NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTAMENTO DE INFORMÁTICA

BERNARDO RAFAEL PEREIRA DE SOUSA
Licenciado em Engenharia Informática

Análise de sintaxe LR em OCaml-FLAT/OFLAT

MESTRADO DE ENGENHARIA INFORMÁTICA

Universidade NOVA de Lisboa
Setembro, 2022

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTAMENTO
DE INFORMÁTICA

Análise de sintaxe LR em OCaml-FLAT/OFLAT

BERNARDO RAFAEL PEREIRA DE SOUSA

Licenciado em Engenharia Informática

Orientador: Artur Miguel Dias,
Professor Auxiliar, DI-FCT, Universidade NOVA de Lisboa

Júri:

Presidente: Ricardo João Rodrigues Gonçalves,
Professor Auxiliar, Faculdade de Ciências e Tecnologia da Universidade NOVA de Lisboa

Arguente: Andreia Mordido,
Professor Auxiliar, Faculdade de Ciências da Universidade de Lisboa

Orientador: Artur Miguel Dias,
Professor Auxiliar, Faculdade de Ciências e Tecnologia da Universidade NOVA de Lisboa

MESTRADO EM ENGENHARIA INFORMÁTICA

Universidade NOVA de Lisboa
Setembro, 2022

Análise de sintaxe LR em OCaml-FLAT/OFLAT

Copyright © Bernardo Rafael Pereira de Sousa, Faculdade de Ciências e Tecnologia, Universidade NOVA de Lisboa.

A Faculdade de Ciências e Tecnologia e a Universidade NOVA de Lisboa têm o direito, perpétuo e sem limites geográficos, de arquivar e publicar esta dissertação através de exemplares impressos reproduzidos em papel ou de forma digital, ou por qualquer outro meio conhecido ou que venha a ser inventado, e de a divulgar através de repositórios científicos e de admitir a sua cópia e distribuição com objetivos educacionais ou de investigação, não comerciais, desde que seja dado crédito ao autor e editor.

AGRADECIMENTOS

Gostaria de agradecer ao orientador desta dissertação, Professor Artur Miguel Dias, pela paciência e apoio durante a orientação desta dissertação.

Também gostaria de agradecer ao Professor Ravara, e ao Eduardo Silva, que demonstraram interesse no projeto e ofereceram o seu contributo.

Gostaria de agradecer aos meus amigos que me apoiaram durante esta fase e durante a minha jornada na FCT-UNL.

Finalmente, gostaria de agradecer à minha família, aos meus pais, aos meus avós e à minha irmã pelo apoio que me dão dia a dia. Obrigado por tudo.

RESUMO

Os conceitos de FLAT (Linguagens Formais e Teoria de Autómatos) envolvem uma natureza formal e rigorosa, mas também algo complexa, assim trazendo um nível de exigência considerável durante a sua aprendizagem. Para mitigar este problema, têm sido criadas ferramentas pedagógicas ao longo das décadas, com objetivo de promover a assimilação dos conceitos a partir de aplicações que permitem ao utilizador visualizar e interagir por meio de exercícios/exemplos.

Na FCT-UNL, foram desenvolvidas duas dessas ferramentas pedagógicas: a biblioteca OCaml-FLAT, com suporte para os conceitos FLAT, e a aplicação gráfica OFLAT, desenhada para web browsers com o objetivo de promover a aprendizagem do utilizador, com uso de visualizações interativas dos conceitos da biblioteca OCaml-FLAT.

Esta dissertação tem como primeiro objetivo estender a biblioteca OCaml-FLAT com conceitos de parsing (análise sintática) LR. O segundo objetivo é estender a aplicação OFLAT, adicionando suporte para visualização e interatividade dos referidos conceitos. O caráter pedagógico do software a desenvolver será uma preocupação constante.

Inevitavelmente, nesta dissertação irá ocupar bastante espaço, a explicação dos conceitos teóricos que estão subjacentes ao trabalho a realizar, concretamente tudo o que tem a ver com parsing bottom-up determinista LR, nas diferentes variantes clássicas: LR(0), SLR(1), LR(1) e LALR(1).

Palavras chave: Teoria de Linguagens Formais e Autómatos, OCaml, OFLAT, OCaml-FLAT, Programação Funcional, Gramáticas formais, Gramáticas LR, Ferramentas Pedagógicas.

Abstract

The concepts of FLAT (Formal Languages and Automata Theory) have a formal and rigorous nature, also somewhat complex, thus carrying a notable demand during their learning. To mitigate this problem, various tools have been created over the decades, allowing the user to visualize and interact with the concepts through exercises/examples.

At FCT-UNL, two such pedagogical tools were developed: the OCaml-FLAT library, with logical support for the FLAT concepts, and the OFLAT graphical application, which runs on a web browser and offers interactive visualizations for the FLAT concepts.

The first objective of this dissertation is to extend the OCaml-FLAT library with support for LR parsing concepts. The second objective is to extend the OFLAT graphical application, adding support for visualization and interactivity of said concepts.

An important aim of this work is to go after the pedagogical character of the software to be developed.

Inescapably, a large portion of this dissertation will be dedicated to the explanation of the theoretical concepts of the LR deterministic bottom-up parsing, concretely in the different classic variants: LR(0), SLR(1), LR(1) and LALR(1).

Keywords: Formal Languages and Automata Theory, OCaml, OFLAT, OCaml-FLAT, Functional Programming, Formal Grammars, LL(1), Pedagogical Tools.

Índice

1 INTRODUÇÃO 1

1.1 Enquadramento e Motivação 1

1.2 Objetivos 2

1.3 Estrutura 2

2 FERRAMENTAS PEDAGÓGICAS PARA FLAT 3

2.1 Exposição sobre ferramentas pedagógicas 3

2.2 JFLAP 4

2.3 First+Follow+Predict Calculator 7

2.4 LR(0) Parser Visualization 8

2.5 LR(1) Parser Generator 10

3 CONCEITOS TEÓRICOS PRÉVIOS 11

3.1 Gramáticas livres de contexto 11

3.2 Parsers e a sua relação com gramáticas 12

3.2.1 Autómatos de Pilha 13

3.3 Parsers deterministas 16

3.3.1 Conjuntos FIRST(A)/FOLLOW(A) 17

4 PARSING LR 19

4.1 LR(0) 19

4.1.1 Autómato Finito das ações (tabela de análise sintática/tabela de parsing): 21

4.2 SLR(1) 26

4.3 LR(1) ... 30

4.4 LALR(1) ... 36

5 OCAML-FLAT ... 39

5.1 Introdução ao OCaml-FLAT ... 39

5.2 Módulo ContextFreeGrammar ... 40

5.2.1 Módulo LRGrammar ... 41

5.2.2 Módulo LR0Grammar ... 41

5.2.3 Módulo SLR1Grammar ... 46

5.2.4 Módulo LR1Grammar ... 50

5.2.5 Módulo LALR1Grammar ... 52

6 INTEROPERABILIDADE OCAML/JAVASCRIPT ... 55

6.1 Introdução à Interoperabilidade OCaml/JavaScript ... 55

6.2 Representação de tipos JavaScript em OCaml ... 56

6.3 Bindings ... 57

6.4 Criar objetos JavaScript em OCaml ... 58

6.5 Módulos importantes do Js_of_ocaml ... 59

6.5.1 Módulo Js ... 59

6.5.2 Submódulo Js.Unsafe ... 59

6.5.3 Módulo Dom_html ... 60

6.5.4 Módulo Firebug ... 61

7 CYTOSCAPE.JS ... 63

7.1 Introdução ao Cytoscape.js ... 63

7.2 Módulo Data Item ... 63

7.3 Estilos ... 64

7.4 Initialização do Cytoscape ... 65

8 OFLAT ... 67

8.1 Introdução ao OFLAT 67

8.2 Módulo ContextFreeGrammarGraphics 68

8.3 Módulo ContextFreeGrammarLRGraphics 69

8.3.1 Criação de diagramas LR 69

8.3.2 Criação de tabelas de parsing LR 74

8.3.3 Aceitar palavras com técnicas LR 76

9 Apreciação Geral e Conclusões 79

9.1 OCaml-FLAT 79

9.2 OFLAT 80

9.3 Conclusões 80

9.4 Trabalho Futuro 81

1

Introdução

1.1 Enquadramento e Motivação

Os conceitos de FLAT (Linguagens Formais e Teoria de Autómatos) são bastante ricos e profundos, com relações com vários tópicos de Informática, e assim fornecendo um bom complemento para a aprendizagem. No entanto, conceitos de teoria FLAT são exigentes e rigorosos, provando ser um desafio considerável para a maioria dos alunos. Para promover a aprendizagem destes conceitos, é importante dispor de ferramentas interativas e intuitivas com exemplos e/ou exercícios para ajudar a perceber a teoria FLAT.

Na FCT-UNL, desde à cerca de três anos, tem existido um trabalho contínuo para desenvolver as duas seguintes ferramentas pedagógicas: a biblioteca OCaml-FLAT, que disponibiliza uma grande variedade de conceitos FLAT para serem usados em programas OCaml[19], e a aplicação gráfica OFLAT, que permite visualizar e manipular conceitos de FLAT de uma forma intuitiva. Esta dissertação é mais um elemento do caminho que está a ser percorrido.

Esta dissertação irá adicionar ao referido software novos conceitos de FLAT, concretamente tudo o que tenha a ver com LR parsing. Para suportar estes conceitos, iremos introduzir a teoria subjacente ao LR parsing e desenvolver as diferentes variantes clássicas do LR: LR(0), SLR(1), LR(1) e LALR(1).

Como o foco desta dissertação é promover a aprendizagem de técnicas LR, haverá um grande empenho no desenvolvimento pedagógico. A eficiência do código não é o fator mais importante, só precisamos de garantir que o sistema funciona bem com exemplos de dimensão pequena ou média.

1.2 Objetivos

O objetivo desta dissertação é estender a biblioteca OCaml-FLAT e a aplicação OFLAT com suporte de técnicas de LR parsing, num contexto pedagógico, com objetivo de facilitar o estudo destas técnicas para alunos e docentes de informática. Estas duas peças de software já tem suporte para outros tópicos na área de FLAT (Formal Languages and Automaton Theory) como autómatos finitos, gramáticas independentes de contexto e gramáticas LL(1).

Pretende-se que as funcionalidades desenvolvidas nesta dissertação estejam adaptadas para complementar a aprendizagem dos alunos na FCT-UNL e noutras escolas.

O código da biblioteca OCaml-FLAT e da ferramenta OFLAT deve ser escrito a pensar na legibilidade e na extensibilidade.

1.3 Estrutura

Este relatório está organizado em nove capítulos, com a seguinte organização:

Introdução: É o capítulo presente, onde se apresenta as motivações e objetivos desta dissertação

Ferramentas pedagógicas para FLAT: Capítulo de exposição sobre ferramentas que implementam conceitos FLAT, e estado da arte das ferramentas com relevância no contexto de LR parsing.

Conceitos teóricos prévios: Capítulo dos conceitos teóricos prévios necessários para a contextualização do LR parsing.

Parsing LR: Capítulo que explica e demonstra o funcionamento das técnicas clássicas de parsing LR.

OCaml-FLAT: Capítulo em que se explica a biblioteca existente OCaml-FLAT e decisões de implementação dos novos módulos de parsing LR.

Interoperabilidade OCaml/JavaScript: Capítulo que fala sobre a interoperabilidade entre as duas linguagens com recurso ao Js_of_ocaml.

Cytoscape.js: Capítulo que apresenta a biblioteca JavaScript Cytoscape e a sua relevância.

OFLAT: Capítulo em que se descreve a ferramenta pedagógica OFLAT e, relativamente às novas funcionalidades, se explicam as decisões de desenho e de implementação.

Apreciação Geral e Conclusões: Capítulo com alguma análise crítica do trabalho feito, conclusões e indicação de algum possível trabalho futuro.

2

Ferramentas Pedagógicas para FLAT

2.1 Exposição sobre ferramentas pedagógicas

Existe um vasto leque de ferramentas na área de FLAT (Formal Languages and Automata Theory), sendo geradores de parsers, ferramentas particularmente importantes para a aplicação dos conceitos da área. É importante destacar ferramentas, como o Yacc, que permitem gerar parsers LR para diferentes linguagens target como C++, C#, Java, JavaScript e OCaml.

Mas essas ferramentas não têm ambições pedagógicas. Esta secção vai focar-se numa análise de ferramentas pedagógicas que têm o objetivo de promover a aprendizagem de parsing LR. Acontece que ferramentas pedagógicas que lidam com parsing LR são bastante raras. Neste capítulo, apresentam sumariamente as ferramentas mais relevantes que foi possível encontrar.

O tópico de FLAT é muito rico e essencial na área de Ciências e Computação. Em primeiro lugar, é a via através da qual se estuda Teoria da Computação, incluindo o estudo de diferentes mecanismos com poderes computacionais diferentes. Em segundo lugar, fornece técnicas que permitem lidar com linguagens em geral, por exemplo na implementação de uma linguagem de programação.

A teoria FLAT envolve um nível de complexidade consideravelmente elevado. Por esse motivo, ferramentas auxiliares com objetivo pedagógico são de grande interesse para o desenvolvimento académico e profissional desta área. Existem ferramentas para um leque grande de opções envolvendo FLAT, e serão exploradas as ferramentas que promovem conceitos importantes para o LR parsing.

2.2 JFLAP

JFLAP[14] é um software desenhado com o objetivo de promover a exploração de tópicos de linguagens formais como autómatos finitos, máquinas de Turing, gramáticas, parsing entre outros. O JFLAP emergiu por volta de 1990 como uma série de ferramentas desenvolvidas por Susan Rodger e estudantes do Rensselaer Polytechnic Institute sobre a sua direção. Atualmente o JFLAP é uma aplicação integrada desenvolvida em Java, tem evoluído e a versão mais recente 7.1 foi lançada no ano 2018. Atualmente é a ferramenta de FLAT mais usada no meio académico.

Ao executar a aplicação, o utilizador é apresentado com os vários tópicos suportados pela aplicação.

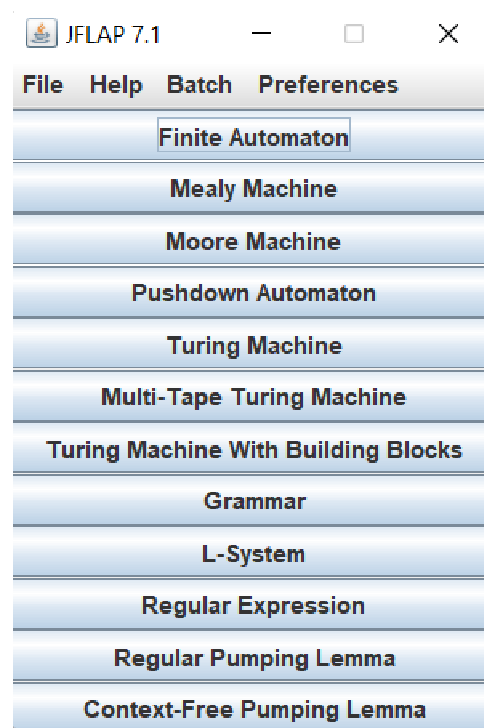


Figura 1:Menu principal do JFLAP

Para o suporte de gramáticas, a aplicação fornece a opção de carregar ficheiros para ler gramáticas já definidas ou introduzir novas gramáticas manualmente. Perante a gramática do utilizador existem várias opções disponíveis.

Temos a funcionalidade Test que permite ao utilizador perceber que tipo de gramática foi introduzida. Ou seja, se é uma gramática irrestrita, livre de contexto, sensível a contexto, regular, etc. Temos a funcionalidade Convert que permite transformar gramáticas em

autómatos de pilha e autómatos finitos. Também possui a capacidade de remover produções inalcançáveis ou vazias e a capacidade de remover símbolos desnecessários nas produções.

Temos a funcionalidade de Parsing[15] que pode ser dividida em várias opções para gramáticas de tipo diferentes, como Build LL(1) Parse Table, Build CYK Parse Table, Brute Force Parse, User Control Parse e a opção mais relevante para LR(k) parsing, Build SLR(1) Parse Table. Esta opção permite criar as os diagramas SLR(1), as respectivas tabelas de parsing passo por passo ou automaticamente. Isto permite à aplicação ser explorada passo a passo para compreender como funciona ou ver o resultado para comparar com resoluções de exercícios.

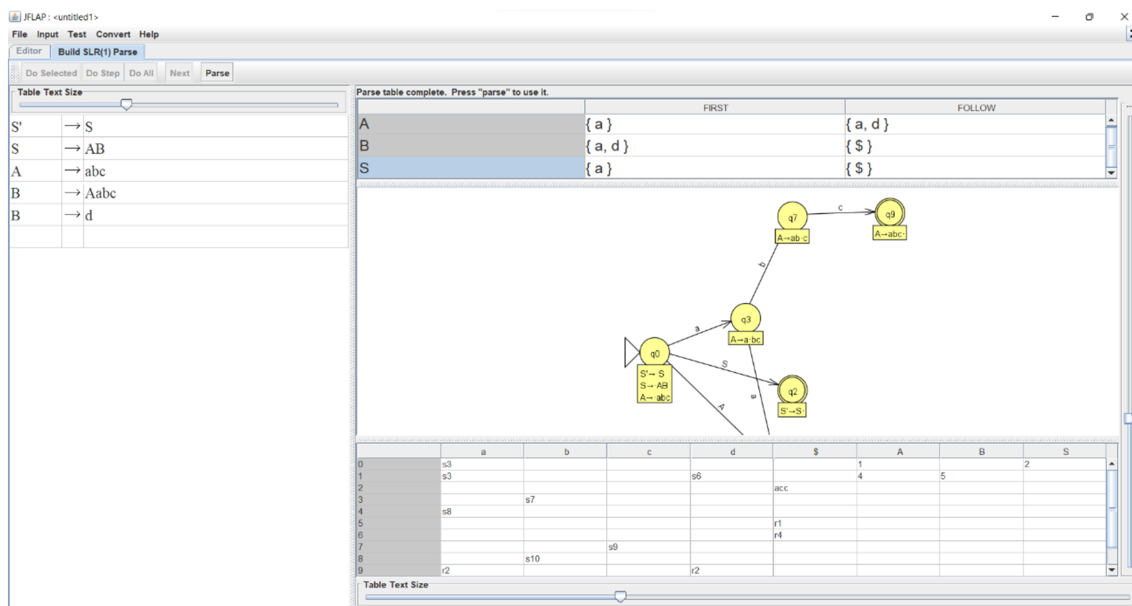


Figura 2: JFLAP SLR(1) diagrama e tabela de parsing

Também é possível utilizar as mesmas tabelas na para validar inputs de teste do utilizador na aplicação.

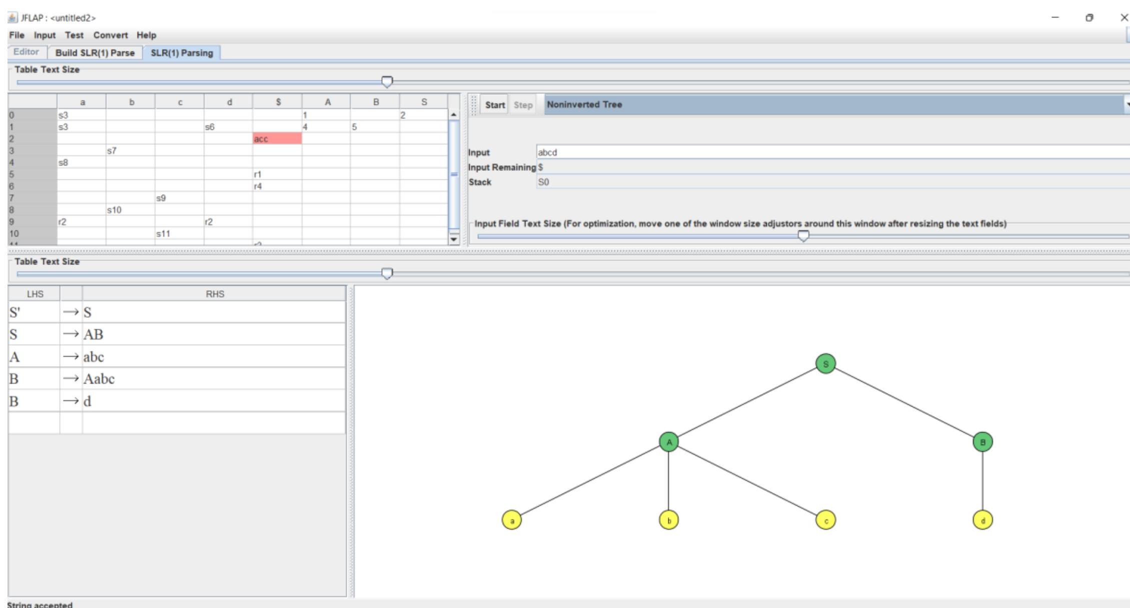


Figura 3:JFLAP parsing do input

Apesar do JFLAP ser uma ferramenta pedagógica útil e poderosa, ao nível do parsing LR, ele apenas suporta LR(0) e SLR(1). Não suporta LR(1) nem LALR(1). Uma vantagem do trabalho que propomos fazer nesta dissertação, é oferecer essas funcionalidades que o JFLAP não oferece.

2.3 First+Follow+Predict Calculator

Desenvolvido pela USNA (United States Naval Academy), é uma ferramenta pedagógica concisa que corre no browser do utilizador. É uma ferramenta muito especializada orientada para o cálculo de alguns aspetos técnicos ligados a parsing LL(1)[16].

