** demonstra o circuito integrado (*microship*). Em 1959, **Noyce** e outros da Fairchild inventam o *microship* de maneira independente. Em 1960, **Paul Baran** inventa a comutação de pacotes. Em 1963, **Licklider**, o diretor-fundador do *Information Processing Thecnical Office* da ARPA, propõe uma "rede de computadores intergalática". **Engelbart** e **Bill English** inventam o . *mouse*. E, 1965, **Ted Nelson** publica o primeiro artigo sobre o "*hipertext*". A Lei de Moore prevê que os microships irão dobrar de potência aproximadamente a cada ano. Em 1966, **Bob Taylor** convence o diretor da ARPA, **Charles Herzfeld** a fundar a *Arpanet*. **Donald Davis** cunha o termo *comutação de pacotes*. Em 1967, as discussões sobre

o projeto *Arpanet*. Em 1968, *Larry Roberts* pede propostas para construir a Arpanet nos USA. **Robert Noyce** e **Gordon Moore** criam a Intel e contratam *Andy Grove*.

Em 1969, a instalação dos primeiros nodos da Arpanet, o embrião da Internet. Em 1971, é revelado o microprocessador Intel 4004. **Ray Tomlinson** inventa o email. Em 1973, **Alan Kay**, **Chuck Tracker** e **Butler Lampson** criam o Xerox Alto, no Xerox Park, que foi um minicomputador pioneiro e o primeiro a utilizar a metáfora da "mesa de trabalho" (*desktop*). XEROX Alto, o minicomputador ALTO, pioneiro a usar uma "área de trabalho".

Figura 81 – XEROX Alto - O minicomputador desktop pioneiro.

Fonte: Imagens Google.

Também em 1973, **Bob Metcalfe** desenvolvia a *Ethernet* no Xerox Park em Palo Alto, USA. Instalação de um terminal compartilhado em Berkeley. **Vint Cerf** e **Bob Kahn** completam o projeto do protocolo TCP/IP para a Internet. Em 1974, o lançamento do microprocessador Intel 8080. Em 1975, surge o computador pessoal Altair da MITS. **Bill Gates** e **Paul Allen** escrevem o *Basic* para o Altair e criam a Microsoft. **Steve Jobs** e **Steve Vosniak** lançam o Apple I. Em 1977, o lançamento do Apple II. Em 1978, primeiro *Bulletin Board System* para a Internet.

Em 1979, criação dos primeiros grupos de notícias na *Usenet*. **Steve Jobs** visita o Xerox Park. Em 1980, a IBM encomenda à Microsoft o desenvolvimento para um sistema operacional para o computadores pessoais (PC). Em 1981, o *modem* Hayes é comercializado para usuários domésticos. Em 1983, a Microsoft anuncia o Windows. **Richard Stallman** começa a desenvolver o GNU, um sistema operacional livre. Em 1984, a Apple lança o Macintosh. Em 1985, a CVC lança o Q-Link que

depois se tornaria a AOL (*American on Line*), o primeiro provedor de Internet. Em 1991, **Linus Torval** lança a primeira versão do kernel do Linux, **Tim Berners-Lee** anuncia a *World Wide Web*. **Tim Berners-Lee**, então no laboratório de física CERN, em Genebra, Suíça, criou a linguagem de marcação de hipertexto (HTML), uma maneira simples de ligar informações entre sites da Internet. Isto, por sua vez, gerou a World Wide Web (WWW), que apenas aguardava por um paginador gráfico para começar a crescer.

Em 1993, **Marc Adreessen** anuncia o navegador *Mosaic* (primeiro navegador para Internet). Talvez a mudança mais importante destes últimos anos tenha vindo de um grupo de estudantes da Universidade de Illinois, pois foi lá que, no início de 1993, **Marc Andreessen**, **Eric Bina** e outros que trabalhavam para o *National Center for Supercomputing Applications* (NCSA) apareceram com o *Mosaic*, uma ferramenta que seria utilizada para paginar a Internet. A ideia de Internet já existia há muitos anos, datando do início dos anos 60, quando o órgão de Defesa de Projetos de Pesquisa Avançada (DARPA) do Pentágono estabeleceu as conexões com computadores de universidades. Enquanto a Internet crescia, o governo transferiu seu controle para os sites individuais e comitês técnicos. Após o lançamento do *Mosaic* ao público, no final de 1993, repentinamente, a Internet, e em particular a Web, podiam ser acessadas por qualquer pessoa que tivesse um computador pessoal, fato auxiliado, em parte, nela possibilidade de transferir livremente a versão mais recente de vários paginadores diferentes. E, dentro de pouco tempo, parecia que todo o mundo estava inaugurando seu site na Web.

Novas versões de paginadores da Web também chegaram rapidamente. A *Netscape Corp.* - uma nova companhia formada por **Andreessen** e **Jim Clark**, que havia sido um dos fundadores da *Silicon Graphics* - logo começou a dominar o ramo de paginadores Web. O *Netscape Navigator* acrescentou vários recursos, inclusive o suporte a extensões (o que, por sua vez, levou a diversas extensões multimídia) e à máquina virtual *Java* (que permitia aos desenvolvedores elaborar aplicativos Java que podiam ser executados dentro do paginador).

Também em 1993, a AOL, de **Steve Case**, oferece acesso direto à Internet. Em 1994, **Justin Hall** lança umWweb log diretório. São criadas as primeira editoras importantes de revistas sobre a WWW. Em 1998, **Larry Page** e **Sergey Brin** lançam a *Google*. Em 1999, **Ev Williams** lança o *Blogger*. Em 2001, **Jimmy Wales** e **Larry Sanger** lançam a *Wikipedia*.

9.15 A Segunda Geração - Transistores e Sistemas Batch (1955 - 1965)

A introdução do transistor em meados dos anos 50 mudou o quadro radicalmente. Os computadores tornaram-se bastante confiáveis para que pudessem ser produzidos e vendidos comercialmente na expectativa de que eles continuassem a funcionar por

bastante tempo para realizar algumas tarefas usuais. A princípio havia uma clara separação entre projetistas, construtores, operadores, programadores e o pessoal de manutenção.

Essas máquinas eram alocadas em salas especialmente preparadas com refrigeração e com apoio de operadores profissionais. Apenas grandes companhias, agências governamentais, ou universidades, dispunham de condições para pagar um preço de multimilhões de dólares por essas máquinas. Para rodar um job (isto é, um programa ou um conjunto de programas), primeiro o programador escrevia o programa no papel (em FORTRAN (FMS - Fortran Monitor System) ou na linguagem Assembly), e então perfurava-o em cartões. Daí, ele levava o *deck* de cartões à sala de recepção e o entregava a um dos operadores.

Quando o computador encerrava a execução de um *job*, um operador apanhava a saída na impressora, a conduzia de volta à sala de recepção onde o programador poderia coletá-lo posteriormente. Então ele tomava um dos *decks* de cartões que tinha sido trazido da sala de recepção e produzia a sua leitura. Se o compilador FORTRAN era necessário, o operador tinha que pegá-lo de uma sala de arquivos e produzir a sua leitura. Muito tempo de computador era desperdiçado enquanto os operadores caminhavam pela sala da máquina para realizarem essas tarefas.

Devido ao alto custo do equipamento, era de se esperar que as pessoas tentassem reduzir o tempo desperdiçado. A solução geralmente adotada era o sistema em "batch". A idéia original era colecionar uma bandeja completa de *jobs* na sala de recepção e então lê-los para uma fita magnética usando um computador pequeno e relativamente barato, por exemplo o IBM 1401, que era muito bom na leitura de cartões, na cópia de fitas e na impressão da saída, porém não era tão bom em cálculo numérico. Outros computadores, máquinas mais caras, tais como o IBM 7094, eram usados para a computação real. Essa situação é mostrada na Figura 82.

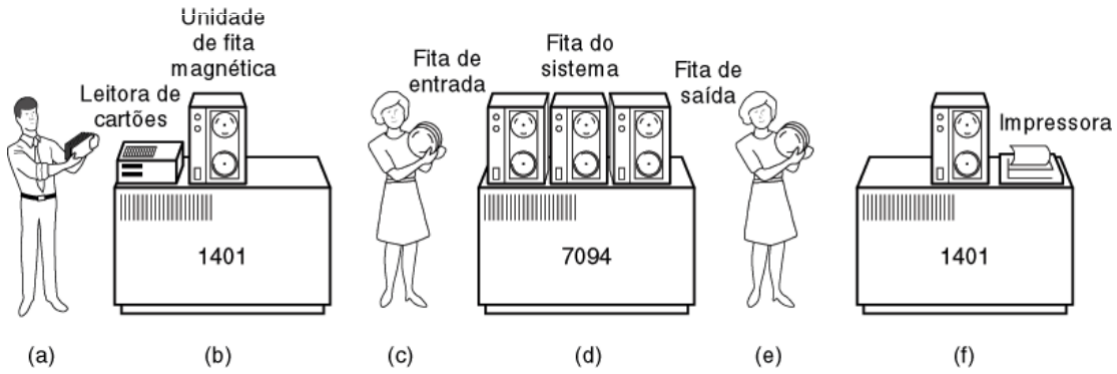


Figura 82 – Um sistem operacional bath - SO tipo lote.

Fonte: Operating Systems: Design And Implementation - Andrew S. Tanenbaum.

Após um tempo coletando um lote de *jobs*, a fita era rebobinada e levada para a sala da máquina onde era montada numa unidade de fita. O operador então carregava um programa especial (o antecessor do sistema operacional de hoje), que lia o primeiro *job* da fita e o executava. A saída era escrita numa segunda fita, em vez de ser impressa. Após o fim da execução de cada *job*, o sistema operacional automaticamente lia o próximo job da fita e começava a executá-lo. Quando todo o *batch* era feito, o operador removia as fitas de entrada e de saída, substituia a fita de entrada pelo próximo *batch* e levava a fita de saída para um IBM-1401 produzir a impressão *off-line* (isto é, não conectada ao computador principal).

A estrutura de um *job* (um programa de usuário) de entrada típico é mostrada na Figura 83.

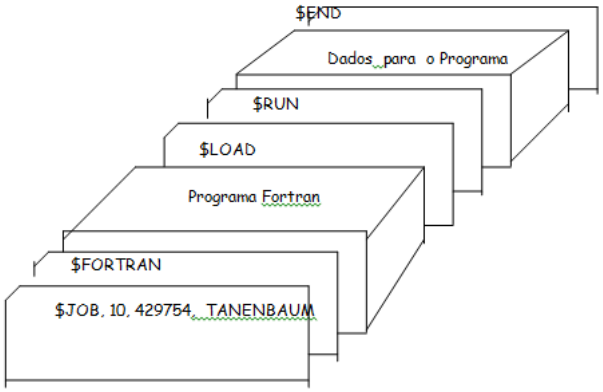


Figura 83 – Um deck de cartões perfurados organizando a estrutura de um job.

Fonte: Operating Systems: Design And Implementation - Andrew S. Tanenbaum.

9.16 A Terceira Geração - CIs e Multiprogramação (1965-1980)

Nos anos 60, muitos fabricantes de computadores tinham duas linhas de produto distintas e totalmente incompatíveis. Por um lado havia os computadores científicos, em grande escala, orientado por palavras, tais como o 7094, que era usado para cálculos numéricos em ciência e engenharia. Por outro lado, havia os computadores comerciais, orientados por caracter, tais como o 1401, que era vastamente usado para classificação em fita e impressão, por bancos e companhias de seguros.

O desenvolvimento e a manutenção de duas linhas de produto completamente diferentes era uma proposta cara para os fabricantes. A IBM, no intuito de resolver os problemas, introduziu o sistema 360. O 360 era uma série de máquinas compatíveis por software, variando de tamanho a partir do 1401 até o mais potente 7094. As máquinas diferiam apenas em preço e performance (capacidade de memória, velocidade do processador, número de periféricos I/O permitidos, e assim por diante). Já que todas as máquinas tinham a mesma arquitetura e o mesmo conjunto de instruções, pelo menos em teoria, programas escritos para uma máquina poderiam

rodar em todas as outras. Além disso, o 360 foi projetado para manusear tanto computação comercial como . Assim, uma única família de máquinas poderia satisfazer às necessidades de todos os clientes. O 360 foi a primeira linha de computadores a usar (em pequena escala) circuitos integrados (CIs), fornecendo uma maior vantagem em preço/desempenho sobre as máquinas da segunda geração, que eram construédas de transistores individuais. Isso foi um sucesso imediato e a idéia de uma família de computadores compatíveis foi logo adotada por todos os outros fabricantes. Nos anos seguintes, a IBM apresentou os sucessores compatíveis com a linha 360, usando uma tecnologia mais moderna, conhecidos como séries 370, 4300, 3080 e 3090. O autor conheceu os IBMs 1130, 360, o 370 e o 4341.

Apesar dos sistemas operacionais da terceira geração terem sido bem apropriados para a execução de programas envolvendo grandes cálculos científicos e de processamento de dados comerciais compactos, eles eram ainda, basicamente, sistemas em "*batch*".

A necessidade de se ter um tempo de resposta menor abriu espaço para "*time-sharing*", uma variante da multiprogramação, em que cada usuário tem um terminal "*on-line*". Embora a primeira série de sistemas em *time-sharing* (CTSS) tenha sido desenvolvida no MIT, num IBM 7094 especialmente modificado, ele não se tornou verdadeiramente popular até que a necessidade de proteção de hardware ficasse mais difundida durante a terceira geração. Após o sucesso do sistema CTSS, o MIT, o Laboratório Bell e a General Electric (então o maior fabricante de computadores) decidiram embarcar no desenvolvimento de um "computador utilitário", uma máquina que suportasse milhares de usuários em "*time-sharing*" simultaneamente. Os projetistas desse sistema, conhecido como MULTICS (*MULTiplexed Information and Computing Service*), tinham em mente uma grande máquina que fornecesse serviço de computação para todos. Mas a idéia de um computador utilitário falhou. Mesmo assim, o MULTICS teve uma enorme influência nos sistemas subsequentes.

Outro importante desenvolvimento durante a terceira geração foi o de mini-computadores, começando com o DEC PDP-1 em 1961. Para certos tipos de trabalhos não-numéricos era quase tão rápido quanto o 7094 e fez surgir uma nova indústria. Foi rapidamente seguido por uma série de outros PDPs (que diferentes da família IBM, eram todos incompatíveis) culminando com o PDP-11.

Um dos cientistas do Laboratório Bell que trabalhou no MULTICS, **Ken Thompson** (ver na seção 9.18), logo depois encontrou um pequeno PDP-7 sem uso e começou a escrever uma versão simplificada mono-usuário do MULTICS. **Brian Kernighan** apelidou esse sistema de UNICS (*UNiplexed Information and Computing Service*), mas sua grafia foi mais tarde trocada para UNIX. Posteriormente foi levado para um PDP-11/20, onde funcionou bem, o bastante para convencer o Laboratório Bell em investir no PDP-11/45 para continuar o trabalho. Outro cientista do Laboratório Bell, **Dennis Ritchie**, juntou-se a **Thompson** para reescrever o sistema numa linguagem de alto nível chamada C, projetada e implementada por **Ritchie**. O Laboratorio

Bell licensiava o UNIX para Universidades, quase gratuitamente e dentro de poucos anos, centenas delas estavam usando-o. O UNIX logo estendeu-se para muitos outros computadores, e logo já era o mais aplicado para computadores do que qualquer outro sistema operacional da história e seu uso está até os tempos atuais.

9.17 1968 - Programação Concorrente

O campo da *programação concorrente* iniciou uma explosiva expansão no final dos anos 60. Um programa ordinário consiste de declarações de dados e instruções executáveis em uma linguagem de programação. As instruções são executadas sequencialmente sobre um processador, o qual aloca memória o código e para os dados do programa. Um programa concorrente é um conjunto de programas sequenciais ordinários os quais são executados em uma abstração de paralelismo. Usamos a palavra *processo* para programas sequenciais e reservamos a palavra programa para o conjunto de processos.

Um breve histórico da pesquisa em programação concorrente para os primeiros computadores é agora contada. Em 1968, **E. W. Dijkstra**: Cooperando Processos Seqüenciais. Em 1971, **E. W. Dijkstra**: Ordem hierárquica de processos seqüenciais. Em 1973 **C. L. Liu** e **J. W. Layland**: Algoritmos de escalonamento para multiprogramação em ambiente de tempo real. Em 1974, **C. A. R. Hoare**: "Monitores, um conceito para estruturar sistemas operacionais". Em 1974, **Leslie Lamport**: "Uma nova solução para o problema da programação concorrente de Dijkstra". Em 1976, **J. H. Howard**: "Provando monitores". Também em 1976, **S. Owicki** e **D. Gries**: "Verificando propriedades de programas paralelos: uma abordagem axiomática". Neste caso, a lógica pode ser usada para expressar tais propriedade. Em 1977, **P. Brinch Hansen**: "A arquitetura de programas concorrentes".

Em 1978 **C. A. R. Hoare** (1934) fez o CSP ("*Comunicating Sequential Processes*"). Mais outras obras significativas foram nas seguintes áreas: os algoritmos de ordenação Quicksort e QuickSelect, a Lógica de Floyd-Hoare (um sistema formal com um conjunto de regras lógicas para um raciocínio rigoroso sobre a corretude na computação), a linguagem formal CSP (usada para especificar as interações entre processos concorrentes), que serviu de inspiração para a linguagem de programação Occam em máquinas de arquitetura paralela, a sincronização entre processos concorrentes utilizando o conceito de monitor (conceito que usa a ideia da ocultação de dados), a especificação axiomática de linguagens de programação,

Em 1978, **E. W. Dijkstra**(1930-2002), **Leslie B. Lamport** (1941-) é um cientista da computação Estadunidense, que juntamente com **A. J. Martin**, **C. S. Sholten** e **E. F. M. Steffens** criaram em cooperação o "*garbage collection*", meio de se coletar lixo em uma memória de computador. E em 1980, **E. W. Dijkstra** e **C. S. Sholten** estudaram sobre "Detecção de terminação".

Figura 84 – Charles A. R. Hoare - criou a ideia do monitor e a álgebra do CSP para especificar e provar sistemas concorrentes.

Fonte: www.en.wikipedia.org/.

Edsger Wybe Dijkstra (1930-2002) foi um cientista da computação Holandês conhecido por suas contribuições nas áreas de desenvolvimento de algoritmos e programas, linguagens de programação, sistemas operacionais e processamento distribuído.

Figura 85 – Edwin Dijkstra - A ideia dos semáforos.

Fonte: cacm.acm.org.

Dijkstra foi um dos membros mais influentes da computação geração fundadora da ciência. Entre os domínios em que suas contribuições científicas são fundamentais são projeto de algoritmo, linguagens de programação, projeto de programas, sistemas operacionais, processamento distribuído, especificação formal e verificação e, projeto de argumentos matemáticos. Além disso, **Dijkstra** foi intensamente interessados no ensino, e nas relações entre Ciência Computação acadêmica e da indústria de software. Durante seus mais de quarenta anos como um cientista de computação, que incluiu posições na academia e da indústria, as contribuições de *Dijkstra* trouxe-lhe

muitos prémios e distinções, incluindo maior honra da Ciência de Computação, o Prêmio Turing ACM.

Figura 86 – Leslie Lamport - Introduziu novos conceitos na ciência computacional, tais como relógios lógicos.

Fonte: https://pt.wikipedia.org/wiki/Leslie_Lamport.

Outros trabalhos relevantes - Em 1981, **G. Ricart** e **A. Agrawala**: "Um algoritmo ótimo para exclusão mútua distribuída". Também em 1981, **G. L. Peterson**: "O problema da exclusão mútua". Já em 1982, **J. Misra** e **K. M. Chandy**: "Detecção de terminação em Communicating Sequencial Processes". Em 1983, **G. L. Peterson**: "Uma nova solução para o prolema de programação concorrente de Lamport usando variáveis compartilhadas". Também em 1983, o *Department of Defense*, USA, criou a linguagem de programação Ada, para programar concorrentemente. Já em 1985, **D. Gelernter** criou a linguagem Linda, baseada num espaço de tuplas. Concorrente de Hoare, **Arthur John Robin Gorell Milner** (1934-2010), conhecido por **Robin Milner**, publicou CCS (Calculus of Communication Systems) e Communication and Concurrency em 1990.

Um programa concorrente é executado por se compartilhar o poder de processamento de um único processador entre os processos desse programa. A unidade de processamento concorrente é o processo. O paralelismo é abstrato porque não requeremos que um processador físico seja usado para executar cada processo, daí chamar-se de *pseudo-paralelismo*. São casos de concorrência, a **sobreposição de I/O e processamento** (Overlapped I/O and Computation), a **multiprogramação** (*multi-programming*), a **multi-tarefação** (*multitasking*). No início dos tempos dos primeiros SOs, controlar I/O não era feito concorrentemente com outra computação sobre um único processador. Mas a evolução do SOs, fez surgir a concorrência, retirando da computação principal, alguns microsegundos necessários para controlar I/O. Entretanto, era mais simples programar os controladores de I/O como processos separados, os quais são executados em paralelo com o processo de computação principal. Assim, era feito nos computadores de grande porte da época. Uma generalização de sobreposição de I/O dentro de um único programa é sobrepor a computação e I/O de diversos programas.

Figura 87 – Robin Milner - CCS, uma estrutura teórica para analisar sistemas concorrentes.

Fonte: Google Images - www.britannica.com.

Multiprogramação veio a ser a execução concorrente de diversos processos independentes sobre um processador. Surgiram os sistemas *time-slicing*, compartilhando o processador entre diversas computações de processos. Ao contrário do que um processo esperar para o término de uma operação de I/O, o processador era já compartilhado através de um hardware (*timer*) usado para interromper uma computação de um processo em intervalos pre-determinados (*time-slice*). Para tal, foi criada a ideia de um programa do SO chamado *Scheduler*, que até hoje, é executado para determinar qual processo deve ser permitido executar no próximo intervalo. Sistemas interativos com *time-slicing* usam multiprogramação com *time-sliced*, para dar a um grupo de usuários a ilusão que cada um tem acesso a um computador dedicado. O *Scheduler* pode também levar em consideração, prioridades dos processos.

Por causa das possíveis interações entre os processos que compreendem um programa concorrente é difícil escrever um programa concorrente correto. Para interagirem, processos precisam se sincronizar e se comunicar diretamente ou não, o que chamado de **sincronização de processos**. Resolvendo um problema por decomposição em diversos processos concorrentes, a execução de diversos aplicativos (programas) por um único usuário, eu uma máquina de um único processador pode ser, por exemplo como:

N: **Integer** := 0;

Process P1 **is begin**
N := N + 1;
end P1;

Process P2 **is begin**

N := N + 1;
end P2;

Se um compilador traduzir estas declarações de alto nível em instruções INC (incremento), como na Figura 88, qualquer intercalação das sequências de instruções dos dois processos darão o mesmo valor. Este programa sendo implementado em instruções INC de um linguagem "assembly" (montagem).

Se o mesmo programa for implementado usando os registradores de uma linguagem Assembly, como na Figura 89, algumas intercalações dão resposta errada.

Processo	Instrução	Valor de N
Inicialmente		0
P1	INC N	1
P2	INC N	2

Processo	Instrução	Valor de N
Inicialmente		0
P2	INC N	1
P1	INC N	2

Figura 88 – Programando concorrência com instruções INC.

Fonte: Ben-Ari, Principles of Concurrent Programming.

Processo	Instrução	N	Reg (P1)	Reg(P2)
Inicialmente		0		
P1	LOAD Reg, N	0	0	
P2	LOAD Reg, N	0	0	0
P1	ADD Reg, #1	0	1	0
P2	ADD Reg, #1	0	1	1
P1	STORE Reg, N	1	1	1
P2	STORE Reg, N	1	1	1

Figura 89 – Programando concorrência com instruções de *Assembly* em registradores.

Fonte: Ben-Ari, Principles of Concurrent Programming.

Então, é extremamente importante definir exatamente quais instruções são para ser

intercaladas, de forma de os *programas concorrentes* sejam corretos em sua execução. **Programação concorrente** pode expressar a concorrência requerida, provendo instruções de programação para a **sincronização** e **comunicação** entre processos.

Se um sistema concorrente é muito extenso, um programador pode ser totalmente confundido pelo comportamento que um programa concorrente pode exibir. Desta forma, ferramentas são necessárias para especificar, programar e verificar propriedades desses programas. A programação concorrente, estuda a abstração que é usada sobre as sequências de instruções atômicas de execução intercalada e define o que significa um programa concorrente ser correto e introduz os métodos usados para provar correção. Portanto, o aluno de graduação precisa aprender as primitivas e as estruturas de programação concorrente clássicas, controladas por semáforos, monitores e Locks, para poder sincronizar processos ou threads, num ambiente multithreading moderno.

Se dois processos rodam em máquinas distintas e eles devem ser comunicar para cooperar um com o outro, esses dois processos precisam se comunicar remotamente, em uma redes de computadores rede de computadores, através do mecanismo de *sockets* existentes em cada um processos.

9.18 O Surgimento do UNIX

Em 1965 formou-se um grupo de programadores, incluindo **Ken Thompson**, **Dennis Ritchie**, **Douglas McIlroy** e *Peter Weiner*, num esforço conjunto da AT&T (Laboratórios Bell), da General Electric (GE) e do MIT (Massachussets Institute of Technology) para o desenvolvimento de um sistema operacional chamado *Multics*.

O *Multics* deveria ser um sistema de tempo compartilhado para uma grande comunidade de usuários. Entretanto, os recursos computacionais disponíveis à época, particularmente os do computador utilizado, um GE 645, revelaram-se insuficientes para as pretensões do projeto. Em 1969, a Bell retirou-se do projeto. Duas razões principais foram citadas para explicar a sua saída. (1) Três instituições com objetivos díspares dificilmente alcançariam uma solução satisfatória para cada uma delas (o MIT fazia pesquisa, AT&T monopolizava os serviços de telefonia americanos e a GE (Gneral Electric) queria vender computadores). (2) A segunda razão é que os participantes influenciados pelos projetos anteriores, queriam incluir no *Multics* tudo que tinha sido excluído dos sistemas experimentais até então desenvolvidos.

Unix é um sistema operacional portável, multitarefa e multiusuário, originalmente criado por **Ken Thompson**, **Dennis Ritchie**, **Douglas McIlroy** e **Peter Weiner**, que trabalhavam nos Laboratórios Bell (Bell Labs) da AT&T. A marca UNIX é uma propriedade do The Open Group, um consórcio formado por empresas de Informática.

Ainda em 1969, **Ken Thompson**, usando um computador PDP-7 ocioso, começou

a reescrever o *Multics*, num conceito menos ambicioso, batizado de *Unics*, usando linguagem de montagem (assembly). Mais tarde, **Brian Kernighan** rebatizou o novo sistema de *Unix*.

Figura 90 – Dennys e Tom Tompson - A construção do sistema operacional UNIX em 1969.

Fonte: ACM - Association Computing Machinery.

Um marco importante foi estabelecido em 1973, quando **Dennis Ritchie** e **Ken Thompson** reescreveram o *Unix*, usando a linguagem C para um computador PDP-11. A linguagem C havia sido desenvolvida por **Ritchie** para substituir e superar as limitações da linguagem B, desenvolvida por **Thompson**. O seu uso é considerado uma das principais razões para a rápida difusão do Unix. Finalmente, ao longo dos anos 70 e 80 foram sendo desenvolvidas as primeiras distribuições de grande dimensão como os sistemas BSD (na Universidade de Berkeley na Califórnia) e os System III e System V (na AT&T).

Em 1977, a AT&T começou a fornecer o Unix para instituições comerciais. A abertura do mercado comercial para o *Unix* deve muito a **Peter Weiner** - cientista de Yale e fundador da Interactive System Corporation. **Weiner** conseguiu da AT&T, então já desnudada de seu monopólio nas comunicações e liberada para atuação no mercado de software, licença para transportar e comercializar o *Unix* para o computador Interdata 8/32 para ambiente de automação de escritório. O *Unix* saía da linha das máquinas PDP, da Digital Equipament Corporation (DEC), demonstrando a relativa facilidade de migração (transporte) para outros computadores, e que, em parte, deveu-se ao uso da linguagem C. O sucesso da Interactive de **Weiner** com seu produto provou que o *Unix* era vendável e encorajou outros fabricantes a seguirem o mesmo curso. Iniciava-se a abertura do chamado mercado *Unix*.

Com a crescente oferta de microcomputadores, outras empresas transportaram o *Unix* para novas máquinas. Devido à disponibilidade dos fontes do *Unix* e à sua simplicidade, muitos fabricantes alteraram o sistema, gerando variantes personalizadas a partir do *Unix* básico licenciado pela AT&T. De 1977 a 1981, a AT&T integrou muitas variantes no primeiro sistema *Unix* comercial chamado de System III. Em 1983, após acrescentar vários melhoramentos ao System III, a AT&T apresentava o novo *Unix* comercial, agora chamado de *System V*. Hoje, o *Unix* **System V** é o padrão internacional de fato no mercado *Unix*, constando das licitações de compra

de equipamentos de grandes clientes na América, Europa e Ásia.

Atualmente, *Unix* é o nome dado a uma grande família de sistemas operacionais que partilham muitos dos conceitos dos sistemas *Unix* originais (GNU/Linux, embora compartilhe conceitos de sistemas da família *Unix*, não faz parte desta família por não compartilhar de código derivado de algum sistema da família *Unix* e não possuir o mesmo objetivo e filosofia no qual o *Unix* se originou e, em grande parte, mantém até hoje), sendo todos eles desenvolvidos em torno de padrões como o POSIX (Portable Operating System Interface) e outros. Alguns dos sistemas operacionais derivados do Unix são: a família BSD (FreeBSD, OpenBSD e NetBSD), o Solaris (anteriormente conhecido por SunOS, criado pela ex-SUN), IRIX, AIX, HP-UX, Tru64, SCO, e até o Mac OS X (baseado em um núcleo Mach BSD). Existem mais de quarenta sistemas operacionais *nix, rodando desde celulares a supercomputadores, de relógios de pulso a sistemas de grande porte.

9.19 Anos 70 - O Conceito de Relação e a Criação dos Bancos de Dados Relacionais

Edgar Frank Codd (1923-2003) foi um matemático Britânico, inventor do banco de dados relacionais. Originado do conceito matemático do produto cartesiano entre conjuntos e de relação entre conjuntos, **Codd** desenvolveu o modelo de banco de dados relacional (baseado no conceito de relação), quando era pesquisador no laboratório da IBM em San José, USA. Em junho de 1970 publicou um artigo chamado "*Relational Model of Data for Large Shared Data Banks*" que foi publicado pela ACM (*Association Computing Machinary*). Este artigo (trabalho interno da IBM publicado no ano anterior, demonstrou os fundamentos da teoria dos bancos de dados relacionais, usando tabelas ("linhas" e "colunas") e operações matemáticas para recuperá-las destas tabelas (UNION, SELECT, SUM, etc).

Devido ao interesse da IBM em preservar o faturamento trazido por produtos pré-relacionais, esta não quis inicialmente implementar as idéias de **Codd**. Este então, buscou grandes clientes da IBM para mostrar-lhes as novas potencialidades de uma eventual implementação do modelo relacional. A pressão desses clientes sobre a IBM forçou-a a incluir, em seu projeto FS, o subprojeto System R com sua linguagem *SEQUEL*. Entretanto, a IBM entregou o projeto a um grupo de desenvolvedores que não "compreendera perfeitamente" as idéias de **Codd**, e isolou o mesmo do projeto. Assim, o produto final do projeto System R foi o *SQL*, linguagem de programação de dados que não é relacional, embora seja suficientemente próxima do modelo de **Codd**, teve várias vantagens sobre os sistemas pré-relacionais da IBM, tendo, assim, alcançado grande sucesso.

Em desgosto pela rejeição a suas idéias, **Codd** uniu-se a seu colega **Christopher J. Date** (1941-) da IBM para deixar a mesma, fundando uma consultoria chamada Codd & Date. Logo após adoeceu e teve de encerrar sua carreira, vindo a falecer em

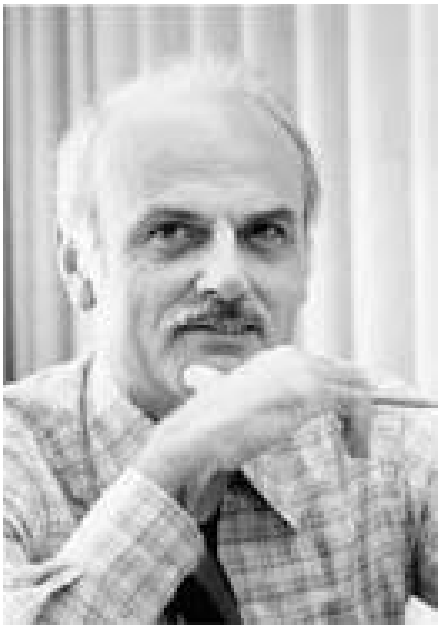


Figura 91 – Edgar Codd - O modelo de banco de dados relacional.

Fonte: en.wikipedia.org.

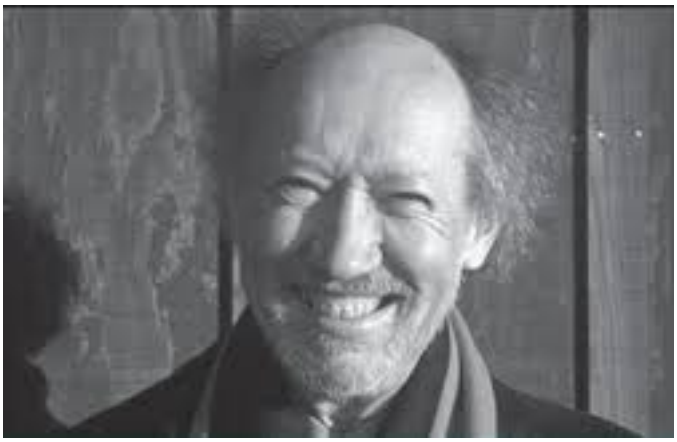


Figura 92 – Christopher J. Date - An Introduction to Database Systems. O continuador das ideias de Codd.

Fonte: Goolge Images - geekswithblogs.net.

2003. **Date** continuou a obra de **Codd**, tornando-se autor de vários livros e co-autor, com **Hugh Darwen**, da proposta de linguagem relacional *Tutorial D*. O modelo relacional chegou a ser implementado ainda em vida de **Codd**, pela própria IBM, num sistema de uso restrito, e por versões iniciais que implementavam a linguagem *QUEL*, baseada numa proposta de linguagem relacional *Alpha* de autoria de **Codd**. Mais recentemente, a linguagem quase-relacional D4 foi implementada, pela empresa de Utah Alphora, a qual criou o *Dataphor*, e um pesquisador inglês criou *Rel*, uma implementação Java do *Tutorial D*.

Álgebra Relacional e Linguagem de Consulta a Bancos de Dados Relacionais

A Álgebra Relacional é um modo teórico de se manipular um banco de dados relacional, de proporcionar uma linguagem de consulta procedural a usuários que especificam os dados necessários e como obtê-los. Consiste de um conjunto de operações de entrada: uma ou duas relações, e operações de saída: uma nova relação resultante.

Suas operações fundamentais tratam sobre: *seleção*, *projeção*, *produto cartesiano*, *renomeação*, *união* e *diferença de conjuntos*. Operações adicionais tratam sobre a *intersecção de conjuntos*, *junção natural*, *divisão* e *agregação*.

9.20 Surge o Minix

O *Minix* é um sistema operacional, escrito em linguagem C e assembly. Ele é gratuito e com o código fonte disponível. O *Minix* foi criado por **Andrew S. Tanenbaum** na Vrije Universiteit em Amsterdam para explicar os princípios dos seu livro-texto, "Operating Systems Design and Implementation" de (1987). Um abreviado das 12.000 linhas de código do Kernel, gerenciador de memória, e sistema de arquivo do MINIX 1.0 estão no livro. A sua editora, Prentice-Hall, também liberou o código fonte MINIX em disquetes com um manual de referência. *MINIX 1* possuia um sistema de chamada compatível com a VII edição do *UNIX*. O *Minix* pode funcionar com quantidades baixas de memória e disco rígido. O *MINIX 3* pode ser usado com apenas 16 MB de memória RAM e 50 MB de disco rígido, mas para instalação de outros software o recomendável é 600 MB de HD. É possível testar pelo Live CD, funcionando sem necessidade de instalação no HD.

O sistema operacional *Unix* foi concebido e implementado em 1969 pela AT&T Bell Laboratories nos Estados Unidos por **Ken Thompson**, **Dennis Ritchie**, **Douglas McIlroy**, e **Joe Ossanna**. Lançado pela primeira vez em 1971, o *Unix* foi escrito inteiramente em linguagem assembly uma prática comum para a época. Mais tarde, em uma abordagem pioneira em 1973, ele foi reescrito na linguagem de programação C por **Dennis Ritchie** (com exceções para o kernel e I/O). A disponibilidade de uma implementação do *Unix* feita em linguagem de alto nível fez a sua portabilidade para diferentes plataformas de computador se tornarem mais fácil. Na época, a maioria dos programas era escrita em cartões perfurados que tinham de ser inseridos

Figura 93 – Andrew S. Tanenbaum - O idealizador do Minix.

Fonte: Google Images - commons.wikimedia.org.

em lotes em computadores *mainframe*.

Apesar de não ser sido lançado até 1992 devido a complicações legais, o desenvolvimento de 386BSD, que veio a partir do NetBSD, OpenBSD e FreeBSD, antecedeu o Linux. **Linus Torvalds** dizia que, se o 386BSD estivesse disponível naquele momento, ele provavelmente não teria criado o Linux.

O núcleo Linux foi, originalmente, escrito por *Linus Torvalds* do Departamento de Ciência da Computação da Universidade de Helsinki, Finlândia, com a ajuda de vários programadores voluntários através da Usenet (uma espécie de sistema de listas de discussão existente desde os primórdios da Internet).

Linus Torvalds começou o desenvolvimento do núcleo como um projeto particular, inspirado pelo seu interesse no *Minix*, um pequeno sistema *UNIX* desenvolvido por **Andrew S. Tanenbaum**. Ele limitou-se a criar, nas suas próprias palavras, "um Minix melhor que o Minix" ("a better Minix than Minix"). E depois de algum tempo de trabalho no projeto, sozinho, enviou a seguinte mensagem para comp.os.minix:

"*Você suspira pelos bons tempos do Minix-1.1, quando os homens eram homens e escreviam seus próprios "device drivers"? Você está sem um bom projeto em mãos e deseja trabalhar num S.O. que possa modificar de acordo com as suas necessidades? Acha frustrante quando tudo funciona no Minix? Chega de noite ao computador para conseguir que os programas funcionem? Então esta mensagem pode ser exatamente para você. Como eu mencionei há um mês atrás, estou trabalhando numa versão independente de um S.O. similar ao Minix para computadores AT-386. Ele está, finalmente, próximo do estado em que poderá ser utilizado (embora possa não ser o que você espera), e eu estou disposto a disponibilizar o código-fonte para ampla distribuição. Ele está na versão 0.02... contudo eu tive sucesso ao executar bash, gcc, gnu-make, gnu-sed, compress etc. nele.*"

Curiosamente, o nome *Linux* foi criado por **Ari Lemmke**, administrador do site **ftp.funet.fi** que deu esse nome ao diretório FTP onde o núcleo Linux estava inicialmente disponível. **Linus Torvalds** inicialmente tinha-o batizado como "Freax".

No dia 5 de outubro de 1991 **Linus Torvalds** anunciou a primeira versão oficial do núcleo Linux, versão 0.02. No ano de 1992, **Linus Torvalds** mudou a licença do núcleo Linux, de uma licença própria para uma licença livre compatível com a GPL do projeto GNU.

Figura 94 – Linus Torvalds - O criador do LINUX.

Fonte: Google Images - www.somenek.com.br.

9.21 O Projeto GNU

Projeto GNU, em computação, é um projeto lançado em 27 de setembro de 1983 por **Richard Stallman** e atualmente a FSF (*Free Software Foundation*) é a principal organização que patrocina o projeto. Já na década de 1980, quase todo o software era proprietário, o que significa que ele possuía donos que proibiam e evitavam a cooperação dos usuários. Isso tornou o Projeto GNU necessário. O objetivo do projeto é criar um sistema operacional, chamado GNU, baseado em software livre. O nome "GNU" foi escolhido porque atende a alguns requisitos; em primeiro lugar, é um acrônimo recursivo para "GNU is Not Unix", depois, porque é uma palavra real. A palavra "livre" em "software livre" se refere à liberdade, não ao preço. Você pode ou não pagar para obter software do projeto GNU. De qualquer forma, uma vez que você tenha o software, você tem quatro liberdades específicas ao usá-lo: a liberdade de executar o programa como você desejar; a liberdade de copiá-lo e dá-lo a seus amigos e colegas; a liberdade de modificar o programa como você desejar, por ter acesso total ao código-fonte; a liberdade de distribuir versões melhoradas e, portanto, ajudar a construir a comunidade. Se alguém redistribuir software do projeto GNU, você pode cobrar uma taxa pelo ato físico de transferir uma cópia, ou você pode simplesmente dar cópias de graça.

Devido a um caso antitruste que a proibia de entrada no negócio de computadores, a AT&T foi obrigada a licenciar o código fonte do sistema operacional para quem quisesse. Com o resultado, o *UNIX* cresceu rapidamente e se tornou amplamente adotado por instituições acadêmicas e diversas empresas. Em 1984, a AT&T se desfez da Bell Labs; livres da obrigação legal exigindo o licenciamento do royalty, a Bell Labs começou a vender o *UNIX* como um software proprietário.

O Projeto GNU, iniciado em 1983 por **Richard Stallman**, teve o objetivo de criar um "sistema de software completamente compatível com o Unix", composto inteiramente de software livre. O trabalho começou em 1984. Mais tarde, em 1985, **Stallman** começou a Free Software Foundation e escreveu a Licença Pública Geral GNU (GNU GPL) em 1989. No início da década de 1990, muitos dos programas necessários em um sistema operacional (como bibliotecas, compiladores, editores de texto, uma Unix shell, e um sistema de janelas) foram concluídos, embora os elementos de baixo nível, como *drivers* de dispositivo, *daemons* e as do kernel foram paralisadas e não completadas.

