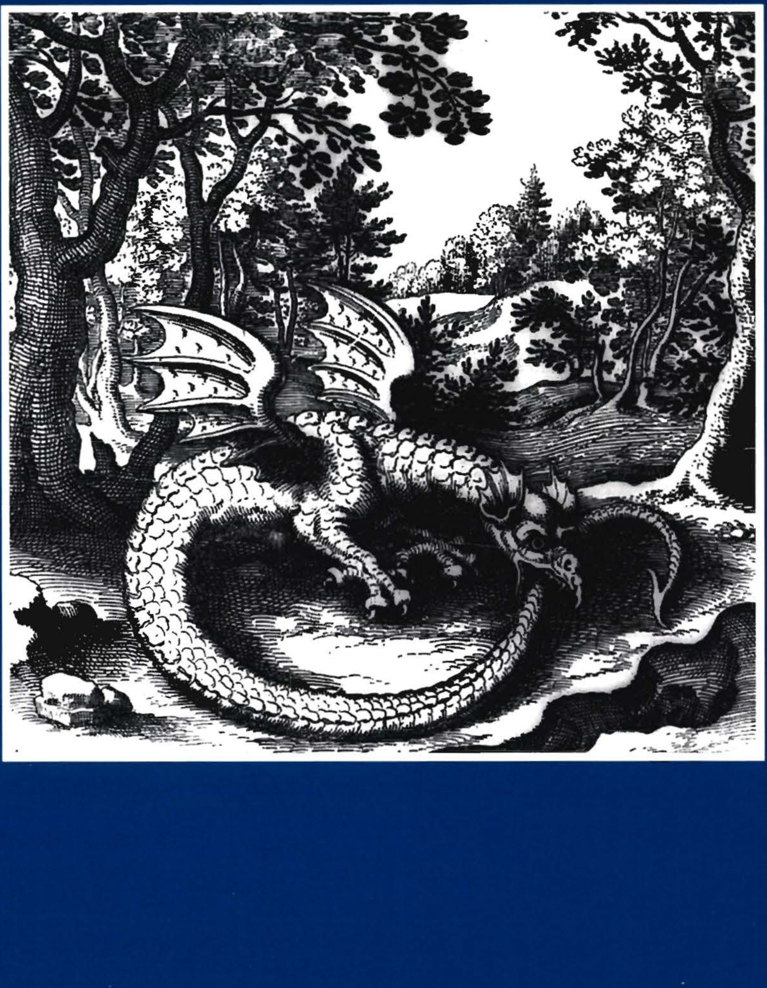
Functional Programming, Program Transformations and Compiler Construction

Lex Augusteijn



Functional Programming, Program Transformations and Compiler Construction

Cover: Lambsprinck: *De lapide philosophico figurae et emblemata*, in: *Musaeum Hermeticum*, Frankfurt 1678, p. 353.

Functional Programming, Program Transformations and Compiler Construction

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Eindhoven,
op gezag van de Rector Magnificus,
prof. dr. J.H. van Lint,
voor een commissie aangewezen
door het College van Dekanen
in het openbaar te verdedigen
op donderdag 7 oktober 1993 om 16.00 uur

door

Alexander Augusteijn

geboren te Hellendoorn

Dit proefschrift is goedgekeurd door de promotoren

prof. dr. F.E.J. Kruseman Aretz
en
prof. dr. S.D. Swierstra

CIP-DATA KONINLIJKE BIBLIOTHEEK, DEN HAAG

Augusteijn, A.

Functional programming, program transformations and
compiler construction / A. Augusteijn. - Eindhoven :
Philips Natuurkundig Laboratorium. - Ill.
Thesis Eindhoven. - With ref.
ISBN 90-74445-04-7
Subject headings: functional programming / program
transformations / compilers.

The work described in this thesis was has been carried out at the Philips Research Laboratories
as a part of the Philips Research program.