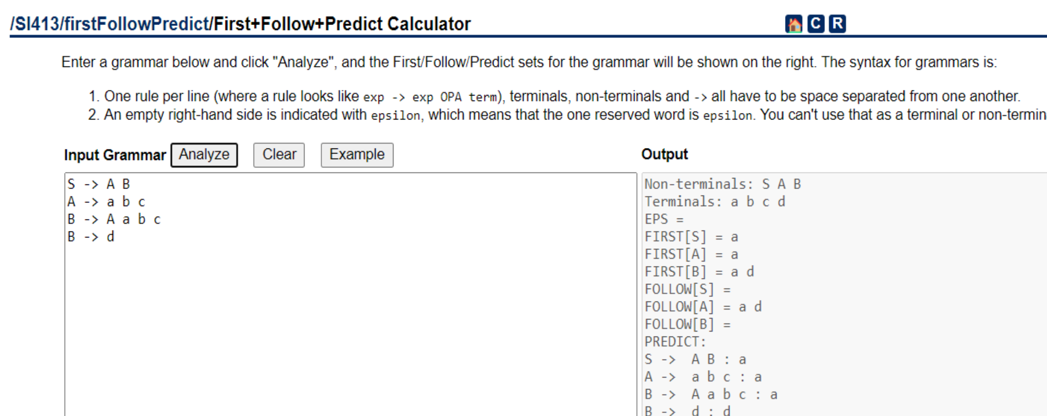


Figura 4: Conjuntos FIRST/FOLLOW/PREDICT para uma gramática de input

Partindo de uma caixa de texto, o utilizador pode inserir gramáticas para serem analisadas. A aplicação retorna informações importantes da gramática para o contexto de parsing como os elementos terminais e não terminais, os conjuntos FIRST, FOLLOW e PREDICT. A aplicação também oferece exemplos de input para demonstrar como a aplicação funciona.

2.4 LR(0) Parser Visualization

Desenvolvida pela universidade de Princeton por Zak Kincaid e Shaowei Zhu, esta ferramenta pedagógica corre no browser do utilizador. Partindo de uma caixa de texto, o utilizador pode inserir gramáticas para serem analisadas. A ferramenta constrói diagramas LR(0) se possível e oferece uma opção para validar o input e mostrar a parser tree gerada na validação[17].

Como LR(k) parsers são bottom-up, esta funcionalidade pode ser retirada do contexto do LR(k) e é utilizada para demonstrar o funcionamento de bottom-up parsers e as suas relações com gramáticas.

2. LR(0) Automaton

State	Item set
0	{S' ::= • S $, S ::= • A B, A ::= • a b c}
1	{A ::= a • b c}
2	{S' ::= S • $}
3	{S ::= A • B, B ::= • A a b c, B ::= • d, A ::= • a b c}
4	{B ::= d • }
5	{B ::= A • a b c}
6	{S ::= A B • }
7	{B ::= A a • b c}
8	{B ::= A a b • c}
9	{B ::= A a b c • }
10	{S' ::= S $ • }
11	{A ::= a b • c}
12	{A ::= a b c • }

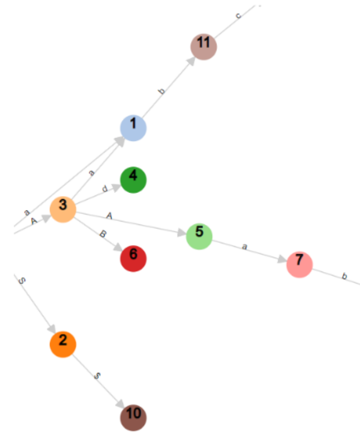


Figura 5:LR(0) autómato e diagrama

3. Parsing

Token stream separated by spaces: a b c d

Start/Reset | Step Forward

Remaining Input

Last action

Shift

Stack

S' ::= • S $
S ::= • A B
A ::= • a b c

S' ::= S • $

S' ::= S $ •

Partial Parse Tree

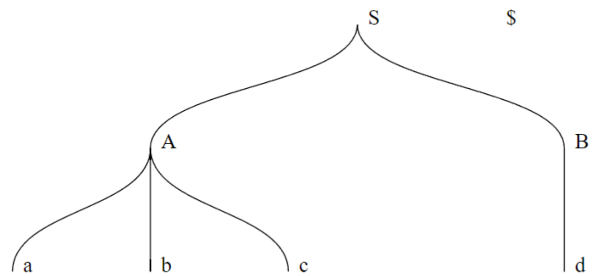


Figura 6:LR(0) parsing e árvore de parsing

2.5 LR(1) Parser Generator

Desenvolvido pelo utilizador apgrgr como parte de uma coleção de ferramentas de ilustração de parsing, designada por JSMachines e está disponível no site Sourceforge. Partindo de uma caixa de texto, o utilizador pode inserir gramáticas LR(1) para serem analisadas. Esta ferramenta calcula e mostra o conjunto FIRST de cada elemento não terminal e os lookaheads consequentes para cada produção da gramática LR(1)[18].

Também fornece a tabela de parsing final e a opção de testar a tabela de parsing com um input do utilizador, demonstrando os passos para validar o input.

Figura 7:LR(1) FIRST/tabelas de fechos/tabelas de parsing

3

Conceitos Teóricos Prévios

3.1 Gramáticas livres de contexto

As linguagens sintéticas, como por exemplo as linguagens de programação, são entidades complexas que envolvem múltiplos aspetos sobre palavras. A natureza da palavra varia muito com a gramática. Por exemplo se se tratar da gramática da linguagem Pascal, uma palavra será um programa completo; se se tratar da gramática dos literais de tipo real, cada palavra será um literal real.

Por uma questão prática é conveniente tratar cada aspeto separadamente. No caso de uma linguagem de programação consideram-se os seguintes aspetos: A estrutura das palavras, a validade das palavras no contexto em que são usadas, a validade das palavras face à sua semântica. As gramáticas livres de contexto são o formalismo que endereça o primeiro aspeto, a estrutura das palavras.[3]

Gramáticas livres de contexto são gramáticas formais[4] compostas por um quádruplo do tipo <V,T,P,S>[12].

V corresponde a um conjunto de símbolos não terminais que funcionam como um conjunto de variáveis matemáticas, representando palavras desconhecidas. São reescritos durante a derivação de uma palavra.

T corresponde a um conjunto de símbolos terminais, símbolos elementares a partir dos quais se constituem as palavras da linguagem. Símbolos terminais não podem ser reescritos durante uma derivação de uma palavra.

P corresponde às produções (também conhecidas como regras) da gramática. Estas regras são expressas na forma A -> α, sendo o símbolo A não terminal e sendo α uma sequência

de símbolos terminais e/ou não terminais, podendo ainda ser uma sequência vazia, que se escreve usando o símbolo ε.

S corresponde ao símbolo inicial da gramática e é a partir dele que todas as derivações começam.

Perante a gramática livre de contexto [A -> Bc ; B -> cd], os símbolos A e B são não terminais, sendo possível reescrever A em Bc e B em cd independentemente do contexto. Os símbolos c e d são terminais pois não existe regra nesta gramática para derivar um conjunto de símbolos terminais e/ou não terminais a partir de c ou d.[2]

3.2 Parsers e a sua relação com gramáticas

Parser é um mecanismo reconhecedor que analisa uma palavra para verificar se a estrutura dela está conforme as regras de uma gramática. Durante esta verificação o parser gera uma árvore sintática para a palavra. Uma árvore sintática captura a estrutura sintática da palavra, e essa estrutura depende das regras da gramática.

Perante a gramática livre de contexto <V = {A,B}, T = {c,d}, P = {A -> Bc , B -> cd}, S = {A}>, e a palavra de input cdc pode ser reconhecida dando origem a seguinte árvore sintática:

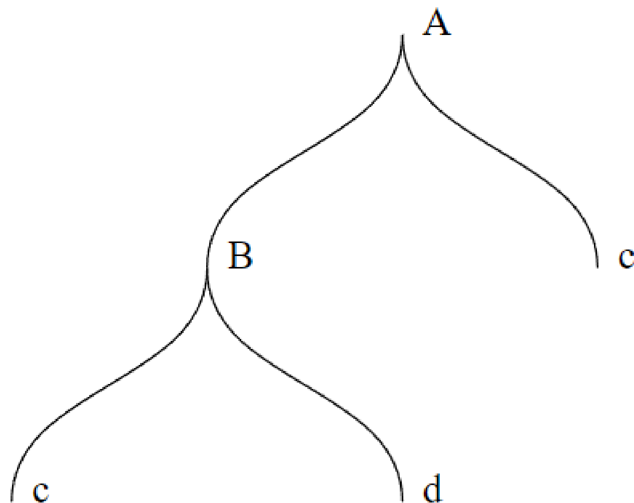


Figura 8: Criação da árvore sintática para a palavra de input a partir das regras da gramática definida

Existem essencialmente duas técnicas para realizar o parsing de palavras: bottom-up parsing e top-down parsing.

Estas técnicas têm pontos comuns: Partem de palavras de input lendo os seus símbolos geralmente da esquerda para a direita e constroem a árvore sintática a partir da derivação da palavra com recurso às regras da gramática.

No entanto, bottom-up parsing produz uma derivação direita pela ordem inversa, usando as regras no sentido α->A, portanto efetuando reduções; enquanto top-down parsing produz uma derivação esquerda pela ordem direta usando as regras no sentido A->α. Bottom-up parsing parte das folhas da árvore sintática e tenta reduzir os símbolos da palavra de input para atingir o símbolo inicial da gramática. Top-down parsing parte da raiz da árvore sintática, o símbolo inicial, e tenta encontrar derivações para a palavra de input.

Em bottom-up parsing, para construir a árvore sintática do recurso 1, lê-se o símbolo c primeiro. Não é possível aplicar regras da gramática no sentido corpo->cabeça ao símbolo c, e por isso continuamos a ler a palavra. Ao ler o símbolo d, podemos aplicar a regra [B -> cd], assim reduzindo a sequência de símbolos cd a B. Como não existe nenhuma regra na gramática para derivar o símbolo B, temos de continuar a ler o input se possível, e encontramos o símbolo c que é o símbolo final da palavra. Como existe uma regra para derivar Bc, podemos reduzir Bc a A com a regra [A -> Bc]. Finalmente, como a palavra de input foi reduzida na sua totalidade e atingimos o símbolo inicial, podemos concluir que a palavra cdc é uma palavra válida para a nossa gramática.

Em top-down parsing, para construir a mesma árvore sintática, lê-se o símbolo c primeiro. Partindo do símbolo inicial, derivamos A para B c com a regra [A -> Bc]. Como ainda não é possível derivar c à esquerda, é necessário derivar B para cd com a regra [B -> cd], assim produzindo uma derivação cdc. Como encontramos o símbolo c lido anteriormente, procedemos para o próximo símbolo da palavra de input d, que está contido na derivação e finalmente lemos o símbolo c que também está contido na derivação. Como já percorremos todos os símbolos da palavra de input e obtivemos uma derivação dela, podemos concluir que cdc é uma palavra válida para a nossa gramática.

3.2.1 Autómatos de Pilha

Um autómato de pilha é um mecanismo computacional que permite exprimir computações gerais[12], e em particular pode ser usado para exprimir as operações de um parser. É por esta segunda razão que nos interessa falar de autómatos de pilha.

Um autómato de pilha é definido como um tuplo de 7 elementos <Q, Σ, Γ,s, Z, δ, F> onde Q é o conjunto finito dos estados do autómato (memória interna), Σ é o conjunto finito dos símbolos terminais do autómato (alfabeto da linguagem), Γ é o conjunto finito dos símbolos da pilha (alfabeto da pilha), s é o estado inicial do autómato, Z é o símbolo inicial da pilha , δ é a relação de transição do autómato com o tipo S × (Σ ∪ {ε}) × Γ × S × Γ∗, e F é o conjunto dos estados finais (aceitação) do autómato.

Para reconhecer uma palavra usando um autómato, é preciso usar uma sucessão de transições, partindo do estado inicial s, com o símbolo Z na pilha e, no final, alcançando uma configuração de aceitação. Há dois critérios alternativos que podem ser usados para caracterizar uma configuração de aceitação.

Critério da pilha vazia: A configuração de aceitação tem a pilha vazia e a palavra foi consumida até ao fim. O estado alcançado é irrelevante para este critério.

Critério dos estados de aceitação: A configuração de aceitação tem um estado que é de aceitação e a palavra foi consumida até ao fim.

Vamos apresentar um autómato de pilha [Q, Σ, Γ, s, Z, δ, F] para reconhecer a linguagem $\{b^n d^n$: n > 0, n ∈ N}:

Q = {p,q,t}

Σ = {b,d}, Γ = {Z,B}, s = p

Z = Z, F= {t}

δ definido na seguinte tabela:

δ	b	d	ε
p,Z	p,BZ		
p,B	p,BB	q, λ	
q,B		q, λ	
q,Z			t,Z

Figura 9:Tabela de transições do autómato

Nota: Cada célula corresponde a uma transição do autómato, partindo do par (estado corrente, topo corrente da pilha) e do próximo símbolo da palavra, e transitando para o próximo estado alterando o topo da pilha. O símbolo λ corresponde à pilha vazia, e o símbolo ε corresponde ao símbolo nulo, que neste caso indica que a palavra foi consumida até ao fim.

Este autómato pode ser representado com o seguinte diagrama:

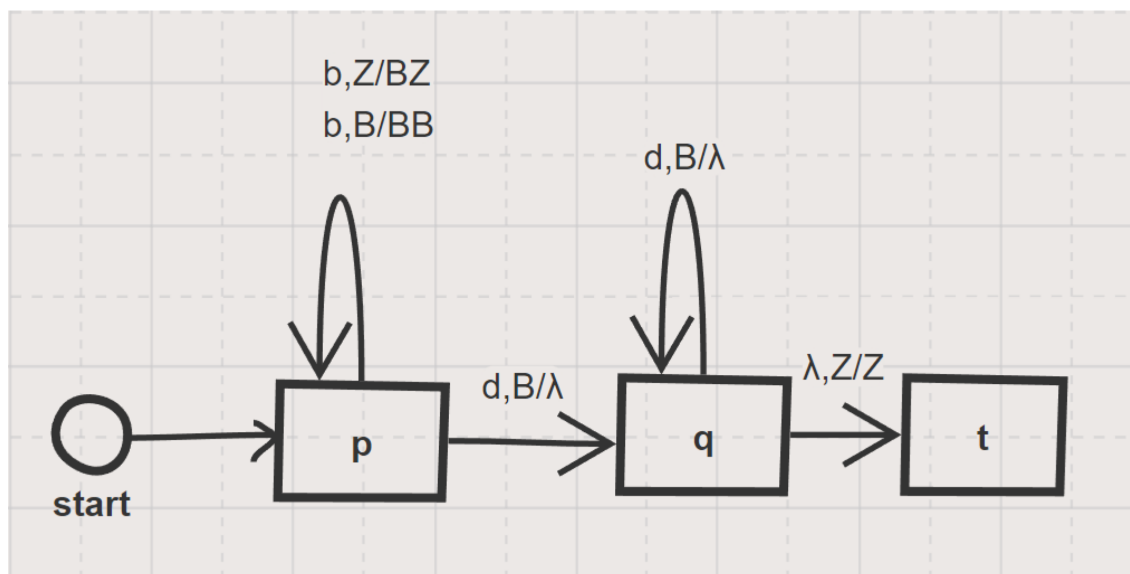


Figura 10:Diagrama do autómato da figura 9

Neste diagrama, uma transição é representada por uma seta que liga o estado corrente ao próximo estado. A etiqueta da seta contém os outros 3 elementos: o próximo da palavra e topo corrente da pilha e o novo topo da pilha.

Para exemplificar descreve-se o significado da casa mais acima à esquerda da tabela: Representa uma transição que só é ativada no estado p, com o símbolo Z no topo da pilha, e como o próximo símbolo da palavra b. O resultado da transição mantém o estado p mas empilha o símbolo B por cima do símbolo Z.

Este autómato reconhece a palavra bbdd. Para mostrar isso basta mostrar uma sucessão de transições que alcance uma configuração de aceitação. Mostra-se agora a sequência usando o critério dos estados de aceitação:

Estado	Pilha	Fita
p (inicial)	Z	bbdd
p	BZ	bdd
p	BBZ	dd
q	BZ	d
q	Z	ε
t (aceitação)	Z	ε

Figura 11:Reconhecimento da palavra bbdd para o autómato da figura 9

Este autómato é determinista. A questão de determinismo ou não determinismo é explicada na secção seguinte.

3.3 Parsers deterministas

Para a classe de linguagens em que estamos interessados (gramática independentes de contexto), os parsers são implementados usando autómatos de pilha.

Diz-se que um autómato é determinista se em cada situação existir no máximo uma transição aplicável. Um autómato é não determinista se existirem situações nas quais exista mais do que uma transição aplicável. Os autómatos deterministas são essencialmente mais simples e são implementados gastando menos recursos. O maior problema da implementação de autómatos não deterministas é que, para explorar as várias transições alternativas aplicáveis temos de usar backtracking ou paralelismo.

Diz-se que um parser é determinista se o autómato que o implementa for determinista. Nós estamos especialmente interessados em classes de gramáticas independentes de contexto, com determinadas propriedades, que permitem construir parsers deterministas.

Vamos considerar a classe de gramáticas LL(k), que estão associadas ao chamado parsing LL, e a classe de gramáticas LR(k), que estão associadas ao chamado parsing LR.

Um parser LL (Left-to-right, Leftmost Derivation) é um parser top-down[28], em que a palavra a reconhecer é processada da esquerda para a direita e o parser funciona construindo uma derivação esquerda da palavra que começa no símbolo inicial[28].

Um parser LR[5] (Left-to-right, Rightmost Derivation) é um parser bottom-up[1] em que a palavra a reconhecer é processada da esquerda para a direita e o parser funciona construindo uma derivação direita da palavra pela ordem inversa[6].

O determinismo dos parsers LL e LR[1] apoia-se muitas vezes no uso de tokens de lookahead que ajudam a tomar decisões durante o parsing[5]. No contexto do parsing determinista, costumam ser usadas as funções FIRST e FOLLOW[2] para determinar qual a regra da gramática a aplicar seguidamente em função do lookahead corrente.

Uma gramática diz-se ambígua se para uma dada palavra existe mais do que uma derivação esquerda dessa palavra. Um parser criado para uma gramática ambígua[3] nunca poderá ser determinista, porque ele tem de ter a capacidade de reconhecer essa palavra das várias maneiras compatíveis com a gramática.

Na secção seguinte vai-se falar em parsing determinista LL e LR. Convém ficar claro que se a nossa gramática for ambígua essas técnicas não são aplicáveis.

3.3.1 Conjuntos FIRST(A)/FOLLOW(A)

No parsing LL e algumas variantes do parsing LR, no tratamento dos tokens lookahead, desempenham um papel importante os conjuntos FIRST(A) e FOLLOW(A).[2]

O conjunto FIRST(A) identifica o conjunto de símbolos terminais que iniciam palavras deriváveis a partir do símbolo não terminal A[12]. Para exemplificar, dada a gramática [A -> aB | aC; B -> b; C -> c; D -> ε | d; E -> DA]:

FIRST(A) ={a}; FIRST(B)={b}; FIRST(C)={c}; FIRST(D) = {d};

FIRST(E) = FIRST(D) U FIRST(A) = {a,d}

O conjunto FOLLOW(A) identifica o conjunto de símbolos terminais que podem surgir a seguir ao símbolo não terminal A na derivação de uma palavra[12]. Para exemplificar, dada a gramática [A -> BC | aC; B -> b; C -> c]:

FOLLOW(A) = {}; FOLLOW(B) = FIRST(C) = {c}; FOLLOW(C) = FOLLOW(A) = {}

4

PARSING LR

4.1 LR(0)

LR(0) é a técnica mais simples na família de parsers LR. Os parsers LR(0) são os mais fáceis de aprender e servem como ponte para as técnicas mais avançadas LR(k). Como consequência da sua simplicidade, estes parsers são relativamente fracos para uso prático exceto para um subconjunto muito limitado e simples das gramáticas livres de contexto.

Para começar a entender a técnica LR vamos começar por mostrar o funcionamento de um parser LR(0) usando um exemplo. Um parser LR é um parser bottom-up e portanto, como já foi dito, ele vai efetuar uma derivação direita da palavra a reconhecer pela ordem inversa. Vamos considerar a gramática aumentada [S' -> S$; S -> SA | A; A -> aSb | ab] e o reconhecimento da palavra aababb$. Uma gramática aumentada, consiste em adicionar um símbolo inicial novo com uma regra gramatical na forma S' -> S$, sendo S o símbolo inicial anterior. Esta técnica ajuda a detetar a aceitação de uma palavra, pois se aplicarmos esta regra, sabemos que estamos a aceitar uma palavra. Note que o símbolo $ representa o final da palavra.

Eis a derivação direita desta palavra[11]:

S' -> S$
-> A$
-> aSb$
-> aSAb$
-> aSabb$
-> aAabb$
-> aababb$

Para o parser conseguir produzir esta derivação pela ordem inversa, ele precisa de usar uma pilha auxiliar, que descreve a estrutura da parte do input já vista. Durante o funcionamento há 3 tipos de ações possíveis:

Transferência - Transfere o próximo símbolo para o topo da pilha;

Redução - Aplica uma regra da gramática no sentido corpo->cabeça;

Aceitação – O passo final, que acontece se a palavra for reconhecida;

Eis a evolução do parser LR(0) a efetuar a derivação referida antes pela ordem inversa:

Pilha	Fita	Ação
λ	aababb$	Transferência
a	ababb$	Transferência
aa	babb$	Transferência
aab	abb$	Redução A -> ab
aA	abb$	Redução S -> A
aS	abb$	Transferência
aSa	bb$	Transferência
aSab	b$	Redução A -> ab
aSA	b$	Redução S -> SA
aS	b$	Transferência
aSb	$	Redução A -> aSb
A	$	Redução S -> A
S	$	Transferência
S$	ε	Aceitação

Figura 12: Derivação direita da palavra aababb

Ao concatenar a pilha e a fita, obtemos as formas direitas na derivação direita anterior pela ordem inversa. Por exemplo, na linha 4, concatenando a pilha aab com a fita aab$, obtemos a palavra completa. Outro exemplo: Na linha 5, concatenando a pilha aA com a fita abb$ obtemos a forma aAabb$ que corresponde ao que se obtém na penúltima derivação.