Figura 95 – Richard Stallman - O criado do Projeto GNU.

Fonte: Google Images - www.thehindu.com.

9.22 A Quarta Geração - Microprocessadores e Computadores Pessoais (1977-1991)

Com o desenvolvimento de circuitos LSI (*Large Scale Integration*), *chips* contendo milhares de transistores em um centímetro quadrado de silício, a era do computador pessoal começava. Em termos de arquitetura, os computadores pessoais não eram diferentes dos minicomputadores da classe do PDP-11, mas em termos de preço eles

eram certamente bem diferentes. Enquanto o minicomputador tornou possível um departamento de uma companhia ou uma universidade ter o seu próprio computador, os com micropocessador em chip, tornou possível uma pessoa ter o seu próprio computador.

A grande variedade de capacidade computacional disponível, especialmente a capacidade de computação altamente interativa com excelentes facilidades gráficas, fizeram crescer a indústria de produção de software para computadores pessoais. Muitos desses softwares eram "amigáveis ao usuário", significando que eles foram projetados para usuários que não tinham conhecimento algum sobre computadores e além do mais não tinha outra intenção a não ser a de orientá-los no uso. Essa foi certamente a maior mudança do OS/360, cujo JCL era tão complexo que livros inteiros foram escritos sobre ele.

Dois sistemas operacionais dominaram a utilização do computador pessoal: o MS-DOS, escrito pela Microsoft para o IBM PC e para outras máquinas que usavam a CPU Intel 8088 e seus sucessores, e *UNIX*, que é predominante em máquinas que usam a CPU da família Motorola 68000. Pode parecer irônico que o descendente direto do *MULTICS*, projetado para o gigante computador utilitário, ficou tão popular em computadores pessoais, mas principalmente mostra como foram boas as idéias sobre o MULTICS e o UNIX. Apesar da primeira versão do MS-DOS ser primitiva, em versões subsequentes foram incluidas diversas facilidades do *UNIX*.

Nos anos 80, com o surgimento dos computadores pessoais, houve grande investimento por parte das corporações, nesses sistemas, em função das tarefas dos seus desenvolvedores da época. Entretando, em pouco tempo descobriu-se que o desempenho retornado pelos desevolvedores, não estava sendo compatível com tal grau de investimento. Este fato fez surgir as redes de computadores, para melhorar o desempenho do desenvolvedor e dimnuir custos quanto ao compartilhamento dos recursos computacionais. Desta forma, um interessante desenvolvimento que começou em meados dos anos 80 foi o crescimento de redes de computadores rodando *sistemas operacionais para rede*.

Num sistema operacional para rede, os usuários têm consciência da existência de múltiplos computadores e podem se conectar com máquinas remotas e copiar arquivos de uma máquina para outra. Cada máquina roda o seu próprio sistema operacional local e tem o seu próprio usuário (ou usuários). Os sistemas operacionais em rede não são fundamentalmente diferentes dos sistemas operacionais de um único processador. Eles obviamente necessitam de um controlador de interface de rede e de algum software de alto nível para gerenciá-lo, bem como de programas para concluir com êxito uma conexão remota e o acesso a arquivos remotos, mas essas adições não mudam a estrutura essencial do sistema operacional.

Com as redes, a partir dos anos 80, os sistemas distribuídos foram sendo desenvolvidos e *sistemas operacionais distribuídos* passaram a ser usados. Um sistema operacional

distribuído, em contraste, aparece para o usuário como um sistema tradicional de um único processador, mesmo sendo composto realmente de múltiplos processadores. Num verdadeiro sistema distribuído, os usuários não têm consciência de onde os seus programas estão sendo rodados ou onde seus arquivos estão localizados; tudo é manuseado automática e eficientemente pelo sistema operacional. Os sistemas operacionais distribuídos requerem mais do que a adição de códigos a um sistema operacional de um processador, porque sistemas distribuídos e centralizados diferem em modos críticos. Sistemas distribuídos, por exemplo, frequentemente admitem rodar programas em vários processadores remotos, ao mesmo tempo, e daí exigem algoritmos de escalonamento de processadores para otimimizar a quantidade de paralelismo que deve ser concluído com êxito. O atraso de comunicação em uma rede frequentemente significa que o tempo em um sistema distribuído, não seja conveniente ser tratado como tempo físico. Essa situação é radicalmente diferente de um sistema de um único processador no qual o sistema operacional tem a informação completa sobre o estado do sistema. Tolerância a falhas é uma outra área em que os sistemas distribuidos são diferentes. É comum para um sistema distribuido ser projetado com a expectativa de que continuará rodando mesmo que parte do hardware deixe de funcionar. Essa exigência adicional ao projeto tem enormes implicações para o sistema operacional.

Em 1980, três tendências convergiram: microcomputadores de alta performance, redes de alta velocidade, e ferramentas padronizadas para computação distribuída de alto desempenho. Essas convergências foram fundamentais para o aprofundamento nas técnicas de sistemas paralelos. A necessidade de alto poder de processamento, seja para aplicações científicas ou para outras aplicações que exijam tal poder de processamento. A implementação de tais aplicações, só foi possível com a aquisição dos supercomputadores. Estes ainda continuavam com preços muito elevados. Com a tecnologia de **cluster** obtém-se alto poder de processamento com baixo custo.

No final de 1993, cientistas da NASA começaram a planejar um sistema de processamento distribuído com computadores *Intel Desktop*. Seu objetivo era fazer com que esses computadores conectados em rede se assemelhassem ao processamento de um supercomputador, e que teriam um menor custo. No início de 1994, já trabalhando no CESDIS *Center of Excellence in Space Data and Information Sciences*, criaram o projeto **Beowulf**. Com 16 computadores Intel 486 DX4 conectados em uma rede Ethernet com sistema operacional Linux fizeram esse primeiro *cluster* chegar a um desempenho de 60 Mflops. A partir daí o projeto *Beowulf* tornou-se de grandes interesses pelos meios acadêmicos, pela NASA entre outros segmentos Tanembaum (2010).

9.22.1 Anos 90 - Programação Orientada a Objeto

Nos sistemas operacionais antigos (como o *UNIX* original), o usuário só podia se comunicar com a máquina, através de uma linha de comandos indicada por um *prompt*. A partir dos anos 90, nos sistemas operacionais mais modernos, esses já proporcionam

uma interface gráfica, pela qual apresenta ícones de programas utilitários, aplicativos e arquivos. Nessas interfaces, o usuário instancia janelas e mais janelas, as vezes até umas sobre as outras. Isto é possível, se o sistema operacional tiver sua interface com o usuário, utilizando o paradigma orientado a objetos. Oriunda da teoria dos **iipos**, o paradigma orientado a objetos, é nos dias de hoje, muito utilizado para a contrução de programas aplicativos, por parte das empresas de Tecnologias da Informação (TI). Com este paradigma podemos criar vários objetos a partir de uma definição de classe (que é um tipo). Um tipo suportado por todas as linguagens orientadas a objeto. O que ocorre em termos de matemática, mais precisamente da teoria dos tipos, é que janelas são objetos criados, que são objetos **equivalentes**, formados (instanciados) a partir de uma mesma classe, formando assim, uma **classe de equivalência** de objetos.

9.22.2 Programação Paralela e Distribuída

O campo da *programação paralela e distribuída* surgiu do campo da programação concorrente. Com a evolução dos microprocessadores em termos de rapidez, a ideia de dividir mais ainda um programa concorrente, fez surgir processos construídos por várias *threads* (linhas de execução internas a processos). Agora, a unidade de programação concorrente passa a ser a *thread*. As linguagens de programação modernas passam a suportar o que se chama de *multithreading*, podendo-se então programar paralelamente usando-se os núcleos internos aos processadores mais modernos (*multicore programming*). Além disso, para processamento paralelo pesado, a programação paralela, hoje, passa por se usar para esse tipo de programação, placas de hardware com diversos processadores paralelos (*many-core programming*) usadas para programação paralela de aplicações gráficas (GPU), como os modernos jogos de computador. Assim, numa placa GPU pode programar, começando-se com a unidade de programação paralela proporcionada por uma thread, pode-se agrupar threads em blocos de threads, e pode-se agrupar blocos de threads de do que se chama uma *grid* de blocos de threads. Aplicações modernas que exigem alto desempenho, fazem uso da programação paralela e, quando necessário, pode ser distribuida na rede.

9.23 Os Projetos e os Fracassos da Quinta Geração

Os computadores a válvulas foram chamados de computadores da primeira geração. Depois, com o aparecimento dos díodos e transístores surge a segunda geração, com o circuito integrado nasce a terceira geração e com o surgimento do microprocessador, deu-se o nome de quarta geração.

Visto que a anterior geração de computadores (quarta geração) tinha como objetivo o aumento do número de elementos lógicos num único CPU, acreditava-se plenamente que a quinta geração iria alcançar completamente a utilização de um grande números de CPUs para um desempenho maior das máquinas.

Através destas várias gerações, e a partir dos anos 50, o Japão tinha sido apenas mais um país na retaguarda de outros, em termos de avanços na tecnologia da computação, construindo computadores seguindo os modelos americano e inglês. Entretanto, em meados dos anos 70, deu-se início a uma visão em pequena escala para o futuro da computação. O Centro Japonês para o Desenvolvimento do Processamento da Informação deveria indicar os caminhos a seguir, e em 1979 ofereceu um contrato de 3 anos, objetivando projetos conjuntos com a indústria e a academia. Foi durante este período que o termo "quinta geração" começou a ser utilizado.

A ideia de que a computação paralela seria o futuro dos ganhos de performance estava tão enraizada que o projeto da quinta geração gerou uma grande apreensão no campo da computação. Após o mundo ter visto o Japão dominar a indústria da eletrônica nos anos 70, não tardou para que projetos de computação paralela fossem implementados também em outros países, como nos Estados Unidos, através da empresa *Microelectronics and Computer Technology Corporation* (MCC), na Inglaterra através da *Alvey* ou através do *European Strategic Program of Research in Information Technology* (ESPRIT).

Nos dez anos seguintes, o projeto da quinta-geração enfrentou dificuldades sobre dificuldades. O primeiro problema era no fato da linguagem de programação escolhida, Prolog, não oferecer suporte para concorrências. Outro problema foi o fato de que a performance das CPUs existentes levaram o projeto a dificuldades inerentemente técnicas, e o valor da computação paralela caiu rapidamente. Para contornar o problema, estações de trabalho com capacidades superiores foram idealizadas e construídas ao longo da duração do projeto. A quinta-geração acabaria por ficar também no lado errado da evolução de tecnologia do software.

Durante o período de desenvolvimento do projeto da quinta geração, a **Apple Computer** introduziu o que é hoje, GUI (Interface Gráfica do Usuário) através do seu sistema operacional voltado amigavelmente ao usuário, como temos atualmente. Por outro lado, a Internet (1992) fez com que as bases de dados armazenadas localmente começacem a se tornar algo do passado e, até mesmo melhores resultados surgiram em termos através de buscas de dados, como prova o exemplo da Google e o vários sistemas de busca de informações.

Finalmente o projeto chegou à conclusão de que as promessas baseadas na programação lógica eram largamente uma ilusão e rapidamente chegaram ao mesmo tipo de limitações que os investigadores de inteligência artificial já antes haviam encontrado embora numa escala diferente.

De fato, pode-se dizer que o projeto se desnorteou como um todo. Foi durante esta época que a indústria informática passou o seu foco principal do hardware para o software. A quinta-geração nunca chegou a fazer uma separação clara, ao acreditar que, tal como se idealizava nos anos 70, o software e o hardware eram inseparáveis.

No final do período de dez anos haviam sido gastos muitos bilhões e o programa terminou sem ter atingido as suas metas. As estações de trabalho não tinham procura comercial num mercado onde os sistemas de uma única CPU, facilmente, lhes ultrapassavam em performance, e o conceito geral ficou completamente em desuso com o amadurecimento e expansão da Internet.

Aqui cabem duas observações: (1) o advento das redes locais (anos 80) evoluiu, posteriormente, ao sistemas cliente-servidor locais, e à construção dos *clusters* de computadores visando computação paralela; (2) o advento da Internet (1992) fez com que a computação evoluisse do sistema cliente-servidor básico, passando pela evolução da Web inicial para o sistemas Web Services, que por sua vez evoluiu para o provimento de serviços através de *grids* computacionais, as quais evoluiram para os sistemas de computação em nuvem atuais; (3) do ponto de vista da computação paralela, a partir de 2002, a evolução do hardware passou das CPUs de poucos núcleos para as tecnologias GPU (*Graphics Processing Unit*) - a computação com aceleração de GPU é a utilização de uma GPU em conjunto com uma CPU para acelerar aplicativos - e FPGA (*Field Programmable Gate Array*) - um circuito integrado projetado para ser configurado por um consumidor ou projetista após a fabricação - que surgiu então, como uma categoria nova de hardware reconfigurável, o qual têm as suas funcionalidades definidas exclusivamente pelos usuários e não pelos fabricantes. CPUs trabalhando em conjunto com GPUs ou FPGAs, formam os ambientes de programação paralela atuais.

9.24 Resumindo as gerações ...

Os últimos 65 anos da Ciência da Computação foram divididos em seis grandes tendências: (1) a **primeira geração** (1946-1954) caracterizada pelas válvulas; (2) a **segunda geração** (1955-1964) caracterizada pelo transistores; (3) a **terceira geração** (1965-1977), com os circuitos integrados feitos de silício nas escalas iniciais SSI (*Short Scale Integration*), MSI (*Medium Scale Integration*), LSI (*Large Scale Integration*) conhecidos como *microchips* integrando um grande número de transistores, o que possibilitou a construção de equipamentos menores e mais baratos, e a multiprogramação de tempo compartilhado nos mainframes com muitas pessoas compartilhando um computador, quando então surgiu o teleprocessamento com terminais remotos sem inteligência (9.16); (4) a **quarta geração** (1977-1991), da escala de integração VLSI (*Very Large Scale Integration*), dos computadores pessoais com um computador para cada usuário, e os sistemas operacionais como MS-DOS, UNIX, Apples Macintosh, linguagens de programação orientadas a objeto como C++ e Smalltalk, discos rígidos utilizados como memória secundária, impressoras matriciais, teclados com os layouts atuais, e quando surgiu a era das redes locais e de longa distância; (5) a **quinta geração** (1992 em diante), caracterizada pela escala de integração ULSI (*Ultra Large Scale Integration*), arquiteturas de 64 bits, processadores que utilizam tecnologias RISC e CISC, a utilização da inteligência artificial (reconhecimento de voz, sistemas inteligentes, redes neurais, robótica, e a

conectividade proporcionada pelas redes de alta velocidade; e a (6) **sexta geração**, a que estamos vivenciando, caracterizada pela *nuvem computacional*, ao mesmo tempo com a *computação móvel*, *pervasiva* e *ubíqua*.

9.25 Bibliografia e Fonte de Consulta

Museu Virtual - The Baby Machine (Manchester Mark I) - ⟨http://piano.dsi.uminho.pt/museuv/1946mmark1.html⟩

Transistor - https://pt.wikipedia.org/wiki/Transístor

Francisco Romero Feitosa Freire (1993). Pró-Censo (com o UNIVAC). Visitado em 30 de outubro de 2011.

Walter Isaacson - Os Inovadores - Uma biografia da revolução digital. Companhia das Letras. 2014.

Operating Systems: Design And Implementation - Andrew S. Tanenbaum

M. Ben-Ari - Principles of Concurrent and Distributed Programming: Algorithms and Models (Prentice-Hall International Series in Computer Science), 1 Ed. 1990, 2 Ed. 2006.

https://www.inf.pucrs.br/ rvieira/cursos/lac/RelFReAB.pdf

Banco de Dados Álgebra Relacional e SQL, Profa. Cristina Dutra de Aguiar Ciferri.

Algebra Relacional e SQL - http://wiki.icmc.usp.br/images/2/2c/SCC578920131-algebraSQL.pdf

Donald Knuth: Leonard Euler of Computer Science - Chapter 1 em ⟨http://www.softpanorama.org/People/Knuth/index.shtml⟩

Negus, Christopher; Christine Bresnahan. In: Starlin Alta Editora e Consultoria. Linux - A Bíblia. 8 ed. Rio de Janeiro: [s.n.]. 9788576087991.

Origins and History of Unix, 1969-1995 (em inglês) faqs.org. Visitado em 05/07/2015. Cópia arquivada em 2015.

Initial Announcement (em inglês) gnu.org. Visitado em 05/07/2015. Cópia arquivada em 2015.

Overview of the GNU System (em inglês) gnu.org. Visitado em 05/07/2015. Cópia

arquivada em 2015.

The Choice of a GNU Generation An Interview With Linus Torvalds (em inglês) gondwanaland.com. Visitado em 14/07/2015. Cópia arquivada em 2015.

What would you like to see most in minix? groups.google.com. Visitado em 14/07/2015.

Primeiro período do original, em inglês: Do you pine for the nice days of minix-1.1, when men were men and wrote their own device drivers?

Lars Wirzenius (27 de abril de 1998). Linux Anecdotes. Visitado em 15 de junho de 2015.

Carlos E. Morimoto. Linux, Ferramentas Técnicas 2ed (em Português). 2 ed. [S.l.]: GDH Press e Sul Editores, 2006. 312 p. ISBN 85-205-0401-9.

Release notes for the version 0.12 of the Kernel Linux Linus (23/06/1993 às 00:00). Visitado em 14/04/2015.

O Projeto e História do UNIX - https://pt.wikipedia.org/wiki/Unix

O Projeto Linux - https://pt.wikipedia.org/wiki/Linux

O Projeto GNU - https://pt.wikipedia.org/wiki/Projeto_GNU (FSF).

Computação da Quinta Geração - https://pt.wikipedia.org/wiki/Computação_da_quinta_geração

www.icot.or.jp - O que são tecnologias de quinta geração? - Instituto para a Nova Geração de Tecnologias de Computação (ICOT)

www.useit.com - The Fifth Generation Computing Conference Report.

www.atarimagazines.com - The Fifth Generation: Japan's Computer Challenge to the World, artigo de 1984 da *Creative Computing*.

Marilza L. Cardi - Evolução da computação no Brasil e sua relação com fatos internacionais. Programa de Pós-Graduação em Computação, Universidade Federal de Santa Catarina, Florianópolis. 2002. Orientador: Jorge Muniz Barreto.

9.26 Referências e Leitura Recomendada

Walter Isaacson - Os Inovadores - Uma biografia da revolução digital, Companhia da Letras, 2014.

O computador analógico MONIAC - ⟨lhttps://en.wikipedia.org/wiki/MONIAC_Computer⟩.

Milestones: Atanasoff-Berry Computer, 1939 IEEE Global History Network IEEE. Visitado em 3 August 2011.

Campbell-Kelly, Martin; Aspray, William (1996), Computer: A History of the Information Machine, New York, NY: Basic Books, p. 84, ISBN 0-465-02989-2.

Copeland, Jack (2006), Colossus: The Secrets of Bletchley Park's Codebreaking Computers, Oxford: Oxford University Press, pp. 101115, ISBN 0-19-284055-X.

Lundstrom, David E.. A Few Good Man from Univac. Cambridge: MIT Press, 1987. 5 p.ISBN 0-262-12120-4.

A Calculus of Communicating Systems, Robin Milner. Springer-Verlag (LNCS 92), 1980. ISBN 3-540-10235-3.

Robin Milner - Communication and Concurrency, Prentice Hall International Series in Computer Science, 1989. ISBN 0-13-115007-3.

C. A. R. Hoare. Communicating Sequential Processes. [S.l.]: Prentice Hall International Series in Computer Science, 1985. ISBN 0-13-153271-5 hardback or ISBN 0-13-153289-8.

O.-J. Dahl, E. W. Dijkstra e C. A. R. Hoare. Structured Programming. [S.l.]: Academic Press, 1972. ISBN 0-12-200550-3.

C. A. R. Hoare. Communicating Sequential Processes. [S.l.]: Prentice Hall International Series in Computer Science, 1985. ISBN 0-13-153271-5 hardback or ISBN 0-13-153289-8.

C. A. R. Hoare e M. J. C. Gordon. Mechanised Reasoning and Hardware Design. [S.l.]: Prentice Hall International Series in Computer Science, 1992. ISBN 0-13-572405-8.

C. A. R. Hoare e He Jifeng. [S.l.]: Prentice Hall International Series in Computer Science, 1998. ISBN 0-13-458761-8.

R. W. Floyd. "Assigning meanings to programs." Proceedings of the American Mathematical Society Symposia on Applied Mathematics. Vol. 19, pp. 1931. 1967.

C. A. R. Hoare. "An axiomatic basis for computer programming". Communications of the ACM, 12(10):576580,583 October 1969. doi:10.1145/363235.363259.

Gonick, Larry. Introdução Ilustrada à Computação. São Paulo: Harper Row do Brasil, 1984. 242 p. p. 34-35.

CAPÍTULO 10

Sobre a Teoria da Complexidade

O riunda da **teoria da computabilidade** (**Turing**, **Church**, **Kleene**, **Post**, ...), surge a **teoria da complexidade**. Como em Sipser (2011) Cap.0 p.1, os problemas computacionais vem em diferentes variedades: alguns são fáceis e outros difíceis. Quatro exemplos de problemas são mencionados a seguir:

1. (a) O problema da ordenação de números: dado como um problema fácil, pois mesmo com um computador de pequeno porte, pode-se ordenar um mihão de números de forma rápida, num tempo de execução aceitável.

2. (b) O problema do escalonamento: como encontrar o escalonamento, quanto a alocação de salas, para uma Universidade inteira, que satisfaça algumas restrições, como duas aulas não poderem ter a mesma sala, ao mesmo tempo. Este é muito mais difícil. Se tivermos 1000 aulas, encontrar o melhor escalonamento pode requerer séculos, até mesmo com um supercomputador.

3. (c) Um outro exemplo muito difícil computacionalmente, é o problema da fatoração de um número natural muito grande, decomposto em fatores de números primos, usado em algoritmos de criptografia.

4. (d) Outro problema computacionalmente difícil é o do cálculo de logarimos discretos, utilizado em algoritmos de criptografia.

"O que faz alguns problemas computacionalmente difíceis e outros fáceis?"

Esta é a questão fundamental da teoria da complexidade. Embora, se tenha sido bastante pesquisada nos últimos 40 anos, ainda não sabemos a resposta para esta questão. O que se conseguiu foi um modo de se classificar os problemas conforme sua dificuldade computacional. Usando um esquema de classificação, pode-se demonstrar um método para dar evidência da complexidade computacional, de que certos problemas são difíceis, mesmo que sejamos incapazes de provar que eles são.

A *teoria da computabilidade* e a *teoria da complexidade* estão relacionadas. Na teoria da computabilidade, o objetivo é classificar os problemas como solúveis ou não são solúveis. Já na teoria da complexidade, o objetivo é classificar os problemas como fáceis e difíceis. Na maioria das áreas um problema computacionalmente fácil é preferível. Mas, mesmo os problemas difíceis tem espaço de uso em determinadas áreas, como, por exemplo, a área antiga da criptografia, que requer especificamente problemas computacionalmente difíceis, porque mensagens cifradas (secretas) precisam ser difíceis de quebrar (chaves secretas). A teoria da complexidade acaba por mostrar aos criptógrafos, o caminho dos prolbemas computacionalmente difíceis. Assim, novos algoritmos criptográficos podem ser projetados.

Para um texto mais completo, o leitor pode expandir seus conhecimentos lendo a parte três sobre a teoria da complexidade em Sipser (2011)(Parte 3, Cap.7-10) ou Sudkamp (1988) (Parte IV, Decidability and Compputability, Cap.14 sobre Computational Complexity).

10.1 Teoria da Complexidade - Histórico da Pesquisa

Começamos relembrando, **David Hilbert** no final da década dos anos 20 e **Turing** em meados dos anos 30.

Em 1928, na conferência international de matemática, **David Hilbert** propôs três questões, que mais tarde tiveram importância no início da pesquisa para a computação. A terceira delas ficou conhecida como "*O Problema de Decisão de Hilbert*".

Relembrando **Gödel**, em 1931, ele publica o trabalho "*On Formally Undecidable Propositions of Principia Mathematica and Related Systems*". **Gödel** prova que em um sistema lógico formal existem afirmações verdadeiras que não podem ser provadas. Mesmo lidando com aritmética, o sistema axiomático que a inclui não pode ser simultaneamente completo e consistente. Isto significa que, se o sistema é consistente, então existirão proposições que não poderão ser nem comprovadas nem negadas por este sistema axiomático. Se o sistema for completo, então ele não se poderá validar a si mesmo sendo, portanto, inconsistente.

Relembrando **Turing** (Capítulo 4), em 1936, ele propôs uma máquina, que passaria a ser chamada na literatura de máquina de **Turing**, imaginada como um modelo para a possível máquina programável a ser construída. Até aquele momento, é a formalização mais convincente da noção de uma função computável algoritmicamente, como **Turing** e **Church** definiram . Com a máquina abstrata já consolidada, em 1937, **Turing** publicou o trabalho "*On Computable Numbers with an Application to the Entscheidungsproblem*", com o termo em alemão: *Entscheidungsproblem* significando "**problema de decisão**". Como consequência, surgiram as provas de impossibilidade relacionadas à computação de determinadas tarefas.

- **Turing** reformula os resultados obtidos por **Gödel** no célebre *Entscheidungsproblem*: "qualquer cálculo que possa ser feito por um ser humano poderá ser feito por este tipo de máquina".
- **Turing** prova que nenhum algoritmo pode decidir num número finito de passos, se uma formula arbitrária do cálculo dos predicados é satisfatível.
- Além disso, **Turing** prova que o "**Problema da Parada**" é indecidível, ou seja, não é possível decidir algoritmicamente se a máquina de Turing irá parar ou não e, assim, provou-se que não há solução para o *Entscheidungsproblem*.

Com a indecibilidade definida, dada a máquina de Turing, com um modelo computacional muito bem definido, e uma teoria associada para explicar que problemas podem ou não, serem resolvidos por ela, surge naturalmente a próxima questão:

"**Qual é a dificuldade computacional de computar funções?**"

Este é o tema fundamental da complexidade computacional.

Michael O. Rabin nasceu na Alemanha em 1931, e mudou-se com seus pais para a Palestina em 1935. Aos 11 anos interessou-se por ler livros de matemática. **Rabin** terminou o ensino médio com 16 anos. Como a maioria de seus colegas, ele foi então convocado para o exército para lutar pela independência do então novo Estado de Israel. Um dos livros que leu foi de **Abraham Fraenkel**, em Jerusalém, sobre teoria dos conjuntos, e **Rabin** tanto se interessou que escreveu para o autor. **Fraenkel**, impressionado com a profundidade da correspondência, se reuniu com **Rabin**, e mais tarde ele foi estudar na Universidade de Jerusalém. Ele foi admitido diretamente no programa de mestrado para estudar álgebra, e se formou em 1953. Sua tese resolveu um problema em aberto, bastante significativo, que tinha sido proposto pelo matemático alemão **Emmy Noether**. Com a importância da tese, ele foi aceito em um programa de doutorado em Princeton, onde estudou sob a orientação de **Alonzo Church** e se formou em 1957.

Depois que **Rabin** concluiu seu doutoramento, ele foi convidado pela IBM para participar de um *workshop* de pesquisa para um grupo de jovens cientistas. Foi lá que ele e **Dana Stewart Scott** (1932-) escreveram e colaboraram no famoso artigo "*Autômatos Finitos e seus Problemas de Decisão*", que o levou a receber o **Prêmio Turing** em 1976. **Dana Stewart Scott** é um matemático, lógico, informático e filósofo Estadunidense. Mais tarde, nos anos 1970-1982, **Dana Scott** trabalhou em vários temas de pesquisa como *lattices*, teoria matemática da computação, tipos de dados, λ-Cálculo, conjuntos ordenados e semântica denotacional. **Dana S. Scott** contribuiu, significativamente, para a teoria dos autômatos, teoria dos modelos e semântica de linguagens de programação.

Rabin também é conhecido por seu trabalho em criptologia em conexão com os números primos e e no âmbito da teoria dos autômatos. Sua filha **Tal Rabin** dirige o

Grupo de Pesquisas sobre Criptologia e Privacidade no Centro de Pesquisas **Thomas J. Watson** da IBM.

Figura 96 – Michael Rabin - Autômatos Finitos e seus Problemas de Decisão

Fonte: http://amturing.acm.org/award_winners/rabin_9681074.cfm.

Figura 97 – Dana Scott - Teoria dos autômatos, teoria dos modelos e semântica de linguagens.

Fonte: Google Images - www.heidelberg-laureate-forum.org.

Em 1959/1960, 22 anos depois de **Turing** levantar as primeiras questões, **Michael O. Rabin** publicou os trabalhos:

- (a) *Speed of Computation and Classification of Recursive Sets.*
- (b) *Degree of Difficulty of Computing a Function and a Partial Ordering.*

10.2 Gödel e a Teoria da Complexidade

Como em Zach (2006), um outro trabalho de **Gödel**, desempenhado um papel no desenvolvimento e na aceitação gradual da Tese de Church (embora o próprio **Gödel**

aparentemente se convenceu da verdade da tese somente através de trabalho de **Turing**), foi um *abstract* sobre o comprimento de provas (Gödel, 1936). Ao afirmar sua tese, **Church** (1936b) introduziu a noção de funções computáveis em uma lógica S: f é computável em S, se houver algum termo ϕ, de modo que, para cada m numeral, existe uma N numeral com $S \vdash \phi(m) = n$ se, e somente se $f(m) = n$ (seguindo **Kleene** 1952, tais funções são também chamadas *reckonable* em S). Em uma nota acrescida à prova, **Gödel** (1936) observou que esta noção de computabilidade é absoluta, no sentido de que se uma é função calculável em um sistema de ordem superior S, já é calculável no da aritmética de primeira ordem, ou seja, as funções recursivas gerais são todas as funções computáveis em qualquer sistema consistente S contendo aritmética. A razão para isso é, se o sistema é formal, no sentido de que as suas provas são recursivamente enumeráveis, em seguida, a função é computável por pesquisar através de todas as provas até encontrar um dos $\phi(m) = N$, e este processo é insensível para a expressividade da lógica da teoria S. Esse resultado serviu tanto para **Church**, e mais tarde também para **Gödel**, como evidência para a tese de *Church-Turing* (veja Gödel 1946 e Sieg 1997, 2006).

A parte principal de (Gödel, 1936), no entanto, não estava preocupado com computabilidade, tanto como, com a prova de complexidade. O resultado que **Gödel** anunciou relativo ao *speed-up* de provas (medida em número de símbolos) entre a ordem n-ésimo e $(n + 1)$-ésimo da aritmética. (Buss 1994 contém uma prova do resultado) 20 anos depois, e **Gödel** outra vez voltou-se novamente a pensar em prova de complexidade. Em uma carta a **John von Neumann**, em 20 de Março 1956 (Gödel, 2003b, 372-377), **Gödel** discutia a complexidade de decidir por uma fórmula A da lógica de primeira ordem, se A tivesse uma prova com símbolos k ou menos. **Cook** mostrou que este problema é NP-completo (ver Hartmanis 1989 e Buss 1995).

Ao contrário dos primeiros trabalhos de **Gödel**, na carta a **von Neumann**, seus pensamentos sobre prova de complexidade e a viabilidade computacional, não teve impacto sobre o histórico do desenvolvimento da teoria da computabilidade e complexidade. No entanto, mostra que questões sobre a natureza da computabilidade, mesmo que essas não estivessem na vanguarda do pensamento de **Gödel** ou proeminentemente em suas publicações, isto ocupava **Gödel**, ao longo de sua carreira profissional.

Rabin foi um dos primeiros cientistas a tratar desta questão explicitamente, da seguinte forma:

"**O que significa dizer que computar a função f é mais difícil que computar a função g?**"

Outro cientista importante neste contexto é o Venezuelano **Manuel Blum** (1938), que prossegue a pesquisa sobre a teoria da complexidade, e neste sentido desenvolveu:

"**A Machine Independent Theory of the Complexity of Recursive Functions**"

Blum foi agraciado com o Prêmio Turing de 1995, em reconhecimento à sua contribuição aos fundamentos da complexidade computacional algorítmica. Na teoria da complexidade computacional, os axiomas de complexidade de **Blum** são axiomas que especificam propriedades desejáveis de medidas de complexidade no conjunto de funções computáveis. Os axiomas foram inicialmente definidos por **Manuel Blum** em 1967. Os axiomas de **Blum** podem ser usados para definir classes de complexidade sem se referir a um modelo computacional concreto.

Figura 98 – Blum - Os fundamentos da complexidade computacional algorítmica.

Fonte: Google Images - amturing.acm.org.

Juris Hartmanis (1928) é um cientista Estadunidense. Foi laureado com o Prêmio Turing de 1993, juntamente com **Richard Stearns**, por pesquisas na áres de complexidade computacional. Em 1965, **Hartmanis & Stearns** publicaram o trabalho

"On the Computational Complexity of Algorithms"

Este trabalho torna-se referência que dá o nome a esta área, e também, propõem como medida de complexidade o tempo de computação em *máquinas de Turing multifita*. Uma máquina de Turing multifita é uma máquina de Turing comum com várias fitas. Inicialmente a entrada aparece na fita 1 e todas as outras vazias, como cada fita tem sua própria cabeça para ler e escrever, a função de transição pode ler, escrever e mover as cabeças em algumas ou todas as fitas simultaneamente. Uma definição formal pode ser encontrada na referência indicada no final do capítulo.

Em 1965, **Alan Cobham** publica o trabalho caracterizando a complexidade intrínseca:

"The Intrinsic Computational Difficulty of Functions"

Em 1967, **Cobham** enfatiza o termo "intrínseco", i.e., ele estava interessado numa teoria que fosse independente de uma máquina. A questão colocada por **Cobham**, era se a multiplicação é mais difícil que a adição e, acreditava que essa questão não poderia ser resolvida até que surgisse uma teoria mais adequada.

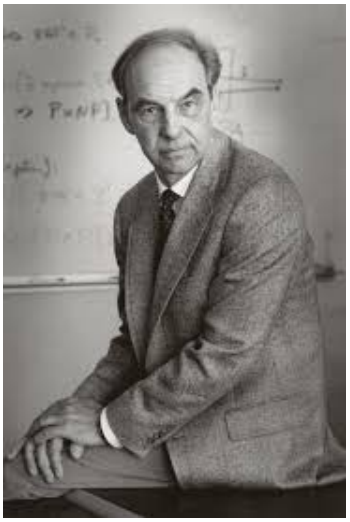


Figura 99 – Juris Hartmanis - Propõe o tempo de execução como medida de complexidade.

Fonte: Google Images - library24.library.cornell.edu.

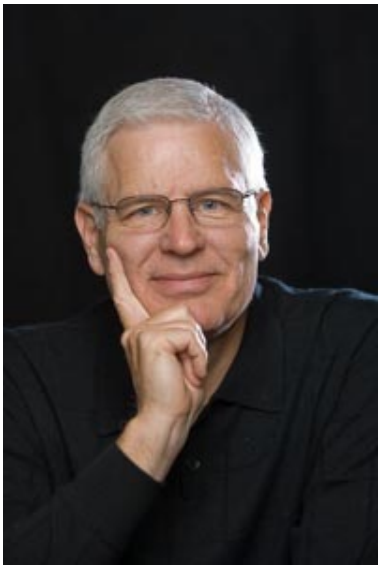


Figura 100 – Richard Stearns - Propõe com Hartmanis, o tempo de computaçãoa medida de complexidade.

Fonte: Google Images - en.wikipedia.org.

Figura 101 – Alan Cobham - A dificuldade computacional de funções.

Fonte: Google Images -en.wikipedia.org.

Alan Cobham também definiu e caracterizou uma classe importante de funções que ele chamou de $\mathcal{I}$, i.e., funções nos números naturais computáveis em tempo limitado por um polinômio no tamanho da entrada. A tese de Cobham diz que:

"**problemas computacionais podem resolvidos em algum dispositivo computacional, somente se eles podem ser computados em tempo polinomial; isto é, se eles se encontram na classe de complexidade P.**"

Formalmente, ao dizer que um problema pode ser resolvido em tempo polinomial, quer dizer que existe um algoritmo que, dada uma instância de n bits do problema como entrada, pode produzir uma solução em tempo $O(n^c)$, em que c é uma constante que depende do problema, mas não o caso particular (instância) do problema.

Na época, a questão que passou a dominar:

"**Qual é a medida de complexidade computacional correta/adequada?**"

As escolhas mais naturais são: *tempo* e *espaço*.

Mas, não havia um consenso que essas métricas fossem as corretas ou as únicas. Então, **Cobham** sugeriu que "alguma medida relacionada a noção física de trabalho computacional poderia conduzir a uma análise mais satisfatória".

Rabin propôs axiomas que uma medida de complexidade deveria satisfazer. **Rabin** sugere um arcabouço axiomático que define a base para a teoria de complexidade abstrata desenvolvida por **Blum** e outros. Após 56 anos depois dos primeiros trabalhos da área, está claro que tempo e espaço estão certamente entre as métricas mais

importantes.

No entanto, existem outras métricas importantes para medir complexidade:

- Tempo e espaço para modelos computacionais paralelos.
- Tamanho do hardware (complexidade combinatória ou de circuitos booleanos): suponha que uma função f receba como entrada *strings* finitos de *bits* e gere também como saída strings finitos de *bits* e a complexidade $C(n)$ de f seja o tamanho do menor circuito booleano que computa f para todas as entradas de tamanho n.

Cobham formaliza o *conceito de classe de problemas* que podem ser resolvidos em uma quantidade de tempo limitada por um polinômio do tamanho da entrada para o algoritmo. Uma *classe de complexidade* é um conjunto de funções que podem ser computadas dentro de certos limites de recursos.

10.3 Caracterização das classes P e NP

Na teoria da complexidade computacional, **P** é o acrônimo em inglês para **Tempo polinomial determinístico** (Deterministic Polynomial time) que denota o conjunto de problemas que podem ser resolvidos em tempo polinomial por uma *máquina de Turing determinística*. Qualquer problema deste conjunto pode ser resolvido por um algoritmo com a **taxa de crescimento do tempo de execução**, **polinomial**. Podemos ter a classe **P** como a classe de problemas que são solúveis para um computador real.

Na teoria da complexidade computacional, **NP** é o acrônimo em inglês para **Tempo polinomial não determinístico** (Non-Deterministic Polynomial time) que denota o conjunto de problemas que são decidíveis em tempo polinomial por uma *máquina de Turing não-determinística*.

Por outro lado, existe, um pior caso, em que um conjunto de problemas tem **taxa de crescimento do tempo de execução**, **exponencial**. Para esses problemas podem existir modos de simplificá-los, isto é, no lugar de se poder buscar a melhor solução (a solução ótima), podemos nos contentar com alguma solução que não é a ótima, mas que é aceitável. O que é importante salientar é que, nesses casos, **não existem, computacionalmente, algoritmos eficientes**, que possam ser implementados em computador para resolver esses problemas.

A verdade é que existem muitas coisas na natureza que nunca conseguiremos saber através de método científico, isto é, empiricamente, fazendo experiências ou criando-se modelos matemáticos e fazendo simulações em computadores. Se tudo fosse computável todos os fenômenos da natureza seriam perfeitamente descritos ou totalmente indescritíves Janos (2009).

Com base nos trabalhos de **Turing**, **Church**, **Kleene**, **Post**, se decidiu definir uma função computável como aquelas que podem ser calculadas por uma máquina de Turing, mesmo que para isso seja necessário tempo e capacidade de memória infinita. Esta mesma ideia pode ser expressa dizendo que qualquer função dada por um algoritmo é computável. Mas, veremos que mesmo **uma função computável pode não ter solução computacional**.

Vejamos um exemplo. O leitor pode examinar o caso clássico, do *Problema do Caixeiro Viajante*, mostrado em vários livros, como apresentado em Janos (2009) ou em livros sobre a Teoria dos Grafos. Este é o problema em que o caixeiro viajante deverá procurar fazer um plano de viagem, sobre um determinado número de cidades, o mais eficiente possivel, ou seja, aquele que ele percorrerá a viagem de menor custo, ou equivalentemente, encontrar o caminho de menor custo no grafo das cidades e caminhos entre essas, em que o caixeiro sai de um ponto (uma cidade origem) do gravo. Deve passar por todos os outros pontos e voltar ao ponto inicial. Muito problemas reais são similares ao **Problema do Caixeiro Viajante**. Este é um problema típico de logística, como por exemplo uma empresa que transporta e distribui produtos geograficamente, ou o caso de uma empresa automobilística que precisa encontrar a melhor sequência de montagem numa linha de produção. Para o problema do caixeiro viajante, estamos falando de um problema prático real, mas que na medida que o números de cidades cresce (sem contar a cidade inicial), o cálculo torna-se impraticável em um computador, devido a ordem de grandeza do número de cidades, o qual corresponderá ao número de permutações do número de cidades, que redundará no cálculo do fatorial do número de cidades, menos a cidade de origem. Isto é, fatorial de $(n-1)$ cidades.

Em muitas situações podemos nos defrontar com o fato de não sabermos se existe ou não solução para um problema. Ou seja:

... antes mesmo de procurar uma solução, saber se ela pode existir ou não.

Se pudermos provar que uma solução existe, não significa que encontramos a solução, mas simplesmente que temos uma chance em potencial para encontrá-la.

Vejamos o Problema do Caixeiro Viajante. A medida que o número de cidades cresce, o acréscimo no tempo de processamento é brutal. O problema é facilmente resolvido para um número pequeno de cidades, digamos 4. Caso, sejam 5, mesmo sem um computador podemos encontrar o melhor caminho. Mas, quando o tempo de processamento passa de um função polinomial para uma função exponencial, o tempo de processamento que era, digamos, de milisegundos pode passar para anos.

Por isto, "**não é possivel, na prática, resolver um problema em tempo exponencial. Assim, temos um problema que é computável, mas que na**

prática, não é possivel resolvê-lo com computadores".

Uma expressão é *satisfatível* se existe alguma atribuição de valores as variáveis, que faz toda a expressão verdadeira.