ISBN 90-74445-04-7

Acknowledgements

The work presented in this thesis grew out of the Esprit project ‘Parallel architectures and languages for Advanced Information Processing – A VLSI-directed approach’. The work conducted in that project has enabled the development of the `Elegant` system and the numerous discussions with many people made the system mature in a relatively short period of time. For this I would like to thank Paul Gorissen, Séan Morrison, Loek Nijman, Eddy Odijk, Frits Schalij, Juul van der Spek, Mark Tucker, and Rob Wieringa.
I am greatly endebted to Séan Morrison for the acronym `Elegant`.
The system was ported by Rob Wieringa from Modula-2 to C. Without this porting activity, `Elegant` would never have reached the state that it has now.
In later periods, the stimulating discussions with Paul Gorissen and Paul Jansen have increased the orthogonality of the system and the language.
The enthousiastic discussions with René Leermakers and Frans Kruseman Aretz have contributed in many ways to the ideas expressed in this thesis.
I greatly appreciate the careful reading of draft versions of this thesis by Frans Kruseman Aretz, René Leermakers, Doaitse Swierstra, Paul Klint, Roland Backhouse and the anonymous referee from the ‘Philips Translation Services’. Especially Frans’ remarks have been a challenge to sharpen my ideas and to express them much more clearly.
During the work on this thesis, my wife Else has supported me with her love and perseverence.
[illegible]

Contents

Chapter 1

Introduction

This thesis deals with the development of the `Elegant` tool set. By means of this set of tools, an implementation for a certain class of software systems, among which compilers, can be generated from a specification written in the `Elegant` programming language. The `Elegant` project started with a simple compiler generator based on attribute grammars and has developed into a (compiler for a) very high level programming language. During this development, many evolutions in computer science have influenced the design decisions that were taken. The development itself can be characterized as an experiment in language design. Once `Elegant` had been boot-strapped (its very first application was its self-generation), it has been used to generate several hundreds of new versions of itself. Numerous new features have been added and five complete re-designs have taken place. For each new feature that was proposed for incorporation into `Elegant`, it was required that the feature was simultaneously a proper language construction, orthogonal to the existing ones, and efficiently implementable. As a result, `Elegant` is now a programming language that combines a high level of abstraction with an efficient implementation.
In this thesis we describe the sources of inspiration for the abstractions that are incorporated in `Elegant`. The most important of these sources is the field of functional programming, that offers very interesting abstraction mechanisms, however combined with a lack of efficiency due to the absence of side-effects. We show how many of these abstraction mechanisms can be incorporated into an imperative language and form an enrichment of imperative programming. An important aspect of the lack of side-effects in functional programming is that functional programs can be transformed by means of algebraic identities into new programs, much like one rewrites $a(b+c)$ into $ab+ac$ in algebra by applying the law of distributivity of multiplication over addition. We will make ample use of such transformations throughout this thesis; they are the most important technical instrument that we will be using.
Both the field of compiler construction and that of functional programming, which is based on the λ-calculus, belong to the most extensively studied areas of computer science. Nevertheless, they have been related weakly, at least until recently. Compilers were implemented in imperative formalisms rather than in functional ones and compiler generators generate

imperative code. In this thesis we show how the two fields can be related much more strongly and how their interaction is of mutual benefit. The theory of compilers, especially of parsers can be cleanly formulated in a functional formalism. The techniques developed in compiler construction in their turn positively influence functional languages and their implementation. Using standard compiler construction techniques, functional languages can be implemented much more efficiently than by basing them on the λ-calculus. The type systems of functional languages can take advantage of notions developed for imperative languages and their implementation.
The contributions of this thesis to compiler construction and functional programming languages are the following:

- We show that not only top-down, but also bottom-up parsers can be derived as a set of mutually recursive functions. This way of expressing them frees the theory of parsing from the use of stack automata.
- We show how non-deterministic parsers can be expressed as functional programs and how deterministic versions are special cases of the non-deterministic ones.
- We show that many different bottom-up parsers can be derived from a single top-down parser by means of algebraic program transformations.
- We show how the technique of memoization can be applied to parsers. The complicated Earley and Tomita parsers become trivial results of the application of this technique.
- We show how a simple, yet effective, error recovery mechanism for functional bottom-up parsers can be designed.
- We show that attribute grammars can be modeled much more lucidly than is usually done by founding them on a functional formalism.
- We show how the notion of attribute grammars can be generalized from functions over an input string to functions over an arbitrary data structure, without losing their expressiveness and efficiency.
- We show how the very general class of attribute grammars that is supported by a functional implementation allows the specification of a general attribution scheme that prescribes the construction of a standardized attribute grammar for a given language.
- We show how scanners can be expressed in a functional language and how this formulation makes the use of finite state automata in the theory of scanning obsolete.
- We compare type systems in functional and imperative languages and show how their features can be merged to form a new type system that combines sub-typing, parameterized types, higher order functions and polymorphism. In this type system, pattern matching becomes equivalent to type analysis.

- We show that the notion of empty types is useful, especially in the construction of code generators.
- We show how laziness can be incorporated in an imperative language.
- We show how these results have been incorporated into the design of `Elegant`.

Throughout this thesis, the reader should be aware that the name `Elegant` is overloaded: it is simultaneously:

- A compiler generator system, consisting of several cooperating tools.
- One of these tools, namely a compiler for attribute grammars.
- A very high level programming language.
- The compiler for this programming language, which is the same tool as the compiler for attribute grammars.

We trust that the reader will be able to judge from the context which of these meanings is appropriate in the different parts of this thesis. Most of the times when we refer to `Elegant`, we mean the programming language however.

1.1 Historical background of compiler construction

A compiler can be defined as a function from a structured input string onto an output string. As most input strings are structured in one way or another, we need a more narrow definition than this one. We obtain this definition by requiring that the compiler function can be written as a composite function $\mathsf{compiler} = \mathsf{code-generation} \circ \mathsf{optimize} \circ \mathsf{make-data-structure} \circ \mathsf{parse} \circ \mathsf{scan}$. When applied to a string, i.e. a sequence of characters, the compiler first performs a *scan* of the input string, by which it maps sub-sequences of characters onto words, so-called *terminal symbols*. The resulting sequence of terminal symbols is subsequently operated on by the *parser* which parses the terminal sequence into a so-called *syntax tree*. This syntax tree describes the structure of the terminal sequence in terms of a (context-free) *grammar*. Subsequently, the syntax tree is mapped onto a data structure that represents the input-string in a more abstract way than the syntax tree. For properly designed languages, this mapping is compositional to a high degree. This data structure is a graph that can be transformed into another graph, a process which we call *optimization*, since the latter graph represents the input string as well, but usually in a more efficient way. Finally, the graph can be mapped onto the output string that is a sequence of characters by the *code-generator*. The sub-function $(\mathsf{make-data-structure} \circ \mathsf{parse} \circ \mathsf{scan})$ is called a *front-end*, while the sub-function $(\mathsf{code-generation} \circ \mathsf{optimize})$ is called a *back-end*. For computer science, the construction of compilers has a long history. In the fifties, compilers were written in machine code in an ad-hoc fashion. The development of the theory of context-free grammars in the early sixties allowed the formal definition of a syntax for a programming language. Several ways to derive parsers directly from such