4.1.1 Autómato Finito das ações (tabela de análise sintática/parsing):

No parsing LR(0), a escolha da próxima ação é potencialmente baseada no conteúdo da pilha completa que descreve tudo o que já foi analisado até ao momento; a decisão não é baseada apenas no topo da pilha. Para tornar o parsing eficiente, toda a informação relevante do contexto esquerdo[11] é codificada num número que designa um estado do parser; para cada gramática LR(0), há um número finito de estados do parser. Incorporando esta otimização, o funcionamento do autómato de pilha irá ser um pouco diferente do exemplo ingénuo mostrado antes. A tabela de análise sintática LR(0) é bastante sofisticada pois descreve um autómato finito de ações indiretas sobre um autómato de pilha.

Estado	S	A	a	b	$	Ação
0	1	2	3			Transferência
1		5	3		4	Transferência
2						Redução S -> A
3	6	2	3	7		Transferência
4						Aceitação
5						Redução S -> SA
6		5	3	8		Transferência
7						Redução A -> ab
8						Redução A -> aSb

Figura 13:Tabela de análise sintática LR(0) para a gramática [S' -> S$; S -> SA | A; A -> aSb | ab]

A maneira mais simples de explicar esta tabela de análise sintática é através da observação da sua utilização durante o reconhecimento de uma palavra. Um pouco mais abaixo aparece o exemplo do reconhecimento da palavra aababb$.

Quando o reconhecimento começa, na pilha encontra-se apenas o estado 0 e a palavra está toda por ler. Esta situação corresponde à primeira linha do reconhecimento.

A tabela de análise sintática mostra que no estado 0 para o input a, se efetua uma transferência com transição para o estado 3. O resultado desta ação pode ser observado na linha 2 do reconhecimento. Note que é empilhado o símbolo a juntamente com o estado 3.

Seguem duas transferências, concretamente dos símbolos a e b, e o autómato fica no estado 7. A tabela de análise sintática mostra que neste estado ocorre uma redução usando a regra [A -> ab]. A redução concretiza-se da seguinte forma: Como do lado direito da regra há dois símbolos, temos a certeza que eles ocorrem no topo da pilha. Esses dois símbolos são desempilhados juntamente com os estados associados. Portanto é desempilhada a sequência a3b7. Fica exposto no topo da pilha o estado 3. E agora é empilhado o símbolo A, da cabeça

da regra, juntamente com o estado 2, porque é isso que está previsto na tabela de análise sintática para o par (3,A).

Vamos saltar para a explicação do passo final. No topo da pilha encontra-se a sequência 0S1$4, e a tabela de análise sintática diz que no estado 4 se vai efetuar a aceitação da palavra. Portanto, acontece a aceitação e o reconhecimento termina com sucesso. Note que a palavra foi lida até ao fim e que a tabela da análise sintática garante que isso acontece, porque a transição para o estado 4 só acontece depois de ler o símbolo $.

Se o autómato só tivesse ações de transferência não seria necessário guardar os estados na pilha. Bastaria tomar nota do estado corrente fora da pilha e usar esse estado nas decisões de transferência de acordo com a tabela de análise sintática. Mas existem as ações de redução, que não podem ser aplicadas indiscriminadamente, mas apenas em determinados estados, que terão de aparecer no topo da pilha na altura da redução. Uma redução requer identificar no topo da pilha, o lado direito de uma regra. Se o estado do topo da pilha permitir a redução, os símbolos correspondentes ao lado direito de uma regra são desempilhados, e é empilhada a cabeça da mesma regra. O estado que agora se empilha é determinado pelo estado anterior na pilha.

Reconhecimento da palavra aababb com autómato de pilha:

Pilha	Fita	Ação
0	aababb$	Transferência
0a3	ababb$	Transferência
0a3a3	babb$	Transferência
0a3a3b7	abb$	Redução A -> ab
0a3A2	abb$	Redução S-> A
0a3S6	abb$	Transferência
0a3S6a3	bb$	Transferência
0a3S6a3b7	b$	Redução A -> ab
0a3S6A5	b$	Redução S -> SA
0a3S6	b$	Transferência
0a3S6b8	$	Redução A -> aSb
0A2	$	Redução S-> A
0S1	$	Transferência
0S1$4	λ	Aceitação

Figura 14: Reconhecimento da palavra abab com autómato de pilha.

A criação da tabela sintática envolve uma análise profunda da gramática e do processo de parsing. Vamos explicar o procedimento que permite criar a tabela sintática LR(0) com base na gramática.

Cada estado reflete toda a informação do contexto esquerdo que é relevante para tomar decisões sobre o passo seguinte. Cada estado corresponde a uma coleção de itens LR(0).

Um item [A -> $\alpha.\beta$] é uma regra da gramática anotada no seu corpo com um ponto que separa o contexto esquerdo relevante (Prefixo viável[11]) α do que vem a seguir β. Este item indica uma possibilidade de redução futura, quando o topo da pilha é α. Espera-se que a evolução da pilha acabe por colocar o β em cima do α e nessa situação futura o $\alpha\beta$ de topo poderá ser reduzido para A.

Um item da forma [A -> α.], portanto sem nada a seguir ao ponto, chama-se um item completo. Este item indica a possibilidade de uma redução imediata, quando o topo da pilha é α.

Uma regra da gramática pode dar origem a vários itens, mas cada item está definido a partir de uma única regra da gramática.

Exemplo: Partindo da regra [A->ab] é possível criar 3 itens diferentes:

[A->.ab]; [A->a.b]; [A->ab.]

No item [A->.ab], estamos à espera de realizar no futuro uma redução de ab para A. Para isso acontecer, os próximos símbolos da palavra devem ser ab para poderem ser transferidos.

No item [A->a.b], estamos à espera de realizar no futuro uma redução de ab para A. O topo da pilha é a, e espera-se que o próximo símbolo seja b.

No item completo [A->ab.], podemos realizar a redução do ab para o símbolo A, porque o topo da pilha é ab.

Num estado do parser podemos não saber antecipadamente qual a regra que será usada para concretizar a redução futura. Por isso, em geral associamos múltiplos itens a cada estado do parser. Por exemplo, na nossa tabela sintática vemos que a única forma de alcançar o estado 3 é por transferência do símbolo a. Os itens que correspondem a esta situação são os seguintes, [A->a.Sb] [A->a.b]. Este é o chamado núcleo do estado 3. Em geral, o núcleo de um estado é um conjunto de itens que partilham o mesmo símbolo antes do ponto. Foi esse símbolo que causou uma transição para o dito estado. Existe apenas a exceção do estado inicial 0, que é constituído pelos itens derivados das regras que têm o símbolo inicial na cabeça adicionando o ponto na primeira posição do corpo.

Um item que tenha um ponto antes de um símbolo não terminal, como por exemplo [A->a.Sb], indica que o parser tenciona reconhecer a seguir o símbolo não terminal S. Um aspeto

importante da técnica LR(0) é que cada estado precisa de conter todos os itens dos símbolos não terminais que ocorrem a seguir a pontos. Desta forma o parser consegue gerir todas as formas de obter esse símbolo não terminal no futuro a curto prazo.

Agora já dispomos de todos os conceitos necessários para definir um estado do parser. Partimos da identificação do núcleo, e a partir deste adicionamos recursivamente novos itens para todos os símbolos não terminais que ocorrem a seguir a pontos. O processo é concluído quando todos os símbolos não terminais que ocorrem a seguir a pontos já foram tratados. Este é o procedimento para o cálculo do fecho, um conjunto de itens que descrevem o estado atual do parser.

Para calcular o fecho, partindo de um núcleo na forma [S' -> .S$] (passo inicial da pilha no estado 0), sabemos que o próximo símbolo esperado é o símbolo não terminal S. Como tal, esse símbolo pode ser obtido através de uma redução e temos de adicionar os itens que têm como cabeça da regra o símbolo S e que ainda não realizaram ações.

Assim adicionamos os itens [S->.SA] e [S->.A] ao fecho. Como estamos à espera de ler símbolos não terminais nesses itens, temos de executar mais um passo do cálculo do fecho para estes itens. Como os itens do fecho resultantes de S já foram calculados, não é necessário executar outra vez o passo para o símbolo não terminal S. Mas temos de calcular o fecho para o símbolo não terminal A, obtendo no fecho os novos itens [A->.aSb] e [A->.ab]. Nestes itens já só esperam ler símbolos terminais, logo completamos o cálculo do fecho para o estado 0.

Estado 0:

Núcleo - [S'->.S$]

Fecho calculado - [S->.SA] [S->.A] [A->.aSb] [A->.ab]

É importante notar que as transições que saem do estado 0 ficam registadas na tabela de parsing para o estado 0. Só há transições para os símbolos S, A e a, a partir do estado 0. No total temos 3 transições a sair do estado 0. Associada a cada uma destas transições temos sempre a ação transferência.

Para exemplificar a obtenção de um novo estado, vamos partir do estado 0 e da transição associada ao símbolo S. Vamos designar o novo estado por estado 1.

O núcleo do estado 1 é obtido copiando todos os itens do estado 0 que têm o S a seguir ao ponto. Nos itens copiados, o ponto avança uma posição, para indicar que já existiu a transição por S. Os itens do novo núcleo serão [S' -> S.$] [S->S.A].

Novamente, para o estado 1, temos de calcular o respetivo fecho, o que resulta na adição dos novos itens [A->.aSb] [A->.ab].

O cálculo do fecho do estado 1 ficou terminado porque olhando para todos os símbolos que seguem a pontos, já não há mais nenhum não terminal que esteja por tratar.

Estado 1:

Núcleo - [S' -> S.$] [S->S.A].

Fecho calculado - [A->.aSb] [A->.ab].

Continua a repetir-se este procedimento para todos os estados que vão aparecendo, e no final quando já não ocorrem estados novos, ficamos com o chamado diagrama LR(0).

Para a nossa gramática [S' -> S$; S -> SA | A; A -> aSb | ab] conseguimos gerar o diagrama LR(0) seguinte:

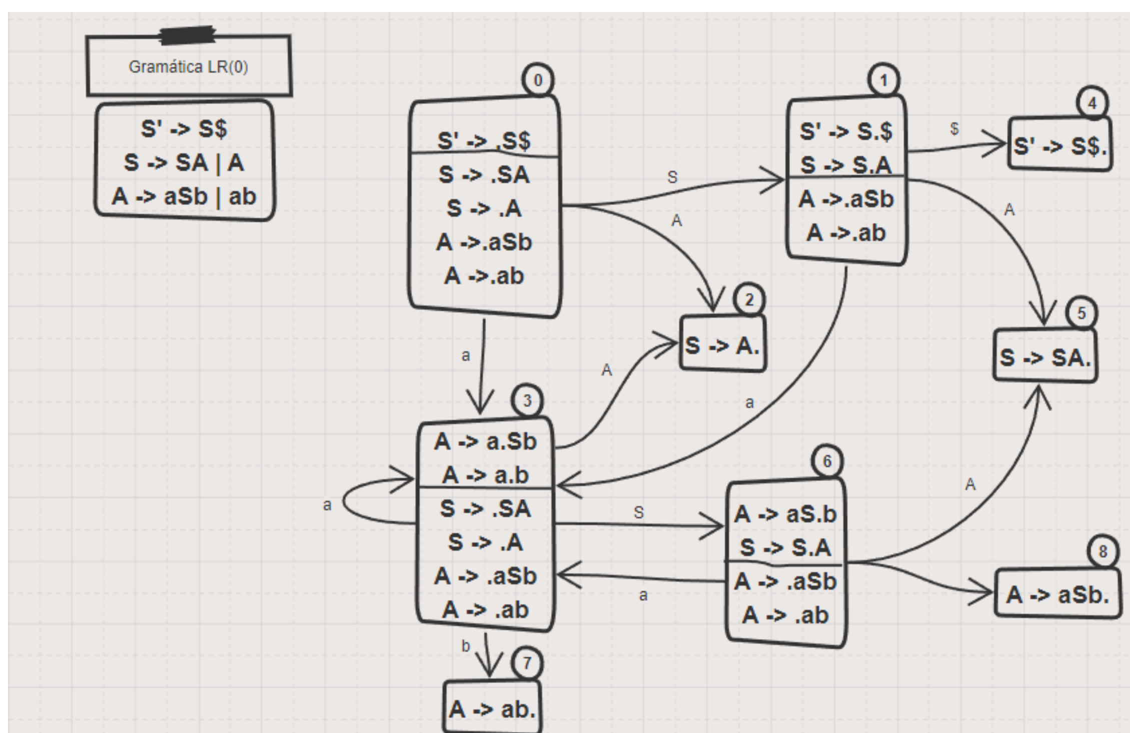


Figura 14:Diagrama LR(0)

Este diagrama LR(0) merece ainda mais alguns comentários. O primeiro comentário é que confirmamos que o diagrama descreve um autómato finito. O segundo comentário, é que, no estado 3, quando se calcula o novo estado para a transição do terminal a, descobrimos que o novo estado já existe e é o próprio estado 3; por isso, no diagrama fica um arco para o próprio estado etiquetado com o símbolo a. O último comentário é que, tendo o estado 4 apenas um item completo, já não ocorre nada a seguir a um ponto, e, portanto, não existe qualquer transição a sair do estado 4.

A tabela de análise sintática obtém-se reescrevendo o autómato finito sobre a forma de tabela, ignorando os itens que se escondem dentro dos estados.

Uma otimização possível, que é muitas vezes feita na prática, consiste em guardar na pilha apenas os estados, porque eles já dão indiretamente informação sobre os símbolos ausentes. No entanto, decidimos não fazer isso na nossa implementação, por questões pedagógicas. Queremos mostrar aos alunos, no topo da pilha, os símbolos do lado direito de uma regra antes de efetuar uma redução.

4.2 SLR(1)

Mas quais são as limitações do LR(0)? Tomando como exemplo a gramática:

[S -> X$; X -> aBbA; X -> aA; A -> b; A -> ε; B -> b; B -> bD; D -> c]

Podemos obter o seguinte diagrama LR(0) a partir desta gramática:

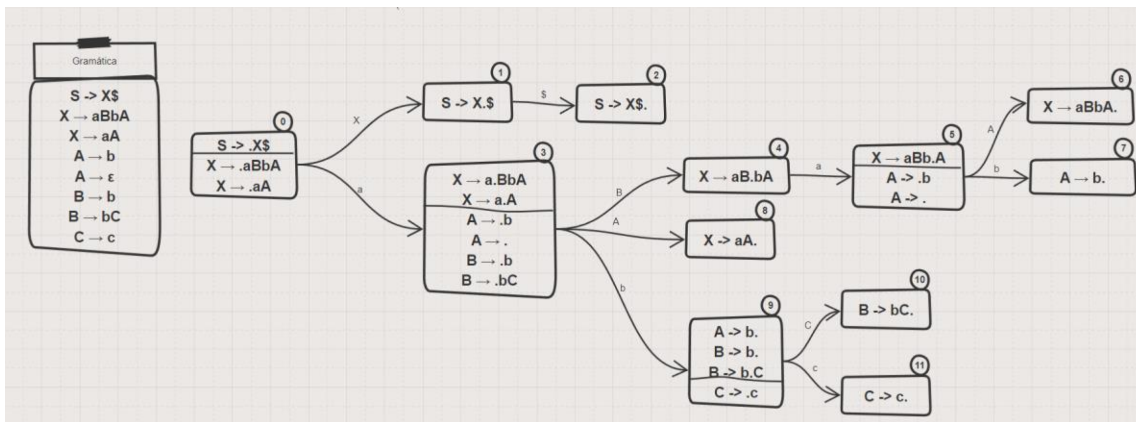


Figura 15: Diagrama LR(0) para uma gramática mais complexa

Eis a correspondente tabela de análise sintática LR(0):

Estado	a	b	c	$	X	A	B	C	Ação
0	3				1				Transferência
1				2					Transferência
2									Aceitação
3		9				8	4		Transferência/Redução A -> ε
4		5							Transferência
5		7				6			Transferência/Redução A -> ε
6									Redução X -> aBbA
7									Redução A -> b
8									Redução X -> aA
9			11					10	Transferência/Redução A -> b e Redução B -> b
10									Redução B -> bC
11									Redução C -> c

Figura 16:Tabela de análise sintática LR(0) para o diagrama da figura 16

E é aqui que encontramos um problema do parsing LR, que afeta mais o caso LR(0) do que os outros casos. Olhando para o estado 9, estamos perante um dilema triplo na decisão do parser - é possível realizar a transferência do símbolo c, reduzir pela regra A -> b, ou pela regra B -> b. Para o estado 3, há um dilema entre uma redução e uma transferência. Para o estado 5, também há um dilema entre uma redução e uma transferência. Estes dilemas chamam-se de conflitos, e são assinalados na tabela de análise sintática com múltiplas entradas na coluna das ações.

De forma geral, podem ocorrer dois tipos de conflitos[7]: redução-transferência e redução-redução. Conflitos transferência-transferência nunca acontecem devido à maneira como os estados são criados no diagrama LR(0).

Um conflito redução-transferência acontece quando um estado tem pelo menos um item completo na forma [A -> α.], juntamente com um item com terminal x à direita do ponto, na forma [A -> α.xβ].

Um conflito redução-redução acontece quando um estado tem pelo menos dois itens completos.

Este tipo de problemas leva-nos à próxima técnica de LR parsing, SLR(1).

Para tentar resolver estes conflitos, o SLR(1)[8] adiciona ao LR(0) a noção de tokens de lookahead, espreitando o próximo símbolo para decidir qual dos itens em conflito é aplicável para o estado atual. A classe das gramáticas SLR(1) contém a classe das gramáticas LR(0). Uma gramática LR(0) é trivialmente uma gramática SLR(1), onde os símbolos de lookahead são usados de forma fútil.

Uma gramática é SLR(1) se as seguintes condições forem cumpridas nos estados do diagrama LR(0)[7]:

1: Para qualquer estado, se existir um item de forma [A ->α.xβ], sendo x um símbolo terminal, e também existir um item completo [B ->μ.], então o x não pode pertencer ao conjunto FOLLOW(B).

2 Para qualquer estado, se ocorrerem dois itens completos [A ->α.] e [B -> β.], então os conjuntos FOLLOW(A) e FOLLOW(B) têm de ser disjuntos.

Para exemplificar, considerando o estado 9, vejamos se é possível resolver o conflito transferência-redução entre os itens [B-> b.] e [C -> .c]. Para isso, calculamos o conjunto FOLLOW(B) = {FIRST(b)} = {b}. Como o símbolo terminal c não pertence ao conjunto FOLLOW(B), o conflito entre estes itens é resolvido. Para o outro conflito transferência-redução, entre [A -> b.] [C -> .c], aplica-se o mesmo raciocínio. Como FOLLOW(A) = FOLLOW(X) = {$} e o símbolo terminal c não pertence ao conjunto FOLLOW(A), o conflito fica resolvido.

Para o conflito entre [A -> b.] e [B -> b.], calculamos os conjuntos FOLLOW(A) e FOLLOW(B). Como FOLLOW(A) = {S} e FOLLOW(B) = {b}, os dois conjuntos são disjuntos, e provamos que não ocorre conflito entre estes itens.

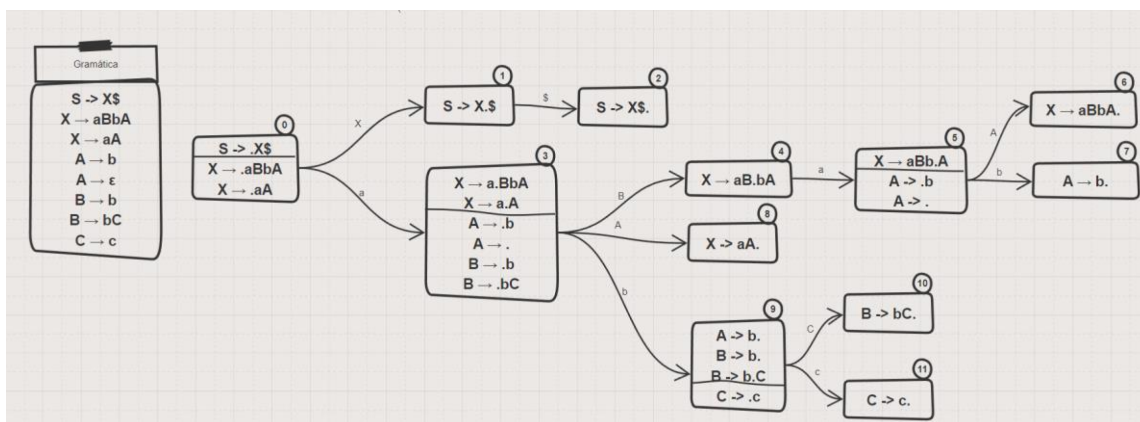


Figura 17:Diagrama SLR(1) para a mesma gramática

Note que o diagrama LR(0) é sempre o mesmo que o diagrama do SLR(1). Já o mesmo não se passa com as tabelas de análise sintática. Uma novidade nas tabelas de análise sintática

SLR(1), é que a linha correspondente a um estado pode conter ações de transferência e redução ao mesmo tempo (veja a tabela abaixo). As escolhas a efetuar pelo parser são assinaladas nas células das transições.

Por exemplo, usa-se a notação R4 para indicar uma redução usando a regra de gramática número 4. Note que na tabela LR(0), a linha corresponde a uma ação de redução tendo todas as células das transições em branco, e isso deixa de acontecer no SLR(1).