Em 1971, **Stephen Cook** publica o primeiro problema **NP-completo**: o problema da satisfabilidade booleana. Na teoria da complexidade computacional, o teorema de **Cook-Levin**, também conhecido como teorema de **Cook**, afirma que o problema de satisfatibilidade booleana é **NP-completo**. Isto quer dizer que, qualquer problema em NP pode ser reduzido em tempo polinomial por uma máquina de Turing determinística para o problema de determinar se uma fórmula booleana é satisfatível.

Figura 102 – Stephen Cook - O primeiro problema NP-Completo.

Fonte: Google Images - amturing.acm.org.

Uma importante consequência desse teorema é que se existe um algoritmo de tempo polinomial para resolver a satisfatibilidade, então existe um algoritmo de tempo polinomial para resolver todos os problemas em NP (*Nondeterministic Polinomial Time*). Crucialmente, o mesmo acontece para qualquer problema **NP-completo**.

A questão, se tal algoritmo existe, é chamada de **Problema P versus NP** e esse é amplamente considerado o mais importante problema sem solução da teoria da computação. O teorema é atribuído a **Stephen Cook** e **Leonid Levin**.

Richard Manning Karp (1935) é um cientista da computação e teórico computacional da Universidade da California, Berkeley, reconhecido pela sua pesquisa sobre teoria dos algoritmos, pelo qual recebeu um **Prêmio Turing** em 1985, **Medalha Benjamin Franklin em Computação e Ciência Cognitiva** em 2004, e o **Prêmio Kyoto** em 2008.2

Em 1972, **Karp** propõe a notação usada em complexidade computacional. Em particular, a classe de problemas P que podem ser resolvidos em tempo polinomial. **Karp** fez a identificação da computabilidade em tempo polinomial com a noção

intuitiva da eficiência do algoritmo, e, mais notável, contribuições para a teoria da NP-completo. **Karp** introduziu a metodologia padrão atual para provar que problemas são **NP-completos** o que levou à identificação de muitos outros problemas práticos e teóricos como sendo de dificuldade computacional.

Figura 103 – Richard Karp - A notação para a complexidade computacional.

Fonte: Google Images - www.nae.edu.

Em 1972, **Karp** também publica uma lista de 21 problemas **NP-completo**, mostrando a importância da área. O conceito de **NP-completo** foi desenvolvido do final dos anos 60 ao começo dos anos 70 em pesquisas nos Estados Unidos e União Soviética. Nos, EUA em 1971, **Stephen Cook** publicou o documento "The complexity of theorem proving procedures", na conferência de procedimentos da recém fundada *ACM Symposium on Theory of Computing*. O documento subsequente de **Richard Karp**, "Reducibility among combinatorial problems", gerou um novo interesse no documento de **Cook** criando a lista dos 21 problemas **NP-completos**. **Stephen Cook** e **Richard Karp** receberam um Prêmio **Turing** por esse trabalho.

10.4 A Complexidade de Problemas

Problemas intratáveis ou difíceis são comuns na natureza e nas áreas do conhecimento. Problemas podem ser classificados em:

- "fáceis" (tratáveis): resolvidos por algoritmos polinomiais.
- "difíceis"intratáveis): os algoritmos conhecidos para resolvê-los são exponenciais.

Complexidade de tempo da maioria dos problemas é **polinomial** ou **exponencial**.

Caracterizando um *problema*:

- Conjunto de parâmetros (definição das instâncias).
- Conjunto de propriedades (restrições do problema).

Tamanho das instâncias - Quantidade de *bits* necessária para representar as instâncias em computadores digitais. A questão é:

"extbfQual o número de computações (operações em *bits*) necessárias para se obter a melhor solução (solução ótima)?"

10.5 A arte de programar de Donald Knuth

De formação matemática, vale a pena terminarmos este capítulo, enfatizando o trabalho de **Knuth**. **Donald Ervin Knuth** (1938-) é um cientista computacional de renome e professor emérito da Universidade de Stanford. É o autor do livro *The Art of Computer Programming*, uma das principais referências da Ciência da Computação. Ele praticamente criou o campo análise de algoritmos e fez muitas das principais contribuições a vários ramos da teoria da computação. Ele também criou o sistema tipográfico TEX, o sistema de criação de fontes METAFONT.

Donald E. Knuth não é somente Professor Emérito de *The Art of Computer Programming* na Universidade de Stanford, também é autor do multi-volume work-in-progress *The Art of Computer Programming*, atualmente em quatro volumes,é membro da Academia Americana de Artes e Ciências, da Academia Nacional de Ciências e da Academia Nacional de Engenharia. Ao contrário de muitos outros portadores de um número semelhante de títulos de prestígio e livros publicados, ele é, hoje, possivelmente a única legenda viva entre os investigadores em Ciência da Computação.

Figura 104 – Knuth - The Art of Computer Programming.

Fonte: Google Images - www-cs-faculty.stanford.edu.

Apesar da opinião de alguns, software livre existia muito antes de **Projeto GNU**. Além disso, o atual software livre/aberto é amplamente baseado no que veio antes dele. O Professor **Donald Knuth** é um dos maiores contribuintes para este conjunto de conhecimentos. Além disso, os programas livres são tão bons, como os algoritmos que eles estão usando. É de se rejeitar que suas contribuições para o movimento do software livre como irrelevantes como "combatente da liberdade de software profissional". **Richard Stallman** (ver **Slashdot / 2nd Annual Fundação Software Livre Awards**) foi pelo menos desrespeitoso, esquecendo o fato de que TeX é um dos mais importantes programas de software livre já escrito. Tanto é verdade, que toda a indústria tipográfica mundial acabou por adotar TeX, como a base para seus produtos.

Apesar da explosão da informação em Ciência da Computação, o Professor **Knuth** ainda está tentando terminar a sua monumental *The Art of Computer Programming*, uma enciclopédia escrita por um autor de algoritmos e Ciência da Computação. Isso realmente faz com que ele seja o último homem do período renascentista da Ciência da Computação. Ele é um dos poucos que conseguiu fazer contribuições para um espectro muito diversificado de temas de Ciência da Computação. Entre eles:

- Escreveu um dos primeiros compiladores *Algol* compactos com a idade de 22 anos (1960).
- Obteve 1271 dígitos da constante de **Euler**, usando a soma de **Euler-Maclaurin** (1962).
- Projetou a linguagem de simulação *SOL*.
- Sugeriu o nome "*Backus-Naur Form*" (a notação BNF para descrever gramáticas)- (1964).
- Introduziu e refinou o algoritmo de análise sintática LR (ver Knuth, DE 1965. "Sobre a Tradução de Línguas da esquerda para a direita", Information and Control, Vol. 8, pp. 607-639. O primeiro trabalho sobre LR análise.)
- Publicou o primeiro volume de *The Art of Computer Programming* com a idade de 30 anos (1968). Seus três volumes de *The Art of Computer Programming* (1968, 1969 e 1973) desempenharam um papel importante no estabelecimento e definição de Ciência da Computação como uma disciplina rigorosa, intelectual.
- Publicou seu trabaho inovador "Um estudo empírico dos programas de Fortran." (Software — prática e experiência, vol 1, páginas 105-133, 1971), que lançou a pedra fundamental de um estudo empírico de linguagens de programação.
- Teve a contribuição crucial para o debate de programação sobre a "Programação estruturada sem goto", que foi um golpe decisivo para os fundamentalistas de programação estruturada liderados por **Edsger Dijkstra**, publicando um artigo com programação estruturada em ACM Computing Survey. 6 (4): 261-301 (1974).

- Projetou e escreveu os sistemas de documentação TeX e Metafont (escrito em WEB para Pascal). **Knuth** desenvolveu a primeira versão do TeX em 1971-1978, a fim de evitar problemas com diagramação da segunda edição de seus volumes TAOCP. O programa provou popular e ele produziu uma segunda versão (em 1982), que foi a base do que usamos hoje. Todo o texto do programa foi publicado em seu livro O Livro Didático (Addison-Wesley, 1984, ISBN 0-201-13447-0, paperback ISBN 0-201-13448-9).

- Inventou vários algoritmos importantes, incluindo algoritmo LR análise e o algoritmo Knuth-Morris-Pratt de busca de *strings* (1977).

- Teve contribuição crucial para dissipar a mitologia sobre a "prova de correção": *Here is his famous citation about correctness proofs*: *On March 22, 1977, as I was drafting Section 7.1 of "The Art of Computer Programming", I read four papers by Peter van Emde Boas that turned out to be more appropriate for Chapter 8 than Chapter 7. I wrote a five-page memo entitled "Notes on the van Emde Boas construction of priority deques: An instructive use of recursion" and sent it to Peter on March 29 (with copies also to Bob Tarjan and John Hopcroft). The final sentence was this: "**Beware of bugs in the above code; I have only proved it correct, not tried it.**"*

10.6 Bibliografia e Fonte de Consulta

Hartmanis, Juris. 1989. Gödel, von Neumann and the P=?NP problem. Bulletin of the European Association for Theoretical Computer Science (EATCS) 38:101107.

Buss, Samuel R. 1994. On Gödels theorems on lengths of proofs I: Number of lines and speedups for arithmetic. Journal of Symbolic Logic 39: 737756.

Buss, Samuel R. 1995. On Gödels theorems on lengths of proofs II: Lower bounds for recognizing k symbol provability. In Feasible Mathematics II, eds. P. Clote and J. Remmel, 5790. Basel: Birkhäuser.

Michel Janos (2009). Matemática e Natureza. Livraria da Física.

Antonio Alfredo Ferreira Loureiro (2009). Teoria da Complexidade, Projeto e Análise de Algoritmos, DCC/INEX/UFMG. Em ⟨homepages.dcc.ufmg.br/~loureiro/alg/091/paa_Complexidade.pdf⟩

Biografia de Michael O. Rabin - ⟨http://amturing.acm.org/award_winners/rabin_9681074.cfm⟩

Teoria da Complexidade - ⟨www.teoriadacomplexidade.com.br/teoria-da-complexidade.html⟩

Richard Stearns - ⟨https://en.wikipedia.org/wiki/Richard_Stearns_(World_Vision)⟩

Richard Karp - ⟨https://pt.wikipedia.org/wiki/Richard_Karpouhttp://www.eecs.berkeley.edu/~karp⟩

Stephen Cook - ⟨https://pt.wikipedia.org/wiki/Teorema_de_Cook-Levin⟩

Complexidade P - ⟨https://pt.wikipedia.org/wiki/P_(complexidade)⟩

Complexidade NP - ⟨https://pt.wikipedia.org/wiki/NP_(complexidade)⟩

Máquina de Turing Multifita -

10.7 Referências

Computational Complexity Theory - ⟨http://plato.stanford.edu/entries/computatio\penalty\@M\hskip\z@skip\discretionary{-}{}{}\penalty\@M\hskip\z@skipnal-com\penalty\@M\hskip\z@skip\discretionary{-}{}{}\penalty\@M\hskip\z@skipplexity/⟩, First published Mon Jul 27, 2015.

Gödel, K. (1931). On formally undecidable propositions of Principia Mathematica and related systems I), Monatshefte für Mathematik und Physik, 38:173-198. [Reimpresso e traduzido em Gödel (1986, pp. 144-195)].

Rabin, Michael O. and Dana Scott,"Finite Automata and Their Decision Problems", IBM Journal of Research and Development, Vol. 3, Num. 2, 1959, pp. 114-125. Reprinted in: Moore E. F. (editor), Sequential Machines, Selected Papers, Addison Wesley Publishing Company, Reading, MA (1964), pp. 63-92. Este artigo contém a pesquisa sobre automata não-determinístico para o qual os dois autores receberam o 1976 Turing Award.

Rabin, Michael O., "Degree of Difficulty of Computing a Function and a Partial Ordering of Recursive Sets", Technical Report No. 2, O.N.R., Hebrew University, Jerusalem, 1960. Artigo inspirado no problema de John McCarthy pelos "spies and guards".

Gödel, K. (1986). Collected Works, vol. 1, S. Feferman et al., eds., Oxford: Oxford University Press.

Filmes e Vídeos - Teoria da Complexidade: ⟨www.teoriadacomplexidade.com.br/filmes-e-videos.html⟩

J. Hartmanis, R. E. Stearns: On the computational complexity of algorithms. Trans. Amer. Math. Soc. 117:285-306, 1965.

The Origins of Computational Complexity - ⟨www.dijkstrascry.com/sites/default/files/.../nEssay.pd...⟩

A Short History of Computational Complexity - ⟨people.cs.uchicago.edu/~fortnow/papers/history.pdf⟩

Alan Cobham (1965), "The intrinsic computational difficulty of functions", Proc. Logic, Methodology, and Philosophy of Science II, North Holland.

Blum, M. (1967). A Machine Independent Theory of the Complexity of Recursive Functions. Journal of the ACM, 14(2):322-336, April.

Richard M. Karp. In: R. E. Miller and J. W. Thatcher (editors). Complexity of Computer Computations. [S.l.]: New York: Plenum, 1972. 85-103 p.

Karp, R.M. Karp (1972). "Reducibility Among Combinatorial Problems", in R.E. Miller and J.W. Thatcher (editors). Complexity of Computer Computations. New York: Plenum, pp. 85-103.

Association for Computing Machinery. ACM Award Citation/Richard M. Karp. http://awards.acm.org/award_winners/karp_3256708.cfm , Visitado em 2010-01-17

Cook, S.A. (1971). The Complexity of Theorem Proving Procedures. Proceedings of the Third Annual ACM Symposium on Theory of Computing, pp. 151-158.

Reserva de Mercado da Informática no Brasil - ⟨https://pt.wikipedia.org/wiki/Reserva_de_mercado⟩

CAPÍTULO 11

Modelos de Computação em Grafos

Este capítulo tem por objetivo apresentar as estruturas matemáticas mais importantes para modelar sistemas de computação. O capítulo dá ênfase às estruturas baseadas em grafos orientados e suas diversas variantes, tais como, um sistema de transição rotulado, uma máquina de estados finito, os autômatos finitos e as redes de Petri. Estas estruturas são básicas e úteis para a construção de ferramentas de computação e formam o modelo-base para determinados métodos formais que tem linguagens textuais, mas que descrevem exatamente como o modelo-base funciona. O desenvolvimento de algoritmos para manipular grafos é um tema importante para a Ciência da Computação.

As seguintes abreviações são consideradas neste capítulo: **Grafos orientados** (G), **Máquinas de Estados Finito** (MEF), **Autômatos Finito Determinísticos** (AFD), **Autômatos Finito Não-Determinísticos** (AFND), **Árvores** (A) e **Redes de Petri** (RP). Todas estas estruturas são matemáticas, muito importantes para a **Ciência da Computação**, e podem ser implementadas num computador usando-se alguma linguagem de programação. Na realidade, um programa de computador é um autômato finito, determinístico ou não-determinístico. E existem algumas boas ferramentas em sites (CPN Tools, PIPE2 Tool, etc) para se modelar, simular e analisar sistemas concorrentes e paralelos através de Redes de Petri como mostradas neste capítulo.

11.1 A Teoria dos Grafos

Como em Deo (1974), o problema da **pontes de Königsberg** é talvez o exemplo mais bem conhecido em teoria dos grafos. É um problema de londa data proposto por **Leonhard Euler**, em 1736. **Leonhard Paul Euler** (1707-1783) foi um grande matemático e físico Suíço de língua Alemã que passou a maior parte de sua vida na Rússia e na Alemanha. **Euler** é considerado um dos mais proeminentes matemáticos

do século XVIII, e foi um dos mais prolíficos matemáticos. Calcula-se que toda a sua obra reunida teria entre 60 e 80 volumes.

Figura 105 – Leonhard Euler - O originador da teoria dos grafos em 1736.

Fonte: Graph Theory with Applications to Engineering and Computer Science, Deo (1974).

Euler, escreveu a primeira publicação em teoria dos grafos e assim tornou o originador da teoria, bem como do restante da topologia. O problema é descrito na Figura 106. São duas ilhas C e D, formadas por Pregel River in Königsberg (então, a capital da East Prussia, agora renomeada Kaliningrad na West Russia) que foram conectadas e as suas margens A e B com sete pontes, como mostrado na Figura 106. O problema é iniciar em qualquer das 4 áreas da cidade: A, B, C ou D, caminhar sobre cada das sete pontes exatamente uma vez, e retornar ao ponto de partida (sem nada através do rio).

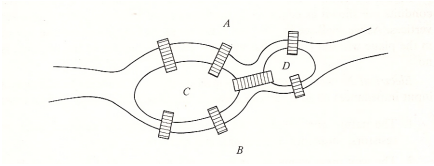


Figura 106 – O problema das sete pontes de Königsberg.

Fonte: Graph Theory with Applications to Engineering and Computer Science, Deo (1974).

Euler representou esta situação por meio de um grafo, como mostrado na Figura 107. Os vértices representam as áreas de terra e o arcos representam as pontes. **Euler** provou que uma solução para este problema, *não existe*.

O problema das pontes de Königsberg é o mesmo problema de desenhar figuras sem tirar a caneta do papel e sem retraçar uma linha.

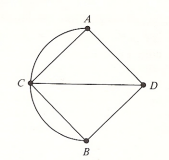


Figura 107 – O grafo do problema das sete pontes de Königsberg.

Fonte: Graph Theory with Applications to Engineering and Computer Science, Deo (1974).

11.1.1 Grafos não-orientados

Um grafo $G = (V, E)$ consiste de um conjunto de objetos $V = \{v_1, v_2, \ldots\}$ chamados vértices, e um utro conjunto $E = \{e_1, e_2, \ldots\}$, cujos elementos sáo chamados arcos, tais que cada arco e_k é identificado com um par não-ordenado (v_i, v_j) de vértices. A mais comum representação de um grafo é por meio de um diagrama, no qual os vértices são representados por pontos e cada arco, como um segmento de linha ligando vértices. A Figura 108, por exemplo, é um grafo. Observe que esta definição permite um arco ser associado com um par de vétices (v_i, v_i). Este caso e chamado um "*self-loop*", as vezes chamado um "*loop*". Em aplicações da teoria de redes elétricas, um "*loop*" tem significado diferente de "*self-loop*", e para evitar confusão, neste contexto chamaremos de "*self-loop*", como em Deo (1974).

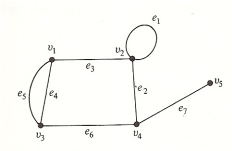


Figura 108 – Exemplo de um grafo não-orientado.

Fonte: Graph Theory with Applications to Engineering and Computer Science, Deo (1974).

Para mostrar a relação da estrutura de um grafo com a teoria da matrizes, observe a matriz do grafo na Figura 109. A matriz é chamada de *matriz de incidência*.

11.1.2 Grafos Orientados

Definição (Grafo Orientado) - Como em Deo (1974), um *grafo orientado* é uma estrutura matemática consistindo de um conjunto de N e uma relação binária A sobre N. Os elementos de N são chamados nodos ou vértices do grafo, e os membros de A são chamados arcos. A é referida como uma relação adjacente e y é dito ser adjacente a x quando $[x, y] \in A$. Um exemplo está na Figura 110 como em Sudkamp (1988).

Em um grafo orientado, um **caminho** de comprimento n de um nodo x para um nodo y é uma sequência de nodos $x_0, x_1, ..., x_n$, satisfazendo:

1. x_i é adjacente a x_{i-1}, para $i = 1, 2, ..., n$
2. $x = x_0$

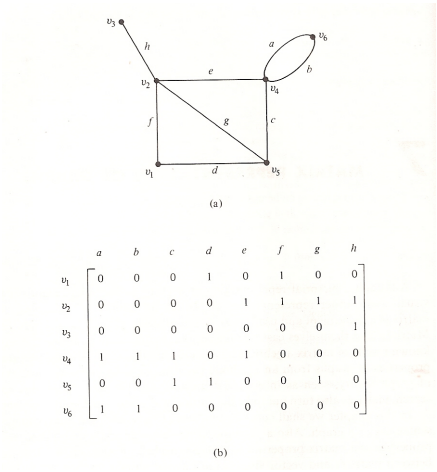


Figura 109 – Exemplo de matriz de um grafo orientado.

Fonte: Graph Theory with Applications to Engineering and Computer Science, Deo (1974).

3. $y = x_n$

O nodo x é o nodo inicial do caminho e y é o nodo terminal. Existe um caminho de comprimento zero de qualquer nodo para ele próprio, chamado de **caminho nulo**.

Um caminho de comprimento 1 ou mais, que começa e finaliza no mesmo nodo é chamado de um **ciclo**. Um ciclo é simples se ele não contém um subcaminho que seja um ciclo.

Um grafo orientado contendo pelo menos um ciclo e dito ser um grafo **cíclico**. Um grafo sem nenhum ciclo é dito ser **acíclico**.

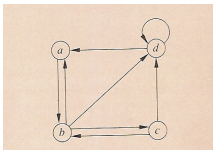


Figura 110 – Exemplo de um grafo orientado.

Fonte: Languages and Machines: An Introduction to Theory of Computer Science Sudkamp (1988).

Definição (**Grafo Orientado Rotulado**)

Um grafo orientado rotulado é uma estrutura $G = (N, L, A)$, onde L é o conjunto dos rótulos e A é a relação sobre $N \times N \times L$. Um elemento $[x, y, r]$ pertencente a A é um arco de x a y, rotulado por r. O rótulo de um arco especifica a relação entre nodos adjacentes. Rótulos sobre um grafo G, correspondentes a arcos, podem representar, distâncias, custos ou alguma grandeza importante para a utilização do grafo. A Figura 111 mostra um segundo exemplo de um grafo orientado, mas neste caso, um grafo rotulado, como em Deo (1974).

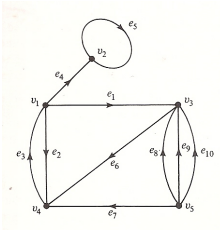


Figura 111 – Exemplo de um grafo orientado e rotulado.

Fonte: Graph Theory with Applications to Engineering and Computer Science, Deo (1974).

11.2 Árvores Ordenadas

Uma outra estrutura matemática muito utilizada na Ciência da Computação é uma árvore ordenada, ou simplesmente uma árvore, um grafo orientado acíclico no qual cada nodo é conectado por um único caminho, a partir de um nodo distinguido chamado **nodo raiz** da árvore. Um nodo raiz tem grau-de-entrada zero e todos os outros nodos tem grau-de-entrada 1.

Definição (**Árvore**)

Uma árvore T é uma estrutura $T = (N, A, r)$, onde N é o conjunto de nodos, A é a relação entre nodos adjacentes, e r, $r \in N$ é o nodo raiz da árvore.

A Figura 112 representa uma árvore ordenada: uma árvore gerada a partir do seu nodo-raiz, como em Sipser (2011).

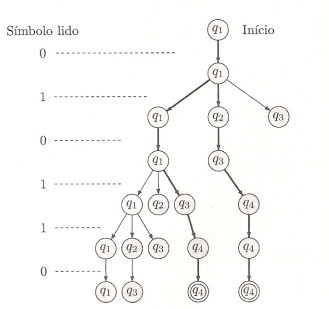


Figura 112 – O exemplo de um grafo que é uma árvore.

Fonte: Introdução à Teoria da Computação, Sipser (2011).

Um nodo y é chamado um **filho** de um nodo x, e x é **pai** do nodo y, se y é adjacente a x. A relação de adjacência é uma ordem sobre o filho de qualquer nodo. Quando uma árvore é desenhada, esta ordenação é geralmente indicada pela listagem dos filhos de um nodo, em uma ordem da esquerda para à direita. Na Figura 112, a ordem dos filhos do nodo q_1 em T é q_1, q_2 e q_3.

Um nodo com grau-de-saída zero é chamado **folha** (em inglês, *leaf*). Todos os outros nodos, que não são folhas, são referidos como **nodos internos** à árvore.

A **profundidade** (em inglês, *depth*) do nodo raiz é zero. A profundidade de qualquer outro nodo é a profundidade de seu pai, mais 1. A profundidade de uma árvore é a máximo das profundidades dos nodos na árvore.

Um nodo y é chamado um descendente de um nodo x, e x um ancestral de y, se existe um caminho de x para y. Com esta definição, cada nodo é ancestral e descendente dele próprio. Ancestral e descedência são relações que podem ser definidas recursivamente, usando a relação de adjacência sobre $N \times N$. O **ancestral comum mínimo** de dois nodos x e y é um ancestral de ambos e um descendente de toso os outros ancestrais. O **ancestral comum mínimo** de x_{10} e x_{11} é x_5, de x_{10} e x_6 é x_2 e de x_{10} e x_{14} é o nodo raiz x_1.

Uma **sub-árvore** de um árvore T é um subgrafo de T que é uma árvore. O conjunto de nodos descendentes de um nodo x e a restrição da relação de adjacência para este conjunto, formam uma su-árvore com raiz x.

A ordenação de nodos irmãos (em inglês, *siblings*) numa árvore T pode ser estendida através da relação $LEFTOF$ sobre $N \times N$. A relação $LEFTOF$ tenta capturar a propriedade de um nodo estar à esquerda de um outro, no diagrama da árvore. Para dois nodos x e y, nenhum dos quais é ancestral do outro, a relação $LEFTOF$ é definida em termos das sub-árvores geradas pelo **ancestral comum mínimo** dos nodos. Por exemplo, seja z um ancestral comum mínimo de dpois nodos x e y. E sejam $z_1, z_2, ..., z_n$ ser os filhos de z em sua ordem correta. Então x está na sub-árvore gerada por um dos filhos de z, chamado z_i. Similarmente, y está na sub-árvore gerada por z_j, para algum j. Visto que z é ancestral comum mínimo de x e y, $i \neq j$. Se $i < j$, então $LEFTOF(x, y)$, o que significa que x está à esquerda de y. De outro modo, $i > j$, então $LEFTOF(y, x)$, o que significa que y é que está à esquerda de x. Com esta definição, nenhum nodo é $LEFTOF$ de seus ancentrais.

A relação $LEFTOF$ pode ser usada para ordenar o conjuntos das folhas de uma árvore. A **fronteira** (em inglês, *frontier*)de um árvore é construída das folhas, na ordem gerada pela relação $LISTOF$. Na Figura 112, a fronteira de T é a sequência dos nodos q_1, q_3 e q_4.

11.3 Sistemas de Transições Rotulados

Com um grafo orientado rotulado, pode-se definir um **sistema de transição rotulado** *LTS* - Labelled Transition System - Milner (1989), como uma tripla:

$$LTS = (S, T, \left\{ \stackrel{t}{\rightarrow} : t \in T \right\})$$

o qual consiste de um conjunto S de expressões de comportamento, um conjunto T de rótulos de transições e uma relação de transição $\xrightarrow{t} \subseteq S \times S$ para cada $t \in T$. Neste sistema de transição rotulado, como em Milner (1989), tomamos S ser um conjunto de expressões de comportamento de agentes, T ser o conjunto Act das ações (externas e internas) de um sistema computacional, e a definição de cada transição $\xrightarrow{t}$. Esta definição permite a estrutura de expressões de comportamento, a qual define as transições de cada agente composto, em termos das transições de seus componentes, e é apropriada quando se trabalha com expressões de comportamneto externo de um sistema.

Nesta estrutura matemática de um LTS, expressões de comportamento de agentes, como em CCS (*Calculus of Communicating System*) de **Robin Milner**, são descritas sobre um conjunto de expressões de agentes $(E, F, ...)$ e operadores sobre essas expressões de agentes, tais como:

1. $\alpha.E$, um prefixo ($\alpha \in Act$).
2. $\sum_{i \in I} E_i$, operador + de escolha determinística, onde i é um índice num conjunto de indexação I.
3. $E_1 \mid E_2$, uma composição paralela de expressões de agentes, que tem a semântica da sincronização dos agentes envolvidos.
4. EŁ, uma restrição, onde L é um subconjunto de ações externas e internas.
5. $E[f]$, uma expressão de rerotulação de ações, onde f é uma função de rerotulação.

11.4 MEF - Máquina de Estados Finitos

Uma outra utilização de um grafo orientado é na definição de uma **Máquina de Estados Finitos** Sudkamp (1988), que é a definição formal de uma máquina, abstraindo-se do *hardware* envolvido na operação de uma máquina, mas, ao contrário, *com a descrição das operações internas, de como uma máquina processa suas entradas*. Entradas correspondem aos eventos (ações) externos e eventos internos (ações internas) provenientes dos usuários de uma máquina. Por exemplo, o que ocorre num máquina de terminal eletrônico de um banco, que realiza as operações financeiras de usuários do banco.

O projeto de uma máquina deve representar simbolicamente cada dos componentes da máquina. A entrada para a máquina abstrata é uma cadeia de símbolos (string). O grafo orientado rotulado, conhecido como **diagrama de estados** é frequentemente usado para representar as transformações do estado interno da máquina. Os nodos do diagrama de estados são os estados da máquina. O estado da máquina no início de uma computação é designado por um **estado inicial**. A partir de um estado inicial, uma sequência de estados, durante o processamento, definem uma **computação**

da máquina, que termina num estado (final da computação) quando a máquina pára, ou volta ao seu estado inicial, como em muitos sistemas de computação atuais. Máquinas de estados finitos (MEF) são os cidadãos de primeira classe da teoria dos autômatos, que define um **autômato finito** como um modelo matemático usado para representar programas de computadores ou circuitos lógicos. Autômatos finitos podem ser determinísticos ou não-determinísticos.

11.4.1 AFD - Autômato Finito Determinístico

A análise de sistemas de computação reais, pode requerer separar os fundamentos do projeto de uma máquina, dos detalhes de implementação. A descrição independente da implementação é frequentemente referida como um **máquina abstrata**. Um grafo orientado pode introduzir uma **classe de máquinas abstratas** cujas computações podem ser usadas para determinar a "aceitabilidade" de cadeias de símbolos (*strings*), que são entradas para a máquina abstrata. Assim, pode-se definir o que é um *Autômato Finito*, como em Sudkamp (1988).

Definição (**Autômato Finito Determinístico**)

Um Autômato Finito Determinístico (AFD) é uma quíntupla $M = (Q, \Sigma, \delta, q_0, F)$, onde Q é um conjunto finito de estados, Σ é um conjunto finito chmado alfabeto, q_0 é estado distinguido conhecido como estado inicial, F é um subconjunto de Q ($F \subseteq Q$) chamado de estados finais e δ é uma função total, $\delta : Q \times \Sigma \to Q$, conhecida com uma função de transição de estados.

Um exemplo dado em Sipser (2011), de um AFD - Autômato Finito Determinístico pode ser visto na Figura 113.

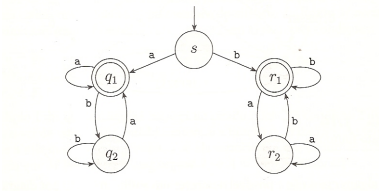


Figura 113 – AFD - Uma máquina de cinco estados.

Fonte: Introdução à Teoria da Computação Sipser (2011).

No caso de um autômato finito não-determinístico (AFND), na definição formal de autômato, a função de transição δ é uma função total, $\delta : Q \times \Sigma \to \mathcal{P}(Q)$. Um exemplo de um AFND, dado em Sipser (2011), pode ser visto na Figura 114. Note que no estado inicial, a cadia vazia ϵ pode conduzir a execução do autômato para dois estados distintos.
Máquinas de estado finito podem modelar um grande número de problemas, entre os quais, em computação para descrever as gramáticas das linguagens formais, em

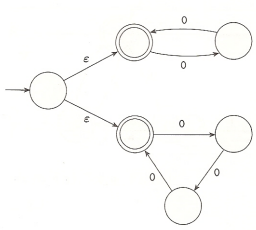


Figura 114 – AFND - Uma máquina não-determinística.

Fonte: Introdução à Teoria da Computação Sipser (2011).

automação, o projeto eletrônico de sistemas, o projeto de protocolos de comunicação, a análise e outras aplicações de engenharia. Autômatos finitos são, por vezes, utilizadas para descrever as gramáticas das linguagens formais usadas na Ciência da Computação.

11.5 Técnicas de Modelagem Matemática de Sistemas

Diversas técnicas de modelagem matemática de sistemas de computação tem sido propostas. O que essas técnicas tem em comum é que usam um **modelo-base** que consiste de um dos tipos de grafo apresentados nas seções anteriores. Um *modelo-base*, apresentado na forma gráfica, pode ser descrito num estilo menos abstrato através de uma linguagem gráfica. Uma linguagem textual, que pode ser um linguagem matemática, define o método formal usado para especificar sistemas. Em Barroca (1992) é apresentada uma classificação dessas técnicas.

1. Técnicas Baseadas em Modelos - Fornecem uma descrição abstrata implícita, textual, sobre estados e operações que transforma os estados. Por exemplo:

    - Z Notation Spivey (1989) - Tendo um modelo-base como uma máquina de estados finito, sem apresentar uma forma explícita para especificar concorrência.
    - Object Z Smith (2000) Duke (2000)- Tendo um modelo-base como uma máquina de estados finitos, apresenta uma forma explícita de especificar concorrência.
    - Zag (Z with agents) Sobral (1996) - Tendo um modelo-base como uma máquina de estados finitos, apresenta um forma explícita de especificar concorrência, baseada em CCS Milner (1989), Z Notation Spivey (1989) e Lógica Temporal Vasconcelos (1989).

2. Técnicas Baseadas em Álgebra de Processos - Tem como modelo-base um sistema de transição rotulado e fornecem uma descrição abstrata, textual, usando expressões de comportamento dos processos componentes da arquitetura do sistema sendo especificado; apresentam forma explícita para especificar concorrência, como por exemplo: CSP Hoare (1985), CCS Milner (1989).

3. Técnicas Baseadas em Lógicas - Tendo um modelo-base implícito de uma máquina de estados finitos, que numa linguagem de uma lógica, especifica propriedades de sistemas; uma variedade desta técnica tem sido propostas, onde se analisam as relações entre estados implícitos, relações temporais ou relações causais, como por exemplo, respectivamente, em Lógica dos Predicados, Lógica Temporal ou Lógica Modal de Ações Goldbaltt (1982).

4. Técnicas Baseadas em Redes - Tendo um modelo-base implícito de uma máquina de estados finitos, que numa linguagem gráfica, modelam o fluxo da execução de um sistema, descrevendo a mudança de um conjunto de estados (um determinado lugar) para outro conjunto de estados (outro lugar), através de transições, podendo especificar explicitamente a concorrência, como no caso das Redes de Petri Peterson (1981).

11.6 Redes de Petri

Focalizaremos, agora, uma técnica que forma um rede explicitada como um grafo: as Redes de Petri, Conforme MACIEL (1996), a teoria inicial foi, primeiramente, apresentada por Carl Adam Petri em 1962, como sua tese de doutorado, da Faculdade de Matemática e Física da Universidade de Darmstadt, Alemanha Ocidental. A partir daí, o trabalho de Petri atraiu a atenção de muitos outros pesquisadores que desenvolveram a teoria matemática. Diversos trabalhos surgiram criando variantes do modelo original, tais como, redes de Petri *predicado-ação*, rede de Petri *predicado-transição*, redes de Petri *temporizadas*, redes de Petri *estocásticas* e redes de Petri *coloridas*. Para todas estas variantes existem ferramentas implementadas, muito úteis quando se deseja analisar uma rede de Petri. Durante uma análise, falas de sistemas podem ser detectadas antes da implementação ser feita.

Redes de Petri é uma ferramenta para estudo de sistemas de computação. É considerada um método formal consolidado, que segue o estilo de linguagem gráfica para especificação e verificação de propriedades de sistemas concorrentes ou paralelos, e, deste modo, validando uma especificação antes de uma implementação do sistema. A teoria de Petri permite sistemas serem modelados como uma representação matemática do sistema. Assim, podem revelar informação sobre a estrutura e comportamento dinâmico do sistema modelado. Essa informação pode ser usada para avaliar o modelo do sistema e sugerir melhoramentos no modelo antes de implementá-lo. Desta forma, Redes de Petri se aplicam à modelagem e projeto de sistemas Peterson (1981). O formalismo é baseado na *Bag Theory*, uma extensão da Teoria dos Conjuntos (*Set Theory*), em que cada *bag* consiste de um conjunto com elementos repetidos, ao contrário do que é um conjunto, que não tem repetição de elementos.

Definição (**O que é uma Rede de Petri**)

Uma estrutura de rede de Petri, C, é uma tupla, $C = (P, T, I, O)$, onde $P =$

$\{p_1, p_2, ..., P_n\}$ é um conjunto finito de , $n \geq 0$. $T = \{t_1, t_2, ..., t_m\}$ é um conjunto finito de transições, $m \geq 0$. O conjunto de lugares e trnasições são disjuntos, $P \bigcap T = \phi$. $I : T \to P^\infty$ é a função-entrada, mapeando de transições para *bag* de lugares. $O : T \to P^\infty$ é a função-saída, um mapeamento de transições para *bags* de lugares. Peterson (1981). A cardinalidade de P é n. E a cardinalidade de T é m. Um elemento arbitrário de P (um lugar) é p_i, $i = 1, ..., n$ e um elemnto arbitrário de T é t_j, $j = 1, ..., m$.

Como explicado em Peterson (1981), um lugar p_i é um lugar de entrada de um transição t_j, se $p_i \in I(T_j)$; p_i é um lugar de saída, se $p_i \in O(t_j)$. As entradas e as saídas de uma transição são *bags* de lugares. O uso de bags, ao contrário de conjuntos, para entradas e saídas de uma transição permite um lugar ser de múltipla entrada e de múltipla saida de uma transição. A multiplicidade de um lugar de entrada p_i para uma transição t_j é o número de ocorrências do lugar na bag de entrada da transição, $\#(p_i, I(t_j))$. Similarmente, a multiplicidade de um lugar de sapida p_i para uma transição t_j é o número de ococrrências do lugar na bag de saída da transição t_j, $\#(p_i, O(t_j))$, as funções de entrada e saída são conjuntos (ao contrário do bags), então a multiplicidade de cada lugar é zero ou 1.

As funções de entrada e saída podem ser estendidas para mapear lugares em bags de transições, em adição, a mapear transições em bags de lugares. Em Peterson (1981), é definido uma transição t_j ser uma entrada de um lugar p_i, se p_i é uma saída de t_j. E uma transição t_j é uma saída de lugar p_i, se p_i é uma entrada de t_j.

Definição (**Estendida**)

Estendemos a função de entrada I e a função de saída O, como em Peterson (1981):

$I : P \to T^\infty$ e $O : P \to T^\infty$, de modo que:

$\#(t_j, I(p_i)) = \#(p_i, O(t_j))$ e $\#(t_j, O(p_i)) = \#(p_i, I(t_j))$.

Uma rede de Petri é um *multigrafo*, porque permite múltiplos arcos de um nodo do grafo para outro. Além disso, visto que os arcos são orientados (dirigidos), uma rede de Petri é um *multigrafo orientado*. Dado que o conjunto de nodos (vértices) do grafo pode ser particionado em dois conjuntos (lugares e transições), de modo que cada arco é orientado de um elemento para um conjunto (lugar ou transição) para um elemento de outro conjunto (transição ou lugar), este será um *multigrafo orientado bipartite*. Grafos para os exemplos dados, são desenhados logo a seguir cada exemplo.

No que segue, alguns exemplos de redes de Petri e seus respectivos grafos, são mostrados como está Peterson (1981).

Exemplo 1 (**Rede de Petri**):

$C = (P, T, I, O)$

$P = \{p_1, p_2, p_3, p_4\}$

$T = \{t_1, t_2, t_3, t_4\}$

$I(t_1) = \{p_1\}$ e $O(t_1) = \{p_2, p_3, p_5\}$

$I(t_2) = \{p_2, p_3, p_5\}$ e $O(t_2) = \{p_5\}$

$I(t_3) = \{p_3\}$ e $O(t_3) = \{p_4\}$

$I(t_4) = \{p_4\}$ e $O(t_4) = \{p_2, p_3\}$

Graficamente temos o seguinte grafo como na Figura 115:

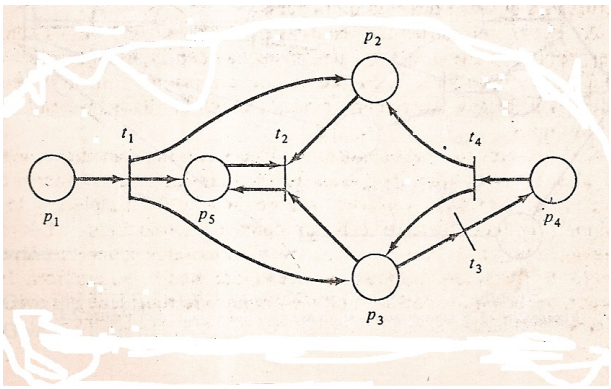


Figura 115 – Rede de Petri equivalente à estrutura do Exemplo 1.

Fonte: Petri Net Theory and The Modeling of Systems, Peterson (1981).

Exemplo 2 (**Rede de Petri**):

$C = (P, T, I, O)$

$P = \{p_1, p_2, p_3, p_4, p_5, p_6\}$

$T = \{t_1, t_2, t_3, t_4, t_5\}$

$I(t_1) = \{p_1\}$ e $O(t_1) = \{p_2, p_3\}$

$I(t_2) = \{p_3\}$ e $O(t_2) = \{p_3, p_5, p_5\}$

$I(t_3) = \{p_2, p_3\}$ e $O(t_3) = \{p_2, p_4\}$

$I(t_4) = \{p_4, p_5, p_5, p_5\}$ e $O(t_4) = \{p_4\}$

$I(t_5) = \{p_2\}$ e $O(t_5) = \{p_6\}$

Graficamente, temos o seguinte grafo como na Figura 116:

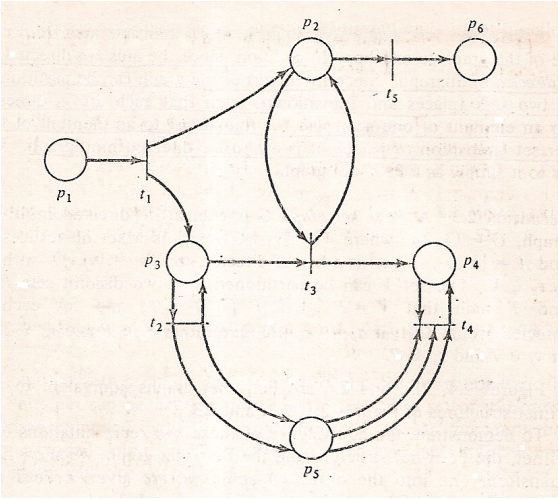


Figura 116 – Rede de Petri equivalente à estrutura do Exemplo 2.