context-free grammars were persued, culminating in the formulation of so-called LR(k) parsers by Knuth [Knu65, ASU86]. The theory of stack automata made it possible to derive automatically a parser from a context free grammar. Especially the LALR(1) variant of these parsers that was incorporated in the `yacc` (for yet another compiler compiler [Joh75]) parser generator tool in the early seventies became the state of the art for the last two decades.
Lexical scanners were described by so-called *regular expressions*, a limited form of context-free grammars, also in the sixties. Again by means of automata theory, a lexical scanner could automatically be derived from a specification in the form of regular expressions. The state of the art in this respect is the `lex` tool that originated in the early seventies and is still widely in uze.
The `lex` and `yacc` approach to compiler construction has developed into almost a de facto standard. Many textbooks on compiler construction concentrate on these tools and many new versions of these tools, differing more in efficiency than in functionality, have been constructed since. However, these tools cover only the analysing part of a compiler. The construction of the data structures and the subsequent operations are not supported at all and the programmer has to fall back on programming in C for these other compiler constituents.
As context free grammars formalize only the syntactical part of a language, different attempts were undertaken to extend them in order to cover also the semantics of a language, or less ambitiously, the context sensitive parts of a language. From the different formalisms that have been developed for this purpose, we mention *two level grammars* and *attribute grammars*.
Two-level grammars were designed by van Wijngaarden for the definition of the programming language Algol 68 [vWMP+76].] They allowed the precise definition of the syntactical and context sensitive part of a language. The dynamic semantics, however, were not covered, although in theory two-level grammars have the expressive power to model also these semantics. The definition of Algol 68 remained the most prominent application of two-level grammars and nowadays, this formalism is hardly used.
The formalism of attribute grammars has been developed by Knuth [Knu68, Knu71]. An attribute grammar describes simultaneously the structure of an input string in the form of a context-free grammar and properties of that input string by means of so-called attributes and attribution rules. These attributes are objects of domains defined auxiliary to the attribute grammar, and usually they form the data structure to be constructed. From an attribute grammar, the second and third sub-function of a compiler can be generated. An attribute grammar is a declarative formalism, that specifies the relations between attributes without specifying the order of computation. Thus, an implementation of attribute grammars should construct such an order (if it exists at all). Many different scheduling techniques for attribute evaluation can be distinguished; see [Eng84] for an overview. Between these techniques, inclusion relations exist, i.e. one technique can be more general than another. Thus, tools based on attribute grammars can be ordered with respect to the class of attribute grammars that they support (i.e. for which they can generate an attribute evaluator). In [DJL84] an overview of many systems based on attribute grammars together with their position in the hierarchy can be found.

Unlike the lex-yacc approach, the use of attribute grammars has been less successful. This is partly caused by the free availability of lex and yacc that are distributed as parts of the Unix system. Until recently, systems based on attribute grammars were also relatively inefficient in space and time behavior, at least for the more general classes of attribute grammars. This changed with the GAG tool [KHZ81], based on so-called ordered attribute grammars [Kas80], a rather general class. GAG, however, generated Pascal, in contrast to lex and yacc which generate C and GAG initially offered a rather spartanic approach to attribute evaluation, namely by means of the special, purely functional but limited, programming language Aladin. It was only later later that an interface to Pascal was added. More recently, the GMD tool-set [GE90] offered a set of tools based on both Modula-2 and C, also supporting ordered attribute grammars.
For the construction of compiler back-ends, i.e. optimization and code-generation, work has concentrated on the generation of target code from formal machine architecture specifications. The mapping between the source data structure and the target code is usually specified by tree-based pattern matching. The GMD tools offer the ability to use a kind of attribute grammars to operate on the data structure, using pattern matching [Gro92].
Where can Elegant be placed in this scheme? Elegant offers a tool for the generation of lexical scanners. It uses attribute grammars for the specification of the remainder of the front-end, supporting a class of attribute grammars located higher in the hierarchy than the most general class distinguished in [Eng84] and [DJL84]. It offers tool support for the specification of data structures and operations on them, using pattern matching and a generalization of attribute grammars. Since our primary interest is not in the generation of target code for processors, but on generating high level code, e.g. in C, no specific support for machine code generation is present. Elegant generates ANSI-C and is available on many platforms. It is efficient, in the sense that Elegant generated compilers can rival with hand written or lex-yacc based ones. Moreover, as the proof of the pudding is in the eating: Elegant is made by means of Elegant.
In this thesis, we discuss the sources of inspiration that led to the current version of Elegant. These sources can be found to a large extent in the field of functional programming, a field in which the author was employed before he devoted himself to compiler generation. We show how the different compiler constituents can be specified by functional programs and use these specifications to obtain more insight in their structure. Although inspired by functional programming, Elegant does not particularly encourage this style of programming. Although functions are first class citizens, Elegant is an imperative programming language: that is where its efficiency comes from.