Eis a correspondente tabela de análise sintática SLR(1):

Estado	a	b	c	$	X	A	B	C	Ação
0	3				1				Transferência
1				2					Transferência
2				acc					Aceitação
3		9		R4		8	4		Transferência/ Redução A -> ε
4		5							Transferência
5		7		R4		6			Transferência/ Redução A -> ε
6				R1					Redução X -> aBbA
7				R3					Redução A -> b
8				R2					Redução X -> aA
9		R5	11	R3				10	Transferência/Redução A -> b e Redução B -> b
10		R6							Redução B -> bC
11		R7							Redução C -> c

Figura 18:Tabela de parsing SLR(1)

Note que para as tabelas sintáticas LR(0), reduções aplicavam-se sempre ao estado correspondente, sem tomar em consideração o próximo símbolo da pilha. Como tal, nas tabelas de análise sintática LR(0), estados com reduções produziam linhas "vazias", pois assumia-se que não poderia haver outra opção.

No entanto, para o SLR(1), estamos a considerar possíveis conflitos e como tal, sempre que aconteça uma suspeita de conflito, calculamos o FOLLOW(Z) para todos os itens completos, sendo Z a cabeça da redução desses itens, e assim obtemos os símbolos não terminais que se podem esperar a seguir a essa redução. Este raciocínio, traduz-se em inserir para cada

célula do mesmo estado, que espera um símbolo pertencente a FOLLOW(Z), a palavra RX, com X sendo o número da regra da gramática utilizada para a redução.

Como exemplo, considerando o estado 3, ao calcular o FOLLOW(A) devido ao possível conflito transferência-redução entre [A ->.] e [A ->.b], sabemos que se espera o símbolo não terminal $ ao A, e como tal, colocamos na célula que resultado do cruzamento do estado 3 com o símbolo $.

Aplicando este raciocínio para todas as possíveis reduções da nossa gramática exemplo, provamos que é possível resolver todos os conflitos na tabela de análise sintática, e assim, podemos concluir que a gramática é SLR(1).

4.3 LR(1)

Como notado anteriormente, o SLR(1) tem a possibilidade de resolver conflitos através do cálculo de FOLLOW para cada regra correspondente a um item completo, isto porque todos os conflitos envolvem pelo menos um item completo. No entanto, o conjunto FOLLOW não tem em consideração os estados da tabela de análise sintática. Isto produz um conjunto de símbolos em excesso para o lookahead, que conduzem a uma discriminação demasiado grosseira.

Assim, o SLR(1) sofre de uma "potência" reduzida, não conseguindo resolver os conflitos para certas gramáticas. Na tabela de análise sintática, os símbolos em excesso para o lookahead refletem-se em demasiadas células de transição preenchidas com o mesmo Rn.

Para resolver este problema, temos de analisar a próxima técnica da família LR parsing, o CLR(1), Canonical LR(1) parser, normalmente referido como LR(1). Para evitar complicações, limitamos a considerar apenas um token lookahead.

Para demonstrar as limitações do SLR(1), concretamente a ocorrência de um estado com conflitos que não se conseguem resolver usando a técnica SLR(1), iremos exemplificar com a seguinte gramática, [S -> X$; X -> aAc; X -> aBd; X -> Bc; A -> z; B -> z;].

Assim, obtemos o seguinte diagrama SLR(1):

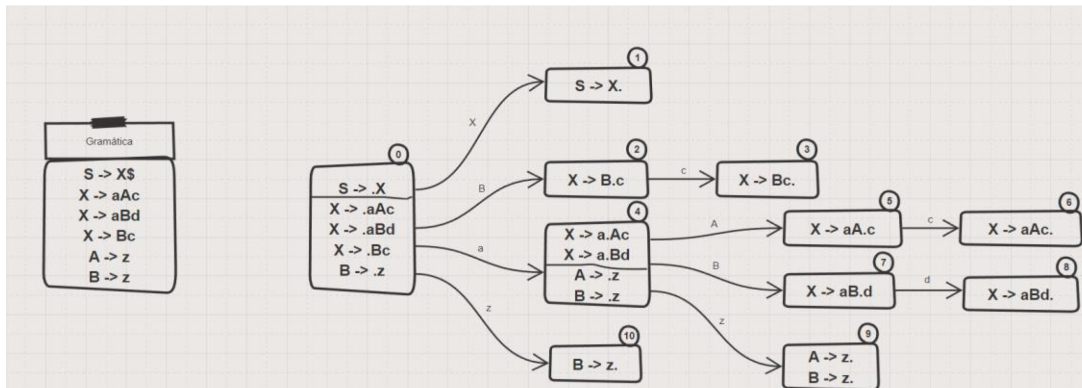


Figura 19:Diagrama SLR(1) para uma gramática mais complexa

Nota: Eliminamos o $ do item do núcleo do estado inicial 0 [S -> .X] para permitir uma comparação com o diagrama LR(1) para a mesma gramática, que aparecerá um pouco mais adiante. Apenas o parsing LR(0) requer a utilização do símbolo $ para indicar final de palavra em alguns itens. A partir do momento em que o conceito de símbolos lookahead começa a ser usado, torna-se mais correto tratar o símbolo $ exclusivamente no contexto dos conjuntos de lookahead.

Eis a resultante tabela de análise sintática SLR(1):

Estado	a	c	d	z	$	A	B	X	Ação
0	5			11			3	1	Transferência
1					acc				Aceitação
2		4							Transferência
3					R3				Redução X -> Bc
4				10		6	8		Transferência
5		7							Transferência
6					R1				Redução X -> aAc
7			9						Transferência
8					R2				Redução X -> aBd
9		R4 R5	R5						Redução A -> z e B -> z
10		R5	R5						Redução B -> z

Figura 20:Tabela de parsing SLR(1) correspondente ao diagrama da Figura 20

Note que na linha do estado 9, existem duas reduções aplicáveis para o símbolo terminal c. Como tal, estamos perante um conflito que o SLR(1) não consegue resolver. Para resolver este problema, iremos utilizar a técnica de LR(1).

A técnica para construir os itens LR(1) estende a técnica do LR(0) [11]. O LR(1) acrescenta um conjunto de símbolos lookahead a cada item, assim gerando itens da forma [A ->α.β , L]. Note que o conjunto de lookahead só afeta verdadeiramente o comportamento do parser em estados onde exista pelo menos um item completo a criar conflito. No entanto, nos outros estados, os conjuntos de lookahead são também importantes pois estão envolvidos nas regras que permitem calcular os lookaheads dos estados seguintes.

Sem perda de generalidade, vamos trabalhar com gramáticas em que há apenas uma regra com o símbolo inicial S na cabeça.

Comecemos por explicar como se cria o estado inicial de um diagrama LR(1). Se a regra inicial da gramática for [S ->α$] ou [S ->α], então o núcleo consiste no item [S ->.α, {$}].

Seguidamente, calculamos o fecho LR(0), adicionamos a cada item novo um conjunto de lookahead vazio e depois vamos preencher os vários conjuntos de lookahead.[10]

O procedimento de preenchimento dos conjuntos de lookahead é simples. Para cada item da forma [A -> .Xβ , L], portanto com símbolo não terminal X à direita do ponto, temos que estudar todas as possibilidades que possam ocorrer a seguir a esse X nas palavras geradas. Vamos a todos os itens que tem o X na cabeça, e adicionamos ao conjunto de lookahead desses itens os elementos de FIRST(βL). Note que no caso em que β = ε, o conjunto de lookahead a adicionar é L.

Repetimos este procedimento até atingirmos uma situação em que os conjuntos de lookahead já não são mais alterados (isto é, atingimos um ponto fixo).

Agora vamos explicar como se cria qualquer estado que não é inicial. Se um estado não é inicial, então pode ser alcançado através de uma transição de um outro estado (que vamos designar de estado anterior), usando um símbolo Σ terminal ou não terminal.

O núcleo do novo estado é constituído por todos os itens do estado anterior que têm o símbolo Σ à direita do ponto, portanto com a forma [A ->α.Σβ, L], sendo feita a adaptação de avançar o ponto para depois do Σ, obtendo a forma [A ->αΣ.β, L]. Obtido o núcleo, o cálculo é igual ao que se já explicou antes.

Comparando o SLR(1) com o LR(1): Nos dois casos, ao atingir um estado com um item completo e pelo menos outro item em conflito utiliza-se os conjuntos de lookahead para resolver o conflito.

A diferença entre as duas técnicas na resolução de conflitos é no conjunto de lookahead. O SLR(1) utiliza o conjunto FOLLOW(A) para todos os itens completos na forma [A ->αβ.]. Para o conflito redução-redução, compara o conjunto FOLLOW calculado para um item completo com outros conjuntos FOLLOW de outros itens completos no mesmo estado. Para o conflito transferência-redução, compara com os símbolos terminais à direita do ponto, para itens na forma [X -> α.β].

O LR(1) utiliza o conjunto de lookahead de cada item para resolver os possíveis conflitos. Desta forma, em vez de se comparar os conjuntos FOLLOW com outros conjuntos FOLLOW, para conflitos redução-redução, ou os conjuntos FOLLOW com os símbolos terminais à direita do ponto, para itens não completos, basta comparar os conjuntos lookahead dos itens em possível conflito, e se os conjuntos forem disjuntos, então não existe conflito, e no caso contrário, a gramática não será LR(1).

Para exemplificar a execução da técnica LR(1), vamos partir da gramática [S -> X$; X -> aAc; X -> aBd; X -> Bc; A -> z; B -> z] que provamos não ser SLR(1) devido a existirem conflitos que o SLR(1) não conseguia resolver.

Partimos do estado 0, com item do núcleo [S -> .X , {$}]. Como X é não terminal, obtemos os itens com cabeça X, [X ->.aAc , {$}], [X -> .aBd , {$}], [X -> .Bc , {$}], com conjunto de lookahead {$}, obtido através do conjunto de lookahead {$} do item do núcleo. Repetimos este procedimento para todos os símbolos não terminais encontrados, ou seja o símbolo B no item [X -> Bc , {$}], obtendo o novo item [B -> .z , {c}], com conjunto de lookahead igual a FIRST(c). Já não encontramos mais símbolos não terminais, e como tal, calculamos todos os itens para o fecho do estado 0.

Se o próximo símbolo esperado for a, então o núcleo do estado correspondente 4 irá conter os itens [X -> a.Ac , {$}] e [X -> a.Bd , {$}], que foram obtidos através dos itens [X -> a.Ac , {$}] e [X -> a.Bd , {$}] do estado 0, onde se avançou o ponto, e herdou-se o conjunto de lookahead.

Repetindo este exemplo para todos os estados, obtemos o diagrama LR(1) seguinte:

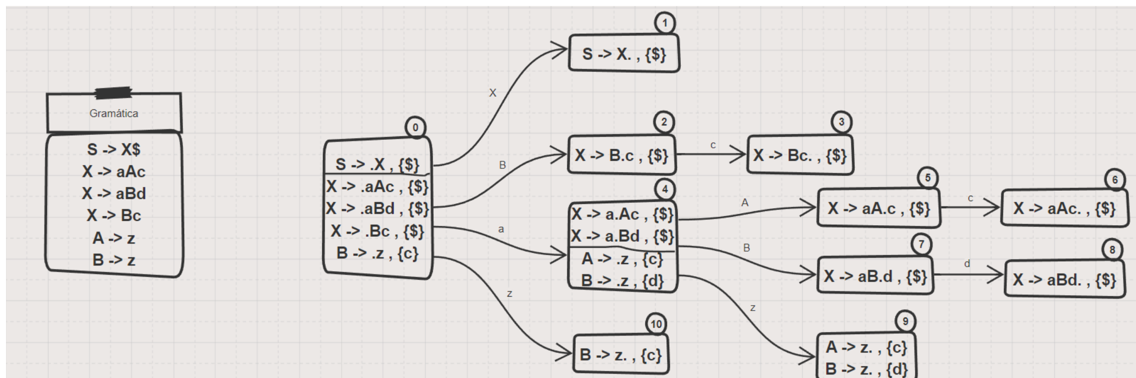


Figura 21:Diagrama LR(1) consegue resolver os conflitos

Repare no item completo [B -> z. , {c}] do estado 10, e no item completo [B -> z. , {d}] do estado 9. Na tabela SLR(1), que pode ser observada atrás, estes dois itens têm o mesmo lookahead, o conjunto FOLLOW(B). No diagrama LR(1), como se vê, o conjunto de lookahead ficou diferente nos dois itens.

Eis a resultante tabela de análise sintática LR(1):

Estado	a	c	d	z	$	A	B	X	Ação
0	5			11			3	1	Transferência
1					Acc				Aceitação
2		4							Transferência
3					R3				Redução X -> Bc
4				10		6	8		Transferência
5		7							Transferência
6					R1				Redução X -> aAc
7			9						Transferência
8					R2				Redução X -> aBd
9		R4	R5						Redução A -> z e B -> z
10		R5							Redução B -> z

Figura 22:Tabela de parsing LR(1)

Como podemos ver, o conflito que o SLR(1) não conseguia resolver para o estado 10, já não é problema nesta tabela de análise de sintática LR(1). A célula do estado 9 para o símbolo terminal c já só contém uma única redução. Aliás, podemos observar que a tabela de análise sintática não tem conflitos, e como tal, estamos perante uma gramática LR(1).

É muito frequente o diagrama LR(1) ficar com mais estados do que o diagrama SLR(1). Mas nem sempre acontece, como se vê no nosso exemplo. Mas, quando o número de estados aumenta, os estados adicionais são uma repetição de estados que já existem, apenas com diferenças nos conjuntos de lookahead.

4.4 LALR(1)

O parser LALR(1) é uma versão simplificada do LR(1) com o objetivo de gerar parsers com menos estados, que consomem menos memória[9]. Começa-se por construir o diagrama dos itens LR(1). A seguir, faz-se uma fusão de todos os grupos de estados cujos itens são iguais, exceto nos conjuntos de lookahead. Se o diagrama resultante não tiver conflitos, diz-se que o parser é LALR(1) e que a própria gramática original também é LALR(1)

Como tal, a técnica LALR(1) é menos poderosa que a técnica LR(1), mas se for aplicável para uma gramática LR(1), conseguindo resolver todos os conflitos, então obtém-se uma tabela de análise sintática mais pequena, assim produzindo um parser mais eficiente.

Para exemplificar, partindo da gramática [S -> X$; X -> CC; C -> cC; C-> d], construímos o seguinte diagrama LR(1):

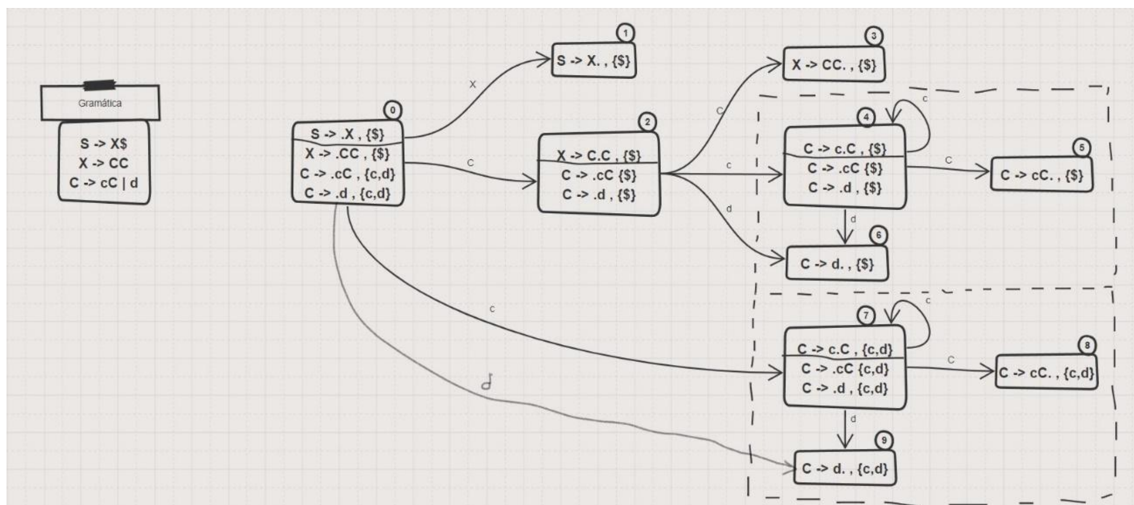


Figura 23:Diagrama LR(1), realçando os estados sujeitos a fusão

Note as duas caixas a tracejado, indicando que o conjunto de estados 4,5,6 tem os mesmos corpos e cabeça nos seus itens que o conjunto de estados 7,8,9. Como têm conjuntos de lookahead diferentes, o LR(1) cria estados diferentes para os dois conjuntos de estados. O LALR(1) tem o objetivo de simplificar diagramas LR(1)[11], sendo aplicável se a tabela de análise sintática resultante não conter conflitos que o LALR(1) não consegue resolver.

Aplicando a fusão referida anteriormente, obtemos o seguinte diagrama LALR(1):

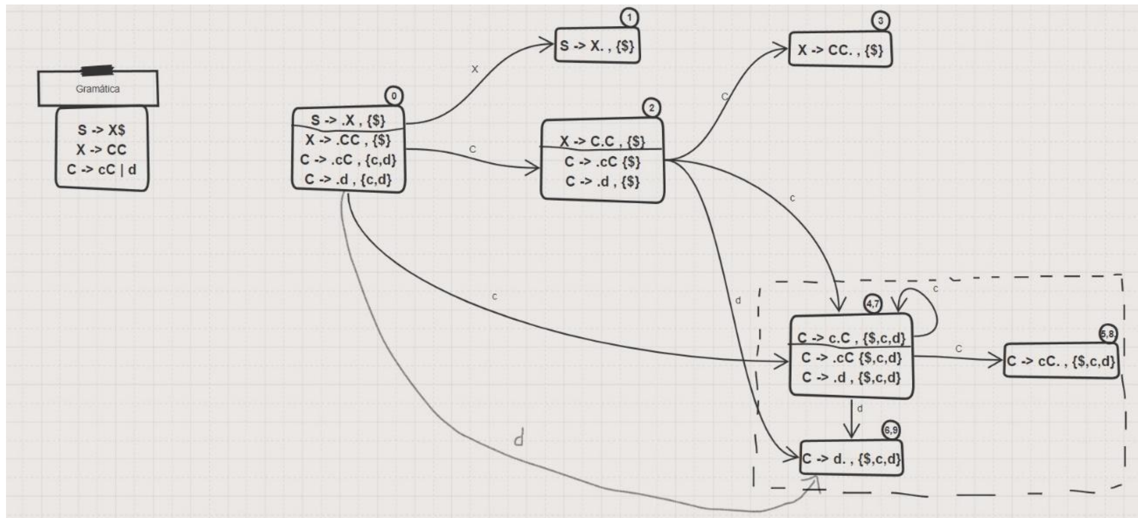


Figura 24: Diagrama LALR(1) resultante da fusão de estados

O estado 4 foi fundido com o estado 7, o 5 com o 8, e o 6 com o 9. O conjunto de lookahead para os itens dos estados de fusão (4,7, 5,8 e 6,9) é a união dos conjuntos dos itens pré-fusão respetivos. As transições do estado 0 para o estado 7 e estado 9 agora ocorrem para o estado 4,7 e o estado 6,9. Aplicamos o mesmo procedimento às transições do estado 2 para estados a ser fundidos. Eis a tabela de análise sintática resultante:

Estado	c	d	C	X	$	Ação
0	4,7	6,9	2	1		Transferência
1					Ac c	Aceitação
2	4,7	6,9	3			Transferência
3					R1	Redução X -> CC
4,7	4,7	6,9	5,8		6	Transferência
5,8	R2	R2			R2	Redução C -> cC
6,9	R3	R3			R3	Redução C -> d

Figura 25:Tabela de parsing para o diagrama LALR(1) da Figura 25

Como a tabela de análise sintática LALR(1) não tem conflitos, não só provamos que a gramática anterior era LR(1), como também provamos que era LALR(1) obtendo uma tabela de análise sintática mais eficiente que a tabela de análise sintática LR(1) que seria obtida pela mesma gramática, visto que o parser iria ter menos estados para computar.

5

OCaml-FLAT

5.1 Introdução ao OCaml-FLAT

O OCaml-FLAT está implementado na sua totalidade em OCaml. Trata-se de uma biblioteca que tem suporte para conceitos FLAT.

Uma particularidade da arquitetura da biblioteca, é que cada categoria de modelo FLAT é implementada num módulo[26, 27] que, além de diversos métodos auxiliares, contém uma classe pública no interior. Essa classe é maior conveniência porque fornece uma forma de acesso ao código usando polimorfismo de inclusão, algo que se revela necessário em diversas situações.

Faz-se uma breve apresentação das componentes, já existentes, da biblioteca OCaml-FLAT que são mais relevantes para esta dissertação:

Entity: Designa-se por Entity qualquer elemento de alto nível que a aplicação suporta. É um módulo abstrato que fatoriza todas as funcionalidades comuns aos tipos de Entidades criadas no OCaml-FLAT. Cada Entidade irá conter o nome do tipo, uma descrição e o nome da entidade. É aqui que se implementa um "construtor" abstrato e flexível que permite receber argumentos de vários tipos.

Exercise: É um módulo que define os exercícios pedagógicos criados no OCaml-FLAT. Estendendo o módulo abstrato Entidade, conseguimos obter por herança funcionalidades como validação, análise de erros e converter para JSON[23]. Estes exercícios são constituídos por uma string que contém o enunciado do problema, um conjunto de testes unitários positivos que correspondem a soluções corretas, um conjunto de testes unitários negativos que correspondem a soluções erradas e as propriedades que a solução deve respeitar.

Model: É um módulo abstrato que fatoriza todas as funcionalidades comuns aos tipos de Modelos criados no OCaml-FLAT. Como exemplos de dois métodos que são introduzidos a este nível de abstração temos o método abstrato accept, sem implementação, que permitirá verificar se uma dada palavra é compatível com o modelo; e temos também o método concreto checkExercise que verifica se o modelo é compatível com um dado exercício, chamando o método accept.