Fonte: Petri Net Theory and The Modeling of Systems, Peterson (1981).

Exemplo 3 (**Redes de Petri**):

$C = (P, T, I, O)$

$P = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9\}$

$T = \{t_1, t_2, t_3, t_4, t_5, t_6\}$

$I(t_1) = \{p_1\}$ e $O(t_1) = \{p_2, p_3\}$

$I(t_2) = \{p_8\}$ e $O(t_2) = \{p_1, p_7\}$

$I(t_3) = \{p_2, p_5\}$ e $O(t_3) = \{p_6\}$

$I(t_4) = \{p_3\}$ e $O(t_4) = \{p_4\}$

$I(t_5) = \{p_6, p_7\}$ e $O(t_5) = \{p_9\}$

$I(t_6) = \{p_4, p_9\}$ e $O(t_6) = \{p_5, p_8\}$

Graficamente, teremos o seguinte grafo como na Figura 117:

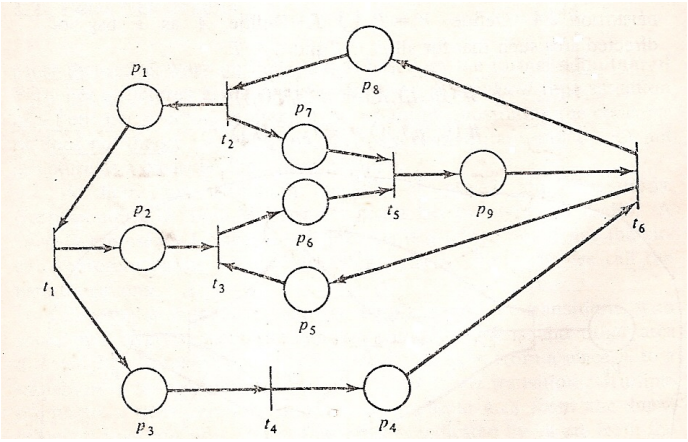


Figura 117 – Rede de Petri equivalente à estrutura do Exemplo 3.

Fonte: Petri Net Theory and The Modeling of Systems, Peterson (1981).

11.6.1 Marcação nas Redes de Petri

Uma marcação μ é uma atribuição de tokens a lugares de uma rede de Petri. Assim, como lugar e transição, o conceito de token é primitivo em Redes de Petri. Tokens são atribuídos e podem ser trazidos a residir em lugares de uma rede de Petri. O número de tokens num determinado lugar pode variar durante a execução de uma rede. Tokens são usados para definir a execução da rede.

Definição (**Marcação**)

Uma estrutura de rede de Petri, C, é uma tupla, $C = (P, T, I, O)$, onde $P = \{p_1, p_2, ..., P_n\}$ é um conjunto finito de, $n \geq 0$. $T = \{t_1, t_2, ..., t_m\}$ é um conjunto finito de transições, $m \geq 0$. O conjunto de lugares e transições são disjuntos, $P \bigcap T = \phi$. $I : T \to P^\infty$ é a função-entrada, mapeando de transições para *bag* de lugares. $O : T \to P^\infty$ é a função-saída, um mapeamento de transições para *bags* de lugares. Peterson (1981).

Uma marcação μ de uma rede de Petri $C = (P, T, I, O)$ é uma função dos conjuntos dos lugares para o conjunto dos inteiros não-negativos, $\mu : P \to \mathbb{N}$. Uma marcação pode ser denotada por um vetor $\mu = (\mu_1, \mu_2, ..., \mu_n)$, onde cada μ_i, $i = 1, ..., n$, correspondente a número de tokens no lugar p_i. As definições de uma marcação como um função ou como um vetor são relacionadas por: $\mu(p_i) = \mu_i$. Uma rede de Petri *marcada* $M = (C, \mu)$ é uma estrutura $C = (P, T, I, O)$ e uma marcação μ, como pode ser denotado por $M = (P, T, I, O, \mu)$.

Sobre o grafo de uma rede de Petri, *tokens* são representados por $\bullet$ nos cículos que representam os lugares de uma rede. Dado que o número de tokens atribuído a um lugar é ilimitado, existem infinitas marcações para uma rede de Petri. O conjunto de todas as marcações de uma rede com n lugares é o conjunto de todos os n-vetores, $\mathbb{N}^n$. Este conjunto, embora seja infinito, mas é **enumerável**.

Exemplo 4 (Um grafo marcado de uma rede de Petri)

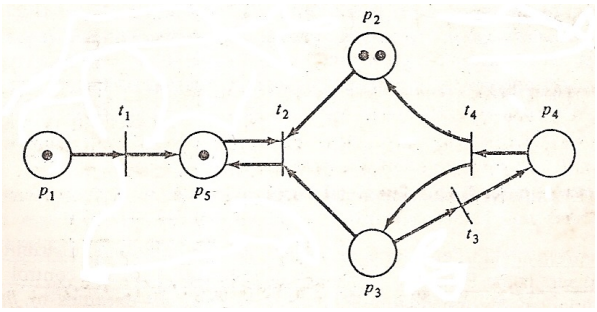


Figura 118 – Rede de Petri marcada.

Fonte: Petri Net Theory and The Modeling of Systems, Peterson (1981).

11.6.2 Regras de Execução

No exemplo da Figura 120, inicialmente, na primeira rede, as transições t_1, t_3 e t_4 estão habilitadas. Na segunda rede é mostrada a marcação resultante do disparo da transição t_4 na primeira rede. E na terceira rede, a marcação resultante da transição t_1 na segunda rede é mostrada.

Duas sequências resultam da execução de uma rede de Petri: a sequência de marcações $(\mu^0, \mu^1, \mu^2, ...)$ e a sequência de transições que foram disparadas $(t_{j_0}, t_{j_1}, t_{j_2}, ...)$.

A execução de uma rede de Petri é controlada pelo número e distribuição de tokens na rede. Tokens controlam a execução de transições. Uma rede é executada por disparar transições. Uma transição dispara por *remover tokens* de seus lugares de entrada, quantos forem os arcos de entrada em uma transição t_j, e por criar novos tokens que são distribuídos em seus lugares de saída para t_j, de acordo com o número de arcos que saem da transição.

Definição (Transição Habilitada)

Uma transição pode disparar se ela estiver *habilitada*. Uma transição está *habilitada*, se cada um dos seus lugares de entrada tem, ao menos, tantos *tokens* quantos forem os *arcos* de entrada na transição. Formalmente:

Uma transição $t_j \in T$ em uma rede de Petri com marcação μ está habilitada, se $\forall\, p_i \in P$, $\mu(p_i) \geq \#(p_i, I(t_j))$.

Seja uma marcação μ. Disparando uma trasição habilitada t_j, resulta em uma nova marcação μ' Formalmente:

$$\mu'(p_i) = \mu(p_i) - \#(p_i, I(t_j)) + \#(p_i, O(t_j)).$$

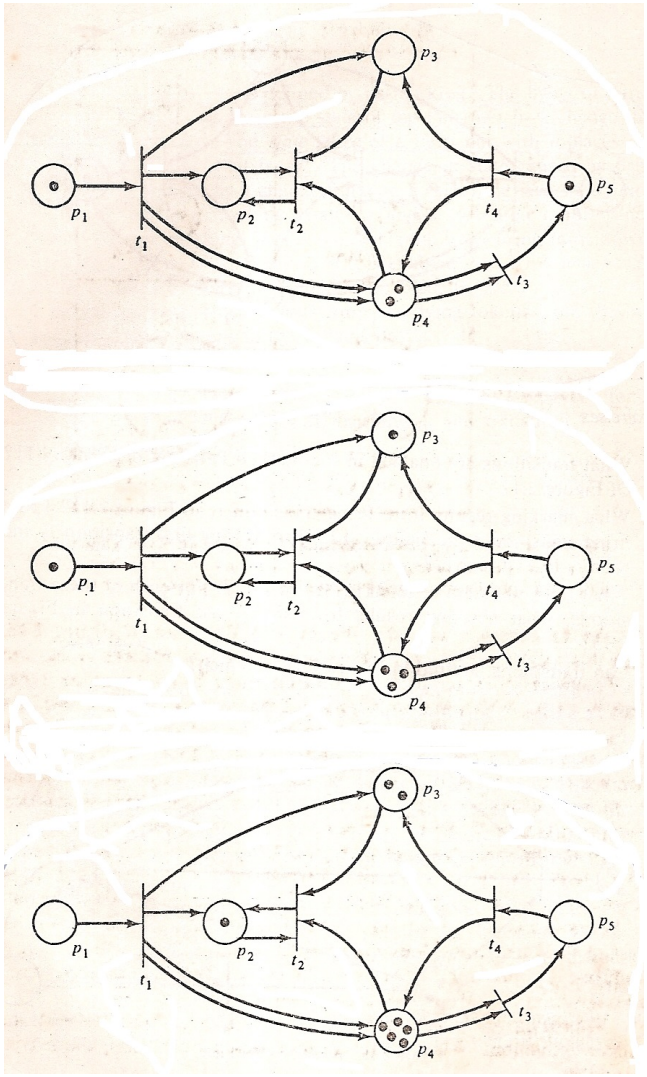


Figura 119 – Rede de Petri ilustrando as regras de disparos das transições.

Fonte: Petri Net Theory and The Modeling of Systems, Peterson (1981).

11.6.3 Espaço de Estados de uma rede de Petri

Pode relacionar uma rede de Petri com seus lugares e transições a um máquina de estado finito (MEF). O estado de uma rede de Petri é definido por sua marcação. O disparo de uma transição representa uma mudançã de estado da rede de Petri, por mudar a marcação da rede. O espaço de estados de uma rede de Petri com n lugares é o conjunto de todas as marcações. A mudança de estado, causada pelo disparo de uma transição é definida pela função δ chamada função de próximo estado. Quando aplicada a uma marcação mu (estado), esta função acarreta a nova marcação (estado) que resulta do disparo de uma transição t_j na marcação mu. Visto que t_j pode disparar somente se ela estiver habilitada, $\delta(\mu, t_j)$ fica não-habilitada na marcação μ. Se t_j estuver habilitada, então $\delta(\mu, t_j) = \mu'$, onde μ' é a marcação resultante, de remover tokens dos lugares de entrada de t_j, e adicionando tokens nos lugares de saída de t_j. Além disso, pode-se ver que uma rede de Petri em que para cada

transição somente exista um lugar de entrada e um lugar de saída, essa é equivalente a uma máquina de estado MEF.

Definição (**Função de próximo estado**)

A função próximo estado $\delta : \mathbb{N}^n \times T \longrightarrow \mathbb{N}^n$ para uma rede de Petri $C = (P, T, I, O)$ com marcação μ e transição $t_j \in T$ é definida se, e somente se, $\mu(p_i) \geq \#(p_i, I(t_j))$ para todo $p_i \in P$. Se $\delta(\mu, t_j)$ está definida , então $\delta(\mu, t_j) = \mu'$, onde:

$$\mu'(p_i) = \mu(p_i) - \#(p_i, I(t_j)) + \#(p_i, O(t_j)), \text{ para todo } p_i \in P.$$

Duas sequências resultam da execução de uma rede de Petri:

- A sequência de marcações $\langle \mu^0, \mu^1, \mu^2, \ldots \rangle$, onde μ^0 é a marcação inicial.
- A sequência de transições que foram disparadas $\langle t_j^0, t_j^1, t_j^2, \ldots \rangle$

Estas duas sequências estão relacionadas pelo relacionamento $\delta(\mu^k, t_j^k) = \mu^{k+1})$, para $k = 0, 1, 2, \ldots$. Dado uma marcação μ^0, podemos facilmente derivar a sequência de marcações para a execução da rede de Petri.

11.6.4 Estrutura das Redes de Petri

Redes de Petri são *estruturas matemáticas* que modelam sistemas de computação. Assim, representam bem a relação da matemática e da computação, como destinamos neste livro. Existem três formas de estruturas para redes de Petri: (a) Estrutura definida em *Bag*, (b) Estrutura definida em *Matriz*,(c) Estrutura definida por *Relações*.

Definição (**Estrutura definida em *Bag***)

Define-se a estrutura de redes de Petri R, como uma quíntupla $R = (P, T, I, O, K)$, onde $P = \{p_1, p_2, \ldots, p_n\}$ é um conjunto finito não-vazio de lugares, $T = \{t_1, t_2, \ldots, t_m\}$ é o conjunto finito não vazio, $I : T \to P^\infty$ é um conjunto de bags que representa o mapeamento de transições para lugares de saída. $K : P \to N \bigcup \{w\}$, é o conjunto das capacidades associadas a cada lugar, podendo assumir um valo infinito. Utiliza-se a notação $\{\ldots\}$ para conjuntos e a notação $[\ldots]$ para representar uma "*bag*" (o mesmo que multiconjunto).

Esta é a estrutura matemática utilizada na definição de redes de Petri, aqui deste texto. Uma *bag* é uma extensão de um conjunto, como se fosse um conjunto contendo elementos repetidos. Imagine uma bolsa contendo 5 bolas azuis iguais, 3 bolas amarelas iguais e 1 bola vermelha. Este é o caso de uma *bag*. Em um conjunto teríamos 1 bola de cada cor.

Definição (**Estrutura definida em *Matriz***)

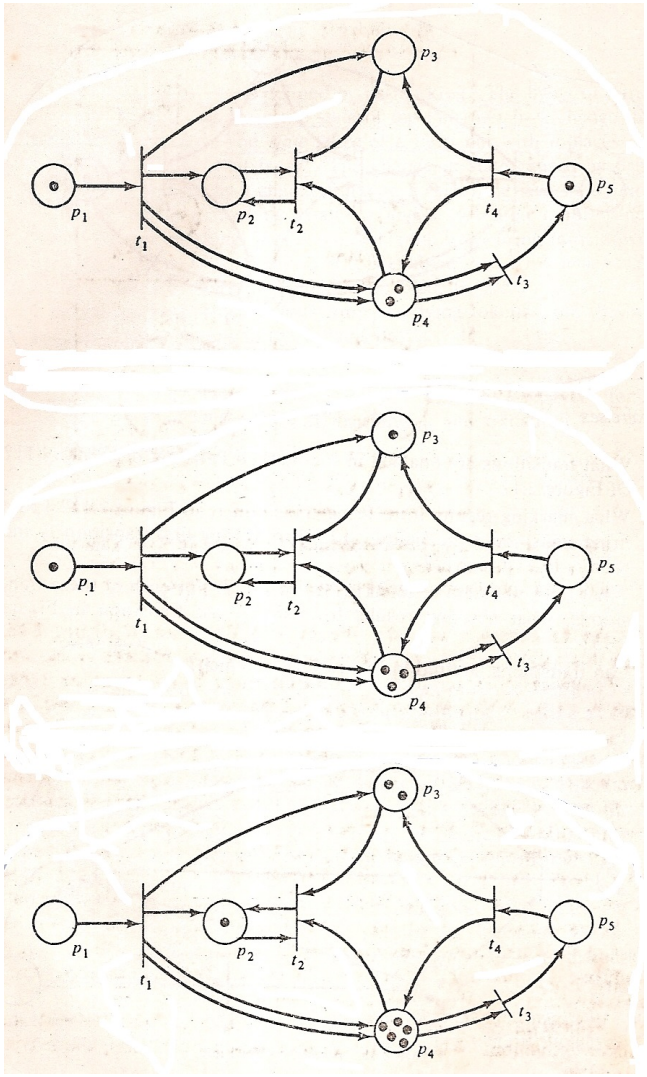


Figura 120 – Rede de Petri ilustrando as regras de disparos das transições.

Fonte: Petri Net Theory and The Modeling of Systems, Peterson (1981).

Dos resultados da álgebra matricial, pode-se utilizar matrizes para formalizar a teoria de Petri, possibilitando a análise de propriedades comportamentais e estruturais. A estrutura de uma rede de Petri definida por matriz será a tupla $C = (P, T, I, O)$, onde P é um conjunto finito de lugares, T é um conjunto finito de transições, $I : P \times T \to \mathbb{N}$ é a **matriz de pré-condições**, e $O : P \times T \to \mathbb{N}$ é a **matriz de pós-condições**, como todo grafo pode ser representado na sua forma matricial. Então, algo que se mostra numa linguagem gráfica, pode ser visualizado numa matriz, seguindo a regra de formação de passagem do grafo para a matriz. A análise de redes de Petri, agora, é baseada sobre a visão matricial de uma rede de Petri.

Uma alternativa a definição (P, T, I, O) é definir matrizes D^- e D^+ para representar as funções de entrada e saída. Cada matriz tem m linhas (uma para cada

transição) por n colunas (uma para cada lugar). Define-se $D^-[j,i] = \#(p_i, I(t_j))$ e $D^+[j,i] = \#(p_i, O(t_j))$. D^- define as entradas para as transições e D^+ define as saídas. A definição matricial de uma rede de Petri (P, T, D^-, D^+) é equivalente à definição padrão.

Exemplo (Matriz de uma rede de Petri como na Figura 121)

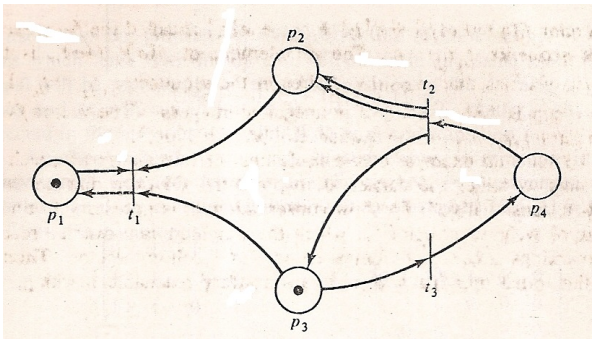


Figura 121 – Rede de Petri para ilustrar a análise da equação matricial.

Fonte: Petri Net Theory and The Modeling of Systems, Peterson (1981).

$$D^- = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$D^+ = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

E a matriz $D = D^+ - D^-$:

$$D = \begin{bmatrix} 0 & -1 & -1 & 0 \\ -1 & +2 & +1 & -1 \\ +1 & 0 & -1 & +1 \end{bmatrix}$$

O desenvolvimento da Teoria de Petri com matrizes provê uma ferramenta útil para estudar o problema da alcançabilidade. Assume-se que uma marcação μ' é alcançável a partir de uma marcação. Então, existe uma sequência δ (possivelmente nula) de disparos de transições que conduzirá de μ para μ'. Isto significa que um *vetor de disparos* é uma solução, em inteiros não-negativos, representado por x na seguinte equação:

$$\mu' = \mu + x.D$$

Se esta equação tem uma solução x em inteiros não-negativos, então μ' é alcançável de μ. Se não tem nenhuma solução então μ' não é alcançável de μ. Isto serve para

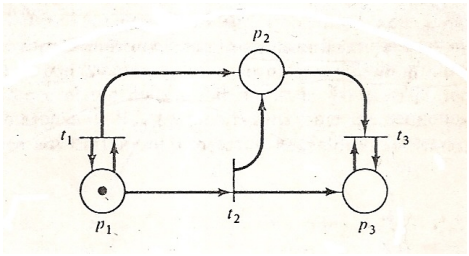


Figura 122 – Rede de Petri para ilustrar a construção de uma árvore de alcançabilidade.

Fonte: Petri Net Theory and The Modeling of Systems, Peterson (1981).

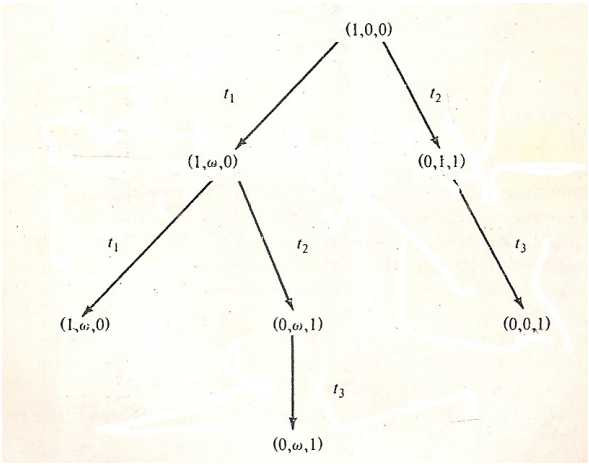


Figura 123 – Ávore de alcançabilidade para a rede da Figura 122.

Fonte: Petri Net Theory and The Modeling of Systems, Peterson (1981).

verificar se uma marcação é alcançável a partir de outra marcação, onde uma solução corresponde a uma sequência de transições. Ter ou não solução, indica a existência ou não de sequências de transições que podem disparar. E isto tem a ver com o problema de verificar a existência de não haver (tem solução) ou haver **deadlock** (não te solução), na rede de Petri que modela um sistema.

Definição (**Estrutura definida por Relações**)

Como em Maciel, Lins e Cunha (1996), define-se uma estrutura por uma quíntupla (P, T, A, V, K), onde T é o conjunto de transições, P o conjunto de lugares, A é o conjunto de arcos, V o conjunto de valorações dos arcos e K o conjunto de capacidades dos lugares. Os elementos de A são arcos arcos que concetam transições a lugares ou lugares a transições ($A \subseteq (P \times T) \bigcup (T \times P)$). Podemos dividir o conjunto A em dois subconjuntos - o conjunto das entradas das transições $I = \{(p_i, t_j)\}$e de saída das transições $O = \{(t_j, p_i)\}$. Uma **relação** é um conceito algébrico, definido na Teoria dos Conjuntos de **Cantor**, que é aqui usada para definir redes de Petri.

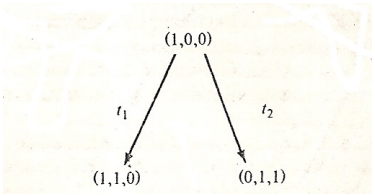


Figura 124 – A primeira etapa para construir uma árvore de alcançabilidade.

Fonte: Petri Net Theory and The Modeling of Systems, Peterson (1981).

11.6.5 Árvore de Alcançabilidade

Quando se implementa uma rede de Petri em computador, a execução da rede de Petri constrói a estrutura de uma árvore. Sobre a execução da árvore, a análise de uma rede é realizada. A análise de redes de Petri, em detalhes, pode ser encontrada nos vários livros especializados em redes de Petri, desde aqueles que utilizam a linguagem gráfica, até aqueles que se utilizam da abordagem matemática mais aprofundada.

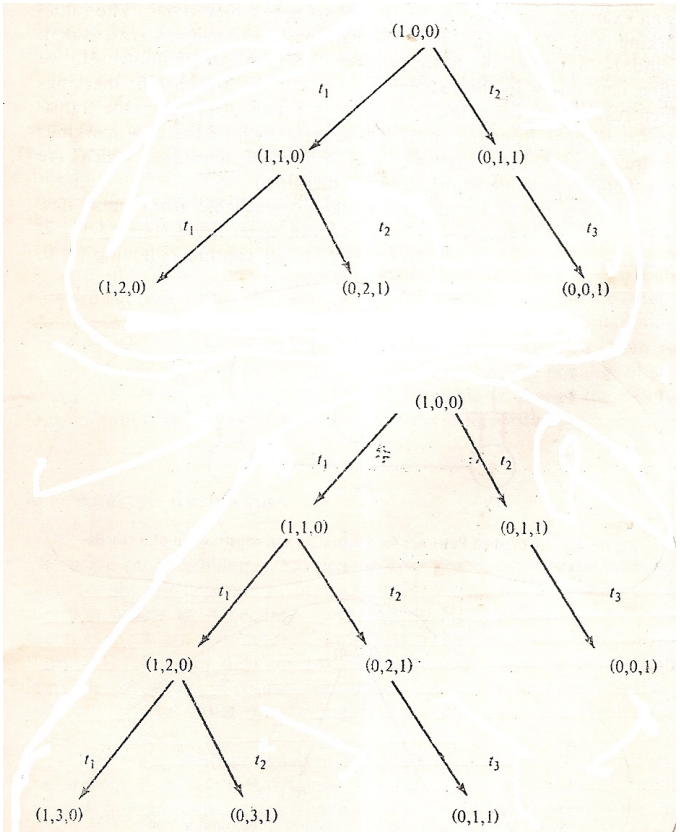


Figura 125 – Duas etapas de construção da árvore de alcançabilidade infinita.

Fonte: Petri Net Theory and The Modeling of Systems, Peterson (1981).

A evolução da execução das transições de um rede de Petri pode conduzir a visualização de uma árvore, chamada árvore de alcançabilidade da rede de Petri: uma árvore das marcações da execução da rede. Considere a rede de Petri da Figura 122. A marcação inicial é (1, 0, 0).

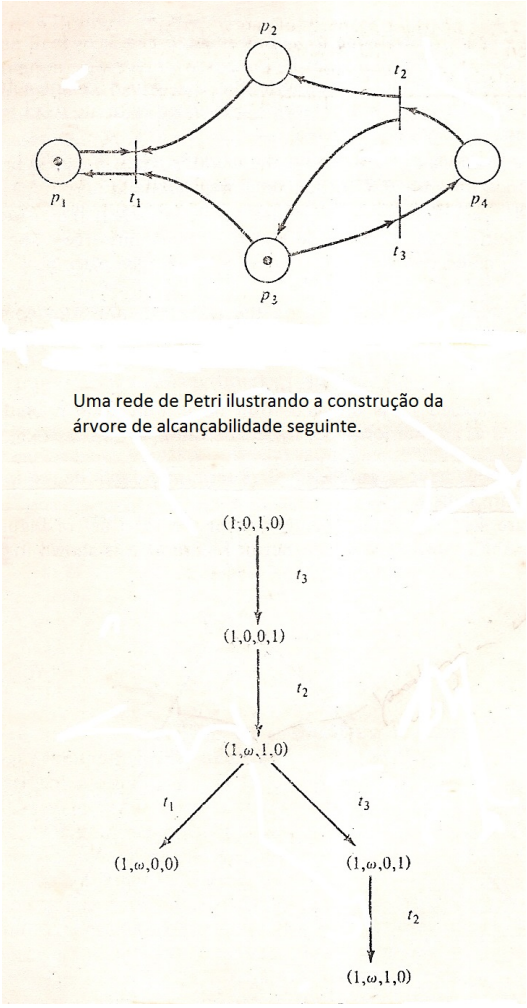


Figura 126 – Ávore de alcançabilidade para a rede de Petri ilustrada acima.

Fonte: Petri Net Theory and The Modeling of Systems, Peterson (1981).

Com esta marcação inical, duas transições são habilitadas: t_1 e t_2. Desejando-se construir a árvore de alcançabilidade inteira, define-se dois novos nodos na árvore para as marcações alcançáveis que resultam dos disparos de novas transições. Isto pode se visto na Figura 124. Estas são as marcações imediatamente alcançáveis a partir da marcação inicial. A partir das marcações da Figura 124 podemos ter todas as novas marcações. Cada um das figuras corresponde a uma etapa de construção da árvore. Duas etapas estão na Figura 125. Note que a marcação (0, 0, 1) morre na Figura 125, porque nenhuma transição é habilitada após a rede chegar nesta marcação. Note também que a marcação produzida pelo disparo de t_3 em (0, 2, 1) é (0, 1, 1) e esta marcação também foi originada diretamente da marcação (1, 0, 0) pelo disparo de t_2. Se este procedimento é repetido, toda marcação alcançável eventualmente (num tempo finito, mas não especificado) será produzida. E assim, a árvore poderá ser infinita. Para qualquer árvore com conjunto de alcançabilidade infinito, a árvore correspondente será infinita. Mas, mesmo uma rede com um **conjunto de**

alcançabilidade finito poderá ter uma **árvore de alcançabilidade infinita**.

A árvore representa todas as possíveis sequências de disparos de transições. Todo caminho na árvore iniciando no nodo raiz, corresponde a um sequência legal de transições. A ideia que foi colocada em prática, usando-se a árvore, é que ela pode ser utilizada para a contrução de ferramentas computadorizadas de análise de redes de Petri. As redes de Petri tem algumas propriedades que se satisfeitas, comprovam uma rede de Petri sem os problemas que podem surgir no funcionamento da rede que, no fundo, corresponde ao funcionamento de sistemas de computação. Por uma árvore de alcançabilidade pode comprovar redes de Petri, com as propriedades: *limitada*, *viva* e *reinicializável*.

Se a árvore é para ser uma ferramenta útil de análise, e esta pode ser infinita, então deve ser encontrado um meio para limitar esta para um tamanho finito. Se a representação de um conjunto infinito é finito, então um número infinito de marcações da árvore deve ser mapeado sobre a mesma representação. Isto em geral, resulta em perda de informação que significa que algumas propriedades da rede não podem ser determinadas, mas isto depende de como a representação é feita. Neste caso, uma redução para uma representação finita da árvore deve ser conseguida. Um meio de limitar novas marcações, quando a árvore cresce, existe, criando-se *nodos-fronteira*, introduzidos em cada etapa da construção da árvore. Isto é conseguido pelas marcações mortas (aquelas que nenhuma transição é habilitada).

Outra classe de marcações na árvore de interesse para este caso, são aquelas marcações duplicadas (nodos duplicados) e nenhum sucessor de nodos duplicados necessitam ser considerados. No final, significa que estes nodos são usados para reduzir a árvore de alcançabilidade para uma representação finita. Portanto, o número infinito de marcações, que resulta destse fatos, pode ser representado, usando-se um símbolo especial, w, que é pensado como infinito e representa o número de tokens que pode ser feito arbitrariamente grande. Para se construir uma árvore finita, existe um algoritmo, que classifica cada nodo tal como: *nodo-fronteira*, *nodo-terminal*, *nodo duplicado* ou um *nodo interno*. Cada nodo i na árvore é associado com uma marcação $\mu[i]$. A marcação é então estendida para permitir o número de tokens em um lugar da rede para ser um inteiro não-negativo ou um símbolo w. Algumas operações são definidas sobre o símbolo w e o algoritmo, o leitor pode conhecer em Peterson (1981), Cap. 4, pagina 94-95. Uma característica importante desse algoritmo para construir árvore de alcançabilidade é o fato de que ele termina. A prova está em Peterson (1981), Cap. 4, pagina 97. Um outro exemplo está na Figura 126.

11.7 Bibliografia e Fonte de Consulta

Narsinhg Deo - Graph Theory with Applications to Engineering and Computer Science, Prentice-Hall, 1974.

Thomas A. Sudkamp - Languages and Machines: an Introduction to the Computer Science Theory, Addison Wesley, 1988.

James L. Peterson - Petri Net Theory and the Modeling of Systems, Prentice-Hall, 1981.

Michael Sipser - Introdução à Teoria da Computação, Cengage-Learning, 2011.

11.8 Referências - Leitura Recomendada

Understanding Petri Nets - Modeling Techniques, Analysis ... - www.springer.com/us/-book/9783642332777.

Introductions to Petri Nets - https://www.informatik.uni-hamburg.de/.../PetriNets/...

Coloured Petri Nets Book - Department of Computer Science - www.cs.au.dk/ cpn-book/

Petri Net, Theory and Applications - InTechOpen - www.intechopen.com/books/petri_-net_theory_and_...

CAPÍTULO 12

Século XX - As Lógicas Clássicas e Não-Clássicas

Este capítulo é sobre as Lógicas clássicas e não-clássicas. Como apontado no capítulo sobre as Bases da Ciência da Computação no volume I desta série, a Ciência da Computação surgiu baseada na lógica, na estrada iniciada com Aristóteles e trilhada posteoriormente por Leibniz, George Boole, Frege, Gödel, Russell e Whitehead, e os demais contemporâneos destas ultimas décadas. Sem lógica, não existiria a Ciência da Computação.

12.1 Lógicas clássicas

Uma lógica clássica identifica uma classe de Lógica matemática que têm sido mais intensamente estudado e mais amplamente utilizado. A classe é, por vezes, chamada de lógica padrão. Elas são caracterizadas por um número de propriedades:

- Lei do terceiro excluído e Dupla negação;
- Princípio da não contradição, e o Princípio de explosão;
- Monotonicidade de vinculação e Idempotência de vinculação;
- Comutatividade da conjunção;
- Teoremas de De Morgan: cada conectivo lógico é duplo a outro;

Enquanto não implicar com as estas condições mencionadas, as discussões contemporâneas da lógica clássica normalmente incluem apenas **Lógica Proposicional** e a **Lógica de Primeira Ordem**. A semântica da lógica clássica é bivalente, avaliando ou interpretando-se em **verdade** ou **falso**.

Exemplos (**Lógicas clássicas**)

1. A reformulação algébrica da lógica de **George Boole**, o seu sistema de Álgebra Booleana.

2. A lógica de primeira ordem encontrada em Gottlob Frege's Begriffsschrift (Lógica e o Cálculo dos Predicados).

Além destas lógicas, existem outras também bem apropriadas para sistemas de computação, que são baseadas nestes mesmos valores semânticos de **verdade** e **falso**: **Lógica Temporal** e a **Lógica Modal**. Ver nas próximas seções.

12.2 Lógica Temporal

Lógica Temporal é qualquer sistema de regras e símbolos para representar e dissertar, sobre proposições qualificadas em termos de tempo. Não em termos específicos de instantes de tempo, mas sobre o intervalo de tempo que transcorre durante a execução de um programa de computador. A lógica temporal é um subconjunto da Lógica Modal (a ser explicada na próxima subseção) que possui como objetivo permitir a variação da veracidade das asserções ao longo do tempo. Ou seja, uma asserção pode ser verdadeira num tempo, mas já não o ser no tempo seguinte.

Se pensarmos no funcionamento de um sistema de computação, através de seus estados,
do sistema e outras que são verdadeiras em determinados estados. O que mostra a ultilidade de uma Lógica Temporal sobre a execução de um sistema de computação. Todo método formal tem um modelo-base, e este, para a Lógica Temporal, é um modelo implícito de estados e transições, dos quais se descreve as propriedades do sistema sobre as suas trajetórias de execução.

Em lógica temporal podemos, por exemplo, expressar sentenças como "Eu estou *sempre* com sede", ou *Eventualmente*, "eu estarei com sede", ou "Eu estarei com sede *até que* eu beba água". Subsequentemente, tem sido desenvolvida por cientistas da computação como **Amir Pnueli** e outros lógicos. **Amir Pnueli** (1941-2009) foi um cientista da da computação, israelense. Introduziu a lógica temporal na Ciência da Computação e contribuiu para a *verificação formal de programas e sistemas concorrentes e distribuídos*, recebendo por isto o Prêmio Turing de 1996.

Lógica temporal encontrou uma importante aplicação em verificação formal das propriedades de sistemas de computação concorrentes, onde é usada para declarar requisitos de sistemas de *hardware* ou *software*. Por exemplo, alguém pode desejar dizer que sempre que uma solicitação é feita, o acesso ao recurso é eventualmente garantido, mas nunca será garantido a dois solicitantes simultaneamente. Uma declaração desse tipo pode ser expressada convenientemente em lógica temporal.

Zohar Manna (1939) é um cientista da computação, estadunidense. É professor da Universidade Stanford e autor de *The Mathematical Theory of Computation*,

Figura 127 – Pnueli - A lógica temporal na Ciência da Computação.

Fonte: www.nae.edu.

um dos primeiros livros a conter cobertura extensiva dos conceitos matemáticos da programação de computadores. Com **Amir Pnueli** foi coautor com Zohar Manna, de uma coleção de livros texto sobre lógica temporal e verificação de sistemas reativos: *The Temporal Logic of Reactive and Concurrent Systems: Specification*, *The Temporal Logic of Reactive and Concurrent Systems: Safety* e *The Temporal Logic of Reactive and Concurrent Systems: Progress*.

Figura 128 – Manna - The Mathematical Theory of Computation.

Fonte: Goolge Images - theory.stanford.edu.

12.2.1 Variantes da Lógica Temporal

Existem algumas variantes da Lógica Temporal, baseadas na proposicional ou na predicativa de primeira ordem. A linha do tempo pode ser visto de forma **linear** (LTL) ou (LTLPo) e **ramificando o tempo** (LTTR). Além destas existe a Lógica de Conhecimento e Crença (LCC) e a Lógica de Conhecimento, Crença e Tempo (LTCC). Excetuando a LTLPo (que é provada ser apenas consistente), as demais são **consistentes** e **completas**. Para entender os conceitos de consistência e completude, ver o capítulo sobre Gödel no volume I desta série. Em Vasconcelos (1989), pode ser visto um estudo completo sobre estas lógicas, mostrando exemplos de utilização na especificação formal de sistemas distribuidos

Lógica Temporal tem aplicação na especificação de sistemas concorrentes, paralelos ou distribuidos de computação, visando a prova de correção da futura implementação de um sistema. Também podem especificar propriedades de trajetórias de estados de um programa de computador, em sistemas que tratam a mobilidade sobre trajetórias de objetos móveis.

A utilização da Lógica Temporal à Ciência da Computação, pode ser vista no exemplo da especificação de uma sistema de computação cliente-servidor, através do método orientado à propriedade, também chamado de método axiomático.

12.2.2 Especificando Sistemas de Computação com Lógica Temporal

Nesta subseção é mostrado um exemplo de especificação Vasconcelos (1989), seguindo um formalismo orientado à propriedades. Um sistema distribuído cliente-servidor, composto por três processos P_1, P_2 e um servidor S, e mais um recurso a ser compartilhado em S pelos dois processos P_i, $i = 1, 2$. O sistema é especificado através de suas propriedades que devem existir durante o funciomento do sistema, utilizando uma Lógica Temporal Linear (LTL). O recurso deve ser utilizado de forma exclusiva (exclusão mútua) por cada dos processos. Tal recurso pode ser uma impressora, um programa de computador, um arquivo ou mesmo um banco de dados acessado via o servidor S. Para maiores detalhes sobre esta lógica temporal, o leitor pode consultar Vasconcelos (1989).

Um exemplo mostrado em Vasconcelos (1989), descreve um esquema deste sistema de computação cliente/servidor, onde os processos estão representados por círculos e o recurso em um retângulo. As retas orientadas representam os canais de comunicação. Também é assumido que não há perdas de mensagens (como se os canais fossem totalmente confiáveis) e, assim todas as mensagens enviadas sobre os canais chegam ao destino.

O sistema funciona da seguinte maneira: o processo P_i manifesta seu desejo de utilizar o recurso enviando um pedido para o processo S. O processo S recebe o pedido e, se o recurso estiver disponível, autoriza o processo P_i a utilizar o recurso.

Esta autorização é uma mesnagem enviada por S para P_i. Quando P_i recbe essa autorização, ele começa a usar o recurso. O uso do recurso pelos processos P_i se realiza em um tempo finito e assim, eventualmente (num tempo finito, mas não especificado) o processo liberará o recurso. Quando isto ocorrer o processo P_i envia para S uma mensagem avisando que está liberando o recurso.

Por questão de simplicidade, não há nenhum tipo de *bufferização* de mensagens - estas são entregues à medida que são enviadas. Por isso, quando um processo atinge um comando de envio de mensagem (ou recebimento de mensagem), ele espera até que o outro processo envolvido na comunicação atinja o comando de recebimento (ou de envio). Isto caracteriza a comunicação síncrona entre P_i e S.

12.2.3 Uma Especificação Formal em Lógica Temporal Linear

Seguindo **??**), a especificação formal do sistema distribuído cliente-servidor (P_i, S), $i = 1, 2$ será feita apenas em função de mensagens enviadas/recebidas. Serão usadas as seguintes proposições com os significados seguintes:

- p_i: o processo P_i enviou um pedido para S, manisfestando a sua necessidade de usar o recurso S.
- a_i: o processo S enviou uma autorização para P_i, autorizando P_i a usar, de modo exclusivo, o recurso.
- l_i: o processo P_i enviou uma mensagem informando que o recurso está liberado.

Como a comunicação das mensagens é sincronizada, o envio e o recebimento de uma mensagem pode ser visto como uma única ação. Portanto, p_i representa tanto o envio, como o recebimento do pedido. O mesmo vale para a_i e l_i.

No que segue, algumas propriedades que o sistema deve possuir:

Propriedade 1 - Como há uma comunicação síncrona, sempre é verdade que, somente um dos fatos, p_i, a_i e l_i pode se verificar a cada vez. Isto é, sempre é verdade que, se p_i for verificado, então nem a_i, nem l_i são verificados, o que pode ser formalizado com a seguinte fórmula da Lógica Temporal Proposicional:

$\psi_1 a : \Box[p_i \supset \sim (a_i \vee l_i)]$

Similarmente, sempre é verdade que, quando a_i for verificado, nem p_i, nem l_i são verificados:

$\psi_1 b : \Box[a_i \supset \sim (p_i \vee l_i)]$

Também, sempre é verdade que, quando l_i for verificado, nem p_i, nem a_i são verificados:

$\psi_1 c : \Box[l_i \supset \sim (p_i \vee a_i)]$

Propriedade 2 - Sempre é verdade que, todo pedido feito deve eventualmente ser autorizado. Isto é formalizado como segue:

$\psi_2 : \Box(p_i \supset \Diamond a_i)$

Propriedade 3 - Assumiu-se que o tempo de uso do recurso pelos processos p_i é finito. Logo, sempre é verdade que, após, uma autorização ser dada pelo servidor, eventualmente, o processo p_i que utiliza o recurso liberará o mesmo:

$\psi_3 : \Box(a_i \supset \Diamond l_i)$

Propriedade 4 - Sempre é o caso que, se dada uma autorização a_i a p_i, então não pode ser dada uma autorização a um p_j, até que p_i libere o recurso:

$\psi_4 : \Box[a_i \supset (\sim a_j \mathcal{U} l_i)]$ para $i, j \in \{1, 2\}$, $i \neq j$

12.3 Lógicas Não-clássicas

Com o advento da lógica algébrica tornou-se evidente que o cálculo proposicional clássico admitisse outras semânticas. Elementos intermediários da álgebra correspondem a outros valores, exceto "verdadeiro" e "falso". O princípio da bivalência prende somente quando a álgebra booleana é considerado como sendo a álgebra de dois elementos, o que não tem elementos intermediários.