1.2 Historical background of functional programming

In this section we discuss briefly the history of functional programming. Functional programming has its roots in the λ-calculus, a formalism that was designed in the thirties [Chu41] and has been extremely well studied [Bar85]. It is a very simple, yet very powerful formalism, that has universal computational power. The λ-calculus offers two major notions as its building block: that of function definition (λ-abstraction) and of function ap-

plication. Thus the λ-calculus deals with *functions*. In the pure λ-calculus, three different forms of expressions exist, a variable, e.g f', function application, denoted by juxtaposition of the function and its argument, e.g. f x and λ-abstraction, denoted $\lambda x.e$, which denotes the function with formal argument x and body e. The semantics of the λ-calculus is given by the simple rule that the expression $(\lambda x.e_1)\ e_2$ can be rewritten into e_1 with every (free) occurrence of x replaced by e_2, i.e. application of a function to an argument is defined by substitution of that argument into the body of the function. Expressions $\lambda x.e_1$ and $\lambda y.e_2$ are equivalent when e_2 can be obtained from e_1 by replacing every (free) occurrence of x by y. A pleasant property of the λ-calculus is that the order of the rewriting of sub-expressions does not matter: every order results in the same result, provided the rewriting process terminates. This property is called the Church-Rosser property and the unique result of an expression is called the normal form of that expression.
The λ-calculus was used as the basis for one of the first computer languages LISP by McCarthy [McC62]. Although very general and powerful, due to the fact that functions were first class citizens (in fact, everything in the λ-calculus is a function), pure LISP was hindered by a lack of expressiveness and efficiency. Thus, many predefined constants (including standard functions) were added and side-effects were introduced. Nowadays, LISP is not considered a pure functional language, due to the presence of these side-effects, while nobody writes pure LISP, due to its inefficiency.
It took almost two decades before the λ-calculus was rediscovered as a basis for another functional programming language, namely SASL [Tur79]. While LISP offers a syntax that is a direct reflection of the λ-calculus, SASL offers a much more user-friendly syntax, resulting in very concise and readable programs. The SASL implementation is based on so-called *combinators*. Combinators are constant functions that can be added to the λ-calculus [Sch24]. These combinators can be used to rewrite an expression in the λ-calculus in such a way that all λ-abstractions, and thus the corresponding variables, are removed. In theory, three combinators[1] suffice to express an expression without λ-abstraction. These three combinators are S, K and I. They are defined by:

$$\begin{aligned} S\ f\ g\ x &= (f\ x)\ (g\ x) \\ K\ x\ y &= x \\ I\ x &= x \end{aligned}$$

i.e. S doubles an expression, K destroys one and I preserves it. By means of the following rewrite rules, every expression in the λ-calculus can be rewritten into an SKI expression:

$$\begin{aligned} \lambda x.(e_1\ e_2) &= S\ (\lambda x.e_1)\ (\lambda x.e_2) \\ \lambda x.x &= I \\ \lambda x.y &= K\ y, \qquad \text{provided } x \neq y \end{aligned}$$

By applying these rules to an expression, all λ-abstractions and consequently, all variables bound by it can be removed. The resulting expression can then be reduced to its normal form by a process called *graph reduction*. In this reduction process, the expression is represented by a graph that is rewritten by applying the transformation rules of figure 1.1 to it. If it is no longer reducible, we say that it is in normal form.

[1]Even two, since I = S K K, or even one, see [Fok92a] for a derivation and overview.