Regular Expression: É um módulo concreto que contém a representação de todas as funcionalidades de uma expressão regular.

Context-Free Grammars: É um módulo concreto que contém a representação de todas as funcionalidades de gramáticas livres de contexto.

Finite Automaton: É um módulo concreto que contém a representação de todas as funcionalidades de autómatos finitos.

5.2 Módulo ContextFreeGrammar

O módulo ContextFreeGrammar, previamente existente, corresponde às funcionalidades de suporte para gramáticas livres de contexto. Possui funcionalidades de suporte para gramáticas nomeadamente: construção da representação interna da gramática, operação de aceitação, geração de palavras, simplificação, cálculo dos conjuntos FIRST e FOLLOW, etc...

Inicialmente desenvolvido como um módulo isolado para gramáticas livres de contexto, este módulo sempre teve como objetivo as funcionalidades essenciais de suporte a gramáticas livres de contexto.

Com a introdução de técnicas de parsing novas, como o LL parsing e o LR parsing, tornou-se conveniente criar módulos separados para organizar essas novas técnicas. Normalmente, tende-se a colocar todas as funcionalidades aplicáveis a uma "entidade" no mesmo módulo, e assim não haveria módulos específicos para as técnicas LL e LR. No entanto, por uma questão de prática de gestão do sistema, foram criados módulos separados para desenvolver estas técnicas.

A solução desenvolvida no OCaml-FLAT utiliza uma hierarquia entre vários tipos de gramáticas livres de contexto. Partimos de um módulo genérico "base" de gramáticas independentes de contexto ContextFreeGrammar, o módulo de nível mais baixo que contém a

definição das gramáticas de contexto e funções aplicáveis a qualquer gramática independente de contexto.

Depois temos o módulo LL1Grammar, um módulo que estende o módulo base com funcionalidades de LL(1) Parsing. A seguir, temos o módulo LRGrammar, que contém as várias técnicas de LR parsing referidas anteriormente. E finalmente, temos o módulo ContextFreeGrammarFull, um módulo cujo objetivo é conter todas as funcionalidades aplicáveis a gramáticas livres de contexto. Como tal, mesmo que sejam introduzidos novos módulos a esta hierarquia, este módulo estará sempre no final desta hierarquia.

5.2.1 Módulo LRGrammar

Este é o módulo central para LR parsing, que foi decomposto em quatro módulos auxiliares: LR0Grammar, SLR1Grammar, LR1Grammar e LALR1Grammar.

5.2.2 Módulo LR0Grammar

5.2.2.1 Definição de conceitos do LR(0) parsing

Começando com o conceito de granularidade mais fina das gramáticas LR, o item LR(k). A definição de um item LR(k) depende da técnica a ser aplicada, onde ambos os LR(0) e SLR(1) utilizam itens LR(0), enquanto o LR(1) e o LALR(1) utilizam itens LR(1).

No caso do LR(0), definimos uma cabeça, e dois "corpos". A cabeça corresponde ao símbolo não terminal que se obtém quando se reduz o corpo do item. Os dois corpos criados resultam do corpo da produção original de uma dada regra gramatical, dividido em duas partes, uma parte que contém os símbolos do contexto esquerdo relevante (Prefixo viável) e a outra parte que contém os símbolos restantes para poder realizar a redução de um item à cabeça.

A seguir, definimos um estado LR(0), que consiste num conjunto de itens LR(0), e um diagrama LR(0), com tipo lr0Diagram, constituído pelo conjunto de todos os estados LR(0), mais as transições que são um conjunto de triplos do tipo: estado LR(0) * símbolo * estado LR(0).

```
1.  type lr0Item = {head:symbol; body1:symbol list; body2:symbol list}
2.  type lr0State = lr0Item set
3.  type lr0Diagram = lr0State set * (lr0State * symbol * lr0State ) set
```

Note que os estados do diagrama não têm identificador associado. A criação do diagrama LR(0) é um procedimento complexo e os identificadores não têm qualquer relevância nessa fase. No entanto, na fase seguinte à criação do diagrama, é necessário associar identificadores aos estados.

O diagrama LR(0) final, com tipo lr0DiagramId, é constituído pelo conjunto de todos os estados LR(0) e os seus identificadores, mais um conjunto de triplos na forma (estado LR(0) * símbolo * estado LR(0)). Este conjunto de triplos corresponde às transições entre estados aplicáveis em cada estado.

```
1.  type stateName = string
2.  type lr0StateId = stateName * lr1State
3.  type lr0DiagramId = lr0StateId set * (lr0StateId * symbol * lr0StateId ) set
```

E para finalizar a descrição dos conceitos da técnica LR(0), falta referir a tabela de parsing LR(0). Esta tabela, com tipo lr1Table, é constituída por um conjunto de entradas da tabela LR(0), onde temos uma entrada por cada estado LR(0). Cada entrada é constituída por o identificador do estado correspondente, um conjunto de pares (símbolo * identificador de estado) que indicam qual será o estado de transição ao encontrar um dado símbolo terminal, e uma ação LR(0) que indica se um dado estado irá realizar uma ação de Transferência/Redução/Aceitação.

```
1.  type lr1Action = Accept | Shift | Reduce of rule
2.  type lr1TableEntry = stateName * (symbol * stateName) set * (symbol * lr1Action set ) set
3.  type lr1Table = lr1TableEntry set
```

5.2.2.2 Criação do diagrama LR(0)

O primeiro passo é criar o estado inicial do nosso futuro diagrama. É a partir deste estado que serão gerados os restantes estados do diagrama LR(0). Para criar o estado inicial, a seguinte função vai buscar todas as regras da gramática aplicáveis ao símbolo inicial e converte-as em itens LR(0) com prefixo viável vazio. A partir desse conjunto de itens, calculamos o fecho. Desta forma obtemos o estado inicial completo com todos os seus itens LR(0). Note que ainda não existem transições e como tal, são representadas com o conjunto vazio.

```
1.   let makeFirstLR0Diagram (cfg:t) : lr0Diagram =
2.             let kernel = Set.map rule2Item (startRules cfg) in
3.             let closure = lr0StateClosure cfg kernel in
4.             (Set.make [closure], Set.empty)
```

A geração do fecho já foi descrita no capítulo da teoria, e aqui faz-se apenas um resumo. Percorremos os itens existentes no estado e calculamos os símbolos diretores desses itens. Os símbolos diretores permitem deduzir novos itens LR(0) no caso em que um dos itens espere um símbolo não terminal que seja cabeça de uma regra da gramática. Se for possível obter novos itens, então acrescentamos os itens ao estado. Repetimos este procedimento até não aparecerem itens novos.

```
1.   let lr0StateClosureStep (cfg: t) currentItems =
2.    let directors = getDirectors currentItems in
3.    let varDirectors = Set.inter directors cfg.variables in
4.    let newRules = Set.flatMap (fun d -> getRulesWithThisHead cfg.rules d) varDirectors in
5.    let newItems = Set.map rule2Item newRules in
6.              Set.union currentItems newItems
7.
8.
9.    let rec lr0StateClosure cfg currentItems : lr0State =
10.             let next = lr0StateClosureStep cfg currentItems in
11.             let next2 = Set.union currentItems next in
12.                       if Set.size next2 = Set.size currentItems then next2
13.                       else lr0StateClosure cfg next2
```

A partir do estado inicial, conseguimos obter o diagrama LR(0) completo da forma que se explica a seguir. O estado inicial é usado como semente. A partir dele todos os outros estados são gerados incrementalmente através da função makeNextLR0Diagram. Este processo é concluído quando já não surge nenhum novo estado.

```
1.  let makeSingularNextLR0Diagram (cfg:t) prevState symbol : lr0Diagram =
2.     let items4Kernel = Set.filter (fun it -> getDirector it = symbol) prevState in
3.     let kernel = Set.map (kernelAdvanceItem) items4Kernel in
4.     let closure = lr0StateClosure cfg kernel in
5.              (Set.make [prevState; closure], Set.make [(prevState ,symbol , closure )])
6.
7.    let makeNextLR0Diagram (cfg:t) prevState : lr0Diagram =
8.     let dirs = getDirectors prevState in
9.     let diagrams = Set.map (fun d -> makeSingularNextLR0Diagram cfg prevState d) dirs in
10.             diagramsJoinList (Set.toList diagrams)
```

Sempre que encontramos estados novos, também adicionamos ao diagrama as transições correspondentes. A função principal que controla este processo chama-se makeLR0DiagramX.

```
1.   let rec makeLR0DiagramX (cfg:t) diagram =
2.    let (states,transitions) : lr0Diagram = diagram in
3.    let next = makeNextLR0DiagramAll cfg states in
4.    let next2 = diagramsJoin2 next (states,transitions) in
5.    let (states2,transitions2) = next2 in
6.              if Set.size states = Set.size states2 && Set.size transitions = Set.size
    transitions2 then next2
7.              else makeLR0DiagramX cfg next2
```

Existe uma função que não é aqui mostrada, que associa identificadores únicos aos vários estados do diagrama. Os diagramas com identificadores são necessários para a próxima fase.

5.2.2.3 Criação da tabela de parsing LR(0)

Para criar a tabela, criamos uma linha por cada estado no diagrama, usando a função makeLR0Table. Para cada linha, vemos se existe uma transição aplicável ao estado correspondente a essa linha. Se não existirem transições possíveis no diagrama LR(0), não iremos colocar estados nas células interiores da linha e sabemos que estamos perante uma redução ou aceitação. Caso contrário, completa a linha com os respetivos estados a obter. Em ambos os casos, coloca-se a ação no final da linha.

```
1.  let makeLR0TableEntry (id, lr0State) (cfg:t) transitions =
2.    let stateTransitions = Set.filter (fun ((a,_),_,_)-> a = id) transitions in
3.     if Set.size stateTransitions = 0 then
4.              let {head = h;body1 = b1;body2 = b2} = List.hd (Set.toList lr0State) in
5.                        if h = cfg.initial then
6.                                  (id,Set.empty,Accept)
7.                        else
8.                                  (id,Set.empty,Reduce ({head = h;body = b1}))

9.     else
10.             let nextShifts = Set.map (fun (a,b,(cId,c)) -> (b,cId)) stateTransitions in
11.                       (id, nextShifts, Shift)
12.
13.   let makeLR0Table (labeledDiagram:lr0DiagramId) cfg : lr0Table =
14.    let (statesId, transitionsId) = labeledDiagram in
15.             Set.map (fun s -> makeLR0TableEntry s cfg transitionsId) statesId
```

Recapitulando, a seguinte linha de código mostra como os vários processos que referimos contribuem para a criação da tabela final.

```
1.  let lr0Table = makeLR0Table (makeLR0DiagramId (makeLR0Diagram cfg)) cfg in …
```

A explicação desta linha de código é simples: Partindo de uma gramática livre de contexto, geramos o diagrama (usando a função makeLR0Diagram). Após gerar o diagrama, atribuímos um identificador a cada um dos estados do diagrama (usando a função makeLR0DiagramId) e geramos a tabela com o diagrama com os novos identificadores (usando a função makeLR0Table).

5.2.2.4 Aceitar palavras LR(0)

Vamos agora descrever como usar a tabela de parsing para concretizar o reconhecimento determinista bottom-up de uma palavra. O procedimento de reconhecimento está implementado na função acceptWordLR0Table. Esta função tem como pré-requisito a disponibilidade da tabela LR(0) já criada.

A pilha do nosso autómato é constituída por símbolos e estados dispostos alternadamente. No início, esta pilha contém apenas o estado inicial.

```
2.  let acceptWordLR0WithTable (word:symbol list) lr0Table cfg : bool =
3.    let revStack = ["0"] in
4.        parseOperation lr0Table (word @ [dollar]) stateRevStack symbolRevStack cfg
```

Cada passo da operação de parsing consiste em ler o estado no topo da pilha (começa com o estado 0), e ir buscar essa entrada correspondente na tabela de estados. Com essa entrada, analisamos a ação LR(0) a tomar.

Se for Accept, verificamos se atingimos o final da palavra, e se tal aconteceu, então a palavra é aceite.

Se for Reduce, temos de consumir elementos da pilha para trocá-los pelo símbolo na cabeça. O número de elementos consumidos será igual ao dobro do número de símbolos no corpo da regra da gramática que está a ser utilizada para a redução. Vamos tomar como exemplo o seguinte caso: Se fizermos uma redução correspondente ao item completo [A -> ab.] assumindo uma pilha na forma [0c1a3b5], iremos desempilhar 4 elementos. Neste exemplo, iríamos desempilhar o id do estado atual 5, o símbolo b, o id do estado anterior 3, que realizou a transferência do símbolo b, e o símbolo a. Ou seja, o conteúdo da pilha após esta redução será [0c1], indicando que o estado no nosso processo de parsing será o estado 1.

Se for Shift, temos de ver se existe uma transição aplicável. Se não existir, a palavra é rejeitada. Se existir, consumimos o símbolo corrente da palavra e empilhamos na pilha esse símbolo juntamente com o estado associado a esta operação de Shift.

```
1.  let rec parseOperation lr0Table word revStack (cfg:t) =
2.    let currentState = int_of_string (List.hd stateStack) in
3.    let (id,shifts,action) = Set.nth lr0Table currentState in
4.    match action with
5.    | Shift ->
6.            begin
7.            match word with
8.            | [] -> false
9.            | s::_ ->
10.           If Set.belongs s cfg.alphabet || Set.belongs s cfg.variables then
11.                   let targetShifts = Set.filter (fun (a,b) -> a = s) shifts in
12.                   If Set.size targetShifts = 0 then false
13.                   else
```

```
14.                     let (nextSymbol,nextState) = Set.nth targetShifts 0 in
15.                     let nextRevStack = [nextState] @ [symb2str nextSymbol] @ revStack
    in
16.
                                 parseOperation lr0Table (getTail word) nextRevStack cfg

17.           else
18.                     false
19.                     end
20.           | Accept -> word = [dollar]
21.           | Reduce({head = h;body = b}) ->
22.                     let popNumber = List.length b in
23.                     let nextRevStack = pop (popNumber*2) revStack in
24.                     let wordWithAddedHead = [h] @ word in
25.                               parseOperation lr0Table (wordWithAddedHead) nextRevStack
    cfg
```

5.2.3 Módulo SLR1Grammar

5.2.3.1 Definição de conceitos do SLR(1) parsing

Como referido no capítulo da teoria, a técnica de parsing SLR(1) também se baseia na utilização de um diagrama LR(0). Como tal, reaproveitamos a implementação anterior para a criação do diagrama. A novidade é que agora vão considerar a possibilidade de haver conflitos, e vamos tentar lidar com eles.

A tabela de parsing SLR(1) usa tokens de lookahead para resolver conflitos, e assume exatamente a mesma forma das tabelas LR(1).

O tipo da tabela de parsing SLR(1) estende o tipo da tabela de parsing LR(0). Cada entrada da tabela SLR(1) substitui a ação (unica) por um conjunto de ações que no momento do accept serão escolhidas em função do símbolo lookahead. Como tal, a ação LR(0) foi substituída por um conjunto de pares na forma (símbolo * conjunto de ações SLR(1)) para um dado símbolo terminal de lookahead.

Uma entrada da tabela pode ter 0,1 ou mais ações aplicáveis: Se > 1, trata-se de um conflito que não pode ser resolvido com o SLR(1); Se = 0 trata-se de uma rejeição.

```
1.   type slr1Action = Accept | Shift | Reduce of rule
2.   type slr1TableEntry = stateName * (symbol * stateName) set * (symbol * slr1Action set )
    set
3.   type slr1Table = slr1TableEntry set
```

Uma parte importante desta técnica é o passo para criar entradas de tabelas LR(1) com itens LR(0). Itens LR(0) não possuem lookahead, e como tal não é possível descobrir que símbolos se podem esperar no próximo passo do parsing LR. A técnica SLR(1) tenta conciliar esta deficiência ao tentar deduzir quais os possíveis símbolos que se podem esperar.

Como não temos tokens de lookahead, a única hipótese é utilizar as regras da gramática para obter esta informação. Como todos os conflitos possíveis têm pelo menos uma redução, iremos utilizar o conjunto FOLLOW(A), sendo A o símbolo não terminal que serve de cabeça para a regra correspondente às reduções em conflito. Este conjunto FOLLOW(A) será utilizado para criar o conjunto de ações SLR(1) para cada possível símbolo de cada estado, assim simulando o lookahead utilizado nas tabelas LR(1). Esta abordagem permite gerar tabelas com o mesmo formato para o SLR(1), LALR(1) e LR(1), que correspondem às técnicas de LR parsing em ordem crescente de capacidade para resolver conflitos.

5.2.3.2 Criação de ações SLR(1)

A criação de ações SLR(1) pega em itens LR(0), produz um lookahead para cada um desses itens e sucessivamente cria as ações SLR(1) para cada símbolo aplicável. Este processo pode ser dividido em duas partes, itens completos e itens de transferências.

```
1. let buildSLR1MixedActionsForOne items symbol cfg=
2.             let reductionItems = Set.filter(fun it -> (Set.belongs symbol (followSet-
   ForSLR1Item it cfg)) && isCompleteLR0Item it) items in
3.             let shiftItems = Set.filter(fun it -> (getNextSymbolForLR0Item it) = symbol)
   items in
4.             let reductionEntries = Set.map (fun it -> Reduce ({head = it.head;body =
   it.body1}) ) reductionItems in
5.                       if(Set.size shiftItems > 0) then
6.                                 Set.union (Set.make [Shift]) reductionEntries
7.                       else
8.                                 reductionEntries
```

Para construir as ações de transferência, temos de identificar o próximo símbolo esperado por transferência desse item. Por exemplo, para o item A -> a.b, o próximo símbolo esperado por transferência é o símbolo b. Então este item indica que para o símbolo b, existe uma ação de transferência. Se também existirem ações de redução para este mesmo símbolo, então estamos perante um conflito.

Para construir as ações de redução, temos de identificar todos os itens completos e obter as cabeças da redução desses itens. Depois, calculamos o conjunto FOLLOW para cada uma dessas cabeças. Depois criamos uma ação SLR(1) num dado símbolo para cada redução cujo conjunto FOLLOW espera esse símbolo. Por exemplo, para um dado conjunto FOLLOW(A) = {a,b,c}, temos de criar uma ação SLR(1) para cada um desses símbolos. Cada ação irá conter a regra da redução correspondente ao item completo em questão.

Neste caso, iremos criar um total de 3 ações SLR(1) para os 3 símbolos terminais {a,b,c}, que corresponde ao conjunto lookahead que nós estamos a emular.

Note que se existir outra ação para esse mesmo símbolo, quer seja uma outra redução ou uma transferência, iremos estar perante um conflito.

5.2.3.3 Criação da tabela de parsing SLR(1)

A criação da tabela SLR(1) tem vários pontos comuns à criação da tabela LR(0). No entanto, o LR(0) cria apenas uma ação por linha da tabela, enquanto que agora vamos criar um conjunto de ações por símbolo, contido dentro de cada linha da tabela.

Utilizando o procedimento de criação de ações referido na secção anterior, conseguimos gerar os conjuntos de ações que vão ser associados a cada símbolo esperado em cada estado. Podemos dizer que o grande progresso entre criar diagramas LR(0) e as tabelas SLR(1) é na produção destes conjuntos de ações.

```
1.  let makeSLR1TableEntry (id, lr0State) (cfg:t) transitions =
2.    let stateTransitions = Set.filter (fun ((a,_),_,_)-> a = id) transitions in
3.              …
4.    if Set.size stateTransitions = 0 then
5.              let {head = h;body1 = b1;body2 = b2} = List.hd (Set.toList lr0State) in
6.                        if h = cfg.initial then
7.                                  let slr1Actions : (symbol * slr1Action set) set =
    Set.make [dollar,Set.make [Accept]] in
8.                                            (id,Set.empty,slr1Actions)
9.              else
10.             …
11.             else
12.                       let completeAlphabet = Set.add dollar cfg.alphabet in
13.                       let slr1Actions = buildSLR1MixedActions lr0State completeAlphabet
    cfg in
14.                                 (id, nextShifts, slr1Actions)
15.
16. let makeSLR1Table (labeledDiagram:lr0DiagramId) cfg : slr1Table =
17.             let (statesId, transitionsId) = labeledDiagram in
18.                       Set.map (fun s -> makeSLR1TableEntry s cfg transitionsId) statesId
```

Em suma, as diferenças entre o LR(0) e as restantes técnicas LR(k) exploradas no processo de criação de tabelas são as seguintes:

Temos de criar conjuntos de ações de cada estado para cada símbolo não terminal do nosso alfabeto, algo que estamos a realizar com a função buildSLR1MixedActions, que recebe um estado LR(0), produz um lookahead para os itens do estado e utiliza esse lookahead para gerar os conjuntos de ações.

Tendo sido desenvolvida a função auxiliar anterior, utilizamos o processo recursivo de criação de tabela para aplicar a nossa função sobre cada estado caso seja apropriado. Existem casos especiais, como o Accept, que não requerem uma função auxiliar para calcular o conjunto de ações, porque um item com ação de aceitação só pode ter um possível lookahead

para uma palavra válida, o símbolo $. Caso contrário, atingimos o final do parsing e não consumimos a palavra na sua totalidade, ou seja, a palavra é inválida.