Lógica *não clássicas* surgiram como sistemas que vão além dessas duas distinções (verdadeiro e falso) que são lógicas não-aristotélicas, ou lógica de vários valores ou ainda lógicas polivalentes. No início do século XX, **Jan Lukasiewicz** (1878-1956)investigou a extensão dos tradicionais valores verdadeiro/falso para incluir um terceiro valor, "possível", ficou reconhecido pelo seu desenvolvimento da lógica polivalente (lógica difusa) e seus estudos sobre a história da lógica, particularmente sua interpretação da lógica aristotélica. Lógicas como a lógica difusa foram então desenvolvidas com um número infinito de "graus de verdade", representados, por exemplo, por um número real entre 0 e 1.

Lógicas não-clássicas é o nome dado aos sistemas formais que diferem de maneira significativa dos sistemas lógicos padrão, que baseados na lógica proposicional e na lógica dos predicados. Existem várias maneiras em que isto é feito, por meio de extensões e variantes. O objetivo é tornar possível construir diferentes modelos de consequência lógica e verdade lógica.

Exemplos (**Lógicas não-clássicas**)

- Lógica computacional é uma teoria formal semanticamente construído de computabilidade, em oposição à lógica clássica, que é uma teoria formal da *verdade*, que se integra e se estende a lógica clássica, linear e lógica intuicionista.
- Lógica polivalente, incluindo a lógica fuzzy, que rejeita a lei do terceiro excluído e permite que um valor de verdade seja qualquer número real entre 0 e 1.
- Lógica intuicionista rejeita a lei do terceiro excluído, a eliminação dupla negativa, e as leis de a De Morgan;
- Lógica linear rejeita idempotência de vinculação;
- Lógica modal estende a lógica clássica com operadores não-verdade-funcionais ("modal");
- Lógica paraconsistente (por exemplo, dialeteísmo e lógica da relevância) rejeita a lei da não-contradição;

12.4 Lógica Difusa

A Lógica Difusa (*Fuzzy Logic*) é uma extensão da lógica booleana que admite valores lógicos intermediários entre o FALSO (0) e o VERDADEIRO (1); por exemplo o valor médio 'TALVEZ' entre 0 e 1 com um valor intermediário de (0,5). Isto significa que um valor lógico difuso é um valor qualquer no intervalo de valores entre 0 e 1.

A lógica difusa deve ser vista mais, como aplicada a uma família de modelos matemáticos dedicados ao tratamento da incerteza, do que uma lógica propriamente dita.

Embora a Lógica Difusa tenha se consolidado ao longo do século XX, suas origens remontam até 2.500 anos. A este respeito, **Aristóteles** acreditava que haviam certos graus de verdade e falsidade. **Platão** também tinha considerado graus de pertinência. O filósofo irlandês do século XVIII e bispo anglicano **David Hume**, acreditava na lógica do bom senso, raciocínio baseada no conhecimento que as pessoas normalmente adquirido através de experiências no mundo. A corrente de pragmatismo fundada no início deste século por Charles Sanders Peirce, foi o primeiro a imprecisão, ao invés de verdadeiro ou falso, como uma abordagem para o mundo e para o raciocínio humano.

O filósofo e matemático britânico **Bertrand Russell**, no início do século XX, estudou a imprecisão da, concluindo com precisão que a imprecisão é um grau. O filósofo austríaco **Ludwig Wittgenstein** estudou as maneiras em que uma palavra poderia ser usada para muitas coisas que têm algo em comum.

A primeira lógica de imprecisão foi desenvolvido em 1920 pelo filósofo **Jan Lukasiewicz**, que visualizou os conjuntos com possíveis graus de pertinência de 0 e 1; depois estendido para um número infinito de valores entre 0 e 1.

No século XX, início dos anos 60, **Lotfi Zadeh**, um engenheiro elétrico iraniano, nacionalizado nos Estados Unidos, Professor de Engenharia Elétrica da Universidade da Califórnia, em Berkeley, e outras universidades americanas de prestígio, Doutor Honoris Causa de várias instituições acadêmicas, trouxe o fundamento teórico do Fuzzy Logic, que combina os conceitos de lógica e dos conjuntos de **Lukasiewicz**, mediante a definição dos graus de pertinência. A motivação original foi lidar com os aspectos imprecisos do mundo real, criando "um sistema que fornece um caminho natural para tratar problemas em que a fonte de imprecisão é a ausência de critérios claramente definidos".

Lotfali Askar-Zadeh(1921) na Figura 129, é conhecido por introduzir em 1965 a teoria de conjuntos difusos ou **Lógica Difusa** (em inglês, Fuzzy Logic).

Figura 129 – Zadeh - o desenvolvedor da Lógica Difusa.

Fonte: www.cs.berkeley.edu, pt.wikipedia.org.

Esta é uma lógica bem aplicada na área da Inteligência Computacional (IC) Goebel. (1998). Por **Inteligência Computacional** (IC), queremos dizer sobre um conjunto de metodologias computacionais e abordagens inspiradas na natureza para lidar com problemas complexos do mundo real, nos quais as abordagens tradicionais são ineficazes ou inviáveis. Nesta linha estão a **computação evolucionária** e os algoritmos genéticos que são tratados nesta área como em Lam HK; Ling (2012).

Por **problemas complexos do mundo real** se almeja problemas sem simplificações, ou pelo menos, com poucas simplificações. Enquanto, por abordagens tradicionais envolvem metodologias desenvolvidas há tempos, em alguns casos, para separar dados, não existe algoritmo. No *controle inteligente*, basicamente se propõe formas que técnicas tradicionais da *teoria de controle* não podem resolver ou resolvem de forma ineficiente. Possivelmente, um exemplo prático e importante, seja a situação de um médico tratando um paciente com câncer. Note o grau de incerteza que deve existir num tratamento complexo como o do câncer. Lógica Fuzzy permitiu o desenvolvimento de aplicações práticas.

Em 1971, **Zadeh** publicou o artigo, *Quantitative Fuzzy Semantics*, que introduz os elementos formais que compõem o corpo da Lógica Fuzzy e suas aplicações, como são conhecidos actualmente. Em 1973, **Zadeh** apresenta a teoria básica ddos controladores nebulosos. A partir desta publicação, outros pesquisadores começaram a aplicar a lógica fuzzy para controlar vários processos, por exemplo, o britânico **Ebrahim Mandani**, que em 1974 desenvolveu o primeiro sistema prático de Controle Fuzzy: regulação de uma máquina a vapor.

A solução implementada pela **Mandani** introduziu os conceitos necessários para a aplicação em áreas industriais. Sua aplicação na área de controle da base de operadores humanos são capazes de realizar em muitos casos, um controle mais eficaz do que controladores automáticos tradicionais, porque eles são capazes de tomar decisões corretas com base na informação linguística imprecisa.

Em 1978 começa a publicação da revista *Fuzzy Sets and Systems*, com uma ou duas edições mensais, apoiando o desenvolvimento da teoria de conjuntos fuzzy e sistemas e aplicações. Esta revista é publicada pela IFSA (International Fuzzy Systems Association).

Também se pode ressaltar que em 1980, o desenvolvimento do primeiro sistema comercial de controle difuso, ao aplicar-se esta técnica para o controle de fornos rotativos, desenvolvido por engenheiros dinamarqueses **Peter Lauritz Holmbland** e **Jens-Jurgen Ostergaard**.

No ocidente, assumiu-se uma relutância, principalmente por duas razões: a primeira, porque a palavra *fuzzy* sugeria algo confuso e sem forma, e segundo, porque não havia nenhuma maneira de provar analiticamente a teoria funcionando corretamente, uma vez que o controle fuzzy não surgiu com base em modelos matemáticos. No entanto, surgiram alguns pesquisadores japoneses no campo da *lógica fuzzy*.

Em 1987, se inaugura no Japão,o metrô de Sendai, um dos sistemas de controle *fuzzy* mais espetaculares criados pelo homem. Desde então, o controlador inteligente tem mantido os trens funcionando eficientemente.

12.4.1 Aplicações da Lógica Difusa

(1) **Faixa Exclusiva de Ônibus em Tráfego Metropolitano** - Umas das aplicações interessantes da Lógica Difusa está no trabalho sobre *Sistema de Controle de Tráfego Metropolitano em Rodovias Dotadas de Faixas Exclusivas para Ônibus* Sousa (2005).

(2) **Aplicação aos Bancos de Dados Relacionais** - Uma aplicação muito interessante da Lógica Difusa à Ciência da Computação está em Biazin (2002). Este trabalho apresenta a intergação de Lógica Fuzzy à Bancos de Dados Convencionais, expondo os métodos e práticas utilizadas para integração das duas tecnologias.

Através da inclusão da uma máquina de inferência fuzzy interna ao SGDB, permite consultar dados em forma nebulosa. Este procedimento utiliza entradas e saídas convencionais, e regras definidas no modelo, possibilitando que uma consulta seja feita sem alterar os padrões estabelecidos na linguagem SQL. Os sistemas de banco de dados atuais, não contemplam a teoria dos conjuntos nebulosos, porém constantemente é necessária sua utilização. Por ser um sistema largamente utilizado em diversas áreas, essa dificuldade é maior percebida nos sistemas gerenciadores de banco de dados relacionais. Portanto nem sempre os valores a serem manipulados são valores exatos, como por exemplo: *selecione todos os clientes que possuam uma credibilidade de pagamento alta*. A credibilidade do cliente depende de vários fatores que devem ser levados em consideração, e não somente se o cliente está com dívidas ou não. Mas, os sistemas financeiros só levam em conta se há dívidas ou não.

(3) **Aplicação à Dozimetria da Pena** - Uma outra aplicação computacional é o sistema de apoio à Dosimetria da Pena utilizando a Lógica Difusa, que está em Guimarães (2004). Das três sucessivas fases que compõem a Dosimetria da Pena, a primeira refere-se às Circunstâncias Judiciais e a segunda às Circunstâncias Legais Agravantes e Atenuantes, previstas nos artigos 59, 61, 62, 65 e 66 do Código Penal Brasileiro. Sendo que as penas dos crimes previstos desde o art. 121 ao art. 359 estão definidas em um intervalo, é necessário ao magistrado aplicar as circunstâncias e extrair um valor exato de pena. Entretanto, para aplicá-las, o Código Penal não prevê uma quantidade ou um *quantum* para cada uma delas. Esta situação gera um fator de incerteza. Assim, os magistrados estão involuntariamente sujeitos a aplicar diferentes penas para crimes similares. Para tratamento de incertezas, a Lógica Difusa ou Nebulosa é considerado uma excelente técnica, sendo usada em diferentes áreas de interesse. A implementação na Dosimetria da Pena inicia-se atribuindo um valor de importância das circunstâncias em uma função matemática, chamada *função de pertinência*, e esta determinará o *grau de pertinência* em subconjuntos difusos. Um grupo de regras de controle tratará condicionalmente da equivalência das circunstâncias em que é dividido o domínio total de variação dos valores atribuídos às circunstâncias. A ação das regras com os respectivos graus de pertinência formam o processo de Fuzzificação. O método do Centro de Gravidade é utilizado no processo de Defuzzificação, resultando um valor exato que representa o "quantum" de agravação ou atenuação de pena referente as duas primeiras fases da Dosimetria da Pena. Certamente, nos processos de julgamento do Supremo Tribunal Federal não se aplica ainda a dozimetria usando-se Lógica Fuzzy.

(4) **Aplicação em Robótica** - Em Hernandez (), *Fuzzy Logic* provou ser uma ferramenta particularmente útil no campo de robótica, caracterizado por Inteligentes (2005):

- A indisponibilidade de um modelo matemático confiável de um ambiente real, quando este atinge um nível mínimo de complexidade.
- A incerteza e imprecisão dos dados fornecidos pelos sensores.

- A necessidade de operar em tempo real.
- A presença de incerteza no conhecimento que temos do meio ambiente.

Nesta aplição, existem diferentes formas de incerteza. Então, se você diz que "*o robô está na loja*" esta é uma informação imprecisa, porque não fornece uma única posição do robô. Se as informações fornecidas é que "*o robô está aproximadamente no centro da loja*" esta informação é vaga porque a posição fornecida não é preciso. Por fim, a frase "*o robô estava ontem na posição (2, 3)*" fornece uma informação confiável, enquanto o robô não possa mais estar nessa posição. Em todos os três casos, a informação pode ser descrito como uma incerta, uma vez que não é possível saber com precisão a posição atual real do robô.

Qualquer tentativa de controlar um sistema dinâmico precisa usar qualquer modelo de sistema de conhecimento ou controle, para o sistema robótico que consiste no próprio robô e o ambiente em que opera. Embora geralmente você pode obter o modelo de robô, não é verdade quando se considera o robô localizado em um ambiente desestruturado. Os ambientes são caracterizados por uma forte presença de incerteza, devido, por exemplo, a existência de pessoas que se deslocam, os objetos podem mudar de posição, novos obstáculos podem aparecer,entre outros.

Além disso, há vários fatores que podem levar a um sistema robótico para um estado errado durante a execução de uma seqüência de tarefas: erros sensoriais devido a fatores do ambiente de trabalho, processo de informações imprecisas, desinformação, entre outros. Neste sentido, o sistema **Fuzzy Logic** incorpora a capacidade de recuperar de quaisquer erros, apresentando, assim, tanto robustez na detecção e recuperação destas condições de erro.

O tratamento da incerteza permite representar, de forma aproximada, a geometria do problema, ordenar as diferentes alternativas (subtarefas) em função da pertinência a estados anteriores, fazendo o tratamento de incerteza nas medições dos sensores.

Uma das aplicações mais comuns de técnicas difusas é projetar comportamentos. Comportamentos são tarefas tais como, por exemplo: evitar os obstáculos fixos, seguir um contorno, evitar obstáculos em movimento, atravessar portas, seguir um caminho, empurrar ou carregar um objeto. Essas tarefas são de complexidade muito diferente. Os controladores *fuzzy* incorporam o conhecimento heurístico em forma de regras do tipo "se-então" e são uma alternativa adequada em caso de não se poder obter um modelo preciso do sistema a se controlar.

(5) **Avaliação da Qualidade da Água** - O trabalho em Avaliação... (2012) aborda a avaliação da qualidade da água utilizando a teoria da lógica *fuzzy* (difusa). A qualidade da água, atualmente, é tratada em todos os âmbitos da engenharia sanitária, com vista a um desenvolvimento sustentável do meio ambiente. Neste

cenário, a avaliação da qualidade da água é essencial e, neste trabalho, esta foi realizada através do monitoramento dos parâmetros indicadores do nível trófico do reservatório de água Jacarecica I no Estado de Sergipe. Neste, foram realizadas análises físico-químicas e constituída uma linguagem *fuzzy* para classificação dos dados ambientais. A lógica *fuzzy* é uma técnica de inteligência artificial que se baseia no conhecimento especialista do fenômeno a ser tratado, para se ter uma linguagem heurística traduzida numericamente. Para utilização dessa teoria *fuzzy*, foi calculado o índice trófico do reservatório através da correlação de Carlson, modificada e estabelecidos os limites para as variáveis *fuzzy*. As variáveis químicas utilizadas foram a clorifila, o fosforo e o nitrogênio total e, foram traduzidas em quatro níveis de eutrofização do corpo hídrico (Oligotrófico, Mesotrófico, Eutrófico e Hipereutrófico). Dessa forma, os resultados mostraram que o nível de qualidade da água do reservatório está entre oligotrófico e eutrófico e a utilização da lógica fuzzy foi aplicada de forma satisfatória para mostrar esses níveis de eutrofização.

(6) **Problemas de Decisão Multicritério** - Em Oliveira (2003b), o trabalho de mestrado em Ciência da Computação "Utilizando integrais fuzzy em tomada de decisão multicritérioos" aborda os métodos tradicionais para avaliação de problemas de decisão multicritério. Estes problemas, geralmente, são tratados por modelos matemáticos que agregam de forma aditiva os fatores submetidos à uma avaliação, como exemplo, a média ponderada. Embora fáceis de se aplicar, as médias, muitas vezes, não contemplam os critérios de forma conjunta, especialmente quando as grandezas a serem medidas não são independentes e não têm uma métrica objetivamente mensurável, ou seja, quando são de caráter subjetivo. Estimulados pelo seu desenvolvimento e pela praticidade de aplicação, as metodologias multicritério vêm sendo amplamente aceitas para apresentar de forma inovadora os modelos de avaliação, quando se propõem a trabalhar com múltiplos critérios, onde define-se limites de valores e graus de confiança. Ao analisarmos o método da Integral Fuzzy, concluímos que a técnica é capaz de auxiliar na busca de decisões, quando participantes do processo expõem suas preferências em variações de valores que são interpretados e executados pelos modelos. Diante do contexto, este trabalho vem apresentar a metodologia da Integral Fuzzy, aplicada em um problema multicritério de tomada de decisão. Aponta também as vantagens da Integral Fuzzy modificada sob sua forma original mostrando um exemplo da aplicação através de uma avaliação de fatores poluentes do Rio Cuiabá (MT).

12.5 Lógica Paraconsistente

A Lógica Paraconsistente inclui-se entre as chamadas lógicas não-clássicas, por derrogar alguns dos princípios basilares da Lógica clássica, como por exemplo, o princípio da contradição. Segundo a Lógica Paraconsistente, uma sentença e a sua negação podem ser ambas verdadeiras.

A Lógica Paraconsistente apresenta alternativas às proposições clássicas, cuja con-

clusão pode ter valores além de verdadeiro e falso - tais como indeterminado e inconsistente. O termo *paraconsistente* significa "além do consistente", e foi cunhado em 1976, pelo filósofo peruano **Francisco Miró Quesada**.

A motivação primordial para a lógica paraconsistente é a convicção de que deve ser possível raciocinar com informações inconsistentes de modo controlado.

Por exemplo, considere a afirmação "*o homem é cego, mas vê*". Segundo a lógica clássica, a sentença é uma contradição; o indivíduo que vê, que é um "não-cego", não pode ser cego; já na Lógica Paraconsistente, ele pode *ser cego para ver algumas coisas*, e *não-cego para ver outras coisas*. Tal exemplo corresponde a um paradoxo e no estudo da semântica, a Lógica Paraconsistente aplica-se especialmente aos paradoxos. No contexto deste livro, sua aplicação é relevante para as áreas da matemática, computação e inteligência artificial aplicada à robótica. Será que se a Lógica Paraconsistente existisse na época de Georg Cantor, o paradoxos encontrados poderiam ser evitados?

Um dos primeiros a desenvolver os sistemas formais da Lógica Paraconsistente foi o brasileiro **Newton Carneiro Affonso da Costa** (Curitiba, 1929). Newton da Costa, como é conhecido é um matemático, lógico e filósofo brasileiro, conhecido por seus trabalhos em lógica.

Figura 130 – Newton da Costa - um dos criadores da Lógica Paraconsistente.

Fonte: pt.wikipedia.org.

12.5.1 Exemplo de Lógica Paraconsistente

Um conhecido sistema de lógica paraconsistente é o simples sistema conhecido como LP (Logical Paradox), primeiro proposto pelo lógico argentino **F. G. Asenjo** em 1966 e depois popularizado por **Priest** e outros 9 . Um modo de apresentar a sistemática para LP é substituir a usual valoração funcional por uma relacional 10 . A relação binária V relaciona uma fórmula a um valor *verdade*:

- V(A,1), significa que A, é verdadeiro.

- V(A,0), significa A, é falso.

Uma fórmula deve ser associada pelo menos um valor verdade, mas não existe requerimento que será associado no máximo um valor verdade. A cláusula da semântica para negação e disjunção são dados a seguir:

- $V(\neg A, 1) \Leftrightarrow V(A, 0)$.
- $V(\neg A, 0) \Leftrightarrow V(A, 1)$.
- $V(A \vee B, 1) \Leftrightarrow V(A, 1)$ *or* $V(B, 1)$.
- $V(A \vee B, 0) \Leftrightarrow V(A, 0)$ *and* $V(B, 0)$.

Outros conectivos lógicos são definidos em termos da negação e da disjunção usual. Ou para colocar o mesmo ponto menos simbolicamente.

- não A é verdadeiro, se e somente se, A é falso.
- não A é falso, se e somente se, A é verdadeiro.
- A ou B é verdadeiro, se e somente se, A é verdadeiro ou B verdadeiro.
- A ou B é falso, se e somente se, A é falso e B é falso.

Quanto a semântica, a consequência lógica é então definida como preservação da verdade.

$\Gamma \vDash A$, se e somente se, A é verdade quando todo elemento de Γ é verdadeiro.

Agora considere a valoração V tal que $V(A, 1)$ e $V(A, 0)$ mas não é o caso que $V(B, 1)$ é fácil checar que o valor constitui um contra-exemplo para ambos, explosão e silogismo disjuntivo. Contudo, é também um contra-exemplo ao modus ponens pelo material condicional de LP. Por esta razão, proponentes de LP, usualmente, defendem a expansão do sistema que inclui um conectivo condicional mais forte que não é definível em termos da negação e disjunção.

12.5.2 Aplicações da Lógica Paraconsistente

(1) **Engenharia de Software** - Lógica Paraconsistente tem sido aplicada, como meio de lidar com inconsistências pervasivas entre a documentação, casos de uso e códigos de grandes sistemas de software **??**).

(2) **Semântica de Linguagens** - No sentido de **Walter Carnielli**, possivelmente, a Lógica Paraconsistente aplicar-se-á à semântica de linguagens na Ciência da Computação.

(3) **Eletrônica** - Projeta rotineiramente uso de um lógica de quatro valores, atuando

em papeis similares a "*não sei*" e "*ambos, verdadeiro e falso*" respectivamente, adicionalmente a verdadeiro e falso.

(4) **Construção de Painel Solar Auto-Orientável** - Uma interessante aplicação da Lógica Paraconsistente é a que está em Prado (2013), uma dissertação de mestrado abordando a construção de um painel solar auto-orientável, destinado a otimizar o rendimento de um painel fotovoltaico para geração de eletricidade, proporcionando carga para um conjunto de baterias em localidades isoladas e desprovidas de energia elétrica, seguindo uma crescente tendência no sentido da busca por dispositivos confiáveis, menos danosos ao meio ambiente e apresentando baixo consumo de recursos em sua operação. Foi executada a implementação prática de um protótipo, baseada em uma placa controladora derivada da Arduino e de um motor de passo para movimentar o painel solar, utilizando para fins de sensoreamento uma amostragem dos próprios valores de tensão oferecidos pelo mesmo.

(5) **Tomada de Decisão, Avaliação e Seleção** - Outra aplicação da Lógica Paraconsistente, é tomada de decisão, processos de avaliação e seleção. Trata-se do processo de tomada de decisão de gestores utilizando-se de algoritmo analisador baseado em lógicas não-clássicas: a lógica paraconsistente anotada (LPA). O método é apropriado ao tratamento de dados incertos, contraditórios ou paracompletos; consiste em estabelecer proposições e parametrizá-las para isolar os fatores de maior influência nas decisões. Especialistas são utilizados para obtenção de anotações sobre esses fatores, atribuindo-lhes graus de crença e descrença. No caso analisado, utilizou-se da LógicaParaconsistente Anotada, como instrumento de apoio ao processo de avaliação e seleção de profissional a ser promovido a gerente, dentre cinco possíveis, em pequena empresa familiar. Os três sócios-gerentes da empresa constituíram o grupo de especialistas, na medida em que conhecem o valor profissional dos cinco funcionários. A interpretação das avaliações realizadas pelos especialistas deu-se por intermédio dos baricentros (o baricentro é um ponto da área de uma figura geométrica plana, como no quadrado, determinado pelo encontro das medianas dos lados e retas partindo dos vértices) no quadrado unitário do plano cartesiano (QUPC), que indicou os graus de inconsistência ou de indeterminação dos dados utilizados.

(6) **Distribuição do Fluxo de Veículos em Trânsito Caótico** - Devido ao trânsito caótico de veículos no capital de São Paulo, observou-se que em muitos sinais de trânsito, existe uma má distribuição do fluxo de veículos. Dado este cenário, o projeto teve como objetivo demonstrar a aplicação de Lógica Paraconsistente, para o tratamento de incertezas com o apoio da engenharia de software e para aumentar o fluxo em diferentes horas, em que principalmente apresenta um movimento maior. Através de um estudo exploratório pesquisa, foi possível observar, identificar e avaliar questões pertinentes para a questão. Com base na extracção de conhecimento do profissional de Companhia da Engenharia de Tráfego (CET), caracteriza-se como especialistas, foram levantadas fluxo característicos do veículo em momentos críticos. Após a remoção dessas informações e enviá-lo para análise, usando-se um aplicativo de software, os resultados ao longo do tempo para mudar um semáforo para outro,

apoiando as decisões sobre a possibilidade de otimização, caso a caso.

12.6 Lógica Modal

Na evolução da lógica como uma ferramenta de formalização, pode-se observar uma crescente habilidade na expressividade das lógicas. O Cálculo Proposicional foi desenvolvido para expressar verdades absolutas ou constantes, afirmando fatos sobre o universo de discurso da Lógica Proposicional. O arcabouço proposicioal lida com a questão de como a verdade de uma sentença composta depende da verdade de seus constintuintes.

A Lógica Modal foi desenvolvida a partir das limitações do conceito de implicação lógica, como notado por **Hugh MacColl** no final do século XIX. A Lógica Modal foi axiomatizada em 1912 por **C. L. Lewis**, utilizando o método de *Principia Mathematica* (1910) de **Bertrand Russell** e **Whitehead**. Segundo estes lógicos, a Lógica e o Cálculo dos Predicados é suficiente para permitir a expressão do conceito de dedução, mas limitado na sua expressividade. Certas noções como *necessidade lógica* e *possibilidade lógica*, dificilmente poderiam ser expressas através da Lógica e do Cálculo dos Predicados.

A *Lógica Modal* se refere a qualquer sistema de lógica formal que procure lidar com *modalidades* para tratar *modos* quanto a *necessidade*, *possibilidade* e *probabilidade*.

Na evolução histórica da Lógica Modal, o fundador da lógica formal moderna, **Gottlob Frege**, duvidava que a Lógica Modal fosse viável. E então, em 1933, **Rudolf Carnap** e **Kurt Gödel** começaram a busca por uma estrutura matemática de uma lógica que lidasse com as três modalidades clássicas (possibilidade, necessidade e probabilidade). Em 1937, **Robert Feyes**, seguidor de **Gödel**, propôs o sistema T de lógica modal. Em 1951, **Georg Henrik Von Wright** propôs o sistema M, que é elaborado sobre o sistema T. Também nos anos 50, **C.I.Lewis** construiu, sobre o sistema M, seus conhecidos sistemas modais $S1$, $S2$, $S3$, $S4$ e $S5$ **??**).

Em 1960 , **Saul Aaron Kripke** (1940) estabeleceu o sistema modal normal mínimo K. **Kripke** é amplamente reconhecido como um dos filósofos vivos mais importantes. Sua obra é muito influente em diversas áreas, desde da lógica à filosofia. Ele é professor emérito na University of Princeton e professor de filosofia na City University of New York (CUNY). Em 2001, ele recebeu o Prêmio Schock em Lógica e Filosofia. **Kripke** é conhecido principalmente pela contribuição para uma semântica para a Lógica Modal e outras lógicas relacionadas, publicadas quando ele tinha menos de vinte anos de idade.

Figura 131 – A evolução da lógica modal.

Fonte: www.cambridge.org.

Mas, o **Cálculo Modal** adiciona outra dimensão à variabilidade e à descrição por *predicados*. Conceitualmente, se pensarmos em **mundos**, onde cada **mundo** é materializado pelo seu universo de discurso. Considera-se, agora, um universo de discurso maior, particionado pelos universos de discursos de cada mundo, de estruturas semelhantes, mas de conteúdos diferentes. Se contemplarmos as transições entre os mundos, o Cálculo Modal propicia uma notação especial para descrever tais transições entre esse mundos.

Um exemplo para se entender a ideia de mundos, considere o sistema planetário, envolvendo a Terra e Vênus. Sabe-se que qualquer raciocínio que seja válido no planeta Terra, pode-se tornar inválido no planeta Vênus, porque conceitos básicos usados naturalmente na Terra, podem assumir significados completamente diferentes em Vênus. Então, o que temos, a respeito de formalismos, é que a variabilidade dentro de um mundo é manipulada variando-se os valores das variáveis dos predicados, enquanto as mudanças (transições) entre mundos podem ser expressas pelo formalismo modal.

Mas, se consideramos que ao invés de vários mundos, interpretarmos o conjunto de mundos possiveis da Lógica Modal, como **um mesmo mundo visto em instantes diferentes**, podemos visualizar neste mesmo mundo, um **conjunto de instantes numa escala temporal qualquer** (segundos, minutos, dias, ...), onde cada instante associado a um, e somente um, estado de um sistema computacional e imaginando-se as transições entre esses estados. Assim, o Cálculo Modal poder ser aplicada a grafos, através do Cálculo da Lógica Temporal. Toda lógica temporal é modal, a qual é uma forma apropriada para a descrição e verificação de propriedades de sistemas comunicantes concorrentes como em Benevides (2015) e Vasconcelos (1989).

Diversos provadores de teoremas, ferramentas para automatizar o uso da Lógica

Modal, são providos em AiML.NET-Advances in Modal Logic (⟨http://www.cs.man.ac.uk/~schmidt/tools/⟩).

12.7 Sobre a Lógica Matemática

Lógica Matemática é uma extensão da *lógica simbólica* em outras áreas, em especial para o estudo da teoria dos conjuntos, teoria da recursão (relevantes para a Ciência da Computação) e também para a teoria dos modelos e teoria da demonstração (relevantes para a matemática). Lógica matemática é o uso da lógica formal para estudar o raciocínio matemático, ou como propôs **Alonzo Church** Church (1996), a "lógica tratada pelo método matemático".

No início do século XX, lógicos e filósofos tentaram provar que a matemática, ou parte da matemática, poderia ser reduzida à lógica. **Gottlob Frege**, por exemplo, tentou reduzir a aritmética à lógica; **Bertrand Russell** e **A. N. Whitehead**, no clássico *Principia Mathematica* (1910-1913), tentaram reduzir toda a matemática então conhecida à lógica, a chamada *lógica de segunda ordem*. Há um certo consenso que a redução falhou, assim como há um certo consenso que a lógica é uma maneira precisa de representar o raciocínio matemático. **Hilbert**, que no seu livro sobre os fundamentos da geometria (1899) reduziu a consistência dos axiomas da geometria aos da aritmética, estava muito interessado na consistência dos axiomas da aritmética. O final desta história é contada no capítulo sobre Fundamentos da Matemática, no volume I desta série.

12.8 Bibliografia e Fonte de Consulta

Wamberto W. M. P. de Vasconcelos - *O Tempo como Modelo: A aplicação de Lógicas Temporais na Especificação Formal de Sistemas Distribuídos*, dissertação de mestrado, Programa de Engenharia de Sistemas e Computação, Setembro de 1989.

Newton C. A. da Costa; Jair Minoro Abe; Afrânio Carlos Murolo; João I. da Silva Filho; Casemiro Fernando S. Leite. Lógica Paraconsistente Aplicada. São Paulo: Atlas, 1999. ISBN 8524422184.

Newton da Costa - ⟨https://pt.wikipedia.org/wiki/Newton_da_Costa⟩

Newton C. A. da Costa - *Sistemas Formais Inconsistentes*. Curitiba: Ed. da UFPR, 1993. xxii, 66p. (Clássicos; n. 03). ISBN 8585132752

Álvaro A. C. Prado. - *Painel Solar Auto-Orientável baseado na Lógica Paraconsistente Anotada Evidencial $E\tau$*. dissertação de mestrado UNIP, 2013.

Cida Sanches, Manuel Meireles, Marcio Luiz Marietto, Orlando Roque da Silva, José Osvaldo De Sordi. UTILIZAÇÃO DA LÓGICA PARACONSISTENTE EM PRO-

CESSOS DE TOMADA DE DECISÃO: UM CASO PRÁTICO. Revista Pensamento Contemporâneo em Administração. Vol 4, No 3, 2010, Rio de Janeiro.

Renan Avanzi, Bruno da Silva Correia, Raphael de Alencar Santos Pereira, Marcelo Nogueira. *Sistema Especialista Paraconsistente baseado em conhecimento para apoiar as decisões de Engenharia de Tráfego utilizando as melhores práticas da Engenharia de Software*. VII International Conference on Engineering and Computer Education. ICECE. September 2011. pag.25-28, Guimarães, PORTUGAL.

12.9 Bibliografia e Fonte de Consulta

The Blackwell dictionary of Western philosophy. [S.l.]: Wiley-Blackwell, 2004. p. 266. ISBN 978-1-4051-0679-5

L. T. F. Gamut. Logic, language, and meaning, Volume 1: Introduction to Logic. [S.l.]: University of Chicago Press, 1991. 156-157 p. ISBN 978-0-226-28085-1.

Gabbay, Dov, (1994). 'Classical vs non-classical logic'. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, (Eds), Handbook of Logic in Artificial Intelligence and Logic Programming, volume 2, chapter 2.6. Oxford University Press.

Shapiro, Stewart (2000). Classical Logic. In Stanford Encyclopedia of Philosophy [Web]. Stanford: The Metaphysics Research Lab. Retrieved October 28, 2006, from ⟨http://plato.stanford.edu/entries/logic-classical/⟩

Haack, Susan, (1996). Deviant Logic, Fuzzy Logic: Beyond the Formalism. Chicago: The University of Chicago Press.

12.10 Referências - Leitura Recomendada

Zohar Manna - The Mathematical Theory of Computation, McGraw Hill, 1974, reimpressão Dover, 2003.

Manna & Pnueli - The Temporal Logic of Reactive and Concurrent Systems: Specification, (Springer-Verlag, 1991).

Manna & Pnueli - The Temporal Logic of Reactive and Concurrent Systems: Safety, (Springer-Verlag, 1995).

Manna & Pnueli - The Temporal Logic of Reactive and Concurrent Systems: Progress, (não publicado; os três primeiros capítulos estão disponíveis em Manna/Pnueli: The Temporal Verification of Reactive Systems: Progress).

Grupo de Sistemas Inteligentes, *Lógica Borrosa y Aplicaciones. Aplicación en*

Robótica, ⟨http://www.puntolog.com/actual/articulos/uni_santiago6.htm⟩, Universidad de Santiago de Compostela.

Jean Yves Béziau, Walter Carnielli e Dov Gabbay, eds.. *Handbook of Paraconsistency*. London: King's College, 2007. ISBN 978-1-904987-73-4

Priest, Graham e Tanaka, Koji (1996, 2009). *Paraconsistent Logic Stanford Encyclopedia of Philosophy*. Visitado em June 17, 2010. (First published Tue Sep 24, 1996; substantive revision Fri Mar 20, 2009)

Oliveira, Wilnice, T. R., *Utilizando Integrais Fuzzy em Tomada de Decisão Multicritérios*, dissertação de mestrado, PPGCC-UFSC, 2003.

Logic for philosophy, Theodore Sider

John P. Burgess. Philosophical logic. [S.l.]: Princeton University Press, 2009. vii-viii p. ISBN 978-0-691-13789-6

Susan Haack. Deviant logic: some philosophical issues. [S.l.]: CUP Archive, 1974. p. 4. ISBN 978-0-521-20500-9

Susan Haack. Philosophy of logics. [S.l.]: Cambridge University Press, 1978. 204 p. ISBN 978-0-521-29329-7

L. T. F. Gamut. Logic, language, and meaning, Volume 1: Introduction to Logic. [S.l.]: University of Chicago Press, 1991. 156-157 p. ISBN 978-0-226-28085-1

Seiki Akama. Logic, language, and computation. [S.l.]: Springer, 1997. p. 3. ISBN 978-0-7923-4376-9

Robert Hanna. Rationality and logic. [S.l.]: MIT Press, 2006. 40-41 p. ISBN 978-0-262-08349-2

John P. Burgess. Philosophical logic. [S.l.]: Princeton University Press, 2009. 1-2 p. ISBN 978-0-691-13789-6 Interpolation and definability: modal and intuitionistic logics. [S.l.]: Clarendon Press, 2005. p. 61. ISBN 978-0-19-851174-8

D. Pigozzi. In: M. Hazewinkel. Encyclopaedia of mathematics: Supplement. Volume III. [S.l.]: Springer, 2001. p. 2-13. ISBN 1-4020-0198-3.

Hazewinkel, Michiel, ed. (2001), *abstract algebraic logic*, Encyclopedia of Mathematics, Springer, ISBN 978-1-55608-010-4.

CAPÍTULO 13

A Visão Abstrata de Dados

Diante das áreas da Matemática, da Lógica e da Ciência da Computação, existem dois mundos em que nos inserimos: o **mundo matemático das abstrações** e o **mundo concreto das soluções**. Esses dois mundos precisam ser modelados e existem duas formas de se modelar esses mundos, que se originam em conjuntos de elementos e proporcionam a formação dos tipos de dados em computação. Assim a **teoria dos onjuntos** é imprescindível para se construir tipos . Por outro lado, no mundo computacional um outro paradigma, que mais reflete os problemas reais, no qual *objetos reais* são projetados para a solução dos problemas é o **paradigma dos objetos**.

Em Ciência da Computação, o conceito de *objeto* teve origem na linguagem de programação *Simula 67*, que foi a primeira linguagem a implementar o conceito de *objeto*. Esses objetos, num programa de computador, correspondem diretamente a objetos reais, como, por exemplo, os componentes de uma máquina. De modo geral, quaisquer que sejam os componentes de uma sistema de computação, esses podem ser implementados e simulados através de objetos de alguma linguagem de programação orientada a objeto. *Simula 67* é uma linguagem de programação, extensão da *ALGOL 60*, projetada para apoiar a simulação de eventos discretos, criadas entre 1962 e 1968 por *Kristen Nygaard* e *Ole-Johan Dahl* no Centro Norueguês de Computação em Oslo, Noruega. *Simula 67* foi a primeira *linguagem orientada a objetos*, porém levou tempo para se consolidar na comunidade acadêmica. Na verdade, a orientação a objeto somente se consolidou pela criação da linguagem *Smalltalk 80*.

13.1 Conjuntos abstratos × Objetos computacionais

Conjuntos e *objetos* são duas formas de se modelar o mundo. O, então, que o *mundo dos conjuntos* e o *mundo dos objetos* tem em comum ?

A resposta para esta pergunta passa por um conceito comum, tanto para a **Teoria dos Conjuntos**, como para o **Paradigma dos Objetos**. Trata-se do conceito

matemático do que é chamado **tipo**, que em Ciência da Computação, chama-se *tipos de dados*.

O conceito de **tipo** veio da **Teoria dos Tipos**, e surgiu no sentido de se organizar o espaço de diversos elementos pertencentes a diversos conjuntos, os quais para a Ciência da Computação são chamados **dados**, separando-os ou classificando-os em *classes de dados*, visando operações sobre determinadas classes formadas a partir de elementos dos conjuntos de dados.

13.2 Teoria dos Tipos de Russell

A *teoria dos tipos* é o ramo da matemática que se preocupa com a classificação de entidades (elementos de conjuntos) em conjuntos chamados *tipos*.

A *teoria dos tipos* moderna foi criada, em parte em resposta ao *Paradoxo de Russell*, publicado em *Principia Mathematica*, dos filósofos e matemáticos britânicos **Bertrand Arthur William Russell** (1872-1970) e **Alfred North Whitehead** (1861-1947). A resposta ao paradoxo veio do próprio **Russell**, com a introdução de sua **teoria dos tipos**. Embora primeiro introduzido por **Russell** em 1903 nos *Princípios*, sua *teoria dos tipos* encontra sua expressão madura em seu artigo *Mathematical Logic como base na Teoria dos Tipos* de 1908, e na obra monumental que ele escreveu, chamada *Principia Mathematica* (1910 , 1912, 1913), com co-autoria de **Alfred North Whitehead** (1861-1947). Sua idéia básica era que a referência a coleções problemáticas (como o conjunto de todos os conjuntos que não são membros de si mesmos que caracterizou o paradoxo sobre a teoria dos conjuntos de Cantor) poderiam ser evitados, organizando todas as sentenças em uma hierarquia, começando com frases sobre os *membros* no nível mais baixo, sentenças sobre *conjuntos de membros* no próximo nível mais baixo, sentenças sobre *conjuntos de conjuntos de membros* no próximo nível mais baixo, e assim por diante. Deste ponto de vista, segue-se que é possível, se referir a uma *coleção de objetos* para os quais uma determinada condição vale, se esses estão todos ao mesmo nível ou do mesmo *tipo*. Surge então, a ideia de *tipos* e a *teoria dos tipos*, que veio ser utilizada na Ciência da Computação, no sentido de se organizar espaço de dados em programas de computador. Neste capitulo, o leitor verá que a *teoria dos tipos* deu origem às *classes* de objetos, e que se pode programar um computador segundo o paradigma da orientação a objetos. Em teoria dos conjuntos, uma *coleção* (também chamada de *classe*) é "quase" um conjunto, ou seja, classes tem várias propriedades em comum com conjuntos, mas não são, necessariamente, conjuntos. A ideia de uma coleção tem a ver com a definição de um multiconjunto (Bag Theory).

13.3 Teoria dos Tipos de Martin-Löf

Per Erik Rutger Martin-Löf (1942-) é um lógico, filósofo, estatístico e matemático sueco, na Figura 132. É internacionalmente reconhecido por seu trabalho

sobre os fundamentos da probabilidade, estatística, lógica matemática e Ciência da Computação.

Figura 132 – Martin Löf - A teoria intuicionista dos tipos.

Fonte: Google Images - michaelt.github.io.

Os avanços no construtivismo decorrentes do trabalho de **Bishop** permitiram que, em 1971, o matemático **Martin-Löf**desenvolvesse a chamada *teoria intuicionista dos tipos*. Entre as formas de construtivismo mostradas até aqui, a **teoria dos tipos é a que deixa mais evidente a relação entre o construtivismo e a computação**. O objetivo desse tópico será limitado a fazer uma introdução simples do que significa a teoria dos tipos e mostrar alguns dos seus usos na computação. Isso deve-se ao fato que é uma teoria bastante complexa, seria necessário um artigo inteiro para explicá-la de forma satisfatória.