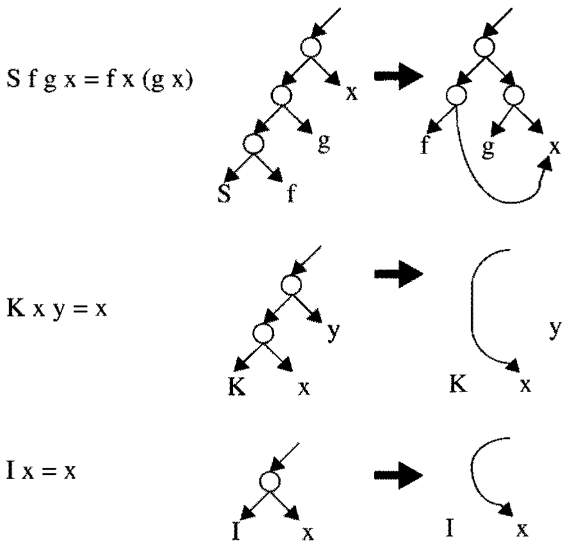


Figure 1.1: Graph reduction rules

The order in which the graph reduction takes place is a matter of choice. When rewriting the outermost expression first, the so-called normal order reduction is obtained. This means that arguments are passed *unevaluated* to a function. In terms of the represented expressions this has advantages, since some functions do not evaluate all their arguments. This evaluation technique is also denoted by the term *lazy evaluation* and it is the technique employed by SASL and many modern functional languages. LISP and ML use the opposite technique, evaluating the innermost terms, resulting in a call by value mechanism where arguments are evaluated before a function is applied. Although potentially more efficient, this technique results in a less expressive formalism, due to the stronger termination requirements.
Combinators and graph reduction became rapidly popular in the eighties and many new functional languages were based on it, of which we mention KRC [Tur82], Miranda [Tur86], ML and Lazy ML [GMW79, Aug84] and Haskell [HWA$^+$92]. They all offer syntactic sugar to the λ-calculus and in that sense they can be seen as merely syntactic variants of the λ-calculus. Yet, the amount and the form of the sugar is such that programming in a modern functional language is a radically different experience to programming in the λ-calculus. A functional language offers a compact and clear notation and the expressiveness of such is a language is considarably enhanced by the use of pattern matching (which allows the specification of different cases for a function without specifying the case analysis process itself), special notational support for lists, of which the so-called list-comprehensions [Tur82] are very convenient, and other forms of syntactic sugar. With respect to imperative languages, functional languages have the additional advantage that a programmer needs not to deal

with the concept of a modifiable state and hence needs not to worry about the order of execution. Moreover, functional languages offer automatic storage management by means of a garbage collector.
Early functional languages like SASL and KRC suffer from two important draw-backs: lack of efficiency and lack of some form of modularity and type checking. The first is caused by the very small steps that SKI reduction performs, the second by the lack of a proper type system. Both problems have been tackled. SKI combinators were first generalized to a richer set of fixed combinators and later to so-called super-combinators [Joh84], which were no longer predefined but generated by the compiler. Further optimizations [BPR88] in this field brought to light that this implementation is equivalent to the traditional compilation technique for imperative languages, the only difference being that invocation frames are stored on the heap instead of on the stack, an idea that dates back to Landin [Lan64] and already used in Simula-67 [DN66]. This traditional compilation technique was used by Cardelli [Car84] for the compilation of a non-lazy functional language and in [Pey87], page 379, it is stated that the super-combinator translation would be necessary to preserve full laziness. The invocation frame-based approach can support full laziness quite easily, namely by storing arguments as unevaluated objects (closures) in an invocation frame. One could say that the semantic foundation on the λ-calculus, and to an even larger extent the associated syntactic foundation, has impeded the development of this obvious translation scheme for more than 10 years.
The lack of protection was overcome by the development of advanced type systems. This was not an easy job, since a type system works like a filter on the set of accepted programs. A functional programming language tends to allow very general and abstract formulations by an abundant use of higher order functions, and a good type system should not limit this expressiveness. Advanced type systems, including polymorphic functions and parameterized types, have been developed. Small untyped functional programs are very concise and readable, and in order to retain this property, so-called *implicit* type systems were developed [Hin69, Mil78] that could deduce the types from an untyped program. Although this gives good support for small educational programs, we believe that large pieces of software should be documented more extensively and that type information forms an essential ingredient of this documentation. However, we do not know of any modern functional language supporting an explicit polymorphic type system, although most languages support the optional specification of types.
For a good overview of the implementation techniques currently used for functional programming languages, see [Pey87].