5.2.3.4 Aceitar palavras SLR(1)

Tal como no LR(0), a função abaixo serve como um resumo dos procedimentos necessários para a técnica SLR(1). Partimos da construção do diagrama LR(0) e usamos esse diagrama para produzir a nossa tabela SLR(1). Agora que temos a tabela, podemos verificar com o nosso parser SLR(1) se uma dada palavra pertence à nossa gramática.

```
1.  let acceptWordSLR1 (word:symbol list) slr1Table cfg: bool =
2.              let revStack = ["0"] in
3.                        parseOperationSLR1 slr1Table (word @ [dollar]) revStack cfg
```

As operações de parsing agora são consideravelmente diferentes, mas não iremos explorar estas operações nesta fase, pois como estamos a tentar emular a técnica de parsing LR(1) com itens LR(0) é importante explorar a implementação LR(1). O procedimento de validar palavras LR(1) é o mesmo que o procedimento do SLR(1). Isto acontece porque temos o mesmo formato de tabelas para o SLR(1), LR(1) e LALR(1), e assim conseguimos ter uniformidade na técnica de aceitação. Assim, as implementações destas técnicas têm o mesmo objetivo de gerar esta tabela comum, e evitam confusões nas implementações destas técnicas. O LR(0) é a única técnica de parsing LR desenvolvida com um formato diferente, devido à falta de lookahead.

5.2.4 Módulo LR1Grammar

5.2.4.1 Definição de conceitos do LR(1) parsing

No caso dos itens LR(1), definimos uma cabeça, dois corpos tal como no caso dos itens LR(0), mas agora acrescentamos também um conjunto de símbolos lookahead, pois os itens LR(1) utilizam-nos para tentar resolver conflitos.

```
1.  type lr1Item = {head:symbol; body1:symbol list; body2:symbol list; lookahead:symbols}

2.  type lr1State = lr1Item set
3.  type lr1Diagram = lr1State set * (lr1State * symbol * lr1State ) set
```

O tipo do diagrama LR(1) é muito parecido com o tipo do diagrama LR(0) onde a única diferença é que cada estado tornou-se num conjunto de itens LR(1). É relevante notar que diagramas LR(1) podem gerar mais estados que os diagramas LR(1), pois agora é possível ter dois estados onde a única diferença é num conjunto de lookahead de um item LR(1).

E para finalizar a definição da técnica LR(1), criamos a tabela de parsing LR(1). Esta tabela tem a mesma definição que a tabela de SLR(1).

5.2.4.2 Criação do diagrama LR(1)

Como estamos a trabalhar com conjuntos de itens LR(1), é possível acontecer casos onde obtemos itens com corpos iguais e lookaheads diferentes. Como tal, quando estamos a preencher um dado estado ao gerar os itens LR(1) correspondentes, iremos ter de fundir esses itens e obter um lookahead correspondente a união dos lookaheads de itens com corpos equivalentes. Existem bastantes funções auxiliares que não vão ser aqui discutidas, mas merecem este pequeno reconhecimento, porque a manipulação de itens LR(1) num dado estado requer vários pequenos procedimentos para obter o fecho completo de um estado LR(1).

Relembrando o LR(0), repetia-se o procedimento de gerar itens LR(0) até não ser possível gerar mais itens. Agora podemos ter passos onde não se acrescentaram novos itens, mas podemos ter gerado um conjunto lookahead maior para um dado item. Assim, temos de guardar o estado num passo anterior e comparar para verificar se realmente já desenvolvemos o fecho completo do estado LR(1).

```
1.  let rec lr1StateClosure cfg currentItems : lr1State =
2.              let next = lr1StateClosureStep cfg currentItems in
3.                        if Set.subset next currentItems then next
4.                        else lr1StateClosure cfg next
```

Temos de comparar os dois estados, e se forem equivalentes, então já não é possível produzir alterações no estado atual.

5.2.4.3 Criação da tabela de parsing LR(1)

Este procedimento de criação de tabela é praticamente equivalente ao procedimento de criação de tabela do SLR(1). Como os itens LR(1) têm os seus próprios conjuntos de lookahead definidos, o processo de criação de tabela LR(1) é ligeiramente mais simples que o do SLR(1). A diferença nestes procedimentos é que o SLR(1) requer um passo adicional que "processa" itens LR(0) como se fossem SLR(1), utilizando o cálculo dos conjuntos FOLLOW para produzir um lookahead, apesar desse lookahead ser potencialmente menos preciso.

5.2.4.4 Aceitar palavras LR(1)

Este procedimento da validação de palavras partilha vários passos com LR(0). Utilizamos uma pilha que serve de memória para o nosso autómato, uma ação de transferência irá acrescentar o símbolo transferido e estado resultante à pilha, uma ação de redução irá consumir n símbolos do topo da pilha, sendo n igual ao dobro do número de símbolos no corpo da regra da redução utilizada. A ação de aceitação irá terminar o procedimento com sucesso.

No entanto, ao contrário do LR(0), agora iremos "espreitar" o próximo símbolo para tomar decisões, o que permite atribuir ações a cada símbolo terminal, algo que seria impossível no LR(0) e que devido aos limites da técnica só conseguia atribuir uma ação por estado. Iremos então discutir os casos novos que surgem com esta abordagem com auxílio de lookahead.

Se "espreitarmos" um símbolo não terminal (variável), então sabemos que podemos acrescentar esse símbolo à pilha e transitar para o estado correspondente. Este é um caso prioritário, que tem precedência sobre qualquer alternativa de redução ou transferência.

```
1.    let peekedSymbol = List.nth word 0 in
2.              if(Set.belongs topSymbol cfg.variables) then
3.                        let targetShifts = Set.filter (fun (a,b) -> a = topSymbol) shifts
   in
4.                        if(Set.size targetShifts = 0) then false
5.                                  else
6.                                  let (nextSymbol,nextState) = Set.nth targetShifts 0 in
7.                                  let nextRevStack = [nextState] @ [symb2str nextSymbol] @
   revStack in                                                              parseOperationLR1
   lr1Table (getTail word) NextRevStackcfg
```

Caso contrário, temos de analisar o símbolo que espreitamos e verificar as ações aplicáveis para esse símbolo. Se não existirem ações possíveis, a palavra é rejeitada. Se existem 2 ou mais ações possíveis, estamos perante um conflito. Caso haja apenas uma única ação a tomar, então podemos tomar essa ação para o nosso estado atual.

```
1.    else
2.              let peekedsymbolAndActions = Set.filter( fun (s,a) -> s = peekedSymbol &&
   Set.size a > 0 ) actionSet in
3.              let nEntries = Set.size peekedsymbolAndActions in
4.                        if nEntries = 0 then
5.                                  false
6.                        else if nEntries > 1 then
7.                                  failwith "ParseOperationLR1: conflito"
8.                        else
9.                                  let (symbol,actions) = Set.hd peekedsymbolAndActions in
10.                                  let action = Set.hd actions in …
```

O procedimento que acabamos de fazer tem como objetivo tentar resolver possíveis conflitos que o LR(0) não conseguia resolver. A partir desta fase, já ultrapassamos os limites do LR(0) e podemos continuar este procedimento com se fosse o LR(0).

Ou seja, iremos realizar uma Transferência/Aceitação/Redução tal e qual como descrito na secção de aceitar palavras com o LR(0).

5.2.5 Módulo LALR1Grammar

5.2.5.1 Conceitos do LALR(1) parsing

Consideremos uma gramática que já provou ser LR(1), e tomemos o respetivo diagrama LR(1). A técnica LALR(1) consiste em tentar reduzir o número de estados do referido diagrama através da fusão de estados, com objetivo de obter uma otimização do LR(1) para reduzir o número de estados.

O procedimento utilizado para gerar um diagrama LALR(1) consiste em criar um diagrama LR(1) e procurar pares de estados onde os seus respetivos itens têm as mesmas cabeças e corpos.

Se existir tal par, então fundimos o par de estados num único estado onde os itens têm o mesmo corpo e cabeça, mas agora o conjunto de lookahead dos itens é correspondente à união dos conjuntos de lookahead diferentes.

Repetimos este procedimento até não ser possível fundir mais estados.

```
1.    let rec lr1StateFusion states  =
2.              match states with
3.              | [] -> []
4.              | x::xs ->
5.                        let ss = lr1StateFusion xs in
6.                        let (a,b) = List.partition (haveSameCores x) ss in
7.                                  match a with
8.                                  | [] -> x::ss
9.                                  | [y] -> mergeLR1States x y::b
10.                                 | _ -> failwith "lr1StateFusionFail"
11.
12.   let translate state fstates =
13.             Set.find (fun s -> haveSameCores state s) fstates
14.
15.
16.   let lr1TransFusion trans fstates =
17.             Set.map (fun (s1,sym,s2) -> (translate s1 fstates,sym,translate s2 fstates))
    trans
18.
```

É importante notar que a técnica de parsing LALR(1) é igual à técnica do LR(1), porque uma gramática LALR(1) é uma gramática LR(1), apesar do contrário não ser sempre verdade.

6

INTEROPERABILIDADE OCAML/JAVASCRIPT

6.1 Introdução à Interoperabilidade OCaml/JavaScript

Neste capítulo, iremos apresentar o Js_of_ocaml[13], uma solução de interoperabilidade entre OCaml[19] e JavaScript. Resumidamente, o Js_of_ocaml traduz para JavaScript código de uma variante do OCaml que tem pequenas extensões linguísticas. O resultado da tradução fica apto a correr num browser web.

Entrado em maior detalhe, esta solução tem 3 componentes[21]:

Js_of_ocaml-ppx: Um pré-processador que usa o mecanismo de extensibilidade do OCaml, que funciona ao nível da AST dos programas.

Js_of_ocaml library: Pequena biblioteca que introduz bindings para diversas categorias de objetos JavaScript: Os objetos intrínsecos da linguagem JavaScript, os objetos do DOM, e os objetos de mais algumas bibliotecas, por exemplo para registar logs na consola do browser.

Js_of_ocaml compiler: Programa que traduz para código JavaScript, ficheiros de código objeto OCaml com formato bytecode.

É importante descrever, pelo menos sumariamente, como é que se escreve código OCaml que tem por objetivo interagir com o JavaScript.

Vamos analisar o seguinte exemplo, onde iremos obter uma referência de uma tabela HTML existente, criar uma linha nova na tabela e inserir apenas uma célula com a string “New row” nessa linha da tabela.

Se estas ações fossem escritas diretamente em JavaScript, o código seria o seguinte:

```
1.  let table = document.getElementById('tbl');
2.  let newRow = table.insertRow(-1);
3.  let newCell = newRow.insertCell(-1);
4.  let newText = document.createTextNode('New row');
5.      newCell.appendChild(newText);
```

Eis o código OCaml equivalente, que realiza as mesmas ações:

```
1.  let table = Js.Unsafe.coerce (Dom_html.getElementById ("tbl")) in
2.  let newRow = table##insertRow (-1) in
3.  let newCell = newRow##insertCell (-1) in
4.  let newText = Dom_html.document##createTextNode (Js.string "New row") in
5.      newCell##appendChild newText
```

Eis alguns detalhes interessantes deste código OCaml. Os cardinais duplos são uma extensão linguística que serve para chamar métodos de objetos JavaScript e que é tratada pelo pré-processador Js_of_OCaml-ppx. A função Unsafe.coerce é a função identidade no domínio dos objetos JavaScript. Mas ela tem um tipo especial que faz com que o seu resultado seja tratado como um objeto JavaScript de uso irrestrito, para o qual os acessos não são validados. Este módulo Unsafe irá ser explorado em mais detalhe na secção de Módulos deste capítulo.

6.2 Representação de tipos JavaScript em OCaml

O suporte para elementos JavaScript dentro da linguagem OCaml leva à introdução de novos tipos para representar estes elementos. Um objeto básico JavaScript irá ser do representado como α Js.t em OCaml. É possível atribuir uma especificação mais precisa do tipo do objeto JavaScript ao instanciar α com um tipo concreto. Ou seja, um objeto int Js.t corresponde a um inteiro em JavaScript[22].

Certos tipos, tais como int e float, não é necessário converter em objeto JavaScript porque a mesma representação interna é usada nas duas linguagens. Assim, por exemplo, torna-se redundante criar um objeto int Js.t.

No entanto, é preciso invocar métodos de conversão explícita para os outros tipos de objetos JavaScript, como estes exemplos:

bool ---> bool Js.t

string ---> Js.Js_string Js.t

array ---> Js.Js_array Js.t

Objetos JavaScript contêm propriedades e métodos. As propriedades são representadas através do tipo paramétrico Js.prop, com caráter de escrita/leitura. Também é possível definir propriedades para apenas leitura, Js.readonly_prop, e propriedades para apenas escrita, Js.writeonly_prop. Os métodos são definidos usando o tipo paramétrico Js.meth. Em cada uso concreto, estes tipos têm de ser instanciados. Eis alguns exemplos: float Js.prop uma propriedade de escrita/leitura com tipo float, int -> Js.Js_string Js.t Js.meth um método que produz uma string a partir de um inteiro, int -> unit Js.meth um método que não produz resultado e recebe um inteiro como argumento.

Para definir um tipo novo de objeto JavaScript, com vários campos, recorremos ao tipo dos objetos em OCaml e instanciamos o tipo paramétrico Js.t. Eis um exemplo:

```
1.  type myobj =
2.     < number : int Js.prop; append : js_string t -> unit Js.meth> Js.t
```

Neste caso, estamos a definir um tipo de objeto JavaScript que possui uma propriedade de escrita/leitura de tipo int, nomeada de number, e um método que recebe uma string e não devolve nada, nomeado de append.

6.3 Bindings

Quando falamos de interoperabilidade de duas linguagens diferentes, é importante falar sobre a criação de bindings. Os bindings suportados pelo Js_of_ocaml tornam prática a ligação entre as duas linguagens.

Para aceder de forma prática a objetos JavaScript, define-se em OCaml os tipos desses objetos como se fossem objetos OCaml. Esses tipos são usados na validação do código OCaml, concretamente a validação dos acessos às propriedades e aos métodos. No final, o compilador do Js_of_ocaml gera o código JavaScript correspondente a esses acessos.

A biblioteca do Js_of_ocaml já possui vários bindings definidos, por exemplo para os objetos do DOM. Mas podemos querer introduzir os nossos próprios bindings para aceder a

determinadas bibliotecas JavaScript, por exemplo no nosso caso, a biblioteca Cytoscape.js que iremos apresentar no próximo capítulo.

Por exemplo, vamos definir um binding para um tipo de objetos JavaScript que representa arcos de grafos. Atribuímos ao binding o nome edge. Um arco liga dois nós representados por dois inteiros, e o arco tem um certo peso associado. Este peso pode ser alterado com uma função de updateWeight.

Temos aqui o binding correspondente ao exemplo:

```
1.  class type edge =
2.    object
3.              method firstNode : int Js.prop
4.              method lastNode : int Js.prop
5.              method weight : int Js.prop
6.              Method updateWeight : int -> unit Js.meth
7.    end
```

Neste exemplo, os métodos firstNode, lastNode e weight devolvem um inteiro. O método updateWeight recebe um inteiro e não devolve nenhum valor.

6.4 Criar objetos JavaScript em OCaml

Agora que já temos uma binding definida, vamos explorar como criar um objeto JavaScript em OCaml. Existem duas maneiras diferentes de criar objetos JavaScript em OCaml, com construtores ou objetos literais. Abaixo apresenta-se o construtor em JavaScript para o qual foi criado o binding edge referido.

```
1.  function edge(firstNode,lastNode,weight){
2.    this.firstNode = firstNode,
3.    this.lastNode = lastNode,
4.    this.weight = weight,
5.    this.updateWeight = function(newWeight){
6.              this.weight = newWeight;
7.    }
8.  }
```

Para criar a partir do OCaml um objeto usando o construtor edge temos de invocar o construtor JavaScript:

```
1.  type edgeConstr =
2.    (int -> int -> int) Js.constr;
3.  let edgeC: edgeConstr = Js.Unsafe.global##.edge in
4.  let edge = new%js edgeC 1 2 10 in
5.     …
```

O tipo edgeConstr corresponde ao binding que queremos criar. Definimos a variável edgeC, com o construtor de arcos. A seguir, usamos o construtor para criar um arco concreto. Para criar um objeto usando um construtor JavaScript, a partir do OCaml, usa-se a forma "new%js" <tipo do objeto> args.

Para criar objetos JavaScript em OCaml a partir da forma literal temos o seguinte exemplo:

```
1.  let edge = object$js (self)
2.    val mutable firstNode = 1
3.    val mutable lastNode = 2
4.    val mutable weight = 10
5.    method updateWeight x = self##.weight := x
6.   }
7.  }
```

Assim, temos em OCaml um objeto edge que tem o tipo que queremos. Para aceder a propriedades usa-se a notação "##.", e para invocar métodos usa-se a notação "##". Por exemplo, edge##.firstNode vai aceder à propriedade firstNode do objeto JavaScript edge.

6.5 Módulos importantes do Js_of_ocaml

O Js_of_ocaml[22] possui vários módulos com bindings pré-definidos. Nesta secção iremos referenciar os módulos que merecem algum reconhecimento.

6.5.1 Módulo Js

Este módulo contém bindings sobre JavaScript. Várias destas bindings correspondem a definições de tipos JavaScript, úteis para converter variáveis OCaml em variáveis/objetos JavaScript. Por exemplo, Js.string permite converter uma string em OCaml para um string em JavaScript. Em geral, este é o módulo principal quando queremos fazer operações que requerem interagir com JavaScript num ambiente OCaml.

6.5.2 Submódulo Js.Unsafe

Este submódulo contido no módulo Js permite operações sobre o JavaScript que não são consideradas seguras. Tomando como exemplo a função Js.unsafe.coerce na secção 7:

```
1.  let table = Js.Unsafe.coerce (Dom_html.getElementById ("tbl")) in
```

A função Dom_html.getElementById devolve um objeto de tipo Dom_html.element Js.t. Mas note que este é um tipo abstrato e o tipo concreto do objeto será mais específico. Isto leva-nos à função coerce, que será explicada já a seguir.

Esta função “confia” que fomos buscar um objeto com o tipo correto, e muda-lhe o tipo α Js.t, um tipo totalmente permissivo. Por exemplo, se invocarmos um método que não existe no objeto, isso não dá erro de compilação no OCaml; mas quando o código traduzido para JavaScript for executado, vai ocorrer um erro de tipo em tempo de execução.

Uma técnica que ajuda a resolver o problema consiste em introduzir o tipo do objeto pretendido, assumindo que já existe um binding para esse tipo. Mas não nos podemos enganar nesse tipo, pois isso conduziria igualmente à ocorrência de erros de execução do lado do JavaScript. Veja a nova versão com tipificação:

```
1.  let table : Dom_html.tableElement =
2.      Js.Unsafe.coerce (Dom_html.getElementById ("tbl")) in
```

As operações deste submódulo têm de ser usadas com cuidado para evitar erros de execução. Outras operações como unsafe.get e unsafe.set, que correspondem a obter o valor de uma propriedade ou alterar o valor de uma propriedade, podem ser substituídas pela notação “##.” se estivermos a usar bindings.

Também podemos referir a função Js.Unsafe.global que permite obter uma referência à variável global do JavaScript global, que contém todas as definições globais correntemente ativas no JavaScript. É também possível definir variáveis de teste nesta variável global, e aceder a estas variáveis é como aceder a uma propriedade de um objeto JavaScript.

6.5.3 Módulo Dom_html

Este módulo contém bindings para os tipos dos elementos do DOM HTML. Deste modo é possível guardar variáveis de referência ao documento HTML.

```
1.  let doc = Dom_html.document in …
```

Esta referência ao documento agora poderá ser utilizada para manipular os elementos do documento HTML no nosso ambiente de OCaml. Vamos tomar como exemplo a seguinte função div:

```
1.  let div idtxt =
2.      let d = Dom_html.createDiv doc in
3.      d##.id := Js.string idtxt;
4.      d
5.
```

Esta função recebe uma string, cria um div no documento e identifica o div com a string agora convertida no tipo Js.string. Finalmente devolve o div criado.

Temos de tomar nota de uma função bastante importante, que já referimos várias vezes nesta secção de interoperabilidade, o Dom_html.getElementById. Com ela, é possível obter elementos definidos no Dom_html, tal como nos seguintes exemplos:

```
1.  let enableButton buttonName =
2.      let buttonTo = Dom_html.getElementById buttonName in
3.        buttonTo##removeAttribute (Js.string "disabled")
4.
5.  let defineMainTitle type1 =
6.      let title = Dom_html.getElementById "mainTitle" in
7.        title##.innerHTML := Js.string "";
8.  …
```

6.5.4 Módulo Firebug

As operações de escrita na consola não são apropriadas para debug do nosso código no OFLAT. Isto acontece porque o código gerado para o browser foi convertido de OCaml bytecode para JavaScript bytecode. Para este caso temos o módulo Firebug, que permite visualizar valores e objetos recorrendo à consola do modo de programador do browser.

```
1.  Firebug.console##log edge
```

7

Cytoscape.js

7.1 Introdução ao Cytoscape.js

O Cytoscape.js é uma biblioteca JavaScript para visualização[25] e análise de grafos[13]. Esta biblioteca foi usada na aplicação OFLAT para a representação visual dos autómatos finitos, das expressões regulares e das árvores de derivação. No que diz respeito ao LR parsing, que é a contribuição desta dissertação, iremos gerar representações gráficas de diagramas LR com esta biblioteca. Como já foi referido na secção anterior, podemos utilizar bindings para manipular objetos JavaScript em OCaml. Iremos então definir os bindings necessários para a interoperabilidade OCaml/JavaScript necessária para o uso desta biblioteca no OFLAT.

7.2 Módulo Data Item

O módulo DataItem em OCaml reúne convenientemente dois bindings: t e data. O tipo de classe t é usado para introduzir do lado do OCaml a representação de nós e arcos. Contém diversos métodos e propriedades que estão disponíveis nos nós e arcos. Além disso, contém um elemento de dados chamado data que é gerido pelo desenvolvedor e será explicado abaixo.

```
1.  class type t =
2.      object
3.        method data : data Js.t prop
4.        method data_fromName : js_string Js.t -> js_string Js.t meth
5.        method data_update : js_string Js.t -> js_string Js.t -> unit meth
6.        method position : position Js.t prop
7.        method renderedPosition : position Js.t prop
8.        …
9.      end
10. end
```

O tipo de classe data só contém propriedades mutáveis e serve para o desenvolvedor definir a topologia do grafo, elemento a elemento. Para cada elemento, o campo id corresponde ao identificador de nó ou arco. O campo source indica o nó de partida de um arco e target corresponde ao nó de destino. O campo Label corresponde à etiqueta do nó ou arco. Finalmente, o campo nodeType pode ser usado para o desenvolvedor definir várias categorias de nós.