Para entender melhor a teoria dos tipos, pode-se pensar na teoria dos conjuntos. Na teoria dos conjuntos, o conceito principal é o conjunto e a partir dele, a teoria vai sendo construída. De forma semelhante funciona a teoria dos tipos, mas sendo o principal conceito os tipos. O próprio **Martin-Löf** falou algumas palavras que ajudam entender as ideias por trás dessa teoria. De acordo com ele, devemos pensar acerca de objetos matemáticos e construções. Todo objeto matemático seria de um certo tipo e sempre seria dado junto comesse determinado tipo. O tipo é definido descrevendo o que deve-se fazer para construir umobjeto desse tipo. Assim, um tipo seria bem-definido quando se sabe o que significa se ter um objeto desse tipo Bridges (2013).

Com essas palavras, **Martin-löf** acaba definindo informalmente o que é um tipo . Além disso, ele vai mais além, dando um exemplo. O exemplo é que o motivo das funções $\mathbb{N} \longrightarrow \mathbb{N}$ serem um tipo, não é porque é sabido a existência de alguma função numérica em particular, como por exemplo, as funções recursivas primitivas (soma, multiplicação, ...), mas sim porque a noção de uma função numérica é bem entendida em geral Bridges (2013). Nessa introdução abreviada das ideias de **Martin-Löf**, já fica evidente a relação entre a **teoria dos tipos** e a **computação**. A forma que essa teoria interpreta os objetos matemáticos é semelhante a muitas linguagens funcionais, sendo um bom exemplo *Haskell*.

Em *Haskell*, todos os dados são associados a um certo tipo. Porém, o que deixa essa linguagem mais similar ainda, é que qualquer função ou expressão da linguagem também tem um tipo associado. Por causa dessas semelhanças, algumas linguagens funcionais mais recentes, como as linguagens *Epigram* e *Agda*, basearam-se diretamente na teoria de **Martin-Löf** (teoria dos tipos com lógica intuicionista). Além disso, uma outra grande importância é a relação entre essa teoria e o isomorfismo de *Curry-Howard*. De forma resumida, esse isomorfismo estabelece a equivalência entre programas e provas matemáticas. Isso é visto de forma bem clara na teoria de **Martin-Löf**, pois do jeito que ela foi desenvolvida, facilita a extração de programas a partir de provas Bridges (2013). É por isso que os conceitos da teoria dos tipos são bastante usados para desenvolver programas que tem como objetivo servir de assistente para provas.

Mais sobre sistemas de tipos , o leitor pode encontrar em Pimentel (2008). Neste trabalho, a autora expõe sobre sistemas de tipos e a relação com a definição sintática de linguagens de programação. A ideia básica desses sistemas formais é estabelecer regras de formação das frases da linguagem, a partir de suas subfrases, levando em conta propriedades das construções envolvidas na formação dessas frases. Como as propriedades consideradas são, tipicamente, *tipos de expressões*, tais sistemas são denominados *sistemas de tipos*.

13.4 O que são sistemas de tipos

Com o surgimento de poderosos computadores programáveis e o desenvolvimento de linguagens de programação, a teoria dos tipos encontrou aplicação prática no desenvolvimento de sistemas de tipos de linguagens de programação.

Definições de sistemas de tipos no contexto de linguagens de programação variam, mas a seguinte definição dada por **Benjamin C. Pierce**, professor no *Department of Computer and Information Science*, University of Pennsylvania, na obra *Types and Programming Languages* em Pierce (2002) - corresponde, aproximadamente, ao consenso na comunidade de Teoria dos Tipos:

"**Um sistema de tipos é um método sintático tratável para provar a isenção de certos comportamentos em um programa, através da classificação de frases de acordo com as espécies de valores que elas computam.**"

Em outras palavras, um sistema de tipos organiza os valores (dados) de um programa em conjuntos chamados tipos. Isto é o que é denominado uma *atribuição de tipos*, e torna certos comportamentos do programa ilegais com base nos tipos que são atribuídos para um programa. Por exemplo, um sistema de tipos pode classificar o valor "casa" como uma cadeia de caracteres e o valor 10 como um número, e proibir o programador de tentar adicionar "casa" a 10, com base nesta atribuição de tipos. Neste sistema de tipos, a instrução "casa + 10" seria ilegal. Assim, qualquer

programa permitido pelo sistema de tipos seria livre de erros aos se tentar adicionar cadeias de caracteres a números.

O projeto e a implantação de sistemas de tipos é um tópico quase tão vasto quanto o das próprias linguagens de programação. De fato, os proponentes da teoria dos tipos argumentam que o projeto de sistemas de tipos é a própria essência do projeto de linguagens de programação.

A teoria dos tipos é uma ferramenta na criação e pesquisa sobre novas linguagens de programação. Ela é formada por um amálgama de matemática, lógica e computação. Através dela, tornase possível a implementação de compiladores eficientes e códigos íntegros. Essa teoria é o estudo de sistemas de tipos , que é um conjunto de regras sobre tipos de dados que previne as linguagens de erros decorrentes de más interpretações. Esses erros são comumente erros de tipos, erros de passagem de parâmetro para funções (em número e tipo), alterações do código durante a execução. Geralmente esses sistemas são definidos através de julgamentos de tipos. Estes julgamentos definem se uma expressão foi bem digitada. Uma das funções de um sistema de tipos é prevenir erros de execução. Um erro de tipo é uma expressão onde e não resulta em um valor. Linguagens de programação podem ser definidas por suas estruturas de tipos . Entre os benefícios que isso traz, estão e modularidade com que a linguagem é apresentada, evitando confusões.

Note que a teoria dos tipos , como descrita daqui para frente, se refere a *tipagem estática*, uma disciplina estabelecida pelo sistema de tipos da linguagem, que corresponde a se atribuir tipos de dados antes do tempo de compilação do programa. Isto corresponde a dizer ao compilador de um linguagem (neste caso estamos tratando de linguagens compiláveis) quais tipos deverão ter as variáveis de um programa, e consequentemente, quanto de espaço de memória, para cada das variáveis, deve ser previsto pelo compilador, para quando o programa for executado.

Uma outra forma de tipagem de dados é a dinâmica. Sistemas de programação que aplicam tipagem dinâmica não provam *a priori* que um programa usa valores corretamente; ao invés disso, elas lançam um erro em tempo de execução quando o programa tenta apresentar algum comportamento que use valores incorretamente.

13.5 O que é um Tipo

Podemos entender o que é um **tipo**, do ponto de vista matemático.

O conceito de tipo de dados simples é amplamente usado nas linguagens de programação tipadas (ou fortemente tipadas). É o que o leitor começar a aprender logo na sua primeira linguagem de programação. Para o caso de tipos de dados mais complexos, existe o conceito matemático de tipo de dados chama-se *Tipos Abstratos de Dados* que teve sua origem na matemática e norteia o conceito sintático chamado

classe, cujas as instâncias de classe correspondem aos objetos de uma classe no **paradigma de programação orientado a objeto**. **Objetos** são, modernamente, os que você aprenderá para programar um computador, no estilo de linguagem imperativa.

Definição: (**Operação**) - Chama-se operação sobre um conjunto E a toda aplicação de $E\ X\ E$ em E. Ou seja, uma **operação** é uma aplicação (função) a elementos de um conjunto E, que tem como resultado, um outro elemento de E.

Pode-se generalizar a definição de *operação*, introduz-se o conceito de operação no sentido amplo, isto é, toda aplicação f de n conjuntos, isto é:

$f : E_1 \times E_2 \times ... \times E_{n-1} \times E_n \rightarrow F$,

onde E_i, para $i = 1, n$ são conjuntos quaisquer. Diz-se, neste caso, que f está definida em:

$E_1 \times E_2 \times ... \times E_{n-1} \times E_n$ e assume valores em F.

Definição: (**Tipo**)

Um tipo T é um conjunto de elementos, munido de um número determinado de operações sobre os elementos desse conjunto.

Qualquer tipo T é dotado de um conjunto de operações, tanto faz ser um *tipo de dado simples* ou um *tipo abstrato de dados*.

Pode-se generalizar esta definição, introduzindo-se o conceito de operação no sentido amplo, isto é, toda aplicação f de n conjuntos $E_1 \times E_2 \times ... \times E_{n-1} \times E_n$ em F, onde E_i, para $i = 1, n$ são conjuntos. Diz-se, neste caso, que f está definida em $E_1 \times E_2 \times ... \times E_{n-1} \times E_n$ e assume valores em F.

Exemplo: (**tipo de dados inteiros, reais**) - *Inteiro* ou *real* é o tipo de dado que classificam números, e são os mais usados nos programas de computador, que o leitor irá desenvolver na sua vida acadêmica ou profissional.

Exemplo: (**tipo de dados caracters**) - Podemos ter, também, o tipo chamado *caracter*, o qual classifica num programa de computador, os dados alfa-numéricos.

Exemplo: (**tipo de dados booleanos**) - Tipo *booleano* é atribuído a variáveis que assumem os valores ***true*** (verdade) ou ***false*** (falso).

13.6 O Surgimento das Liguagens de Programação Tipadas

O λ-Calculus Barendegt (1994) Ronchi e L. (2004) é um sistema formal que lida com a teoria de funções. Foi introduzido nos anos 1930 por **Alonzo Church**. Originalmente, **Church** tentou construir um sistema (que continha o λ-Calculus) para a fundamentação da matemática. Mas esse sistema foi mostrado *inconsistente* (por **Rosser**) por ser possível simular o paradoxo de Russell dentro da teoria. Desta foma, **Church** separou a parte do λ-Calculus e a usou para estudar a **computabilidade**. O λ-Calculus é uma teoria que representa funções como regras, ao invés da tradicional abordagem de funções como gráficos. Funções como regras é a noção mais antiga de função e refere-se ao processo de, a partir de um argumento para um valor, o processo avalia o valor associado ao argumento, apenas por uma definição por certas regras. Desta forma, **é possível estudar os apectos computacionais das funções**.

Tipos estão presentes tanto em matemática quanto em computação. Na teoria de conjuntos tradicional, o agrupamento de elementos em um conjunto independe da natureza desses elementos. Quando passamos a trabalhar em aplicações específicas, precisamos classificar os objetos em categorias, de acordo com o seu uso ou aplicação. A noção de tipo origina-se dessa classificação: um tipo é uma coleção de objetos ou valores que possuem alguma propriedade em comum. Em geral, para cada tal conjunto de valores, existe uma classe sintática correspondente, qual seja, de termos que representam esses valores, que também é chamada de tipo.

Em **matemática**, tipos impõem restrições que evitam paradoxos. Universos não tipados, como o da teoria de conjuntos de **Frege**, apresentam inconsistências lógicas (tais como o paradoxo de Russell encontrado na teoria ingênua dos conjuntos de Cantor).

Em **computação**, estudos sobre tipos em linguagens de programação são usualmente desenvolvidos sob o arcabouço do λ-Calculus tipado. O λ-Calculus tipado surgiu a partir do λ-Calculus não tipado, ambos definidos por **Church**, na década de 30. O λ-Calculus (não tipado) provê um modelo muito simples de avaliação de expressões. Apesar dessa simplicidade, o λ-Calculus é um modelo "universal" de computabilidade, no sentido de que qualquer função recursiva, assim como qualquer função computável por uma máquina de **Turing**, pode ser expressa como um termo do λ-Calculus.

O λ-Calculus tipado simples, proposto por **Church**, considera apenas tipos básicos (por exemplo, *Int*, o tipo dos inteiros, e *Bool*, o tipo dos valores booleanos) e tipos funcionais. O λ-Calculus tipado simples pode ser estendido de diversas maneiras, introduzindo-se novos tipos básicos ou novos construtores de tipos. Construtores de tipos, comumente adicionados são construtores para os tipos produto e soma (união disjunta) e tipos recursivos.

13.7 Tipos em Linguagens Imperativas

FORTRAN (Formula Translation) foi a primeira linguagem de programação imperativa criada com repercução na comunidade da Ciência da Computação e engenharia. O primeiro compilador FORTRAN foi desenvolvido para o IBM 704 em 1957, por uma equipe da IBM chefiada por **John W. Backus**. A linguagem introduziu o usos dos tipos de dados simples tais como *integer*, *real* e *character*.

Na evolução do FORTRAN, a primeira versão padronizada da linguagem, conhecida como FORTRAN IV (ANSI X3.9-1966) foi padronizada pela *American Standard Association* (*ASA*). A linguagem foi largamente utilizada por cientistas para a escrita de programas numericamente intensivos. A ampla disponibilidade de compiladores para diferentes computadores, a simplicidade da linguagem, facilidade para ensiná-la (mesmo que naquela época não houvesse programação estruturada), sua eficiência e as vantagens introduzidas pelo uso de subprogramas dos tipos **função** (retorna somente um valor como resultante do processamento da função) e **subrotina** (pode retornar mais de um valor como resposta do processamento desse subprograma). A compilação do programa era independente dos subprogramas (precisava de um programa chamado Link Editor) e mais a capacidade de lidar com números complexos, ajudaram em sua ampla difusão dentro da comunidade científica. Depois do FORTRAN, surgiram várias outras linguagens no paradigma imperativo, utilizando um sistema de tipos simples, até que a Teoria dos Tipos veio a contribuir, definitivamente, com o surgimento da Programação Orientada a Objeto, na qual a linguagens tem como componente principal a implementação de tipos abstratos de dados, materializados como as classes deste paradigma de linguagens de programação.

13.8 Tipo Abstrato de Dado (TAD)

Informalmente, **abstrato** é tudo aquilo que que só existe na ideia, no conceito. Resulta de uma **abstração** , sendo abstração a ausência de detalhes. Também, informalmente, é aquilo que é de difícil compreenção.

Em Ciência da Computação, um Tipo Abstrato de Dados (TAD) é uma especificação de um conjunto de dados e operações que podem ser executadas sobre esses dados. Com um TAD podemos nos abstrair das variáveis envolvidas, em uma única entidade fechada, com operações próprias que podem manipular essas variáveis, no sentido de permitir a informação das variáveis, restritas somente as essas operações, requisitadas pelo ambiente externo ao TAD.

Um exemplo prático é o caso de um funcionário de uma organização. Antes da Teoria de Tipos Abstratos de Dados, um funcionário seria representado por variáveis soltas (como seu nome, sua idade e sua matrícula) que seriam operadas separadamente, sem ligação lógica entre elas, além do conhecimento de quem produz o programa de computador, de que a variável corresponde ao nome da “entidade”

funcionário.

Conceitualmente, na teoria TAD, passou-se a pensar que não há o nome, idade e matrícula do funcionário, mas simplesmente o tipo FUNCIONÁRIO. Este tipo, como um tipo simples (inteiro ou string), deve ter operações próprias. Assim, o FUNCIONÁRIO deve possuir operações desejáveis a quem produz o programa, como, por exemplo, validar sua matrícula, verificar a idade, entre outras operações.

Na prática, o TAD é especificado usando-se um tipo composto pelos valores pertencentes ao TAD (nome, idade, matrícula). E por funções que operam esta estrutura. Veja o exemplo na Figura 13.8 seguinte:

TADFuncionario

$nome : NOME$
$idade : IDADE$
$matricula : MATRICULA$

Declara a assinatura do TAD.

TIPOS

$FUNCIONARIO\ [NOME, IDADE, MATRICULA]$

FUNÇÕES

$MaiorDeIdade60(FUNCIONARIO)$
$ValidaMatricula(FUNCIONARIO)$

Pesquisa sobre Tipos Abstratos de Dados

O trabalho precursor deve-se a Liskov e Zilles (1975) sobre técnicas de especificação para abstração de dados. Depois, Guttag (1977) sobre tipos abstrados de dados e o desenvolvimento de estruturas de dados. Em Guttag (1977), a especificação algébrica de um tipo abstrato de dados está em Guttag J. V. e Horning (1978), e em Guttag, Horowitz e Musser (1978) sobre o projeto de especificação de tipos de dados. Também em Goguen, Thatcher e Wagner (1978) surge uma abordagem algébrica inicial para a especificação, correção e implementação de um tipo abstrato de dados. Uma visão algébrica de um tipo abstrato de dados pode ser estudada em Veloso (1987) e Sette (1988).

13.9 Contribuição para a Computação - Projeto Orientado a Objeto

Para mostrar a contribuição de tipos abstratos de dados à Ciência da Computação, nos voltamos ao que existe de mais atual em termos de paradigma de programação de computadores. A primeira definição do que vem a ser um **projeto orientado a objeto**, é que este é **o método que conduz a arquiteturas de software baseadas sobre os objetos de todo sistema ou que subsistemas manipulam**.

Esta definição provê a linha geral para projetar software no modo orientado a objeto. Mas, aí surgem algumas questões:

- Como descrever os objetos.
- Como descrever as relações e pontos em comum entre objetos.
- Como usar objetos para estruturar programas.

13.9.1 Descrevendo Objetos - Tipos Abstratos de Dados

Uma vez que se tenha estabelecido os tipos de objetos visando a decomposição do sistema, a próxima questão é como esses objetos devem ser descritos.

Quando falando sobre sistemas sendo organizados em torno de estruturas de dados, estamos, claramente, mais interessados em **classes de estruturas de dados**, do que em **objetos** individuais.

Em Meyer (1988), o conceito de classe , é sem dúvida, o termo técnico que será aplicado em linguagens orientadas a objetos para descrever tais classes de estruturas de dados, com seus atributos, caracterizadas por propriedades. Não confundir classe e objeto. A diferença é que **classe** diz respeito a um conjunto de itens, e um **objeto** é um elemento desse conjunto. Uma classe é um conceito sintático, enquanto um objeto é algo como um elemento que é instanciado de uma classe, mas que irá corresponder, num ambiente de computação, a um código executável na memória do computador ou armazenável num dispositivo de memória secundária.

13.10 A necessidade de Tipos Abstratos de Dados

Descrições formais de classes, como estruturas de dados **completas**, **precisas**, **não-ambíguas** é o que antecede a construção de software orientado a objeto. Mas, como conseguir completude, precisão e não-ambiguidade ?

Segundo Meyer (1988), no contexto da construção de software orientado a objeto, *over specification* pode ser entendida para significar redundância ou inconsistência; a especificação tem mais informação mais precisa do que é requerida. Ao contrário,

under specification significa que vários elementos das especificações são incompletas ou insuficientemente precisas. Um critério deve ser encontrado numa construção de software orientado a objeto, para evitar a demasia ou insuficiência de informação numa especificação, e a resposta é a **Teoria dos Tipos Abstratos de Dados**. Uma breve apresnetação de tipos abstratos de dados está em Meyer (1988).

Genericamente, uma especificação de um tipo abstrato de dados descreve uma classe de uma estrutura de dados, através da **lista de serviços** disponíveis sobre a estrutura de dados em questão, e mais as **propriedades** formais desses serviços. As palavras **operações** e **features** também são usadas ao invés de **serviços**.

13.11 Formalmente especificando TAD -Tipos de Dados Abstratos

Duas maneiras formais podem ser utilizadas para especificar um TAD: (a) Tipo abstrado de dados orientado a propriedades , usando-se funções (que não explicitam estados) para a descrição dos servicos; (b) Tipo abstrato de dados baseado em estado , usando-se operações sobre as variáveis de estado.

Um exemplo de um tipo abstrato de dados orientado a propriedades , pode ser mostrado numa especificação formal completa de uma estutura de dados chamada PILHA, como segue, muito utilizada em determinadas implementações de computador. Os currículos dos cursos de graduação em Ciência da Computação tem como disciplina básica, as Estruturas de Dados, tais como *listas*, *filas* e dentre as quais, a *pilha* é um outro exemplo.

Exemplo : (**Tipo Abstrato de Dados PILHA - Orientado a Propriedades**)

O exemplo, em Meyer (1988), mostra a especificação de um **tipo abstrato de dados orientado a propriedades**, chamado TAD (Tipo Abstrato de Dados), como na Figura 13.11, com suas características matemáticas, cuja especificação formal é mostrada a seguir, consistindo de quatro partes:

Neste exemplo as duas primeira partes expressam a **sintaxe do tipo** (suas propriedades estruturais, definindo os serviços disponíveis no tipo), enquanto as duas últimas partes expressam a **semântica do tipo** (definindo as propriedades dos serviços).

TAD

Tipo Abstrato de Dados *PILHA*, orientado a propriedades.

Declara a assinatura do TAD.

TIPOS

PILHA[*X*]

Declara as funções que descrevem os serviços sobre o tipo *PILHA*.

FUNÇÕES

vazia: *PILHA*[*X*] –> *BOOLEAN*
nova: –> PILHA[*X*]
coloca: *X* X *PILHA*[*X*] –> *PILHA*[*X*]
retira: *PILHA*[*X*] -/-> *PILHA*[*X*]
topo: *PILHA*[*X*] -/-> *X*

Declara as pré-condições necessárias para as funções-serviços.

PRECONDIÇÕES

preretira (*s*:*PILHA*[*X*] = (**not** *vazia*(*s*))
pretopo (*s*:*PILHA*[*X*] = (**not** *vazia*(*s*))

Especifica as propriedades dos serviços.

AXIOMAS

Paratodox:*X*, s:*PILHA*[*X*]
vazia(*nova*())
not *vazia*(*coloca*(*x*,*s*))
topo (*coloca*(*x*,*s*)) = *x*
retira (*coloca*(*x*,*s*)) = *s*

No exemplo acima é mostrada a especificação de uma estrutura de dados do tipo *PILHA* (*stack*) vista como uma estrutura sobre a qual os serviços disponíveis são *coloca* (*push*) um novo elemento na *PILHA*, acessa o topo da *PILHA* (textitpop), o qual *retira* um elemento do topo da pilha e testa se a *PILHA* é *vazia*. Estes serviços devem observar a política *last-in-first out* (o último que entra na pilha, é o primeiro que sai) que caracteriza o comportamento de uma pilha.

Tipos abstratos de dados , orientado a propriedades herdaram da matemática, as funções, que descrevem as operações de um tipo . Essas funções podem ser predicados , que são funções que tomam valores no conjunto chamado $BOOL = \{true, false\}$.

Duas características importantes de Tipos Abstratos de Dados devem ser enfatizadas: a ocultação da informação e a instanciação de tipo .

A instanciação de um TAD, pode ser visualizada na construção do exemplo 13.11 do tipo abstrato de dados *PILHA*. Pode-se destacar em sua especifIcação formal, a **identificação** do tipo sendo especificado, sua **assinatura** $PILHA[X]$, que contém um parâmetro fictício X, representando uma estrutura genérica, o qual representa um tipo arbitrário, que representa o tipo de elementos na *PILHA*. Portanto, pilhas de vários tipos podem ser obtidos por prover um tipo para X. Cada pilha individual é chamada de uma **instância do tipo** *PILHA*.

13.12 Ocultação da Informação num TAD

Em inglês, o conceito é conhecido como **Information Hiding**. Como está em Meyer (1988), este é o princípio da ocultação da informação dentro de um tipo abstrato de dados. O princípio *Information Hiding* pode ser explicado como segue:

"**Toda informação sobre um módulo TAD, deve ser privada ao módulo TAD, a menos que seja especificamente declarada pública.**"

A aplicação deste princípio assume que todo o módulo TAD é conhecido ao mundo externo, através de uma **interface**. A razão fundamental por trás deste princípio é que se um módulo TAD mudar, somente seus elementos privados mudam, sem afetar sua *interface* que outros módulos de um programa, chamados *clientes* a conhecem.

Em suma, um **tipo abstrato de dados** é uma **classe de estruturas de dados** descrita por uma visão externa: **serviços** disponíveis e as **propriedades desses serviços**. Note que, da definição do que seja um **tipo** (conceito algébrico), passamos à definição de uma **classe** (conceito sintático) de uma linguagem orientada a objeto. Por isso, que nas linguagens orientadas a objetos, uma classe corresponde a um tipo da linguagem.

Como em Meyer (1988), uma estrutura de dados é, assim, vista como um conjunto de serviços oferecidos ao mundo externo. Usando uma descrição de um **tipo abstrato de dados**, não é preciso cuidar o que a estrutura é; quais problemas ela tem; o que ela pode oferecer para outros elementos de software. Esta é um visão utilitária, consistente com as restrições do desenvolvimento de software em larga escala: preservar a integridade do módulo em um ambiente em constante mudança, todo componente do software deve guardar sua própria lógica. Componentes do software devem somente acessar estruturas de dados, com base nas suas propriedades. Isto está em consonância com o princípio de *Information Hiding* - a ocultação da informação, constante nos módulos que definem um TAD.

O conceito de **Information Hiding** é utilizado, em programação concorrente (processos ou threads), na construção da estrutura de um **monitor** Ben-Ari (2014), como é implementado na linguagem de programação Java Deitel e Deitel P (2005). O **monitor**, então, funciona um módulo que controla a sincronização da execução dos processos (ou threads), dentro da execução de um programa concorrente.

13.13 Tipos Abstratos de Dados Baseados em Estados

Funções matemáticas só explicitam seus argumentos de entrada e sua saída. Portanto, não explicitam **estados**. Os estados são variáveis internas, dentro da "caixa" de uma função. Entretanto, programas de computador, do ponto de vista da **teoria dos autômatos finitos**, explicitam os estados do programa e as transições entre estados. Programas tem o seus estados iniciais, que são considerados os estados iniciais nos autômatos, e dependendo do programa, pode existir um estado final, ou o programa pode ser reinicializado depois de alguns estados e transições ocorrerem.

Visto assim, podemos ter um **tipo abstrato de dados baseado em estados**, o qual especifica uma classe, como explicado anteriormente. Uma classe tem seus atributos (variáveis de estado) e operações (que representam transições) que manipulam essas variáveis, provendo os serviços do TAD baseado em estado.

Desta forma, ao invés de funções matemáticas, descrevemos as operações que podem representar transições que não mudam de estado, ou mesmo, transições que mudam o estado do programa. As as situações podem ser representadas, claramente, numa especificação baseada na linguagem Z, como em Spivey (1989). A linguagem Z foi criada usando a Teoria dos Conjuntos, a Lógica dos Predicados e o λ-Cálculo, e tem a expressividade para descrever tudo o que estes formalismos que a compõem expressam.

PILHA

$$max : \mathbb{N}$$

$$max \leq 100$$

Esta classe tem duas operações.

$$items : \mathbf{seq} Item$$

INIT

$$items = \langle\rangle$$

Push

$$\Delta(items)$$
$$item? : Item$$

$$\#items < maxitems' = \langle item? \rangle \frown items$$

Pop

$$\Delta(items)$$
$$item! : Item$$

$$items \neq \langle\rangle$$
$$items = \langle item! \rangle \frown items'$$

No exemplo da especificação da *PILHA* em Duke e Rose (2000), usada no exemplo anterior 13.11, podemos ilustrar o que vem a ser um TAD baseado em estados. Este corresponde a um **método construtivo** de se especificar programas de computador ou sistemas. Enquanto o método que utiliza somente funções, corresponde a um método axiomático (orientado a propriedades) de se especificar programas ou sistemas. No sentido de se aproximar dos princípios do que é um projeto orientado a objeto, mostramos o mesmo exemplo 13.11 do TAD *PILHA*, a estrutura sintática e a semântica como na linguagem Object-Z Smith (2000). Esta linguagem estende a linguagem Z, por adicionar construções da linguagem em direção ao paradigma orientado a objetos, mais notadamente, sobre o uso de classes (que sáo tipos) e suas operações. Em Object-Z, outros conceitos tais como polimorfismo e herança são também suportados. Object-Z recebeu significante atenção na comunidade de métodos formais e várias pesquisas no sentido de validar os aspectos da sua linguagem existem, tornando-a uma linguagem híbrida, que utiliza a linguagem Z, mas também pode utilizar a Lógica Temporal, para descrever propriedades das trajetórias de execução sobre os estados de um programa de computação. Object-Z Mahony B; Dong (2000) Dong e Duke R; Hao (2005) suporta ferramentas através

da *Community Z Tools project* e também possui um cálculo para refinamentos de especificações (*Refinement Calculi*) Derrick John; Boiten (2014).

O projeto CZT da comunidade de ferramentas Z (CZT) é um projeto para construir um conjunto de ferramentas para a notação Z, um método formal em engenharia de software. As ferramentas incluem suporte para edição, verificação de tipo e animação para especificações Z. Há algum suporte para extensões como Object-Z. As ferramentas são construídas usando a linguagem de programação Java. CZT foi proposta por **Andrew Martin** em 2001, para estabelecer um projeto de comunidade baseada na Internet para construir uma estrutura para a integração da ferramenta Z e, finalmente, outras ferramentas *plug-in* para algum ambiente de desenvolvimento de software Oxford University Computing Laboratory (2001).

Um exemplo de tipos abstratos de dados baseados em estados, utilizando a linguagem de especificação chamada Object Z:

Exemplo (**Service**) - Um TAD chamado *Service* é mostrado. A classe Service refere-se a uma parte da especificação de um metamodelo para descrever as características de modelos referentes a ambientes genéricos de computação pervasiva ou ubíqua como está em Campiolo (2005), Campiolo, Cremer e Sobral (2007).

Service

ServiceState

$$identification : \mathbb{N}$$
$$description : TEXT$$
$$created : TimeStamp$$

$$\forall x, y : identification \bullet x \neq y$$

CreateService

$$\Delta(identification, description, created)$$
$$identification? : \mathbb{N}$$
$$description? : TEXT$$
$$created? : TimeStamp$$

$$identification' = identification?$$
$$description' = description?$$
$$created' = created?$$

Tipos abstratos de dados, baseados em estados, herdaram da matemática a **Teoria**

dos Conjuntos, a **Lógica dos Predicados** e as funções do λ-Cálculo. Da Lógica dos Predicados temos que as funções podem ser **predicados**, que são funções que tomam valores no conjunto chamado $BOOL = \{true, false\}$. A lógica pode expressar propriedades dos estados, relacionando pré-condições e pós-condições, nas transições de estados. Se a Lógica Temporal é usada, TADs baseados em estados também podem comportar (ver em Object Z) a forma de se descrever as propriedades dos caminhos sobre os estados (trajetórias) de execução do programa de computador implementado na classe.

13.14 Benefícios dos Tipos para Linguagens

Em computação, existem diversas linguagens não tipadas (ou seja, que possuem apenas um tipo, que contém todos os valores) como, por exemplo: a linguagem do λ-Calculus, Lisp (List Processing), Self (uma linguagem de programação orientada a objeto dinâmica baseada em protótipo), Perl e Tcl (Tool Command Language). Essas linguagens não dispõem de nenhum mecanismo para a deteção de falhas devidas a operações aplicadas a argumentos impróprios. A ocorrência de um erro dessa natureza não interrompe a execução do programa, sendo possível que o erro seja detectado somente após uma sequência grande de operações subsequentes à ocorrência do mesmo.

O tipo de uma expressão determina em que contextos a ocorrência dessa expressão é válida ou não. Em outras palavras, o agrupamento de valores em tipos permite que se verifique se expressões que denotam tais valores, não são usadas em contextos em que não fazem sentido. Essa verificação, comumente chamada de "checagem de tipo", pode ser feita em tempo de compilação ou em tempo de execução de um programa. Quando a verificação é feita em tempo de compilação, além dos erros de tipo serem detetados antecipadamente (um programa não é executado caso contenha erros de tipos), eles são sempre detetados, podendo ser então corrigidos. No caso de verificação em tempo de execução, um erro existente só será detetado, se alguma execução do programa envolver, de fato, o ponto onde tal erro ocorre. Em outras palavras, o erro só é detetado se a execução do programa constitui um teste para o caso correspondente ao erro de tipo.

Entretanto, existem formas de erro que não são erros de tipo. No entanto, os argumentos apresentados acima, são importantes devido a grande frequência de erros de tipo, usualmente cometidos durante uma tarefa de programação. Em face dos argumentos apresentados, o estudo de tipos em linguagens de programação tornou-se de grande importância, no sentido de sua influência sobre o projeto e a definição de linguagens de programação e, portanto, sobre o desenvolvimento de software em geral.

Apesar da similaridade entre as noções de tipo em matemática e em computação, existem algumas diferenças entre estes dois conceitos. Em primeiro lugar, a finalidade é diferente. Em computação, a noção de tipos é motivada pelos fatores apresentados

acima: estruturação, clareza e eficiência de programas, e detecção de erros. Em matemática, o propósito é o de evitar inconsistências lógicas.

Linguagens de especificação de sistemas ou de programação de computadores, utilizam tipos no sentido de organizar o espaço de dados de interesse para programas de computador. A abstração de informações através de um tipo abstrato de dados permite a melhor compreensão dos algoritmos e maior facilidade de como se programar um compuatador. Como consequência aumentou a complexidade dos programas, tornando-se fundamental em qualquer projeto de software, a modelagem prévia de seus dados. Tipos abstratos de dados foram incorporados à própria linguagem de especificação (como em LOTOS - Language Of Temporal Ordering Specification, ISO) ou programação (Java pode definir um tio abstrato de dados), para o paradigma do que é hoje a **orientação a objetos**. Isto permite o controle do acesso às informações de um tipo (característica de *information hiding*), a aparição do conceito de **polimorfismo** e, do conceito de **herança** de tipo (classe herdando de outra classe), muito utilizado em linguagens de programação orientadas a objeto. Com relação a LOTOS, esta é uma linguagem de especificação formal baseado em ordenação temporal de eventos. LOTOS é usado para especificação do protocolo em normas ISO. É uma linguagem algébrica que consiste em duas partes: uma parte para a descrição de dados e operações, com base nos tipos de dados abstratos, e uma parte para a descrição de processos simultâneos, com base em cálculo processos. O padrão foi concluída em 1988, e foi publicada como ISO 8807 em 1989.

Quanto ao impacto prático da *teoria dos tipos*, linguagens de programação fortemente tipadas podem ser construídas e existem vários exemplos dessas linguagens. Também a análise e a otimização de *programas orientados a tipos*, como também os mecanismos de verificação de segurança de tipos (como a verificação dos bytecodes do Java), são partes de um projeto de linguagem que surgiram com impacto prático da teoria dos tipos. Outros tópicos dizem respeito a noção de tipos abstratos de dados, através da definição formal envolvendo *pré condição*, *pós condição* e *invariantes*, como também, a relação entre tipo abstrato da dados baseado em estados e programação orientada a objeto.

O termo *polimorfismo* é originário do grego e significa "muitas formas". Na programação orientada a objetos, o conceito de **polimorfismo** permite que referências de tipos de classes mais abstratas representem o comportamento das classes concretas que a referenciam. Assim, é possível tratar vários tipos de maneira homogênea.

13.15 Bibliografia e Fonte de Consulta

Alonzo Church, A formulation of the simple theory of types, The Journal of Symbolic Logic 5(2):56-68 (1940)

Typed Lambda Calculus - https://en.wikipedia.org/wiki/Typed_lambda_calculus

Teoria dos Tipos: ⟨http://pt.wikipedia.org/wiki/Teoria_dos_tipos⟩

Estruturas de Dados e Verificação de Programas com Tipos de Dados. Paulo A. S. Veloso. Edgar Blücher. 1987.

Tipos Abstratos de Dados - Uma Visão Algébrica. José Sergio Antunes Sette. VIII Congresso da SBC. VII JAI. Rio de Janeiro. 1988.

Bertrand Meyer. Object Oriented Software Construtions. Prentice-Hall. 1988.

Polimorfismo: ⟨http://pt.wikipedia.org/wiki/Polimorfismo⟩

Object Z: ⟨en.wikipedia.org/wiki/Object-Z⟩

Object Z: ⟨formalmethods.wikia.com/wiki/Object-Z⟩

João Bosco M. Sobral. *Uma Linguagem de Especificação Formal de Sistemas Distribuidos em Alto Nível de Abstração*, Tese de Doutorado, EEL-COPPE, UFRJ, 1996.

Rodrigo Campiolo - Aspectos de Modelagem para Ambientes de Computação Ubíqua, Dissertação de Mestrado, PPGCC-UFSC, 2005.

Vivian Cremer Kalempa - Especificando Privacidade em Ambientes de Computação Ubíqua, Dissertação de Metrado, PPGCC-UFSC, 2009.

13.16 Referências - Leitura Recomendada

Barendregt, H.P., The Lambda Calculus: its syntax and semantics, N.103 in Studies in Logic and the Foundations of Mathematics (revised edition), North-Holland, Amsterdam (1994).

Ronchi Della Rocca S., Paolini L., The Parametric λ-Calculus: a metamodel for computation, Computer Science-Monograph, Springer Verlag, (2004).

Russell, B. and Whitehead, A. N., Principia Mathematica, New York, Cambridge University Press (1927).

Church, A., A formulation of the simple theory of types, Journal of Symbolic Logic 5, pp. 56-68 (1940).

J. P. Spivey. The Z Notation. Prentice-Hall. 1989.

Rose, Gordon; Duke, Roger. Formal Object Oriented Specification Using Object-Z

(Cornerstones of Computing). Publisher: Palgrave Macmillan, 2000. ISBN 10: 0333801-237 ISBN 13: 9780333801239

Bertrand Meyer (1988). Object-Oriented Software Construction. C. A. R. Hoare Series Editor, Prentice-Hall, Series in Computer Science.

Abstract Data Types - http://www.nist.gov/dads/HTML/abstractDataType.html - Abstract

Abstract Data Types - http://www.cs.ucsd.edu/users/goguen/ps/beatcs-adj.ps.gz - Um paper sobre o básico de ADTs, e uma boa lista de referencias. As pages 3-4 são as mais relevantes. (arquivo compactado)

Andrews, Peter B. (2002). An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof, 2nd ed, Kluwer Academic Publishers, ISBN 1-4020-0763-9

Intuitionistic Type Theory - ⟨https://en.wikipedia.org/wiki/Intuitionistic_type_theory⟩

Bengt Nordström; Kent Petersson; Jan M. Smith (1990). Programming in Martin-Löf's Type Theory. Oxford University Press p.90

Per Martin-Löf, Constructive mathematics and computer programming, Logic, methodology and philosophy of science, VI (Hannover, 1979), Stud. Logic Found. Math., v. 104, pp. 153-175, North-Holland, Amsterdam, 1982.

P. Andrews - An Introduction to Mathematical Logic and Type Theory, Academic Press, 1986.

CAPÍTULO 14

O Paradigma Computacional da Computação Ubíqua

Conforme **Weiser** e **Brown** (in "The Coming Age of Calm Thechnolgy", 1996), a computação eletrônica passou por duas grandes eras: o uso dos mainframes e o uso do computador pessoal. No início da era dos mainframes, perdurava a ideia dos usuários irem ao computador. Este fato foi mudando para a ideia dos computadores irem ao usuário, o que prevalece na era dos computadores pessoais. Desde 1984 o número de pessoas usando computadores pessoais é maior que o número de pessoas compartilhando computadores. Nos anos 90, a Internet e a computação distribuída tem sido a transição em direção à computação ubíqua, para muitos computadores compartilharem cada um de nós. Se o objetivo é não obrigar o usuário a ir até ao computador, uma saída possível é ter os dispositivos para que possam ser facilmente, embarcados ou carregados ("vestidos") (*Computação Pervasiva*), enquanto o usuário se movimenta livremente (*Computação Móvel*). A junção dessas tecnologias pode ser chamado de *Computação Ubíqua*. E hoje, após uma transição pelo período da Internet e da Computação Distribuída, já entramos na era da Computação Ubíqua, com muitos computadores compartilhando cada um de nós, com muitos computadores embutidos em paredes, móveis, roupas, carros, aviões, navios e a chamada computação veicular, compartilhando cada um de nós. Alguns desses computadores serão centenas daqueles que as pessoas podem acessar durante alguns minutos de consulta na Internet. A computação ubíqua é fundamentalmente caracterizada pela conexão das coisas existentes no mundo através da computação.

Para se entender e posicionar a Computação Ubíqua é necessário ter em mente alguns conceitos. Resumidamente a Computação Ubíqua está posicionada entre a Computação Móvel e a Computação Pervasiva. Computação Móvel é a capacidade de um dispositivo computacional e os serviços associados ao mesmo serem móveis, permitindo este ser carregado ou transportado mantendo-se conectado a rede ou a Internet. Este conceito define que os meios de computação estarão distribuídos no ambiente de trabalho dos usuários de forma perceptível ou imperceptível.

Uma forma fácil de se situar a Computação Ubíqua beneficia-se dos avanços tecnológicos de ambos os ramos de pesquisa. Portanto, a Computação Ubíqua é a integração entre a mobilidade e os sistemas de presença distribuída, em grande parte imperceptível, inteligente e altamente integrada dos computadores e suas aplicações para o benefício dos usuários.

14.1 O Início da Computação Ubíqua

O termo computação ubíqua foi primeiramente sugerido por **Mark Weiser** em 1988 para descrever sua idéia de tornar os computadores onipresentes e invisíveis. Isto corresponde a tentativa de tirar o computador do caminho entre o usuário e seu trabalho. Seu objetivo é ir além da "interface amigável" e ir mais longe da realidade virtual. O termo Computação Ubíqua, foi definido pela primeira vez pelo cientista chefe do Centro de Pesquisa Xerox PARC, **Mark Weiser**, através de seu artigo "*The Computer for the 21st Century*". **Mark D. Weiser** (1952-1999) nasceu Illinois nos Estados Unidos.

Figura 133 – O computador para o século XXI.

Fonte: www.cs.umd.edu.

Weiser publicou este artigo no final dos anos 80, e já naquela época previa um aumento nas funcionalidades e na disponibilidade de serviços de computação para os usuários finais. Entretanto, a visibilidade destes serviços seria a menor possível. Para ele, a computação não seria exclusividade de um computador, uma simples caixa mesmo que de dimensões reduzidas e, sim, diversos dispositivos conectados entre si. Numa época em que os usuários de computação ao executarem suas tarefas lançavam mão de PCs (*desktops*), e detinham grande parte de sua atenção e conhecimento na operação do computador em si. Weiser teorizou que no futuro o foco destes usuários ficaria voltado para a tarefa, e não para a ferramenta utilizada, utilizando-se de computação sem perceber ou necessitar de conhecimentos técnicos da máquina

utilizada: *The world is not a desktop* - **Mark Weiser** (*Interactions* - Janeiro de 1994 - pp 7-8). Através da evolução dos Sistemas de Informação Distribuídos (SID), percebido inicialmente com o desenvolvimento da Internet, e a ampliação das opções de conexões, verifica-se que a Computação Ubíqua já é realidade comprovado pelos benefícios que a Computação Móvel trouxe aos usuários. Celulares com acesso à Web, *Laptops*, Redes WIFI, Lousas Digitais, *I-Pods* e o maior expoente de todos, o *I-Phone*, permitem ao mais leigo, sem perceber, a utilização a qualquer momento e em qualquer lugar de um sistema de computação, através de um software e/ou uma interface.