1.3 Historical background of `Elegant`

Esprit project 415, 'Parallel architectures and languages for Advanced Information Processing – A VLSI-directed approach', ran from November 1984 until November 1989. In this project, different sub-contractors have developed different parallel architectures for different programming styles, object-oriented, functional and logical. From these sub-projects, sub-project A was contracted by Philips Research Laboratories, Eindhoven (PRLE). Within this

sub-project, a new Parallel Object-Oriented Language, called POOL was developed. Simultaneously, a distributed computer, initially called DOOM (for Distributed Object-Oriented Machine) was developed. This acronym was later on replaced by POOMA (Parallel Object-Oriented MAchine).
In the beginning of the project, when one of the first variants of the POOL language family, called POOL-T [Ame87], had been designed and the completion of the parallel machine was still far away, it was decided to implement POOL-T on a sequential machine and simulate its parallel behaviour. This language was compiled to Modula-2, using the co-routine facility of Modula as a basis for the concurrent execution. A compiler for POOL-T was designed and specified carefully by means of an attribute grammar. This grammar was written in the form of a LaTeXdocument [dHO87], which was hard to maintain. The compiler was appropriately called SODOM, which is an acronym for Sequential Object-Oriented Development On Modula. After its design, it was implemented by means of `lex`- and `yacc`-like tools, which were developed by the Twente University of Technology [SK84]. The newly constructed compiler was embraced by the POOL community and soon demands for bug-fixes and more efficient execution were expressed and satisfied. In this process of compiler maintenance, a deliberate attempt was made to keep both the specification (the attribute grammar [dHO87]) and the implementation consistent. These efforts were in vain, however. Especially the fact that certain optimizations were straightforwardly encoded by means of side-effects, but hard to specify in a declarative formalism like an attribute grammar, caused this deviation. From this experience we learned the lesson that whenever possible, one should generate an implementation from a specification, which meant in this case that a compiler generator based on attribute grammars was needed and that the attribute grammar formalism that would be employed should allow side-effects.
At that time, in the beginning of 1987, we had the GAG-system [KHZ81] at our disposal, and this seemed the most natural tool to use. Just at that time the author of this thesis had developed a new technique for the implementation of lazy (demand-driven) evaluation and it looked ideal as an implementation technique for attribute grammars. Not only was it simple, but it would allow the use of any non-circular attribute grammar [Eng84], which is a wider class of attribute grammars than the class of ordered attribute grammars [Kas80] which where accepted by GAG. The technique looked so good that it was almost immediately decided to exploit it further. To this end, we decided to boot-strap the new compiler generator by means of the GAG system. This version could then be used to generate itself, and the GAG- and self-generated versions could be compared. But first a new acronym had to be found and after a good deal of discussion the new generator was called *Elegant*, for Exploiting Lazy Evaluation for the Grammar Attributes of Non-Terminals, a rather accurate name.
In three weeks time `Elegant` was boot-strapped and made self-generating. The steps in this process are depicted in the diagram of figure 1.2.
In this diagram rectangular boxes indicate input or output of a program, while an executable program is represented by an oval box. In the case of a compiler that generates executable code, the output is also executable and we represent it by an oval box. Two versions of `Elegant` appear in this diagram, one GAG-generated (Elegant-1) and one self-generated (Elegant-2). These versions are functionally equivalent, but behaved very differently in