No caso da nossa aplicação, os estados são representados por nós, cujo nome fica registado na label. As transições são representadas por arcos, cujo símbolo de transição fica registado na label. O campo nodeType tem uma utilização particular no contexto do parsing LL(1), porque se torna conveniente definir duas categorias de nós na representação gráfica da árvore abstrata.

```
1.  module DataItem =
2.  struct
3.
4.    class type data =
5.      object
6.        method id : js_string t prop
7.        method source : js_string t prop
8.        method target : js_string t prop
9.        method label : js_string t prop
10.       method nodeType : js_string t prop
11.     end
```

7.3 Estilos

Estilos são um tipo de objeto no Cytoscape.js que contém propriedades que influenciam o aspeto dos grafos a serem visualizados. Os estilos do Cytoscape são análogos aos stylesheets CSS.

Os estilos que são usados na aplicação OCaml, são descritos usando objetos JavaScript definidos do lado do OCaml usando o mecanismo object%js. São aplicados aos nós e arcos através do método style modifica o estilo desses elementos.

Sabemos que em CSS, um estilo é uma sequência de atributos. Representamos a sequência usando um array JavaScript de objetos de estilo. Um objeto estilo precisa de um seletor, que indica os elementos aplicar o estilo, e as propriedades desse estilo. O node_name_style corresponde ao estilo aplicável a todos os nós desse grafo, indicando a posição do símbolo centrada no nó, as dimensões do nó, o alinhamento e as margens do texto do nó.

O edge_symbol_style indica a label de cada arco, que corresponde ao símbolo de transição entre nós.

O edge_style define o tipo de curva do arco e a forma do alvo do arco.
A ordenação dada no array de estilos é importante, uma ordenação imprópria pode levar a que não se obtenha o resultado esperado. Estilos mais gerais devem ser adicionados primeiro, mais específicos devem ser adicionados no fim.

```
1.  let testStyle : Js.Unsafe.any style Js.t Js.js_array Js.t =
2.    let node_name_style = Js.Unsafe.coerce @@ object%js
3.      val selector = Js.string "node[label]"
4.        val style = Js.def (object%js
5.          val content = Js.string "data(label)"
6.          val textHalign = Js.string "center"
7.          val textValign = Js.string "bottom"
8.          val width = Js.string "40px"
9.          val height = Js.string "40px"
10.         val textMarginY = Js.string "2"
11.       end)
12.   end in
13.   let edge_symbol_style = Js.Unsafe.coerce @@ object%js
14.     val selector = Js.string "edge[label]"
15.       val style = Js.def (object%js
16.         val content = Js.string "data(label)"
17.       end)
18.     end in
19.   let edge_style = Js.Unsafe.coerce @@ object%js
20.     val selector = Js.string "edge"
21.     val style = Js.def (object%js
22.       val curveStyle = Js.string "bezier"
23.       val targetArrowShape = Js.string "triangle"
24.     end)
25.   end in
26.  ...
27.  Js.array [|node_name_style; edge_symbol_style; edge_style;
28.  ...|]
```

7.4 Initialização do Cytoscape

Para inicializar o Cytoscape, utilizamos a seguinte função initCy:

```
1.  let initCy cyContainer =
2.    let g = mk_graph ~style:cfgStyle cyContainer in
3.    let cy = display g in
4.       run_layout (cy##layout layoutOptions);
5.       cy
```

A função mk_graph recebe como argumento o container html onde o grafo irá ser colocado (no caso da nossa aplicação, esse container é um div) e cria um grafo vazio. A função display recebe como argumento o grafo e cria um objeto-gestor Cytoscape através do qual o grafo será gerido. Finalmente, podemos aplicar layouts ao nosso grafo, se for apropriado.

Agora que já conseguimos inicializar o objeto-gestor, podemos adicionar elementos ao nosso grafo da seguinte forma:

```
6.  var cy = initCy “cy”;
7.
8.  var eles = cy.add([
9.    { group: 'nodes', data: { id: 'n0' }, position: { x: 100, y: 100 } },
10.   { group: 'nodes', data: { id: 'n1' }, position: { x: 200, y: 200 } },
11.   { group: 'edges', data: { id: 'e0', source: 'n0', target: 'n1' } }
12. ]);
```

Este objeto-gestor irá conter grupos de nós e arcos. Cada um destes nós e arcos têm um campo para os seus dados. Os dados têm como objetivo guardar qualquer atributo que possa ser relevante para os nossos elementos.

8

OFLAT

8.1 Introdução ao OFLAT

O OFLAT é a ferramenta responsável pela parte gráfica da nossa aplicação. Funciona a partir de um browser e oferece uma interface gráfica e pedagógica que permite gerar gráficos interativos e ricos para uma visualização intuitiva do utilizador. Faz também parte do trabalho desta dissertação a aplicação OFLAT.

O OFLAT está integralmente implementado em OCaml. Existiram duas razões para queremos isto: Aproveitar a validação de tipo estática do OCaml e a procura de uniformidade.

O OFLAT está baseado na arquitetura MVC (Model-View-Controller). Esta arquitetura caracteriza-se por três componentes[27, 28]:

Model: Corresponde à biblioteca OCaml-FLAT na qual todos os modelos FLAT tem uma implementação.

View: Corresponde a um conjunto de módulos que são usados para visualizar os dados. Recorre-se ao DOM do HTML e aos grafos da biblioteca Cytoscape.js.

Controller: Processa o input do utilizador. Recebe eventos tipicamente causados pelo rato ou teclado e processa-os, para isso interagindo com o Model e o View.

Atenção, no contexto do MVC, a palavra Model diz respeito aos dados da aplicação e não tem nada a ver com o módulo Model da biblioteca nem com os modelos FLAT da teoria. Os nomes iguais são coincidência.

8.2 Módulo ContextFreeGrammarGraphics

Este módulo, previamente desenvolvido, tinha como objetivo produzir gráficos para gramáticas livres de contexto com uso da técnica de parsing LL(1). No entanto, quando expandimos a biblioteca OCaml-FLAT com conceitos de parsing LR, decidimos criar vários módulos separados com uso de uma hierarquia.

Deste modo, o módulo ContextFreeGrammarGraphics foi reestruturado em quatro módulos diferentes para suportar a adição do módulo de LR parsing:

ContextFreeGrammarBasicGraphics: um módulo "base" que possui todas as funcionalidades do ContextFreeGrammarFull presente na biblioteca OCaml-FLAT. Neste módulo, não são introduzidas funcionalidades ou conceitos novos, e tem como objetivo ser o ponto de partida na construção dos gráficos para módulos aplicáveis a gramáticas livres de contexto. Este módulo é uma extensão do módulo ContextFreeGrammar.

ContextFreeGrammarLL1Graphics: este módulo possui funcionalidades de suporte no nosso ambiente gráfico para gramáticas livres de contexto, tais como limpar gramáticas, remover produções vazias, o cálculo dos conjuntos FIRST e FOLLOW, etc. Adicionalmente, este módulo também contém funcionalidades de visualização para a técnica de parsing LL(1), tais como factorização à esquerda, transformação para gramáticas LL(1), gerar as tabelas de parsing LL(1), entre outras. Este módulo é uma extensão do módulo LL1Grammar.

ContextFreeGrammarLRGraphics: este módulo contém as funcionalidades de suporte para as 4 técnicas de parsing LR referidas, para o nosso ambiente gráfico. Tais como gerar o diagrama para a técnica LR respetiva, gerar a tabela de parsing respetiva, validar palavras para uma dada gramática com recurso às técnicas LR, e é uma extensão do módulo LRGrammar.

ContextFreeGrammarGraphics: este módulo herda de todos os outros. Contém todas as funcionalidades gráficas dos módulos anteriores.Este módulo contém uma funcionalidade para lidar com um pequeno problema técnico de polimorfismo, concretamente converter um objeto LL(1) num objeto desta mesma classe (a conversão vai em sentido contrário da hierarquia). Este módulo é uma extensão do ContextFreeGrammarFull, o módulo no topo da hierarquia do OCaml-Flat, e contém todas as funcionalidades de suporte para gramáticas livres de contexto.

8.3 Módulo ContextFreeGrammarLRGraphics

Como referido na secção anterior, o ContextFreeGrammarGraphics é o módulo dedicado para funcionalidades de suporte ao LR parsing no nosso ambiente gráfico. Como tal, iremos explorar em mais detalhe os conceitos desenvolvidos neste módulo para gerar os nossos gráficos.

8.3.1 Criação de diagramas LR

Para criar os diagramas com recurso ao Cytoscape, temos de gerar o diagrama LR utilizando o módulo correspondente na biblioteca OCaml-FLAT. Recapitulando, os diagramas têm duas grandes componentes, um conjunto de estados e um conjunto de transições. Estes dois conjuntos serão usados para produzir respetivamente os nós e os arcos do grafo. Note que vamos usar o LR(0) como exemplo, mas muitos dos procedimentos aplicados aqui também se aplicam para as outras técnicas LR.

```
1.  method buildCyLR0Diagram cy =
2.    let lr0Diagram = LR0Grammar.getLR0DiagramId super#representation in
3.       inputLR0NodesV2 cy lr0Diagram;
4.       inputLR0Edges cy lr0Diagram
```

Para gerar os nós:

```
1.  Set.iter (fun (id,state) -> Cytoscape.addNode2 cy id … ) statesId;
```

Para gerar os arcos:

```
1.  Set.iter (fun ((id1,_),symb,(id2,_)) -> (Cytoscape.addEdge cy (id1,symb,id2)) ) transi-
    tions
```

Desta forma, é possível gerar um grafo com nós e arcos correspondentes ao nosso diagrama. No entanto, nós queremos que cada nó do diagrama apresenta graficamente o respetivo fecho. Para popular os estados do diagrama com os itens do fecho, podemos atribuir estilos personalizados a cada nó do grafo em forma de imagem. Tal como referido na secção de estilos do capítulo do Cytoscape, devemos aplicar estilos mais específicos depois dos estilos mais generalizados do nosso grafo. Deste modo, iremos atribuir estes estilos personalizados após a criação dos nós no Cytoscape.

Para criar os estilos, produzimos uma lista com todos os itens do estado a partir da função writeLR0StateItems.

```
1.  Set.iter (fun (id,state) -> Cytoscape.buildLR0NodeStyle cy id (writeLR0StateItems state))
    statesId
```

A função buildLR0NodeStyle pode ser dividida em 3 partes: Produzir a imagem do estilo com a lista de itens do estado atual em forma de string com a função buildStyleImage; codificar a string anterior como um URI com a função encodeURI; inserir o URI produzido no estilo do nó correspondente com a função insertNodeImage.

```
1.  let buildLR0NodeStyle cy node stringItemList =
2.    let styleImage = buildStyleImage stringItemList in
3.    let encodeStyle = encodeURI styleImage in
4.    let itemListSize = List.length stringItemList in
5.    let maxSymbolCount = maxLR0SymbolCount stringItemList 0 in
6.              insertNodeImage cy node encodeStyle itemListSize maxSymbolCount
```

A função buildStyleImage consiste em produzir uma string literal de um URI de uma imagem com recurso ao SVG. Para produzir as dimensões adequadas para o diagrama, determinamos o número máximo de símbolos do estado atual. Depois usamos uma função auxiliar para converter cada item numa string literal com forma textSpan do SVG. Finalmente, definimos um prefixo que contém o tipo da imagem SVG e a sua codificação, e devolvemos a string literal completa que será codificada com um URI.

```
1.  let buildStyleImage stringItems=
2.    let prefix = "data:image/svg+xml;utf8," in
3.    let itemListSize = List.length stringItems in
4.    let maxSymbolCount = maxLR0SymbolCount stringItems 0 in
5.    let svg = Printf.sprintf {| <svg xmlns="http://www.w3.org/2000/svg" height="%s"
    width="%s"> |} (string_of_int ((itemListSize*20) + 30)) (string_of_int ((maxSymbol-
    Count*9) + 10)) in
6.    let text = {| <text x="0" y="0" style="fill:black;"> |} in
7.    let textSpan = tspan stringItems 30 in
8.    let finish =        {|</text>
9.              </svg> |} in
10.   let style =         prefix ^ svg ^ text ^ textSpan ^ finish in
11.      style
```

A função encodeURI utiliza o módulo Js do Js_of_ocaml para aceder à função encodeURI do JavaScript. Assim, convertemos a nossa string numa Js.string para ser aplicável a esta função.

```
1.  let encodeURI svg =
2.    Js_of_ocaml.Js.encodeURI (Js.string svg)
```

A função insertNodeImage vai buscar um nó existente utilizando o id do estado correspondente. Se existir tal nó, então criamos um estilo para esse nó com os vários parâmetros

gráficos adequados, como a forma do nó, as dimensões do nó que escalam com o número de símbolos e o número de itens, a opacidade do nó, entre outras. Neste mesmo estilo, iremos introduzir o URI como backgroundImage.

```
1.  let insertNodeImage cy node itemString itemListSize maxSymbolCount =
2.    ignore(
3.      (Js.Unsafe.coerce cy)##
4.      style##
5.      (selector (Js.string ("#" ^ node)) )##
6.      style(Js.Unsafe.coerce @@ object%js
7.        val opacity = Js.string "0.85"
8.        val backgroundColor = Js.string "white"
9.        val backgroundImage = itemString
10.       val backgroundClip = Js.string "none"
11.       val borderWidth = Js.string "3px"
12.       val borderColor = Js.string "black"
13.       val width = Js.string (string_of_int ((maxSymbolCount*9)) ^"px")
14.       val height = Js.string (string_of_int ((itemListSize*20) + 10) ^"px")
15.       val shape = Js.string "roundrectangle"
16.     end)
17.   )##update
```

Assim, já criamos o grafo do diagrama no JavaScript. Para visualizar estes grafos no browser, iremos usar um controlador para o LR. Para ativar as funcionalidades gráficas de um dado módulo usamos listeners definidos para o controlador desse módulo. Como tal, a criação do diagrama irá ser iniciada através de um destes listeners, para o controlador de LR. O procedimento consiste em obter a gramática definida pelo utilizador, criar uma segunda janela à direita com os limites apropriados, executar a nossa função para construir o diagrama no Cytoscape e reestruturar a janela de acordo com o diagrama gerado.

```
1.  ListenersLR.buildLR0DiagramListener :=
2.              fun () -> try
3.                                          twoBoxes !ctrlL#getCy_opt;
4.                                          let cfg = !ctrlL#getCFG in
5.                                          !changeToControllerCtrlRight();
6.                                          let cy = !ctrlR#getCy in
7.                                          HtmlPageClient.defineCFG();
8.                                          HtmlPageClient.cfgBoxRegex();
9.                                          …
10.                                         cfg#buildCyLR0Diagram cy;
11.                                         cy##resize;
12.                                         cy##fit;
13.                                         …
14.                               with
15.                   _ -> JS.alert (Lang.i18nErrorParsing());;
```

Eis um exemplo de uma gramática introduzida no browser e o respetivo diagrama gerado:

S→A
A→aBC
B→bC
C→ab

Figura 26:Gramática introduzida pelo utilizador a ser apresentada na aplicação

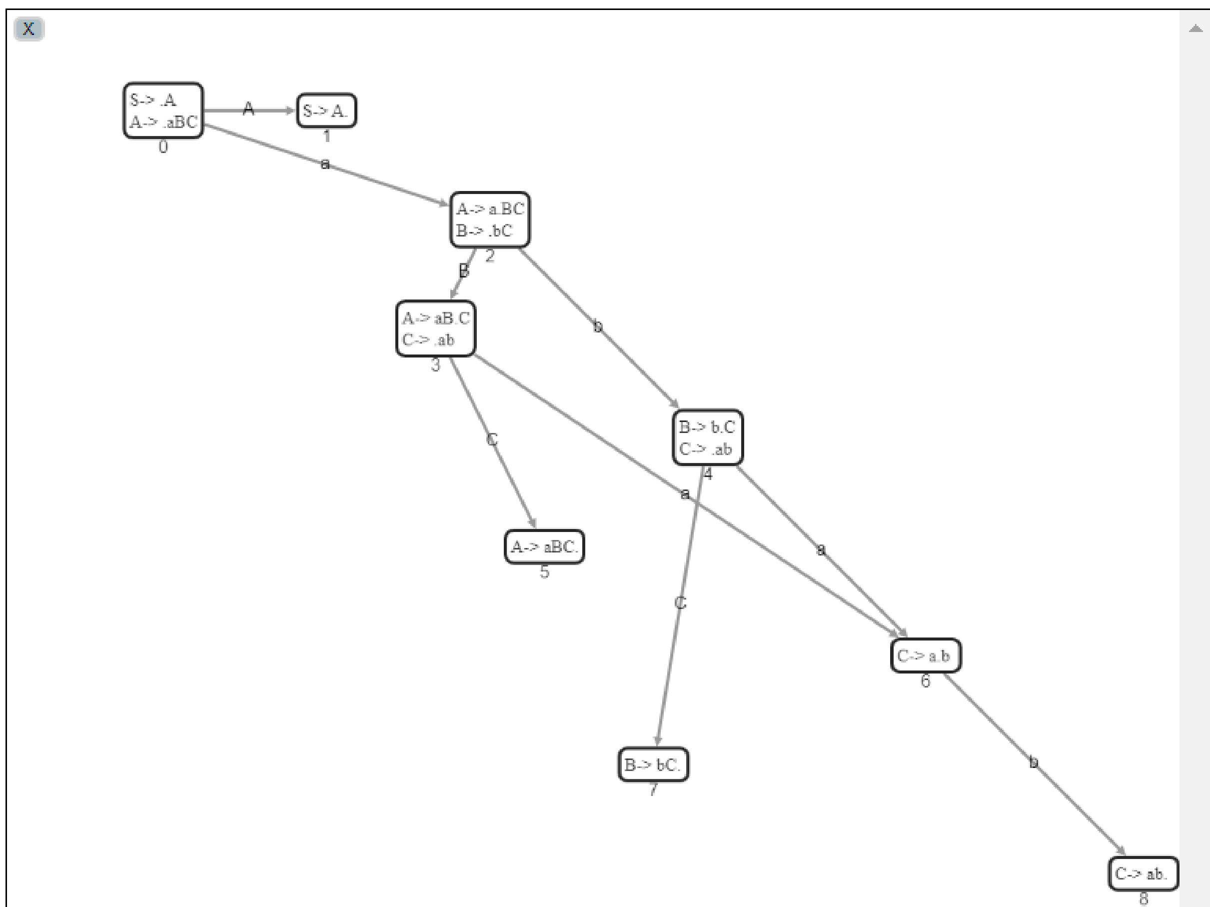


Figura 27:Diagrama LR(0) produzido com a gramática da Figura 27

Se o utilizador pretender ver um diagrama para outra técnica LR, como por exemplo o LR(1), basta carregar no botão para gerar o diagrama LR(1) e rapidamente será apresentado esse diagrama.

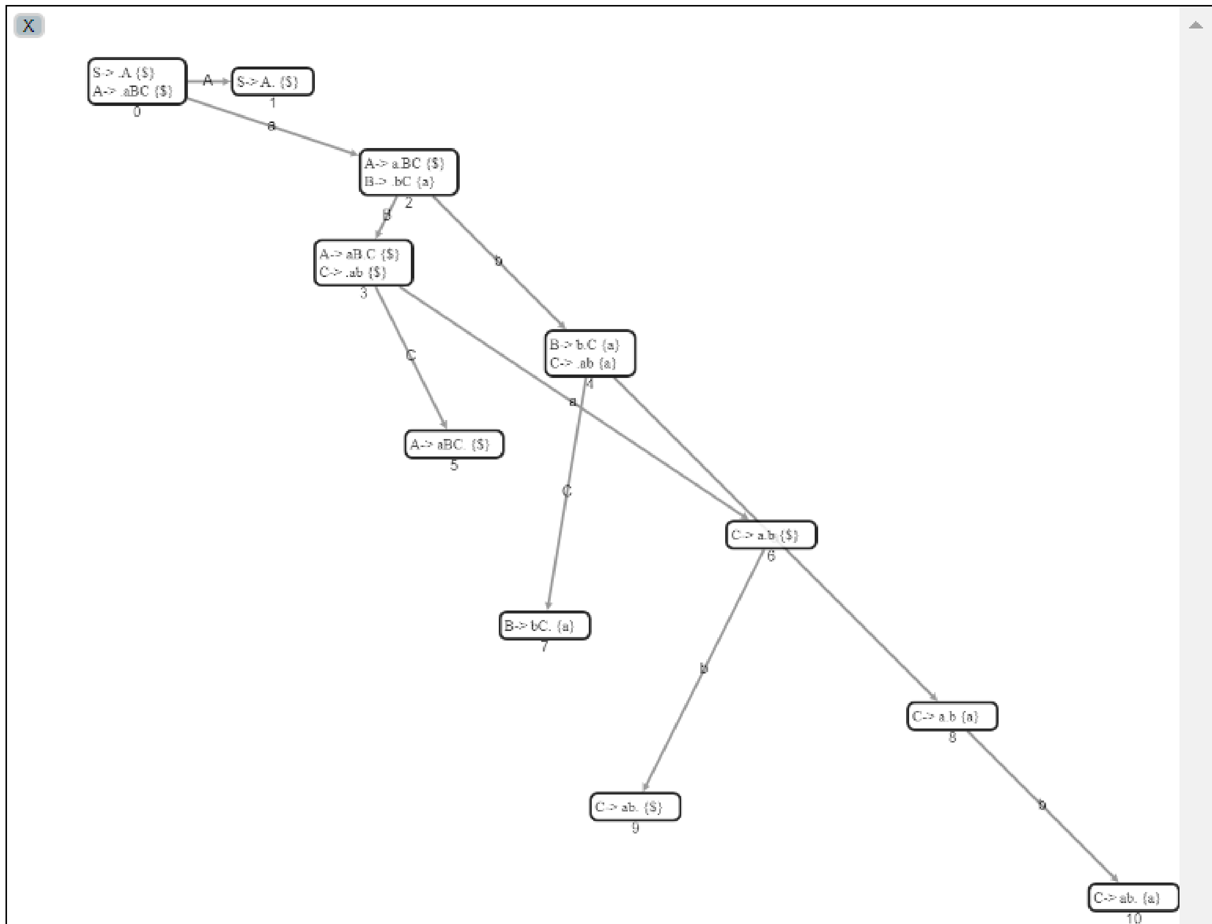


Figura 28:Diagrama LR(1) com a gramática da Figura 27.