14.2 O Pensamento de Mark Weiser

"Ubiquitous computing in this context does not just mean computers that can be carried to the beach, jungle or airport. Even the most powerful notebook computer, with access to a worldwide information network, still focuses attention on a single box. By analogy to writing, carrying a super-laptop is like owning just one very important book. Customizing this book, even writing millions of other books, does not begin to capture the real power of literacy."

A idéia é permitir que o usuário faça seu trabalho com o auxílio de computadores, sem nunca ter que se preocupar em trabalhar nos computadores. A computação ubíqua surgiu em 1988 e desde este ano, o XEROX PARC (Centro de Pesquisa de Tecnologia da Xerox Corp. em Palo Alto-CA-EUA) vem pesquisando e desenvolvendo soluções de computação ubíqua, e a partir de 1990 alguns protótipos foram desenvolvidos e comercializados. Nos laboratórios do *Palo Alto Research Center* (PARC), da Xerox, sendo os projetos lá desenvolvidos concentrados em três classes de dispositivos: *pad*, *liveboard*, *interfaces hands-free* (reconhecimento de voz), computação desagregada (por exemplo, a possibilidade de fazer sua apresentação mover-se para qualquer tela da sala) e o dispositivo *tab*, precursor do *tablet*.

Além da *Computação Móvel* (processamento + mobilidade + comunicação sem fio) e a *Computação Pervasiva* (tecnologia embutida nos mais diversos dispositivos), surgiram outros paradigmas de computação como a *Computação Autonômica* (sistemas que gerenciam a si próprios de acordo com os objetivos do administrador e sem a intervenção humana direta) e os *Ambientes Inteligentes* (redes de sensores sem fio depositadas em ambientes com o objetivo de monitorar condições ambientais ou físicas).
Mark Weiser percebeu a predominância cada vez maior dos dispositivos de computação, conduzindo a mudanças revolucionárias no modo como os usuários usariam os computadores.

- Primeira mudança que **Weiser** previu - "Cada pessoa no mundo utilizaria muitos computadores". A visão de **Weiser**: "uma pessoa, muitos computadores". Em computação ubíqua, computadores aparecem em quase tudo, em forma

e em função, não apenas em número, para acomodar diferentes tarefas. Esse cenário faz surgir questões sobre usability (utilização) e questões econômicas, e toca sobre uma pequena parte de nossas vidas. Mas, nos dá uma idéia do que a "computação em todo lugar" poderia parecer.

- Segunda mudança que Weiser previu - **Weiser** previu que computadores "desapareceriam". Isto reflete a idéia de que computação tornar-se-ia embutida (embarcada): ítens do dia-a-dia que, normalmente, não pensamos ter capacidade computacional, passarão a ter. Máquinas domésticas ou veículos seriam vistos com ou como "dispositivos de computação".

14.3 Computação com Reconhecimento de Contexto

Context-aware computing é uma subárea importante da Computação Ubíqua e Móvel. Este é o caso, por exempolo, do *crachá ativo*, ou melhor, as reações de outros dispositivos (um sensor) à sua presença exemplifica a computação com reconhecimento de contexto. Além da interação explícita com o usuário, o ambiente pode contar com sensores que detectem o que está acontecendo e o que as pessoas estão fazendo de forma geral. Se esta informação for representada de algum modo e disponibilizada para consulta por aplicativos, então estes aplicativos têm uma idéia de o que está acontecendo ao redor do usuário. Isto chama-se *reconhecimento de contexto*, onde sistemas de computadores automaticamente adaptam seu comportamento de acordo as circunstâncias físicas. Tais circunstâncias físicas podem ser, em princípio, qualquer coisa fisicamente mensurável ou detectável. Relacionado à computação com reconhecimento de contexto, quando um ambiente possui uma reapresentação de contexto, pode também ter comportamentos automáticos ativados por determinados acontecimentos, sem nenhuma instrução explícita do usuário. Isso pode ser chamado de "*Ambiente Inteligente*". Se a única maneira de interagir com o ambiente for através de tais comportamentos automáticos, isso pode ser chamado "computação invisível".

14.4 Bibliografia e Fonte de Consulta

WEISER M., BROWN S. John. The Coming Age of Calm Technology. Xerox PARC. Outubro de 1995. Esse artigo foi uma revisão da versão: Weiser Brown. "Designing Calm Technology", PowerGrid Journal, v 1.01. Disponível em ⟨http://www.ubiq.com/hypertext/weiser/acmfuture2endnote.htm⟩.

WEISER, M. The computer for the 21st century. Scientific American, 265(3): pp. 94-104, Setembro de 1991. Disponível em ⟨http://www.ubiq.com/hypertext/weiser/SciAMDraft3.html⟩.

WEISER, M. Hot Topic: Ubiquitous Computing. IEEE Computer, pp. 71-72, Outubro de 1993.

Greenfield, Adam. Everywhere: the dawning age of ubiquitous computing. [S.l.]: New Riders, 2006. p.11-12 p. (ISBN 0-321-38401-6).

14.5 Referências - Leitura Recomendada

WEISER, M., Ellis J., GOLDEBERG D., PETERSEN Karin, GOLD, R., ADAMS Norman, SCHILIT N. Bill, WANT Roy. The ParcTab Ubiquitous Computing Experiment. University of California, 2000. Disponível em ⟨http://citeseer.nj.nec.com/su00mobility.html⟩

WEISER, M. Some computer science issues in ubiquitous computing. CACM, 36(7): pp. 74-83, Julho de 1993. Disponível em ⟨http://www.ubiq.com/hypertxt/weiser/UbiCACM.html⟩

WEISER, M. The world is not a desktop. Interactions, pp. 7-8, Janeiro de 1994. Disponível em ⟨http://www.ubiq.com/hypertext/weiser/ACMInteractions2.html⟩

Hansmann, Uwe. Pervasive Computing: The Mobile Word. [S.l.]: Springer, 2003. ISBN 354-000-218-9.

CAPÍTULO 15

O Futuro - A Computação Quântica

O Paradigma da computação quântica é agora focalizado, como o futuro da ciëncia da computação. Este é um capitulo de caráter didático, direcionado ao leitor, que não necessariamente tenha conhecimentos prévios sobre mecânica quântica. Como em Mattielo et al. (2012), são mostradas questões fundamentais que mostram pesquisas nesta área da ciência, bem como apresentamos as perspectivas de suas aplicações. A **computação quântica** é a ciência que estuda as aplicações das teorias e propriedades da **mecânica quântica** na **Ciência da Computação**. Dessa forma seu principal foco é o desenvolvimento do **computador quântico**, possivelmente, a tentativa de se ultrapassar as limitações dos computadores de hoje, previstas no modelo abstrato da máquina de Turing.

15.1 As Limitações dos Computadores Atuais

Na arquitetura de **von Neumann** dos computadores de hoje, o computador e a computação clássica fazem a distinção clara entre os componentes de processamento e de armazenamento de dados, isto é, possui processador e memória, separados, comunicando-se por um barramento de comunicação, sendo seu processamento de natureza sequencial, mas, também, podendo-se ter computadores com arquitetura paralela nos moldes de **von Neumann**.

Contudo, os computadores atuais possuem limitações. Dessa forma surge a necessidade da criação de um computador diferente dos usuais que possa resolver determinados problemas computacionais, inviáveis no computador usual, como por exemplo, os problemas matemáticos algébricos como a fatoração de números primos muito grandes ou logaritmos discretos.

Apesar de haver um constante crescimento na velocidade de processamento dos computadores, essa evolução pode atingir um certo limite , a um ponto onde não será

possível aumentar essa velocidade e, então se fez necessário uma revolução significativa na construção de computadores, onde uma nova máquina de computação possa superar esse obstáculo, quebrando, assim, as limitações dos computadores usuais. E assim, os estudos em **Computação Quântica** se tornaram muito importantes e a necessidade do desenvolvimento de uma máquina extremamente eficiente se torna maior.

Tudo o que venha a mudar as estruturas clássicas de computação, vem das mudanças que a **Física Quântica** propõe e introduz. E, portanto, a computação quântica pode quebrar alguns paradigmas da computação clássica.

15.2 A Lei de Moore

Até meados de 1965 não havia nenhuma previsão real sobre o futuro do *hardware* quando o então presidente da Intel, **Gordon E. Moore** (1929 - ...), engenheiro estadunidense, fez sua profecia, na qual o número de transistores dos chips teria um aumento de 100%, pelo mesmo custo, a cada período de 18 meses. Essa profecia tornou-se realidade e acabou ganhando o nome de **Lei de Moore**. A Lei de **Moore** afirma que a velocidade de um computador é dobrada a cada 18 meses. Assim sempre houve um crescimento constante na velocidade de processamento dos computadores. Entretanto essa evolução pode atingir um certo limite, um ponto onde não mais será possível aumentar essa velocidade e então se fez necessário uma revolução significativa na computação para que este obstáculo fosse quebrado. E assim os estudos em *Computação Quântica* se tornaram muito importantes e a necessidade do desenvolvimento de uma máquina extremamente eficiente se torna maior a cada dia. No início de 2014 o departamento de pesquisa da IBM anunciou um teste de novos chips de silício com tecnologia de 7nm empurrando para novos limites o previsto fim da *Lei de Moore*. Em Outubro de 2015 foi anunciada uma nova pesquisa da IBM iniciando a caminhada para novos limites na produção de processadores utilizando *nano tubos de carbono*, o que permitiria atingir escalas de 1.8nm.

15.3 História da Computação Quântica

A grande revolução cientifica no século XX teve início, a partir de uma crise que a Física atravessava no fim do século XIX, pois neste período haviam alguns fenômenos naturais que não eram explicados pelas teorias físicas até então existentes. Nesse escopo, existe uma data precisa para o nascimento da Física Quântica, trata-se do dia 14 de dezembro de 1900, quando o cientista **Max Karl Ernst Ludwig Planck** (1858-1947) explica a emissão de radiação do *corpo negro*. Na Física, um *corpo negro* é aquele que absorve toda a radiação eletromagnética que nele incide, nenhuma luz o atravessa e nem é refletida. Um corpo com essa propriedade, em princípio, não pode ser visto , daí o nome *corpo negro*. A explicação elaborada por **Planck** foi baseada na hipótese de que a radiação deveria ser quantizada. **Max Planck** (o substituto de **Gustav Kirchhoff** na Universidade de Berlin) foi um físico alemão. É considerado o pai da física quântica e um dos físicos mais importantes do século XX.

Figura 134 – Gordon Earle Moore - É co-fundador da Intel Corporation, autor da Lei de Moore.

Fonte: en.wikipedia.org.

Figura 135 – Mark Plank - Explicou a emissão de radiação do corpo negro.

Fonte: Google Images - https://en.wikipedia.or/wiki/Mark_plank.

Em fins do século XIX, uma das dificuldades da Física consistia na interpretação das leis que governam a *emissão de radiação por parte dos corpos negros. Tais corpos são dotados de alto coeficiente de absorção de radiações.* Por isso, *parecem negros para a visão humana.*

Em 1899, após pesquisar as radiações eletromagnéticas, descobriu uma nova constante fundamental, batizada posteriormente em sua homenagem como *Constante de Planck* Fazzio (2007). Um ano depois, descobriu a lei da radiação térmica, chamada Lei de Planck da Radiação. Essa foi a base da *teoria quântica*, que surgiu dez anos depois com a colaboração de **Albert Einstein** e **Niels Bohr**. De 1905 a 1909, **Planck** atuou como diretor-chefe da Deutsche Physikalische Gesellschaft (Sociedade Alemã de Física). Em 1913 foi nomeado reitor da Universidade de Berlim. **Planck** foi laureado com o Nobel de Física de 1918, por suas contribuições na área da Física Quântica.

Alguns anos mais tarde, em 1905, o renomado físico alemão **Albert Einstein** (1879-1955) elucida o *Efeito Fotoelétrico*, que era outro fenômeno não explicado pela teoria ondulatória da luz. O *efeito fotoelétrico consiste na emissão de elétrons por uma superfície metálica bombardeada por um feixe de luz.* Embora muitos de nós não saibamos, tal efeito é bem familiar, pois está presente em nosso cotidiano, como por exemplo, como era o funcionamento dos tubos de raios catódicos (em televisão e vídeos em computadores mais antigos) as portas que abrem e fecham automaticamente têm o funcionamento baseado nesse fenômeno. **Einstein** explicou o efeito fotoelétrico admitindo a hipótese de que a luz é constituída por pacotes concentrados de energia, que atualmente denominamos *fótons*, e assim, o fenômeno em questão é facilmente explicado quando consideramos a colisão entre os *fótons* da radiação incidente e os dos metais*elétrons dos metais.*

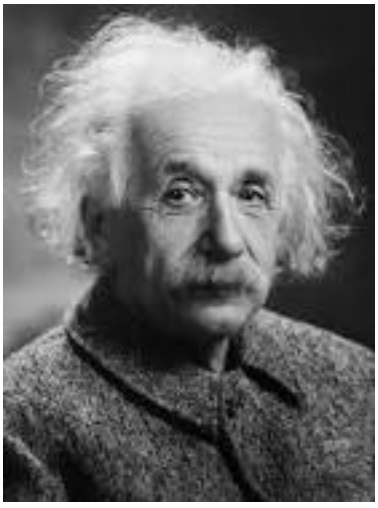


Figura 136 – Albert Einstein - Quem elucidou o Efeito Fotoelétrico.

Fonte: Google Images - https://en.wikipedia.or/wiki/Albert_Einstein.

É importante notarmos que os trabalhos de **Einstein** e **Planck** sugeriam a quan-

tização, mas não explicavam o porquê. Um problema parecido surgiu quando o físico dinamarquês **Niels Bohr**, em 1913, introduziu o seu modelo atômico, no qual supôs que o *elétron* poderia se mover somente em órbitas determinadas, onde não emitia radiação eletromagnética. A radiação era emitida somente quando o *elétron* "saltava" de uma órbita para outra. Com esse modelo, **Bohr** solucionou a estabilidade atômica e explicou o espectro discreto de radiação para o átomo de hidrogênio, porém, não ficou claro o motivo pelo qual o *elétron* não poderia ocupar posições intermediárias no espaço. Assim, por meio desses exemplos, percebemos que a teoria quântica desenvolvida até o primeiro quarto do século XX possuía bases teóricas e conceituais frágeis (com uma rosa, as teorias eram frágeis, mas não fracas), pois os princípios eram esparsos e os enunciados eram criados com a finalidade específica de atender a uma necessidade (ou problema) pontual. Nesse escopo, os físicos ressentiam-se de postulados autênticos e princípios gerais dos quais poderiam formular uma teoria consistente, eficiente e abrangente. Esse desejo dos físicos se tornou realidade com o surgimento da **Mecânica Quântica** Pires (2008), Griffiths (1995).

15.4 A Mecânica Quântica

A **Mecânica Quântica** é considerada uma teoria científica muito bem sucedida da história da ciência. Esse fato se deve a sua infalibilidade, até o momento, de suas previsões serem constatadas mediante os experimentos. A **Mecânica Quântica** é a teoria física utilizada para tratar as partículas microscópicas, de ordem de tamanho atômico ou molecular, pois se utilizarmos as leis de **Newton** para analisarmos o comportamento de partículas com essa ordem de tamanho, chegaríamos a resultados incompatíveis com os experimentos. Sendo assim, a **Mecânica Quântica** constitui a base da Física Atômica, da Física Nuclear, da Física do Estado Sólido e da Química Moderna, no sentido que um grande número de utensílios com valor tecnológico agregado tenham seus princípios de funcionamento embasados na Mecânica Quântica. Para se ter uma idéia, desde o pós-guerra, cerca de um terço do produto interno bruto dos Estados Unidos é oriundo da aplicação da Mecânica Quântica Fazzio (2007).

Dessa forma, quando observarmos, por exemplo, os modernos telefones celulares e televisores, além de diversos outros equipamentos eletrônicos, devemos nos lembrar que são oriundos da vasta aplicabilidade da teoria quântica. E ainda, há projeções que indicam que a partir da segunda década do século corrente, boa parte dos empregos em manufatura no mundo estarão ligados à **nanotecnologia**, e para se trabalhar nessa escala é indispensável um conhecimento em **Mecânica Quântica**.

Deixemos a **Mecânica Quântica** um pouco de lado para focalizarmos fatos históricos da **Ciência da Computação**. De certa forma, podemos dizer que a ciência da computação nasceu com o notável trabalho do matemático **Alan Turing** em 1936. Naquele trabalho, **Turing** desenvolveu a noção abstrata do que conhecemos como computador programável, o que ficou conhecido como *máquina de Turing*. Este importante artefato proposto por **Turing** operava com sequências lógicas de unidades

de informação chamadas *bits* (binary digit), os quais podem adquirir os valores "0" ou "1".

A idéia de **Turing** foi tão importante para o desenvolvimento da humanidade que os computadores que utilizamos hoje, desde o simples computador, até o mais potente computador de um grande centro de pesquisa, certamente tratam de uma realização física da máquina de Turing. Assim, toda a informação fornecida a um computador é lida, processada e retornada sob a forma de sequências de bits. Um fato que revela o poder da máquina de Turing é decorrente da chamada **tese de Church-Turing**, a qual estabelece que a máquina de **Turing**, apesar de simples, é capaz de resolver qualquer problema computacional solúvel por qualquer outro tipo de computador. Essa tese implica que se existir algum problema que seja insolúvel para a máquina de **Turing**, tal problema não poderá ser resolvido por nenhum outro tipo de computador. A **tese de Church-Turing** diz que não precisamos considerar máquinas diferentes da máquina de **Turing** para saber se um problema é computável ou não, mas a tese não diz nada sobre o tempo necessário para solucionar um problema que possa ser computável. Isso possibilitou que os computadores evoluíssem em velocidade, mas sem perder o princípio fundamental de funcionamento baseado nos bits.

Nesse sentido, chegará um momento na evolução dos computadores que será inevitável. Esse grande acontecimento abrirá as portas para o que chamamos de **Computação Quântica**, que poderá revolucionar a forma atual que a humanidade trata a informação. Sendo assim, o objetivo deste trabalho é apresentar a **Computação Quântica** de uma forma pedagógica, na perspectiva de sua introdução nos níveis iniciais de um curso de graduação, bem como na informação do público em geral.

15.5 A Pesquisa em Computação Quântica

A pesquisa para o desenvolvimento da computação quântica iniciou-se já na década de 50 quando imaginaram em aplicar as leis da física e da mecânica quântica nos computadores. Em 1981 em uma conferencia no MIT o físico **Richard Feynman** apresentou uma proposta para utilização de sistemas quânticos em computadores, que teriam então uma capacidade de processamento superior aos computadores comuns. Já em 1985, **David Deutsch**, da Universidade de Oxford, descreveu o primeiro computador quântico, uma **Máquina de Turing Quântica**, onde ele simularia outro modelo de computador quântico. Depois de Deutsch apenas em 1994 surgiram novas notícias da computação quântica, quando em Nova Jersey, no Bell Labs da AT&T, o professor de matemática aplicada **Peter Shor** desenvolveu o **Algoritmo de Shor**, capaz de **fatorar grandes números numa velocidade muito superior à dos computadores convencionais**. Em 1996, **Lov Grover**, também da Bell Labs, desenvolveu o *Speedup*, o **primeiro algoritmo para pesquisa de base de dados quânticos**. Nesse mesmo ano foi proposto um modelo para a correção do erro quântico. Em 1999 no MIT foram construídos os primeiros protótipos de computadores quânticos utilizando montagem térmica. No ano de 2007 surge o Orion,

um processador quântico de 16 **q-bits** que realiza tarefas praticas foi desenvolvido pela empresa canadense D-Wave. Em 2011 a *D-Wave* lançou o primeiro computador quântico para comercialização, o D-Wave One, que possui um processador de 128 **q-bits**. Porém o *D-Wave One* ainda não é totalmente independente, precisa ser usado em conjunto com computadores convencionais.

15.6 As Propriedades dos q-bits

Podemos fazer uma representação bastante simplória de um bit utilizando uma moeda. Para esse fim, representaremos o resultado "cara" com o "0" e o resultado "coroa" com o "1". Conforme intuímos, a moedas são **objetos macroscópicos**, governados pela física clássica, e por isso só obtemos um dos resultados ("0" ou "1") em cada jogada. No entanto, se as moedas se comportassem como os **objetos microscópicos**, que obedecem os princípios da mecânica quântica, teríamos que as faces, cara e coroa, poderiam ser vistas ao mesmo tempo, ou seja, no lançamento de uma moeda, o resultado cara e coroa coexistiriam. Esse resultado é possível graças a uma das propriedades da mecânica quântica, denominada de **superposição**. Se novamente supusermos que uma moeda é um objeto quântico, os resultados "1" e "0" são denominados de *estados* e são representados por $\mid 0 >$ e $\mid 1 >$ (pronuncia-se: 'ket 0' and 'ket 1'). E neste modo, um **q-bit** (bit quântico) é representado por meio da expressão abaixo, como em Mattielo et al. (2012).

$$\mid \varphi > = \alpha \mid 0 > + \beta \mid 1 >$$

onde α e β são números complexos, que fornecem a probabilidade de se encontrar $\mid 0 >$ e $\mid 1 >$, respectivamente, no lançamento de uma moeda como um objeto microscópico.

A *superposição* de estados, apesar de parecer um pouco estranha, pode ser entendida mediante uma analogia conhecida como **gato de Schroedinger**. Com o objetivo de explicar as minúcias das soluções da equação fundamental da mecânica quântica e do princípio da superposição, **Schröedinger** propôs o experimento imaginário no qual utiliza um gato que supostamente pode estar vivo ou morto ao mesmo tempo. **Erwin Rudolf Josef Alexander Schrödinger** (1887-1961) foi um físico teórico austríaco, conhecido por suas contribuições à mecânica quântica, especialmente a equação de Schrödinger, pela qual recebeu o Nobel de Física em 1933. Propôs o experimento mental conhecido como o *Gato de Schrödinger*.

Esse exemplo trata de uma forma simples de analisar o princípio da *superposição* das soluções da equação de **Schroedinger**. Tal princípio afirma que se para um determinado problema a equação de **Schroedinger** admitir duas soluções distintas e , então o sistema pode ser descrito por meio da superposição das duas soluções, o que é matematicamente escrito como está em Eisberg e Resnick (1979), Cohen-Tannoudji, Diu e Laloe (1992):

$$\mid \varphi > = \mid a > + \mid b >$$

Figura 137 – Schrödinger em 1933 - Explicou o princípio da superposição de estados da Mecânica Quântica.

Fonte: https://pt.wikipedia.org/wiki/Erwin_Schrrdinger.

onde os estados quânticos $\mid a >$ e $\mid b >$ existem simultaneamente. Contudo, quando observarmos o sistema, ou seja, quando efetuarmos uma medição, um dos estados entra em colapso (deixa de existir) e o outro é então detectado. Nesse sentido, o referido gato de Schroedinger faz alusão a essa curiosa propriedade quântica, conforme explicitamos a seguir.

O gato de **Schroedinger** - Considere um gato preso numa caixa onde há um recipiente com material radioativo que tem 50% de chance de emitir uma partícula radioativa a cada hora, e um contador *Geiger*. O contador *Geiger* é um aparelho utilizado para detectar radiação. Se o material liberar partículas radioativas, o contador percebe a sua presença e aciona um martelo, que, por sua vez, quebra um frasco de veneno. Evidentemente, ao se passar uma hora só terá ocorrido um dos dois casos possíveis: o material radioativo emitiu uma partícula radioativa ou não a emitiu (a probabilidade que ocorra um ou outro evento é a mesma). Como resultado da interação, no interior da caixa o gato estará vivo ou morto. Porém, isso não poderemos saber a menos que se abra a caixa para comprovar as hipóteses. Se tentarmos descrever o que ocorreu no interior da caixa, servindo-nos das leis da mecânica quântica, chegaremos a uma conclusão muito estranha. O gato viria descrito por uma função de onda extremamente complexa resultado da superposição de dois estados, combinando 50% de "gato vivo" e 50% de "gato morto". Ou seja, aplicando-se o formalismo quântico, o gato estaria por sua vez "vivo" e "morto"; correspondente a dois estados indistinguíveis! Assim, o *estado* do gato seria dado por:

$$\mid \varphi > = \mid vivo > \; + \; \mid morto >$$

A única forma de averiguar o que "realmente" aconteceu com o gato será abrir a

caixa e olhar dentro. Em alguns casos encontraremos o gato vivo e em outros um gato morto. Isso ocorre por que ao realizar a medida, o observador interage com o sistema e o altera, rompendo a superposição dos dois estados, fazendo com o que o sistema seja observado em um dos dois estados possíveis. E isso é uma forma simplista de explicar o que se passa na computação quântica, que é uma característica inerente ao processo de medição em mecânica quântica.

Nessa perspectiva, um **q-bit** pode existir num estado contínuo entre $| 0 >$ e $| 1 >$ até que ele seja observado. E então, quando um **q-bit** é medido, o resultado será sempre "0" ou "1", probabilisticamente. Vale destacar que os **q-bits** são objetos matemáticos com certas propriedades específicas que podem ser implementados como objetos físicos reais.

O q-bit é descrito por um *vetor de estados* em um sistema quântico de dois níveis. Usa-se a notação seguinte para representá-los:

$$| 0 > = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \text{ e } | 1 > = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Assim, o estado de um q-bit pode ser representado por: $| \varphi > = | 0 > + | 1 >$, e o conjunto $\{| 0 >, | 1 >\}$ forma uma base no espaço de **Hilbert** de duas dimensões (que poderá ser estudado em outro livro), chamada de base computacional. Este conjunto forma uma base no **espaço de Hilbert** de duas dimensões, chamada de base computacional.

15.7 A Matemática da Mecânica Quântica

As formulações matemáticas da mecânica quântica são os formalismos matemáticos que permitam uma descrição rigorosa da mecânica quântica. Estas, por sua vez, se distinguem do formalismo matemático da mecânica clássica, (antes do início de 1900) pelo uso de estruturas matemáticas abstratas, tais como **espaços de Hilbert de dimensão infinita e operadores sobre estes espaços**. Muitas destas estruturas são retiradas da *análise funcional*, uma área de pesquisa dentro matemática que foi influenciada, em parte, pelas necessidades da mecânica quântica. Em resumo, os valores físicos observáveis, tais como *energia* e *momento* já não eram considerados como valores de funções em espaço de fase, mas como *autovalores*, mais precisamente: como valores espectrais de *operadores lineares* nos **espaços de Hilbert** como em Byron e Fuller (1992). Mas esta matemática é assunto para um volume independente.

A *Análise Funcional* é o ramo da matemática, e mais especificamente da Análise Matemática, que trata do estudo de *espaços de funções*. Tem suas raízes históricas no estudo de *transformações*, tais como a *Transformada de Fourier*, e no estudo de *equações diferenciais* e *equações integrais*. A palavra *funcional* remonta ao *Cálculo de Variações*, implicando uma função cujo argumento é uma função (ver em Oliveira (2001)). Em matemática, em especial *álgebra linear* e *análise matemática*, define-se

como um *funcional*, toda função cujo domínio é um espaço vetorial e a imagem é um *corpo* de escalares. Intuitivamente, pode-se dizer que um funcional é uma "função de uma função".

O *Cálculo de Variações* é um problema matemático que consiste em buscar máximos e mínimos (ou, mais geralmente, extremos relativos) de funções contínuas definidas sobre algum espaço funcional. Constituem uma generalização do cálculo elementar de máximos e mínimos de funções reais de uma variável. Ao contrário deste, o cálculo das variações lida com os funcionais, enquanto o cálculo ordinário trata de funções. Funcionais podem, por exemplo, ser formados por integrais envolvendo uma função incógnita e suas derivadas. O interesse está em *funções extremas* - aquelas que fazem o funcional atingir um valor máximo ou mínimo - ou de funções fixas - aquelas onde a taxa de variação do funcional é precisamente zero. Um grande impulso para o avanço da análise funcional durante o século XX foi a modelagem, devida a **John von Neumann**, da mecânica quântica em espaços de **Hilbert**. O leitor pode ler sobre **John von Neumann** (matemático, lógico e ciência da computação) e **David Hilbert** (matemático, idealizador do que se chama *formalismo*), nos capítulos referentes a eles no volume I (Dos Primórdios da Matemática aos Sistemas Formais), nesta mesma série. Para entendimento destas áreas da Matemática, as disciplinas de Cálculos (I, II, III, IV) como são ensinadas em cursos de graduação, são muito bem-vindas. Mas, como ressaltadas no Capítulo I do Volume I desta série, fazem parte da base matemática para se estudar os *sistemas contínuos*, não considerados nestes volumes I e II, os quais tratam da *matemática dos sistemas discretos*.

Talvez a melhor contribuição para a matemática tenha sido os **espaços de Hilbert**. Esses são uma generalização do conceito de espaço euclidiano, mas que não precisam estar restrita a um número finito de dimensões. É um espaço vetorial dotado de produto interno, ou seja, com noções de distância e ângulos. Os espaços de **Hilbert** permitem que, de certa maneira, noções intuitivas sejam aplicadas em espaços funcionais (de funções). Os espaços de **Hilbert** são de importância crucial para a Mecânica Quântica.

Os elementos do **espaço de Hilbert** abstrato são chamados vetores. Em aplicações, eles são tipicamente *sequências funções*. **Espaços de Hilbert** desempenham um papel fundamental em toda a Física Quântica e em várias áreas da Matemática. Em Mecânica Quântica, por exemplo, um sistema físico é descrito por um espaço de Hilbert complexo (o corpo dos números complexos é usado) que contém os vetores de estado, e todas as informações do sistema e suas complexidades. Por isto, forma a base matemática de qualquer pesquisa em direção ao estudo da **criptografia quântica**, poosivelmente a área da computação quântica mais bem pesquisada e desenvolvida. Para os pretendentes em criptografia quântica, aqui está a definição de um **espaço de Hilbert**.

Um **espaço de Hilbert** é um espaço vetorial $\mathcal{H}$ sobre o corpo dos complexos $\mathbb{C}$ e dotado de um produto escalar $u,\, v \in \mathcal{H} \mapsto \langle hu \rangle \in \mathbb{C}$. $\mathcal{H}$ é dito ser um **espaço de**

Hilbert, se for completo em relação à métrica d definida por esse produto escalar:

$$d(u, v) = \|u - v\| = \sqrt{\langle u - v, u - v \rangle}, \text{ onde } u,\ v \in \mathcal{H}.$$

Para a manipulação dos estados quânticos utiliza-se o seguinte raciocínio: suponhamos agora que temos dois **q-bits**. Se estivéssemos trabalhando com a computação clássica, teríamos quatro estados possíveis: $\{00, 01, 10, 11\}$. Como consequência, temos que um sistema de dois q-bits possui quatro estados na base computacional: $| 00 >$,$| 01 >$, $| 10 >$ e $| 11 >$. Contudo, de acordo com o *princípio da superposição*, exemplificado por **Schroedinger** mediante o seu exemplo do gato, um par de q-bit também pode existir em superposições desses estados, ou seja, temos o seguinte estado:

$$| \varphi > = \alpha \, | 00 > \ + \beta \, | 01 > \ + \chi \, | 10 > \ + \delta \, | 11 >$$

onde os coeficientes α, β, χ e δ são números complexos, sendo úteis nos cálculos de probabilidades. Na essência, um computador quântico manipula a informação, como se os quatro estados de dois **q-bits** existissem ao mesmo tempo, e essa propriedade tornaria possível uma capacidade computacional muito além da que temos acesso na atualidade.

15.8 Relacionando q-bit × bit

Como, agora interpretar estados quânticos como informação? Um bit clássico pode representar o estado de um transístor em um processador, a magnetização de uma superfície em um disco rígido e a presença de uma corrente elétrica num cabo, podem ser usados para representar bits em um mesmo computador. Através da alocação de elétrons, um bit clássico é capaz de assumir uma única informação como positiva ou negativa, ou ainda 0 ou 1. Toda a computação atual é construída em cima desta base binária na qual, em essência, toda informação assume apenas duas possibilidades diferentes, de maneira independente.

Na computação quântica, qualquer sistema de 2-níveis pode ser usado como um **q-bit**. Sistemas multinível podem também ser usados, conquanto possuam dois estados. Muitas implementações físicas que se aproximam de sistemas de 2-níveis em vários graus foram feitas com sucesso. Um **computador quântico** pode vir a usar várias combinações de **q-bits** em seus padrões.

Um computador quântico é um dispositivo que executa cálculos fazendo uso direto de propriedades da mecânica quântica, tais como *sobreposição* e **interferência**. Teoricamente, computadores quânticos podem ser implementados e o mais desenvolvido atualmente trabalha com poucos q-bits de informação (ver a Tabla 7).

Um *bit quântico*, ou **q-bit** (às vezes q-bit) é a menor unidade de informação quântica. Esta informação é descrita por um vetor de estado em um sistema de mecânica quântica de dois níveis o qual é normalmente equivalente a um vetor de espaço

bidimensional sobre números complexos. **Benjamin Schumacher** descobriu uma maneira de interpretar estados quânticos como informação. Ele apresentou uma forma de comprimir a informação num estado, e armazenar esta num número menor de estados, o que é agora conhecido por *compressão Schumacher*.

Figura 138 – Benjamin Schumacher - Ele descobriu uma maneira de interpretar estados quânticos como informação.

Fonte: www.thegreatcourses.com.

Um *bit* é a base da informação computacional clássica. Independente de suas representações físicas, ele sempre é lido como 0 ou 1. Por outro lado, um *q-bit* (bit quântico) possui algumas similaridades com o bit clássico, mas é bem diferente no geral. Como o bit, um **q-bit** pode ter dois possíveis valores - normalmente um 0 ou um 1. A diferença é que, enquanto um bit deve ser 0 ou 1, um **q-bit** pode ser 0, 1, ou uma **superposição/sobreposição** de ambos.

Na representação **q-bit**, os estados em que um **q-bit** pode ser medido são conhecidos como estados básicos. Para qualquer estado quântico usa-se a notação para representar dois estados básicos computacionais, convencionalmente escritos como $| 0 >$ e $| 1 >$. Na Tabela 7 é mostrado a correspondência teórica que se sabe que se poderá existir num computador quântico, em relação aos computadores binários.
Logo, 1 bit equivale (equivale, não é igual) a 1 q-bit e armazena uma única informação. Mas, enquanto 2 bits juntos armazenam apenas duas informações, 2 q-bits armazenam 4 informações diferentes, do mesmo modo que 3 bits armazenam 3 informações contra 8 informações armazenadas por 3 q-bits. Enquanto a informação total armazenadas pelos bits é igual à soma direta deles ($1 + 1 + 1 + ... = n$), a informação armazenada por um conjunto de q-bits cresce exponencialmente ($2 \times 2 \times 2... = 2^n$). Cada bit adiciona uma única informação ao conjunto, já um único **q-bit** dobra a capacidade de informações do mesmo. De uma forma geral, na **computação quântica** temos: n **q-bits** $= 2^n$ **bits**.

Tabela 7 – Correpondência entre q-bit e bit

q-bits	**bits**
2 q-bits	4 bits
3 q-bits	8 bits = 1 **byte**
4 q-bits	16 bits
5 q-bits	32 bits
6 q-bits	64 bits
7 q-bits	128 bits
8 q-bits	256 bits
9 q-bits	512 bits
10 q-bits	1.024 bits = 1 **Kilobyte** = 1 **Kb**
20 q-bits	1.048.576 bits
30 q-bits	1.073.741.824 bits
40 q-bits	1.099.511.627.776 bits
41 q-bits	2.199.023.255.552 bits
42 q-bits	4.398.046.511.104 bits
43 q-bits	8.796.093.022.208 bits = 1 **Terabyte**

Fonte: http://www.tecmundo.com.br/.

Um *computador quântico* não possui qualquer limite teórico para sua capacidade. Um *bit* comum armazena informação de forma bem menos complexa, do que um q-bit (bit quântico) que pode armazenar informações de forma infinitamente mais complexas em seu nível quântico. Desde que operações sobre os q-bits se tornem suficientemente eficazes, computadores poderão ser extremamente mais rápidos do que a mais rápida das máquinas de hoje.

15.8.1 Propriedades da Computação Quântica

Registrador Quântico

Um número de **q-bits** entrelaçados tomados juntos, forma um *registrador quântico*. Computadores quânticos realizam cálculos através da manipulação dos q-bits dentro de um registrador quântico.

Entrelaçamento Quântico

Uma importante diferença entre o *q-bit* e o bit clássico é que vários **q-bits** podem exibir **entrelaçamento quântico**. *Entrelaçamento* é uma propriedade que permite que um conjunto de **q-bits** consiga uma *correlação* maior (possivelmente, no sentido de proporcionar significado com relação à informação) do que o esperado em sistemas clássicos.

Exemplo (Comunicação q-bit entre Alice e Bob)

Considere que estes dois *q-bits* entrelaçados sejam separados, e sejam dados um para Alice e outro para Bob. Alice faz uma medida do q-bit dela, obtendo com igual probabilidade $| 0 >$ ou $| 1 >$. Por causa do **entrelaçamento dos q-bits**, Bob deve agora conseguir a mesma medida que Alice, isto é, se ela mediu $| 0 >$, Bob deve medir o mesmo, uma vez que é o único estado onde o q-bit de Alice é $| 0 >$.

Diferentemente dos bits clássicos que podem apenas ter um valor de cada vez, e poder manter dois estados 0 e 1. O **entrelaçamento** também permite outros estados serem estabelecidos simultaneamente. O conceito de entrelaçamento é um ingrediente chave para a **computação quântica**. **Entrelaçamento** não pode ser feito em um computador clássico. A ideia da computação e comunicação quântica fazem uso do **entrelaçamento**, sugerindo que o **entrelaçamento** é imprescindível para a **computação quântica**.

15.9 Algoritmos Quânticos

Qualquer pessoa que tenha algum conhecimento sobre computação, sabe que para um computador desenvolver determinada tarefa é necessário programá-lo. E, antes disso, é imprescindível elaborar um bom algoritmo, que por sua vez, são implementados transformando-se em programas de computador. Contudo, com o advento da computação quântica, esses programas ficarão obsoletos. Ou melhor, não somente esses programas, mas a teoria utilizada para elaborá-los ficará obsoleta. Dessa forma, quando essa nova revolução ocorrer, os programas deverão ser construídos a partir de algoritmos quânticos. É justamente neste ponto que aparece um novo desafio, pois com esse novo paradigma, os futuros programadores deverão conhecer bem a forma como a informação deve ser tratada na perspectiva quântica, de forma que deter conhecimento sobre mecânica quântica deixará de ser um privilégio restrito aos físicos.

Atualmente, já existem alguns algoritmos quânticos propostos, que de certa forma apresentam considerável vantagem sobre os algoritmos clássicos. Um desses algoritmos quânticos foi desenvolvido por **Peter Shor** em 1993.

Peter Williston Shor (1959- ...) é um matemático estadunidense. É professor de matemática aplicada no Instituto de Tecnologia de Massachusetts, Departamento de Matemática (MIT). É conhecido por seu trabalho em **computação quântica**, em particular pela elaboração do algoritmo de Shor, um algoritmo quântico para fatorar exponencialmente mais rápido que o melhor algoritmo conhecido atualmente rodando em um computador clássico. O trabalho de **Shor**, equivale nos tempos de hoje, ao trabalho de **Turing** que, na década de 30, pensava em computação sem computador. **Shor** pensa em computação quântica sem computador quântico.

Com um computador digital crê-se ser capaz de simular qualquer dispositivo de computação física com um aumento no tempo de cálculo de um fator de, no máximo,

Figura 139 – Peter Shor - O algoritmo quântico de fatoração de números primos grandes.

Fonte: en.wikipedia.org.

polinomial. Isso não pode ser verdade quando a mecânica quântica é levado em consideração. O trabalho de **Peter Shor** em (⟨http://arxiv.org/abs/quant-ph/9508027⟩) considera fatorar inteiros e encontrar logaritmos discretos, dois problemas que são geralmente dificeis em um computador clássico e têm sido usados como a base de vários sistemas criptográficos propostas (ver Terada (2008)). Algoritmos aleatórios eficientes são dadas para estes dois problemas em um computador quântico hipotético. Estes algoritmos tomam uma série de etapas e é polinomial no tamanho da entrada, por exemplo, o número de dígitos do número inteiro a ser fatorado.

Quando propôs o algoritmo, **Shor** trabalhava na empresa AT&T e desenvolvia pesquisas que apontavam vantagens dos computadores quânticos em relação à máquina de **Turing**. Nesse panorama, **Shor** formulou um algoritmo quântico que permitia decompor um número com muitos algarismos em seus fatores primos. O detalhe fundamental é que o algoritmo de **Shor** realiza essa tarefa em tempos muitos menores do que os gastos por algoritmos clássicos. O problema da fatoração é essencial para os sistemas criptográficos atuais, mas que não trairiam vantagens quanto a inviabilidade computacional útil na criptografia convencial. Notamos, ainda que, os sistemas criptográficos de segurança baseados em chave pública ficarão totalmente obsoletos a partir do momento em que o primeiro computador quântico iniciar o seu funcionamento. É conhecido do ponto de vita teórico, o elevado potencial computacional dos sistemas quânticos.

Um outro algoritmo quântico que merece destaque foi proposto pelo indiano **Lov Grover** (1961- ...) é um cientista indiano-americano, originador do algoritmo de

busca em bancos de dados usado em computação quântica. Enquanto trabalhava nos laboratórios de pesquisa Bell, nos Estados Unidos. **Grover**, em 1996, propôs um algoritmo de busca, o qual, como o próprio nome já sugere, realiza a tarefa de buscar numa base de dados, encontrando itens que tenham certas propriedades desejadas. Estamos acostumados a utilizar algumas espécies de sistemas como esses quando usamos a Internet. Assim como o algoritmo de **Shor**, a vantagem computacional apresentada pelo algoritmo de **Grover** foi estupenda, pois em geral, numa determinada tarefa onde classicamente precisamos fazer n buscas, quanticamente são necessárias $\sqrt{n}$, que é um número muito menor.