Este exemplo demostra o aumento de consumo de recursos do LR(1), que devido à adição do conjunto lookahead, obteve um número maior de estados que o LR(0).

8.3.2 Criação de tabelas de parsing LR

Esta funcionalidade é acionada no browser com um botão correspondente. Quando o utilizador carrega no botão, o controlador LR começa por buscar a gramática do utilizador. Depois, executa a função prepareCFG2Tables que irá construir os elementos html correspondentes às tabelas de FIRST/FOLLOW e a tabela de parsing[24]. Estas tabelas estarão vazias após a criação, e serão preenchidas nas bibliotecas gráficas das gramáticas livres de contexto.

```
1.  ListenersLR.buildLR0TableListener :=
2.              fun () ->
3.                  let cfg = !ctrlL#getCFG in
4.                  twoBoxes !ctrlL#getCy_opt;
5.                  HtmlPageClient.prepareCFG2Tables ();
6.                  cfg#createFirstAndFollowTableHtml;
7.                  cfg#createLR0ParsingTableHtml;;
8.
```

A função createFirstAndFollowTableHtml vem do ContextFreeGrammarLL1Graphics, uma biblioteca desenvolvida antes desta dissertação. Esta função calcula os conjuntos FIRST/FOLLOW para cada símbolo não terminal, e será usada para preencher as linhas da tabela FIRST/FOLLOW, onde cada linha corresponde a um símbolo não terminal e as duas colunas correspondem ao tipo de conjunto (FIRST/FOLLOW). O valor de cada célula será um conjunto de símbolos terminais resultantes do cálculo do conjunto apropriado de acordo com o símbolo da linha.

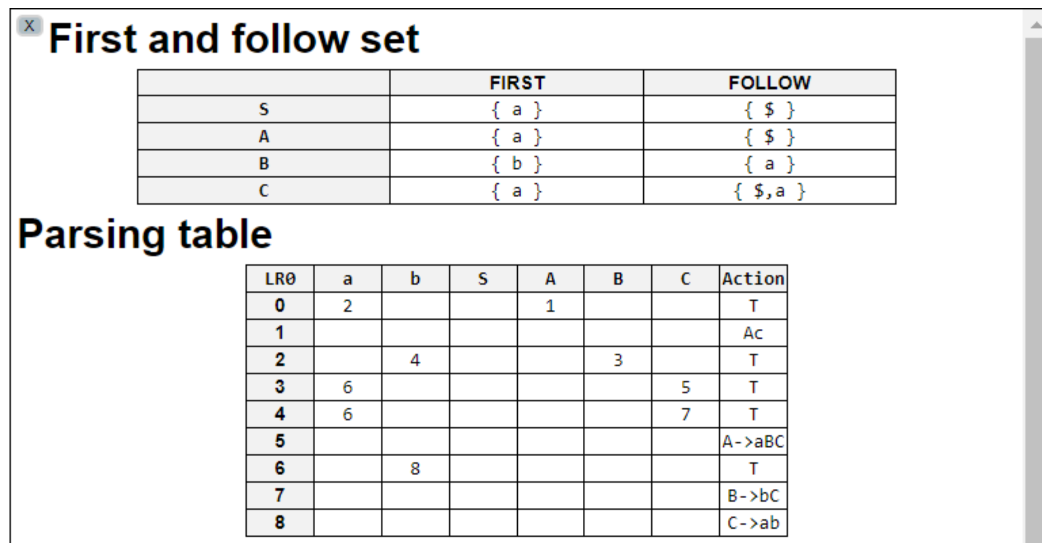
A função createLR0ParsingTableHtml vai buscar a tabela de parsing que criamos e preenche essa tabela em vários passos: Primeiro, criamos a linha do header, que irá indicar a técnica que estamos a aplicar, o alfabeto e as variáveis da gramática, e finalmente a/as ações que cada estado poderá realizar.

Após criado o header[24], vamos preencher as linhas da tabela, onde cada linha é um estado do respetivo diagrama LR. Para produzir o conteúdo de cada linha, utilizamos uma função fillRow, sendo nomeada de fillLR0Row no caso do LR0. Esta função é aplicada a todas as linhas da tabela LR0 usando a função getLR0Table do módulo LR0Grammar presente na biblioteca OCaml-FLAT. Esta função recebe um conjunto de símbolos correspondente à união do alfabeto com as variáveis e percorre a linha da tabela do estado atual à procura de transições para cada um desses símbolos.

Finalmente, após preencher a tabela, acedemos ao seu estilo e alteramos as dimensões para produzir uma tabela com células legíveis. Isto pode acontecer quando uma tabela tem demasiadas colunas, o que irá comprometer o tamanho da cada célula na tabela.

```
1.  let createLR0ParsingTableHtml (cfg:t) =
2.    let table = Js.Unsafe.coerce (Dom_html.getElementById ("cfgParsingTable")) in
3.    let lr0Table = LR0Grammar.getLR0Table cfg in
4.    let row = table##insertRow (-1) in
5.    let newCell = row##insertCell (-1) in
6.        newCell##.innerHTML := Js.string ("LR0");
7.        ignore (newCell##.classList##add (Js.string "monospaceClass"));
8.    let symbSet = Set.union cfg.alphabet cfg.variables in
9.        Set.iter ( fun symb ->
10.             let newCell1 = row##insertCell (-1) in
11.                       newCell1##.innerHTML := Js.string (symb2str symb);
12.                       ignore (newCell1##.classList##add (Js.string "monospaceClass"));
13.                       ) symbSet;
14.             let newCell1 = row##insertCell (-1) in
15.             newCell1##.innerHTML := Js.string ("Action");
16.             ignore (newCell1##.classList##add (Js.string "monospaceClass"));
17.                       Set.iter ( fun entry ->
18.                                 let row = table##insertRow (-1) in
19.                                           fillLR0Row row entry symbSet
20.                       ) lr0Table;
21.             let numberOfCells = (Set.size symbSet) + 2 in
22.             let cellWidth = 50 in
23.             let width = Printf.sprintf "width:%dpx" (numberOfCells*cellWidth) in
24.                       table##.style := (Js.string width)
```

Usando a gramática do exemplo anterior na criação do diagrama, eis as respetivas tabelas FIRST/FOLLOW e parsing:



First and follow set

	FIRST	FOLLOW
S	{ a }	{ $ }
A	{ a }	{ $ }
B	{ b }	{ a }
C	{ a }	{ $,a }

Parsing table

LR0	a	b	S	A	B	C	Action
0	2			1			T
1							Ac
2		4			3		T
3	6					5	T
4	6					7	T
5							A->aBC
6		8					T
7							B->bC
8							C->ab

Figura 29:Tabelas FIRST/FOLLOW e tabela de parsing LR(0) da gramática da Figura 27

Se o utilizador pretender ver uma tabela de parsing para outra técnica LR, como por exemplo o LR(1), basta carregar no botão para gerar as mesmas tabelas LR(1) e rapidamente serão apresentadas essas tabelas, devido ao polimorfismo da nossa solução.

First and follow set

	FIRST	FOLLOW
S	{ a }	{ $ }
A	{ a }	{ $ }
B	{ b }	{ a }
C	{ a }	{ $,a }

Parsing table

LR1	a	b	S	A	B	C	a	b	$
0	2			1			T		
1									Ac
2		4			3			T	
3	6					5	T		
4	8					7	T		
5									A->aBC
6		9						T	
7							B->bC		
8		10						T	
9									C->ab
10							C->ab		

Figura 30:Tabelas FIRST/FOLLOW e tabela de parsing LR(1) da gramática da Figura 27

Aqui destaca-se as linhas de ação para cada estado, onde no LR(0) era apenas uma única ação por estado, e agora temos n linhas, onde n corresponde ao número de símbolos terminais da gramática. Se ocorrer um conflito para um dado símbolo, a célula da ação irá conter o texto "Conf".

8.3.3 Aceitar palavras com técnicas LR

Para produzir uma visualização de cada passo do processo de validação de palavras, foi necessário definir um tipo novo de dados na biblioteca OCaml-FLAT, nomeado de lr0TableStep.

```
1.  type truelr0TableStep = symbol list * string list * lr0Table * bool
```

Este tipo guarda uma lista que contém os símbolos restantes da palavra a consumir, uma lista de strings correspondentes à pilha do nosso autómato, a tabela de parsing para validar a palavra entre passos e um bool que começa como true e só é alterado no caso em o autómato detete que a palavra não poderá ser aceite, onde neste caso vai tomar o valor false.

Para construir a tabela que irá representar o procedimento de validação de palavras, precisamos de construir uma lista destes passos. Assim conseguimos apresentar o

funcionamento do autómato desde o passo inicial até ao passo final. Eis um exemplo com a função buildLR0Steps (referida em baixo) onde vamos produzir essa lista:

```
1.  let buildLR0Steps word2 cfg =
2.              let word = str2word word2 in
3.              let open LR0Grammar in
4.              let firstStep = acceptWordLR0InitV2 word cfg in
5.              let stepList : truelr0TableStep list = [firstStep] in
6.              let nextStep = firstStep in
7.              let (word,revStack,lr0Table,valid) = nextStep in
8.              let valid = ref valid in
9.              let word = ref word in
10.             let stepList = ref stepList in
11.             let nextStep = ref nextStep in
12.                       while(!valid && !word != [dollar]) do
13.                                 nextStep := acceptWordLR0StepV2 !nextStep cfg;
14.                                 let (word2,revStack,lr0Table,valid2) = !nextStep in
15.                                           stepList := !stepList @ [!nextStep];
16.                                           valid := valid2;
17.                                           word := word2;
18.                                 ()
19.                       done;
20.                       !stepList
```

Esta função consiste em definir o passo inicial do nosso autómato, constituído pela palavra completa, o identificador do estado inicial, a tabela de parsing e o booleano com valor true. Também criamos referências para várias variáveis, as quais irão sofrer alterações durante o funcionamento do autómato passo a passo. Deste modo, conseguimos criar todos os passos e colocá-los na lista de passos stepList.

O procedimento de criar a tabela[24] para este processo é bastante parecido com o caso anterior da tabela de parsing. Neste caso, iremos ter sempre 3 colunas, e um número potencialmente grande de linhas, onde cada linha é um passo do autómato.

Eis um exemplo para a mesma gramática utilizada nas seções anteriores, onde o autómato está a executar o processo de validação para a palavra ababab$:

Accept

Palavra	Pilha	Ação
ababab$	0	Shift
babab$	0a2	Shift
abab$	0a2b4	Shift
bab$	0a2b4a6	Shift
ab$	0a2b4a6b8	C->ab
Cab$	0a2b4	Shift
ab$	0a2b4C7	B->bC
Bab$	0a2	Shift
ab$	0a2B3	Shift
b$	0a2B3a6	Shift
$	0a2B3a6b8	C->ab
C$	0a2B3	Shift
$	0a2B3C5	A->aBC
A$	0	Shift
$	0A1	Ac
	S	

Figura 31:Validação da palavra ababab$ para a gramática da figura 27.

9

Apreciação Geral e Conclusões

Neste capítulo vai ser feita uma apreciação sobre os aspetos positivos e limites da implementação sobre as várias componentes desta ferramenta. Também serão apresentadas as conclusões desta dissertação.

9.1 OCaml-FLAT

Para esta biblioteca, foi criado o módulo LRGrammar. Este módulo estende as funcionalidades de suporte a gramáticas livres de contexto que já existiam. Foram implementadas várias funcionalidades novas como a criação de diagramas LR, a criação de tabelas de parsing LR e teste de aceitação das palavras. Isto foi feito para as várias técnicas da família LR. Foram também definidas múltiplas funcionalidades de suporte, nomeadamente: determinar se uma gramática é LR(0) ou SLR(1) ou LR(1) ou LALR(1) ou nenhum destes; detetar conflitos; produzir uma sequência com o tracing da tentativa de reconhecimento de uma palavra. Todas estas operações estão disponíveis na biblioteca para utilização geral, mas foram também usadas na programação da aplicação gráfica OFLAT.

Relativamente à validação do código, foi criado um módulo de testes com numerosos exemplos para a cada técnica LR, concebido para detetar o máximo de situações de possível erro na implementação. Ajudou a detetar alguns problemas nas funcionalidades implementadas.

Houve uma pequena liberdade no processo de aceitação de palavras, onde a transferência de um símbolo não terminal após uma redução foi tratada com um passo distinto do autómato. No modelo tradicional, esse passo fica oculto dentro de um passo mais abrangente. Foi decidido autonomizar o passo para promover a clareza e o carácter pedagógico da nossa solução.

9.2 OFLAT

Quando este trabalho se iniciou, a aplicação OFLAT já oferecia uma interface gráfica pedagógica para gramáticas independentes de contexto, inclusivamente com suporte para parsing LL(1). Estendemos o OFLAT para incluir funcionalidades gráficas interativas para as técnicas de parsing LR.

A interface para as funcionalidades de LR, foi desenhada com foco na simplicidade e pedagogia. A solução faz questão de ajudar o utilizador a comparar as diferenças entre várias técnicas de parsing LR. Por exemplo, através de dois botões é possível alternar entre visualização do diagrama LR(0) e LR(1) e assim identificar as diferenças. Esta funcionalidade é realmente útil e para se poder concretizar, foi necessário conceber um algoritmo de colocação uniforme dos nós no plano e depois usar o mesmo algoritmo para todos os diagramas.

No contexto da biblioteca Cytoscape.js, o suporte básico no sentido de criar nós sem qualquer conteúdo no interior; as etiquetas aparecem fora dos respetivos nós. Mas a biblioteca também fornece um mecanismo sofisticado e complicado de usar, a que tivemos de recorrer. Realmente, nos nossos diagramas LR, cada nó tem conteúdo próprio, concretamente uma sequência de itens LR. Para criar uma visualização gráfica do conteúdo de cada nó, é preciso gerar dinamicamente uma imagem no formato SVG.

Um detalhe técnico com algum impacto visual que não estava tratado no caso das tabelas LL(1) foi agora tratado no caso das tabelas LR: as tabelas agora possuem um mecanismo dinâmico de reestruturação que leva em conta o número de linhas e o número de colunas, evitando situações de desformatação, mesmo no caso de tabelas muito grandes.

Para validar a qualidade pedagógica do suporte para parsing LR, houve bastante interação com docentes atuais e antigos da cadeira de Teoria da Computação. As sugestões e críticas tiveram uma influência positiva neste trabalho.

9.3 Conclusões

Nesta dissertação houve um esforço considerável para desenvolver todo o código na linguagem OCaml de acordo com o paradigma da programação funcional[20]. Para além disso, ouve um investimento considerável para desenvolver uma solução com código próximo da teoria, deste modo promovendo o valor pedagógico das funcionalidades da biblioteca OCaml-FLAT para os algoritmos do LR, que possuem um nível de complexidade notável.

Os algoritmos do LR são complexos e foi necessário um grande esforço para produzir código legível, bem organizado e próximo das definições matemáticas. Grande parte das funções que foram escritas precisaram de ter várias versões.

Do lado do OFLAT, foi necessário um estudo duro e dedicado para aprender a usar os mecanismos de interoperabilidade entre OCaml e JavaScript. Também foi preciso compreender a arquitetura existente e integrar o código novo no complexo contexto dessa arquitetura, onde já tinham colaborado várias pessoas.

Deste modo, esta dissertação contribuiu vastamente para a minha aprendizagem da linguagem OCaml, de mecanismos de interoperabilidade e as técnicas de parsing LR, que têm um nível de interesse respeitável atualmente, pois provam ser bastante poderosas. Também reforçou o meu conhecimento de JavaScript, que apesar de não ter sido escrito código nesta linguagem, foi necessário explorar os mecanismos desta linguagem para desenvolver uma interoperabilidade adequada.

9.4 Trabalho Futuro

Como trabalho futuro, pode-se identificar como mais importante: a adição de árvores de derivação tanto da biblioteca como na parte gráfica, e a criação de um certo número de exercícios pedagógicos para acrescentar à biblioteca de exercícios do OCaml-FLAT/OFLAT. Algo um pouco menos importante seria a geração de parsers LR no formato de programas em C ou OCaml por exemplo (imitando ferramentas como o Yacc).

Bibliografia

[1] Donald E. Knuth, "On the Translation of Languages from Left to Right, Information and Control", 8(6) (1965). (ver p16 e p16)

[2] Aho Alfred Vaino, Lam Monica Sin-Ling, Sethi Ravi Ullman e Jeffrey David, "Compilers: Principles, Techniques, and Tools" (2 ed.). Boston, Massachusetts, USA: Addison-Wesley. ISBN 0-321-48681-1 (2006). (ver p12 e p16 e p17)

[3] Noam Chomsky, "On Certain Formal Properties of Grammars" (1959). (ver p11 e p16)

[4] Noam Chomsky e Marcel-Paul Schützenberger, "The Algebraic Theory of Context-Free Languages". Editado por Paul Braffort e Dan Hirschberg em "Computer Programming and Formal Systems". (ver p11)

[5] Matthew M. Geller e Michael A. Harrison, "Strict Deterministic Versus LR(0) Parsing" (Extended Abstract) (1973). (ver p16)

[6] Stephen Chong, Universidade de Harvard, "CS153: Compilers Lecture 11: LR Parsing" (2019). (ver p16)

[7] Maggie Johnson e Julie Zelenski, Universidade de Standford, "SLR(1) and LR(1) Parsing" (2011). (ver p27)

[8] Franklin L. DeRemer, "Simple SLR(k) Grammars". ACM, 14(7), (1971). (ver p28)

[9] Franklin L. DeRemer e Thomas Pennello, "Efficient Computation of LALR(1) Look-Ahead Sets", ACM, 4(4) (1982). (ver p36)

[10] East Carolina University, "Canonical LR(1) Parsers", URL: LR(1) parsers (ecu.edu), (acedido em 25/06/2022). (ver p33)

[11] Luís Monteiro, "Slides da disciplina de linguagens formais e autómatos", (2005). Material privado apenas disponível a alunos/professores autorizados. (ver p19, p21, p23, p32 e p36)

[12] António Ravara, "Slides da disciplina de teoria da computação" (2020). Material privado apenas disponível a alunos e professores autorizados. (ver p11, p13, p17 e p17)

[13] "Cytoscape.js", URL: https://js.cytoscape.org/ (acedido em 28/09/2022). (ver p55 e p63)

[14] "JFLAP", URL, https://www.jflap.org/ (ver p4) (acedido em 27/06/22). (ver p4)

[15] Eric Gramond e Susan H. Rodger, "Using JFLAP to Interact with Theorems in Automata Theory" (1999). (ver p5)

[16]"First+Follow+Predict Calculator", URL: https://www.usna.edu/Users/cs/wcbrown/courses/F20SI413/firstFollowPredict/ffp.html (ver p7) (acedido em 30/06/2022). (ver p7)

[17] "LR(0) Parser Visualization", URL: LR(0) Parser Visualization (princeton.edu) (acedido em 30/06/2022). (ver p8).

[18] "LR(1) Parser Generator", URL: https://jsmachines.sourceforge.net/machines/lr1.html (acedido em 30/06/2022). (ver p10)

[19]"OCaml", URL: https://ocaml.org/docs, (acedido em 26/09/2022). (ver p55)

[20] Yaron Minsky, Anil Madhavapeddy e Jason Hickey, "Real World OCaml" (2022). (ver p80)

[21] Jerome Vouillon e Vincent Balat, "From Bytecode to JavaScript: the Js_of_ocaml Compiler". (2011). (ver p55)

[22]"Js_of_ocaml - Reference Manual", URL: https://ocsigen.org/js_of_ocaml/latest/manual/overview, (acedido em 23/09/2022). (ver p56 e p59)

[23] "Introducing JSON", URL: https://www.json.org/json-en.html (acedido em 12/07/2022). (ver p39)

[24] "HTMLTableElement", URL: HTMLTableElement - Web APIs | MDN (mozilla.org). (ver p74, p74 e p77)

[25] Max Franz, Christian T. Lopes, Gerardo Huck, Yue Dong, Onur Sumer e Gary D. Bader, "Cytoscape.js: a graph theory library for visualisation and analysis" (2015). (ver p63)

[26] João Gonçalves, "OCaml-FLAT - An OCaml Toolkit for experimenting with formal language theory", dissertação de mestrado, FCT-UNL (2020). (ver p38)

[27] Rita Macedo, "OCaml-FLAT on the Ocsigen framework", dissertação de mestrado, FCT-UNL (2020). (ver p38 e p67)

[28] Eduardo Silva, "Gramáticas LL(1) em OCaml-FLAT/OFLAT", dissertação de mestrado, FCT-UNL (2021). (ver p16, p16 e p67)

2022 Bernardo Sousa Análise de sintaxe LR em OCaml-FLAT/OFLAT

NOVA

NOVA SCHOOL OF SCIENCE & TECHNOLOGY