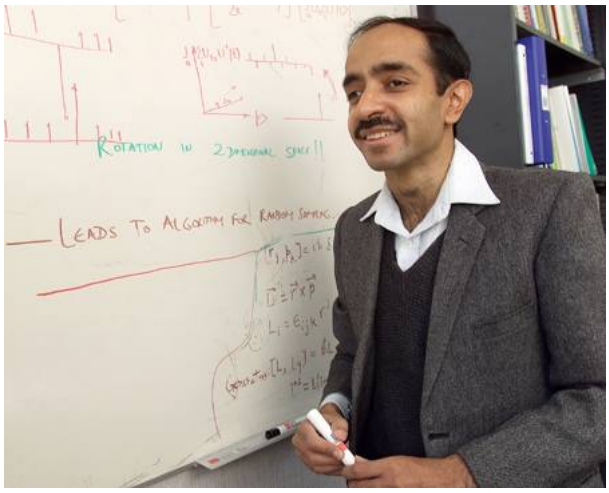


Figura 140 – Lov Grover - O inventor do algoritmo de busca em um banco de dados quântico.

Fonte: Google Images - www.kennislink.nl.

A título de ilustração, considere uma tarefa na qual classicamente necessitaríamos 10000 buscas; dessa maneira, quanticamente seriam necessárias apenas 100 buscas. O algoritmo de **Grover** pode ser aplicado com sucesso em problemas práticos da biologia molecular e engenharia genética. Assim, mediante esses dois exemplos de algoritmos quânticos, percebemos que computadores quânticos poderão, de fato, revolucionar a forma como tratamos a informação, sendo necessário, para isso, que novos algoritmos quânticos sejam elaborados. Este é um grande desafio para o futuro da ciência da computação Nielsen e Chuang (2000).

15.10 Experiências de Computadores Quânticos

Como em Mattielo et al. (2012), a computação quântica experimental está enfrentando atualmente um panorama muito parecido ao que a computação clássica encontrou na década dos anos 30, pois não se sabia qual seria a melhor tecnologia para computadores Silva e Martins (1996), Oliveira (2003a). De forma análoga, diversas alternativas práticas estão sendo testadas para simular os q-bits da computação quântica, dentre as quais podemos citar: pontos quânticos, ressonância

nuclear magnética em líquidos, armadilha de íons, supercondutores, dentre outros sistemas. Assim, vários protótipos de computadores quânticos, que utilizam pouco mais de uma dezena de q-bits, já foram testados com sucesso em laboratórios. Esses testes demonstraram o funcionamento dos algoritmos quânticos descobertos até agora. O grande desafio atual é o aumento do número de q-bits de forma controlada, e, certamente, as pesquisas pertinentes a esse tema se apoiarão na *nanotecnologia*. A nanotecnologia ou é o estudo de manipulação da matéria numa escala atômica e molecular. Geralmente lida com estruturas com medidas entre 1 a 100 nanômetros (1 nanômetro $= 1 \times 10^{-9}$ metro), e incluí o desenvolvimento de materiais ou componentes associados a diversas áreas (medicina, eletrônica, ciência da computação, física, química, biologia e engenharia dos materiais) de pesquisa e produção na escala nano (escala atômica).

Nesse caminho, mesmo não se sabendo qual seria a melhor tecnologia para o desenvolvimento dos computadores quânticos, já conhecemos os quatro requisitos básicos para a implementação experimental da computação quântica. Esses requisitos são os seguintes: (1) a representação dos q-bits; (2) preparação de estados iniciais de q-its; (3) medida do estado final dos q-bits e (4) a evolução unitária controlável, quanto ao aumento dos q-bits. No contexto desses requisitos básicos aparece uma dificuldade adicional na implementação dos computadores quânticos: **ler os dados durante a execução do programa sem perder todo o processamento**. Essa grande dificuldade emerge de um dos princípios da mecânica quântica que torna a computação quântica interessante, pois segundo a mecânica quântica, não é possível medir ou observar um sistema quântico sem destruir a superposição de estados. Porém, isso foi conseguido mediante a utilização de uma técnica apropriada, a qual permite a correção de erros sem comprometer o sistema. Para esse fim, a técnica utiliza a observação indireta para efetuar a correção de erros e manter a coerência do sistema.

Nesse panorama de implementação física de computadores quânticos, merece destaque o trabalho realizado em dezembro de 2001 por um grupo de cientistas do Centro de Pesquisas da IBM, localizado em Almaden, San Jose, Califórnia, USA. Eles construíram um computador quântico de 7 q-bits e o utilizaram para fatorar o número 15. Apesar da simplicidade da experiência realizada, a máquina construída pôde comprovar a viabilidade da computação quântica, onde ficou evidente que as principais dificuldades encontradas até o momento são mais tecnológicas do que teóricas. O computador quântico da IBM foi implementado através de uma molécula com 7 spins, no sentido que o núcleo da molécula era constituído por 5 átomos de **fluorina** e 2 átomos de *carbono*. Do ponto de vista funcional, a programação do computador é realizada através de pulsos de rádio-frequência e a leitura dos dados é feita mediante o uso de técnicas de ressonância magnética nuclear (RMN). A operação desse computador quântico requer temperaturas baixas, a fim de reduzir a incidência de erros. O computador quântico da IBM, por enquanto, é apenas um instrumento de pesquisa. Mesmo assim, a aplicação desse tipo de máquina na solução de problemas criptográficos já despertou o interesse no departamento de defesa de diversos países Mattielo et al. (2012).

Já existem equipamentos que manipulam informações quânticas em laboratórios de pesquisa, mas que não são capazes de lidar com mais do que alguns poucos q-bits simultaneamente. Além do evidente aumento da quantidade de armazenamento e velocidade, sob o ponto de vista quântico, outro grande desafio a ser superado é a redução das dimensões para que o equipamento possa se tornar acessível e viável economicamente. Um computador quântico que depende ainda do funcionamento do computador atual não é viável computacionalmente e comercialmente. O que é animador é a matéria prima do silício, que terá a mesma utilidade que tem nos computadores atuais. Após se estudar o comportamento de materiais muito mais caros em pequena escala, o silício demonstrou se comportar muito bem a natureza exigida para a manipulação quântica e provavelmente continuará sendo o material empregado na geração dos computadores quânticos.

15.11 Redes Neurais e Computação Quântica

Uma aplicação bastante interessante dos conceitos de computação quântica são as *redes neurais*. O modelo de redes neurais foi construído baseado no cérebro humano, pois este processa informações de uma maneira completamente diferente de um computador digital convencional. O cérebro pode ser considerado um computador altamente complexo, não-linear e paralelo, que por meio de seus constituintes chamados neurônios realiza processamentos, como percepção e controle motor, muito mais rapidamente do que qualquer moderno computador digital. Em seu livro, **Haykin** Haykin (1999) diz que o cérebro humano no momento do nascimento tem uma grande estrutura e a habilidade de desenvolver suas próprias regras através do que usualmente denominamos "experiência". Esta vai sendo acumulada com o tempo, sendo que o mais dramático desenvolvimento acontece durante os dois primeiros anos de vida e continua muito além desse estágio. Assim, um neurônio em "desenvolvimento" é sinônimo de um cérebro plástico em que a plasticidade permite que o sistema nervoso em desenvolvimento se adapte ao meio ambiente.

A *plasticidade* é fundamental para o funcionamento dos neurônios como unidades de processamento de informação do cérebro humano e também ela o é para a formação das redes neurais construídas com neurônios artificiais. Segundo **Haykin** Haykin (1999) uma rede neural, em sua forma mais geral, é uma máquina que é projetada para modelar a maneira como o cérebro realiza uma tarefa particular ou função de interesse; a rede é normalmente construída utilizando-se componentes eletrônicos ou é simulada por programação em um computador digital.

Pode-se definir uma *rede neural* como uma máquina adaptativa da seguinte forma:

"*Uma rede neural é um processador maciçamente paralelamente distribuído, constituído de unidades de processamento simples, que têm a propensão natural para armazenar conhecimento experimental e torná-lo disponível para o uso. Ela se asse-*

melha ao cérebro em dois aspectos: O conhecimento é adquirido pela rede a partir de seu ambiente através de um processo de aprendizagem. Forças de conexão entre neurônios, conhecidos como pesos sinápticos, são utilizados para armazenar o conhecimento adquirido."

O procedimento voltado ao processo de treinamento de aprendizado da rede é chamado de algoritmo de aprendizagem, cuja função é modificar os pesos sinápticos da rede de uma forma ordenada para alcançar um objetivo do projeto desejado. É possível também para uma rede neural modificar sua própria topologia, o que é motivado pelo fato dos neurônios do cérebro humano poderem morrer e que as novas conexões sinápticas possam crescer.

Benefícios de uma rede neural

Dois benefícios se destacam quando se fala em utilizar o método de redes neurais. (1) O primeiro diz respeito a sua estrutura maciçamente paralela e distribuída, e o segundo (2) é a sua habilidade de aprender e, portanto, generalizar. A generalização se refere ao fato de a rede neural procurar saídas adequadas para entradas que não estavam presentes durante o treinamento (aprendizagem). Estas duas capacidades de processamento de informação possibilitam às redes neurais resolverem problemas complexos e muitas vezes intratáveis.

Embora o método de redes neurais possa resolver uma gama considerável de problemas, como em Andrade et al. (2005), existem limitações que impedem a resolução acurada de muitos desses problemas. Esses autores citam como exemplo o tempo de treinamento das redes neurais, que, dependendo do algoritmo de aprendizagem (Backpropagation, na maioria dos casos) tende a ser muito longo. Em alguns casos são necessários milhares de ciclos para se chegar a níveis de erros aceitáveis, principalmente se o algoritmo estiver sendo simulado em computadores que realizem operações de forma sequencial, já que o processador deve calcular as funções para cada unidade e suas conexões separadamente, o que pode ser problemático em redes muito grandes, ou com grande quantidade de dados.

A natureza multidisciplinar e as limitações de Redes Neurais Artificiais associadas à hipótese de que as *sinapses* - conexões - entre neurônios poderiam ser tratadas por fenômenos quânticos Penrose (1994), motivaram o interesse de pesquisas em Redes Neurais que incorporassem conceitos de Física Quântica. Assim, o estudo de Redes Neurais Quânticas mostrou-se um ramo bastante inovador no campo da Computação Quântica embora seja uma área ainda incipiente (Altaisky (2004), Gupta e Zia (2001)). O trabalho realizado por em Andrade et al. (2005) traz em seu bojo uma proposta de um modelo de neurônio quântico e descreve o seu funcionamento. O artigo também faz uma descrição dos princípios da computação quântica e circuitos quânticos. Segundo os autores o modelo proposto foi simulado utilizando o simulador de circuitos quânticos Zeno Cabral, Lula e Lima (2005) desenvolvido na *Universidade Federal de Campina Grande*. Em Herbster et al. (2004) no seu trabalho: *O Estado da*

Arte em Redes Neurais Artificiais Quânticas relata que as Redes Neurais Quânticas surgiram como uma nova abordagem no campo da Computação Quântica, possuindo propriedades que permitem resolver os paradigmas encontrados nas Redes Neurais Clássicas.

Na literatura é possível encontrar trabalhos Shor (1997), Deutsch (1989), Feynman (1982), Nielsen e Chuang (2000) que evidenciam a importância da utilização de conceitos quânticos para mitigar ou até mesmo eliminar problemas inerentes aos métodos de computação clássica, uma vez que a computação quântica tem como principais características o processamento e a transmissão de dados armazenados em estados quânticos de uma forma muito mais eficiente que os modelos de computação convencionais. Esses trabalhos fortalecem a ideia de que o advento da Física Quântica promoveu uma verdadeira revolução em todos os campos da tecnologia e em particular na computação.

15.12 Perspectivas da Computação Quântica

Este capítulo uma revisão bibliográfica básica sobre a computação quântica é apresentada. Esta promissora área da ciência propõe a fusão entre as idéias da **mecânica quântica** e da *ciência da computação*. Apesar da incipiência de projetos para a construção de computadores quânticos, muitos desenvolvimentos mostraram-se possíveis e aplicáveis, imaginando-se recursos computacionais como na computação clássica. Percebemos também que a adoção do paradigma quântico na computação trata-se de um trajeto natural, pois caminha concomitante com a diminuição dos dispositivos eletrônicos presentes no computador, como já previa a *Lei de Moore*.

Segundo (MATTIELO et al., 2012), a computação clássica não se esgotará, mas o que o futuro nos reserva é trata-se de um novo *paradigma de computação*, que pode ter profundas consequências, não só para a *tecnologia*, mas também para a *teoria da informação*, para a *Ciência da Computação*, e para a *ciência* em geral. Imaginamos que da mesma forma que a computação iniciada no século passado trouxe inúmeras aplicações que contribuíram para o desenvolvimento da humanidade nas mais variadas áreas, a computação quântica também propiciará aplicações que alcancem a melhor qualidade de vida das pessoas. É esperado que cada vez mais pessoas conheçam esse novo campo da ciência e, que alguns mais aptos, desenvolvam aptidão por pesquisar sobre esse amplo tema.

Entretanto da mesma maneira que a superposição de estados permite a criação do computador quântico é essa mesma propriedade que tornou difícil, até o momento, a criação deles. A superposição é muito sensível a qualquer microruído eletromagnético que pode alterar o estado do q-bit, fazendo com que a informação que ele contenha ainda seja perdida. Outro fato em questão é o superaquecimento dessas máquinas.

15.13 Bibliografia e Fonte de Consulta

Lei de Moore - https://pt.wikipedia.org/wiki/Lei_de_Moore#Segunda_Lei_de_Moore

Computação Quântica - https://pt.wikipedia.org/wiki/Computação_quãntica

O que é o q-bit - http://www.tecmundo.com.br/computação-quântica/2627-o-que-e-qubit-o-bit-quantico-.htm

Pires, A. S. T. - Evolução das Ideias da Fisica. São Paulo: Editora Livraria da Fisica, (2008).

Eisberg, R. and Resnick, R. - Fisica Quãntica. Rio de Janeiro: Editora Campus, (1979).

Cohen-Tannoudji, C. and Diu, B. and Laloe, F. - Quantum Mechanics. vol. 1, NY: Wiley Interscience, (1992).

Landau, L. D. and Lifshitz, E. M. - Quantum Mechanics. 3rd. Edition, Oxford: Pergamon Press, (1976).

Feynman, R. P., et al. - The Feynman Lectures on Physics. Vol.3, NY: Addison-Wesley, (1982).

Piza, A. F. R. T. - Mecânica Quântica. São Paulo: EDUSP, (2003).

Souza, A. M. C.Topicos de Fisica Contemporãnea. Sergipe, (2002).

Fazzio, A., et al. Fisica para um Brasil competitivo: Estudo encomendado pela CAPES visando maior inclusão da Fisica na vida do pais. Brasília: Sociedade Brasileira de Fisica, (2007).

Nielsen, M. A.; Chuang, I. L. Computacão Quântica e Informação Quântica. Porto Alegre, Bookman, 2005.

Haykin, S., Neural Networks: A Comprehensive Foundation. 2a ed., New Jersey: Prentice Hall, 1999.

Silva, C.; Martins, R. Revista Brasileira de Ensino de Fisica, 18, n.4, (1996), 313.

Oliveira, I. S., et al, Ciência Hoje, vol. 33, n. 193, (2003), 22.

Andrade, W. L., Araújo B. C., Gomes, H. M., Fechine, J. M. Proposta de um Neurônio Quântico, Congresso Brasileiro de Redes Neurais, 2005, Natal, RN, Anais do CBRN 2005, 2005 p. 1-5.

GRIFFITHS, D. J. Introduction to Quantum Mechanics, Prentice Hall; (1995).

Galvão, E. - O que é Computação Quantica?, Rio de Janeiro: Vieira & Lent, 2007.

Penrose, R. Shadows of the Mind. Vintage Science, 1994.

Gupta, S., Zia, R. K. P. Quantum Neural Networks. Journal of Computer and System Sciences, vol.63, pages 355-383, 2001.

Cálculo das Variacões - https://pt.wikipedia.org/wiki/Calculo_de_variações

Espaco de Hilbert - https://pt.wikipedia.org/wiki/Espaço_de_Hilbert

15.14 Referências e Leitura Recomendada

Altaisky, M. V. Quantum neural network. Joint Institute for Nuclear Research, Russia. Technical report, Available online at the Quantum Physics repository: http://arxiv.org/PS_cache/quantph/pdf/0107012.pdf, las accessed 31/5/2014.

Cabral, G. E., Lula, B., Lima, A. F. Zeno: a New Graphical Tool for Design and Simulation of Quantum Circuits, Proceedings of SPIE Quantum Information and Computation III - Defense and Security Symposium, Orlando, 2005.

Herbster, R. F., Andrade, W. L., Gomes, H. M., Machado, P. D. L. O Estado da Arte em Redes Neurais Artificiais Quânticas. Revista de Iniciação Científica da SBC, ano IV, Número IV, 2004.

Shor, P. W. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM Journal on Computing, 26(5): 1484-1509, October, 1997.

Deutsch, D. - Quantum Computational Networks, Proceedings of the Royal Society of London, A 425, pages 73-90, 1989.

Rieffel, E. and Wolfgang, P. - An Introduction to Quantum Computing for Non-Physicists. ACM Computing Surveys, Vol. 32, No. 3, September 2000, pp. 300-335.

Gershenfeld, N. and West, J. - The Quantum Computer. Scientific American, 2000.

Sajurai, J. J. Modern Quantum Mechanics: Revised Edition. NY: Addison-Wesley, (1994)

Freire, O. J.; Pessoa, O. J.; Bromberg, J. L. Teoria Quântica: estudos históricos e

implicações culturais. São Paulo: Livraria da Física, (2010).

Bouwmeester, D. and Ekert, A. and Zeilinger, A. - The Physics of Quantum Information (Springer Verlag, 2001). Bouwmeester, A. Ekert and A. Zeilinger - The Physics of Quantum Information (Springer Verlag, 2001).

Williams, C.P. and Clearwater, S.H. - Explorations in Quantum Computing (Springer Verlag, 1998).

Erwin Schrodinger - https://pt.wikipedia.org/wiki/Erwin_Schrrdinger, acessado em 07 de Dezembro de 2015.

https://pt.wikipedia.org/wiki/Computação_quântica, acessado em 07 de Dezembro de 2015.

Formulação Matemática da Mecânica Quântica -
J. von Neumann, Mathematical Foundations of Quantum Mechanics, Princeton University Press, 1955. Reprinted in paperback form.

Mathematical Formulations of Quantum Mechanics - ⟨www.iue.tuwien.ac.at⟩

Edwards, D, - The Mathematical Foundations of Quantum Mechanics, Synthese, 42 (1979),pp. 170.

Gleason, A. - Measures on the Closed Subspaces of a Hilbert Space, Journal of Mathematics and Mechanics, 1957.

Mackey, G. - Mathematical Foundations of Quantum Mechanics, W. A. Benjamin, 1963 (paperback reprint by Dover 2004).

Teschl, G. - Mathematical Methods in Quantum Mechanics with Applications to Schrödinger Operators, American Mathematical Society, 2009, em www.mat.univie.ac.at

Weaver, N. - "Mathematical Quantization", Chapman & Hall/CRC 2001.

Weyl, H. - The Theory of Groups and Quantum Mechanics, Dover Publications, 1950.

Referências

ALTAISKY, M. V. *Quantum neural network*. Russia, 2004. Technical report, Available online at the Quantum Physics repository:. Disponível em: ⟨http://arxiv.org/PS_cache/quantph/pdf/0107012.pdf,Accessed31/5/2004⟩. Citado na página 275.

AMERICAN DOCUMENTATION. *Information science: what is it?* 1968. Citado na página 100.

ANDRADE, W. L. et al. Proposta de um neurônio quântico. *Congresso Brasileiro de Redes Neurais*, 2005. Citado na página 275.

ARAúJO, C. A. Ávila. Correntes teóricas da ciência da informação. *Ci. Inf., Brasília, DF, v. 38, n. 3, p.192-204, set./dez., 2009*, v. 38, n. n. 3, p. p. 192–204, setembro-dezembro 2009. Citado na página 100.

AVALIAçãO da Qualidade da Agua utilizando a Teoria Fuzzy. 2012. Disponível em: ⟨http://www.scientiaplena.org.br/sp/article/view/1024⟩. Citado na página 221.

BARENDEGT, H. The lambda calculus: its syntax and semantics. In: *Studies in Logic and the Foundations of Mathematics*. Amsterdam: North-Holland,, 1994. Citado na página 237.

BARENDREGT, H. The impact of the lambda calculus in logic and computer science. *The Bulletin of Symbolic Logic*, Vol.3, n. (2), p. 181–215, 1997. Citado na página 39.

BARROCA, J. A. M. L. Formal methods: Use and relevance for the development of safety-critical systems. *The Computer Journal*, 1992. Citado na página 195.

BEN-ARI, M. *Principles of Concurrent and Distributed Programming*. 2nd editon. ed. [S.l.]: Addison-Wesley, 2014. ISBN-13: 978-0321312839. Citado na página 244.

BENEVIDES, M. *Apostila de Logica*. 2015. Citado na página 227.

BIAZIN, A. *Um Modelo de Integração de Lógica Fuzzy à Bancos de Dados Convencionais*. Dissertação (dissertação de mestrado) — PPGCC-UFSC, 2002. Citado na página 219.

BOLTER, J. D. *Turings man, western culture in the computer age*. [S.l.]: Universidade da Carolina do Norte, 1984. Citado na página 10.

BRASIL, R. P. M. Sobre o número "e". *Revista do Professor de Matematica*, 2007. Citado na página 12.

BRIDGES, D. *Constructive Mathematics*. 2013. The Stanford Encyclopedia of Philosophy(Spring 2013 Edition), Edward N. Zalta(ed.),. Disponível em: ⟨http://plato.stanford.edu/archives/spr2013/entries/mathematics-constructive⟩. Citado 2 vezes nas páginas 233 e 234.

BYRON, F. W.; FULLER, R. W. *Mathematics of classical and quantum physics*. [S.l.]: Courier Dover Publications, 1992. Citado na página 265.

CABRAL, G. E.; LULA, B.; LIMA, A. F. Z. A new graphical tool for design and simulation of quantum circuits. *Proceedings of SPIE Quantum Information and Computation III Defense and Security Symposium,*, 2005. Orlando. Citado na página 275.

CAMPIOLO, R. *Um Metamodelo para Ambientes de Computacao Ubiqua*. Dissertação (Mestrado) — PPGCC-UFSC, 2005. Citado na página 246.

CAMPIOLO, R.; CREMER, V.; SOBRAL, J. B. M. O. M. f. P. C. E. On modeling for pervasive computing enviroments. *ACM MSWiM*, 2007. Citado na página 246.

CAPURRO, R. Foundations of information science: review and perspectives. *In: INTERNATIONAL CONFERENCE ON CONCEPTIONS OF LIBRARY AND INFORMATION SCIENCE*, 1991. Proceedings Tampere: University of Tampere, 1991. Available: ¡http://www.capurro.de/tampere91.htm¿. Acessado em 14 de Agosto de 2015. Citado na página 99.

CAPURRO, R. C. B. H. /www.capurro.de/infoconcept.html¿. Access: 15 May 2004 (A versão online não é idêntica à versRafael. The concept of information. *Annual Review of Information Science and Technology*, v. 37, p. p. 343–411, 2003. Em www.capurro.de/infoconcept.html. Access: 14 Agosto de 2015. Citado 2 vezes nas páginas 99 e 100.

CARDI, M. d. L.; BARRETO, J. M. Primordios da computacao no brasil. *CLEI 2002*, 2002. Citado na página 135.

CARNIELLI, R. l. E. W. A. *Computabillidade, Fiuncoes Computaveis,Logicae os Fundamentos da Matematica*. [S.l.]: Editora UNESP, 2009. Citado na página 83.

CARNIELLI, W.; EPSTEIN, R. *Computabilidade, Funcoes Computaveis, Logica e os Fundamentos da Matematica*. [S.l.]: Editora FAPESP, 2005. Citado na página 57.

CHURCH, A. A set of postulates for the foundation of logic. *Annals of Mathematics*, v. 33, n. Series 2, p. 346366, 1932. Citado na página 41.

CHURCH, A. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, Vol 58, n. 2, p. pp. 345–363, April 1936. Citado na página 41.

CHURCH, A. A formulation of the simple theory of types. *Journal of Symbolic Logic*, Vol 5, 1940. Citado na página 41.

CHURCH, A. *Introduction to Mathematical Logic.* 10th ed.. ed. Princeton, New Jersey, USA: Princeton University Press, 1996. ISBN 978-0-691-02906-1. Citado na página 228.

COHEN-TANNOUDJI, C.; DIU, B.; LALOE, F. *Quantum Mechanics.* [S.l.]: Wiley Interscience, 1992. Citado na página 263.

COLEMAN, R. *Stochastic Processes.* [S.l.]: Springer-Verlag, 1974. Citado na página 4.

DAVIS, M. Why godel didnt have churchs thesis. *Information and Control*, n. 54, p. 3–24, 1982. Citado na página 41.

DEITEL, H. M.; DEITEL P, J. *Java - Como Programar.* [S.l.]: Pearson Education, 2005. Citado na página 244.

DEO, N. *Graph Theory.* [S.l.]: Prentice Hall, 1974. (Series in Automatic Computation). Citado 5 vezes nas páginas 187, 188, 189, 190 e 191.

DERRICK JOHN; BOITEN, E. A. *Refinement in Z and Object-Z.* (2nd ed.). [S.l.]: Springer, 2014. ISBN 978-1-4471-5355-9. Citado na página 246.

DEUTSCH, D. Quantum computational networks,. *Proceedings of the Royal Society of London*, 1989. Citado na página 276.

DIVERIO, T.; MENEZES, P. B. *Teoria da Computacao: Maquinas Universais e Computabilidade.* 3. ed.. ed. [S.l.]: Editora Bookman, 2011. Vol. 5. (Série Livros Didáticos Informática UFRGS, Vol. 5). Citado na página 3.

DONG, J.; DUKE R; HAO, P. Integrating object-z with timed automata. *Engineering of Complex Computer Systems*, p. 488497, 2005. Citado na página 245.

DUKE, G. R. R. *Formal Object Oriented Specification Using Object-Z.* [S.l.]: Palgrave Macmillan, 2000. (Cornerstones of Computing). ISBN 9780333801239. Citado na página 195.

DUKE, R.; ROSE, G. *Formal Object-Oriented Specification Using Object-Z.* [S.l.]: McMillan, 2000. Citado na página 245.

EISBERG, R.; RESNICK, R. *Física Quântica.* [S.l.]: Editora Campus, 1979. Citado na página 263.

EPSTEIN, W. A. C. R. L. *Computability: Computable Functions, Logic and The Foudations of Mathematics.* [S.l.]: Wadsworth & Brooks/Cole, 1989. (Mathematics Series). Citado na página 69.

FAZZIO, A. e. a. *Física para um Brasil Competitivo.* Brasilia, 2007. Estudo encomendado pela Capes visando maior inclusão da Física na vida do país. Citado 2 vezes nas páginas 260 e 261.

FEYNMAN, R. P. Simulating physics with computers. *Int. Journal inTheorectical. Physical*, v. 21, n. 467-488, 1982. Citado na página 276.

GALLAGER, R. G. *Stochastic Processes*. [S.l.]: Cambridge University Press, 2014. Citado na página 4.

GALVAO, E. F. *O que e Computacao Quantica*. [S.l.]: Vieira & Lent Casa Editorial Ltda, 2007. Citado na página 9.

GODEL, K. Uber formal unentscheidbare s¨atze der ¨ principia mathematica und verwandter systeme i. *Monatshefte f¨ur Mathematik und Physik*, Vol. 38, p. 173–198, 1931. Reprinted and translated in (Godel, 1986, 144195). Citado na página 39.

GODEL, K. *Collected Works*. Oxford: Oxford University Press, 2003. vol. 4. Citado na página 39.

GOEBEL., D. P. A. M. R. *Computational Intelligence: A Logical Approach*. [S.l.]: Oxford University Press, 1998. Citado na página 218.

GOGUEN, J. A.; THATCHER, J. W.; WAGNER, K. G. *An Initial Algebra Approach to the Specification Correctness and Implementation of Abstract Data Typs*. [S.l.]: Prentice-Hall, 1978. IV, p.80-149. Citado na página 239.

GOLDBALTT, R. *Applications the Logic of Computer Programming*. [S.l.]: Springer-Verlag, 1982. Citado na página 196.

GRIFFITHS, D. J. *Introduction to Quantum Mechanics*. [S.l.: s.n.], 1995. Citado na página 261.

GUIMARãES, M. G. *Um sistema de apoio à Dosimetria da Pena utilizando Fuzzy Logic*. Dissertação (dissertação de mestrado) — PPGCC-UFSC, 2004. Citado na página 220.

GUPTA, S.; ZIA, R. K. P. Quantum neural networks. *Journal of Computer and System Sciences*, v. 63, p. 355–383, 2001. Citado na página 275.

GUTTAG, J. V. Abstract data typtypes the development of data structures. *CACM*, v. 20, n. 6, p. 396–404, 1977. Citado na página 239.

GUTTAG, J. V.; HOROWITZ, E.; MUSSER, D. P. The design of data type specification. *Current Trends in Programming Methodology*, IV, 1978. Prentice-Hall. Citado na página 239.

GUTTAG J. V., J. V.; HORNING, J. J. The algebraic specification of abstract data type. *Acta Informatica*, v. 10, n. 1, p. 27–52, 1978. Citado na página 239.

HAYKIN, S. *Neural Networks: A Comprehensive Foundation*. [S.l.]: Prentice Hall, New Jersey, 2 ed., 1999. Citado na página 274.

HERBSTER, R. F. et al. O estado da arte em redes neurais artificiais quânticas. *Revista de Iniciação Científica da SBC*, Ano IV, n. IV, 2004. Citado na página 275.

HERNANDEZ, A. S. *LA LÓGICA DIFUSA. CARACTERÍSTICAS Y APLICACIONS*. Ciudad de la Habana. Cuba. Citado na página 220.

HILBERT, D.; BERNAYS, P. *Grundlagen der Mathematik*. Berlin: Springer, 1939. Citado na página 39.

HINDLEY, C. e. *History of Lambda-calculus and Combinatory Logic*. [S.l.: s.n.], 2006. Citado na página 41.

HOARE, C. A. R. *Communicating Sequencial Process*. [S.l.]: Prentice Hall, 1985. (Series Editor). Citado na página 195.

INTELIGENTES, G. de S. *Lógica Borrosa y Aplicaciones. Aplicación en robótica*. 2005. Disponível em: ⟨http://www.puntolog.com/actual/articulos/uni_santiago6.htm⟩. Citado na página 220.

JANOS, M. *Matemática e Natureza*. [S.l.]: Editora Livraria da Fisica, 2009. ISBN 978-85-7861-038-8. Citado 2 vezes nas páginas 177 e 178.

LAM HK; LING, S. N. *Computational Intelligence and its applications: Evolutionary Computation, Fuzzy Logic, Neural Network, and Support Vector Machine Technique*. [S.l.]: Imperial College Press, 2012. Citado na página 218.

LISKOV, B. H.; ZILLES, S. N. Specification thecniques for data abstration. *IEEE Transactions on Software Engineering*, SE-1, n. 1, p. 7–19, 1975. Citado na página 239.

MACIEL, P. R. M.; LINS, R. D.; CUNHA, P. R. F. *Introdução às Redes de Petri*. [S.l.]: SBC, X Escola de Computação, 1996. Citado na página 206.

MAHONY B; DONG, J. S. T. C. O. Z. I. T. o. S. E. . . . d. Timed communicating object z. *IEEE Transactions on Software Engineering*, Vol 26, n. (2), p. 150177, February 2000. Citado na página 245.

MAOR, E. *e: A Historia de um Numero*. 1. ed. Rio de Janeiro: Editorial Record, 2003. Tradução de Jorge Calife. Citado na página 12.

MATHEUS, R. F. Rafael capurro e a filosofia da informação: abordagens, conceitos e metodologias de pesquisa para a ciência da informação. *Perspectivas na Ciência da Informacao*, v. 10, n. 2, p. p.140–165, julho-dezembro 2005. Belo Horizonte. Citado na página 99.

MATIJACEVIC, Y. V. Enumerable sets are diophantine. *Doklady Akademii Nauk SSSR*, v.191, n. n.2, p. 279–282, 1970. In Russian. Citado na página 53.

MATTELART, M. M. A. *Historia das Teorias da Comunicacao*. Oitava edicao 2005. [S.l.]: Edicoes Loyola, 1999. ISBN 85-15-01770-9. Citado na página 100.

MATTIELO, F. et al. Decifrando a computacao quantica. *Caderno de Fisica da UEFS*, v. 10, n. 1 e 2, p. 31–44, 2012. Citado 5 vezes nas páginas 257, 263, 272, 273 e 276.

MEYER, B. *Object-Oriented Software Construction*. [S.l.]: Prentice-Hall International, 1988. (Series Editor). Citado 3 vezes nas páginas 240, 241 e 243.

MILNER, R. *Communication and Concorrency*. [S.l.]: Prentice Hall, 1989. (Series Editor). Citado 3 vezes nas páginas 192, 193 e 195.

NIELSEN, M. A.; CHUANG, I. L. *Quantum Computation and Quantum Information*. [S.l.]: Cambridge University Press, 2000. Citado 2 vezes nas páginas 272 e 276.

OLIVEIRA, C. R. *Intoducao a Analise Funcional*. [S.l.]: IMPA, OCLC 49254749, 2001. Citado na página 265.

OLIVEIRA, I. e. a. Ciência hoje,. *Ciência Hoje,*, v. 33, n. 193, p. 22, 2003. Citado na página 272.

OLIVEIRA, W. T. R. *Utilizando Integrais Fuzzy em Tomada de Decisão Multicritérios*. Dissertação (dissertação de mestrado) — PPGCC-UFSC, http://repositorio.ufsc.br/xmlui/handle/123456789/84549, 2003. , Orientado por Paulo Sergio da Silva Borges. Citado na página 222.

OXFORD UNIVERSITY COMPUTING LABORATORY. *Proposal: Community Z Tools Project (CZT)*. 2001. Web site. Disponível em: ⟨http://www.cs.ox.ac.uk/people/andrew.martin/CZT/proposal.html⟩. Citado na página 246.

PENROSE, R. *Shadows of the Mind*. [S.l.]: Vintage Science, 1994. Citado na página 275.

PETERSON, J. L. *Petri Net Theory and the modeling of Systems*. [S.l.]: Prentice-Hall, 1981. Citado 13 vezes nas páginas 196, 197, 198, 199, 200, 201, 202, 204, 205, 206, 207, 208 e 209.

PIERCE, B. S. *Types and Programming Languages*. [S.l.]: MIT Press, 2002. Citado na página 234.

PIMENTEL, E. G. *Fundamentos de Matematica*. [S.l.], 2008. Citado na página 234.

PIRES, A. S. T. *Evolução das Ideias da Física*. [S.l.]: Editora Livraria da Fisica, 2008. Citado na página 261.

PRADO, l. A. C. *Painel Solar Auto-Orientável baseado na Lógica Paraconsistente Anotada Evidencial Eτ*. Dissertação (Dissertação de Mestrado) — UNIVERSIDADE PAULISTA - UNIP, 2013. Citado na página 225.

RONCHI, D. R. S.; L., P. The parametric lambda-calculus: a metamodel for computation,. In: *Computer Science-Monograph*. [S.l.]: Springer Verlag, 2004. Citado na página 237.

SCHECHTER, L. M. *A Vida e o Legado de Turing*. 2015. Disponível em: ⟨http://www.dcc.ufrj.br/~luisms/turing/Seminarios.pdf⟩. Citado na página 56.

SELDIN, J. P. *The logic of Curry and Church*. Amsterdam: Elsevier, 2006. vol. 5. (In Handbook of the History of Logic, vol. 5). Forthcoming. Citado na página 39.

SETTE, J. S. A. *Tipos de Dados - Uma Visao Algebrica*. Rio de Janeiro, 1988. Citado na página 239.

SHOR, P. W. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, v. 26, n. 5, p. 1484–1509, October 1997. Citado na página 276.

SIEG, W. Step by recursive step: Churchs analysis of effective computability. *Bulletin of Symbolic Logic*, n. 3, p. 154–180, 1997. Citado na página 41.

SILVA, C.; MARTINS, R. Revista brasileira de ensino de física. *Revista Brasileira de Ensino de Física*, v. 18, n. 4, p. 313, 1996. Citado na página 272.

SIPSER, M. *Introducao a Teoria da Computacao*. [S.l.]: Cengage Learning Editores, 2011. Citado 9 vezes nas páginas 4, 50, 53, 70, 169, 170, 191, 194 e 195.

SMITH, G. *The Object-Z Specification Language*. [S.l.]: Springer, 2000. ISBN 978-1-4615-5265-9. Citado 2 vezes nas páginas 195 e 245.

SOBRAL, J. B. M. *Especificacao Formal de Sistemas Distribuidos em Alto Nivel de Abstracao*. Tese (Doutorado) — EEL-COPPE-UFRJ, 1996. Citado na página 195.

SOUSA, J. N. de Paula e. *Sistemas de Controle de Tráfego Metropolitano em Rodovias Dotadas de Faixas Exclusivas para Ônibus*. Dissertação (Mestrado) — Porgrama de Pós-Gradiação em Sostemas e Computação, COPPE, UFRJ, 2005. Citado na página 219.

SPIVEY, J. M. *The Z Notation*. [S.l.]: Prentice-Hall, 1989. Citado 2 vezes nas páginas 195 e 244.

SUDKAMP, T. A. *Languages and Machines*. [S.l.]: Addison Wesley, 1988. Citado 14 vezes nas páginas 4, 54, 66, 69, 82, 83, 84, 85, 86, 170, 189, 190, 193 e 194.

TANEMBAUM, A. S. *Computer Networks*. [S.l.]: Pearson Education, 2010. Citado na página 161.

TERADA, R. *Seguranca de Dados*. [S.l.]: Editora Blucher, 2008. Citado na página 271.

TORRES, D. F. M. *O Computador Matematico de Post*. [S.l.], 2000. Citado na página 77.

VASCONCELOS. *O Tempo como Modelo*. Dissertação (Mestrado) — Program de P[os-Graduação em Sistemas de Computação - COPPE - UFRJ, 1989. Citado 3 vezes nas páginas 195, 214 e 227.

VELOSO, P. A. S. *Estruturação e Verificação de Porgramas com Tipos de Dados*. [S.l.]: Editora Edgar Blücher, 1987. Citado na página 239.

ZACH, R. *Kurt Gödel and Computability Theory*. 2006. Research supported by the Social Sciences and Humanities Research Council of Canada. Acessado em 12 de Outubro de 2015. Disponível em: ⟨http://people.ucalgary.ca/~rzach/static/cie-zach.pdf⟩. Citado 3 vezes nas páginas 3, 38 e 172.

Índice

M

Y

Este texto foi composto em Minion Pro, de Robert Slimbach, e Myriad Pro, de Robert Slimbach e Carol Twombly.

Este texto foi composto em fontes EBGaramond
(⟨http://www.ctan.org/tex-archive/fonts/ebgaramond⟩).

SÉRIE PENSAMENTO MATEMÁTICO @ CIÊNCIA COMPUTAÇÃO

Volume II: Da Computabilidade Formal às Máquinas Programáveis

A partir das máquinas calculadoras mecânicas não programáveis do século XIX e das primeiras ideias de avaliar funções no início dos anos 30, no século XX, surge a ideia da computabilidade formal para máquinas programáveis. Das ideias das funções recursivas por Herbrand e Gödel, passando por Alonzo Church e seus famosos estudantes Kleene e Turing, mesmo sem computador, constituíram-se importantes contribuições para as teorias dos algoritmos e computabilidade, conduzindo para fundamentar a Ciência da Computação. A estrada da Lógica, iniciada em Aristóteles, sistematizada por Leibniz, passando por Boole e Frege, chegou a Church, Kleene, Rosser, Turing, Post e, posteriormente a Shannon, John von Neumann e a engenharia de Konrard Zuse e outros, fazendo surgir o computador digital programável. Da ideia da computabilidade passou-se à complexidade computacional, e temos hoje, o desenvolvimento fantástico da Ciência da Computação.

Este livro pretende servir de material de apoio às disciplinas da ciência da computação teórica, em cursos de graduação, e seu texto, seguindo o mais que possível a ordem cronológica do surgimento das grandes ideias, tenta mostrar ao estudante, do ponto de vista histórico, como determinadas ideias geniais, serviram para a construção da máquina programável que hoje é o computador digital. O material aqui exposto, faz a parte da organização do segundo volume da série Pensamento Matemático @ Ciência da Computação, baseando-se na exposição das ideias dos grandes personagens, os verdadeiros donos das ideias, as vezes narrando fatos e suas ideias geniais. É uma tentativa de proporcionar este conhecimento histórico sobre as raízes da Ciência da Computação, proporcionando uma visão que outros livros, que tratam de temas específicos, não contemplam do ponto de vista histórico e conceitual.

Sobre o autor:

João Bosco M. Sobral é Bacharel em Matemática pelo Instituto de Matemática da UFRJ em 1973, M.Sc. pelo Programa de Sistemas e Computação da COPPE-UFRJ em 1977, e realizou seu doutorado no Programa de Engenharia Elétrica da COPPE-UFRJ em 1996. Como docente durante quase quatro décadas na ciência da computação da UFSC, ao participar nas disciplinas em cursos de graduação e mestrado em computação, teve a oportunidade de entender e vivenciar o elo existente entre a Matemática, a Lógica, e a Ciência da Computação. Agora, tenta disseminar o que aprendeu sobre o elo fascinante destas ciências, no sentido de motivar o leitor a ficar conectado com o mundo da Matemática e da Lógica, como ciências construtoras da Ciência da Computação passada e futura.

Agência Brasileira do ISBN
ISBN 978-85-902995-3-0
9 788590 299530