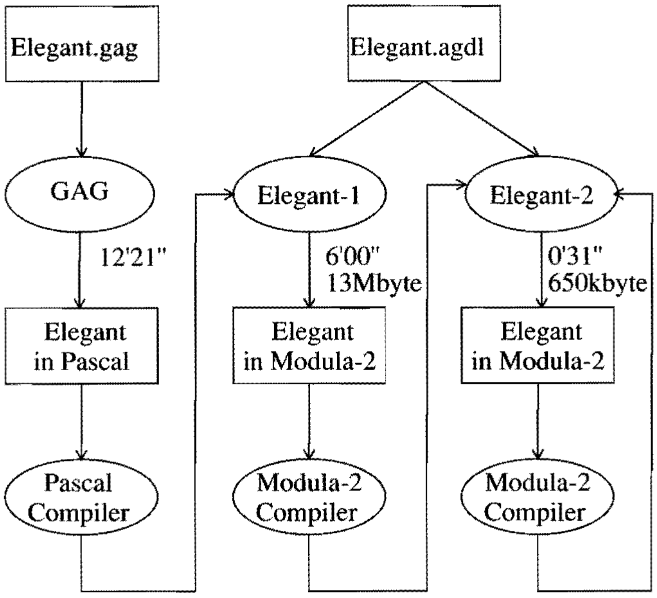


Figure 1.2: The `Elegant` boot-strapping process

terms of performance. The self-generated version consumed 12 times less time and 20 times less memory than the GAG-generated one. This was mainly caused by the fact that the GAG programming language (Aladin) was by that time strictly applicative and did not allow side-effects, which forced us to encode the code-generation by the construction of a single string, which could be written to a file at the end of the compilation process by a special facility. Of course, these results encouraged us and GAG was swiftly put aside, while `Elegant` was extended with a scanner generator, context checks and many more facilities to turn it into a professional tool. `Elegant` has been used subsequently for the construction of the compilers for two other POOL languages, POOL2 and POOL-X [Ame88, Ame89], and for numerous other compilers within Philips. `Elegant` has survived its source; DOOM and POOMA are no longer operational at PRLE, the POOL implementations are no longer supported, but `Elegant` has developed from a compiler generator developed to ease the implementation of POOL, initially just capable of compiling attribute grammars, into a professional compiler generator tool-set. It now implements a modern programming language (also-called `Elegant`), offering attribute grammars as a sub-formalism, and features like polymorphism, parameterized types, sub-typing, pattern matching and laziness. Several other tools were added and nowadays `Elegant` is not only a programming language, but also a system consisting of various tools, each of them being a compiler and each of them generated by `Elegant`. These tools are, of course, `Elegant` itself [Aug92b], a scanner generator [Aug92f], a compiler from EBNF rules to attribute grammars [AM92] and a compiler from EBNF rules to syntax diagrams [Aug92c]. Several libraries implementing simple and complex polymorphic functions are available [Aug92e]. An overview of the system can be found in [Aug90a, Aug92d] and in this thesis. A tutorial, which serves as a gentle introduction to `Elegant`is [Jan93].
The structure of a compiler created with `Elegant` is depicted in figure 1.3.
We concentrate on the part of this picture contained in the dashed box on the left-hand side first. Assuming one wishes to implement a formal language `X`, starting with a context-free grammar, one first specifies this grammar in an EBNF form, which is compiled by the `Bnf` tool into a scanner specification which must be completed by the user and into an grammar in the `Elegant` formalism. These specifications are compiled by `ScanGen` and `Elegant`, respectively and implement a context-free parser for the language `X`. This parser can be extended by adding attributes to the file `X.agdl` and implementing auxiliary operations in the `Elegant` programming language in specification/implementation module pairs, as indicated by the modules outside of the dashed box. In this way, a complete compiler can be constructed. For a good pictorial representation, the `Diagrams` tool can be used, which accepts a superset of EBNF rules and compiles them into syntax diagrams in PostScript™. The syntax diagrams in the appendix to this thesis have been generated by this tool.

1.4 Compiler construction

The structure of an `Elegant` generated compiler corresponds to the forementioned decomposition of a compiler into sub-functions, namely compiler = code-generation $\circ$ optimize $\circ$ make-data-structure $\circ$ parse $\circ$ scan. This is depicted in figure 1.4.

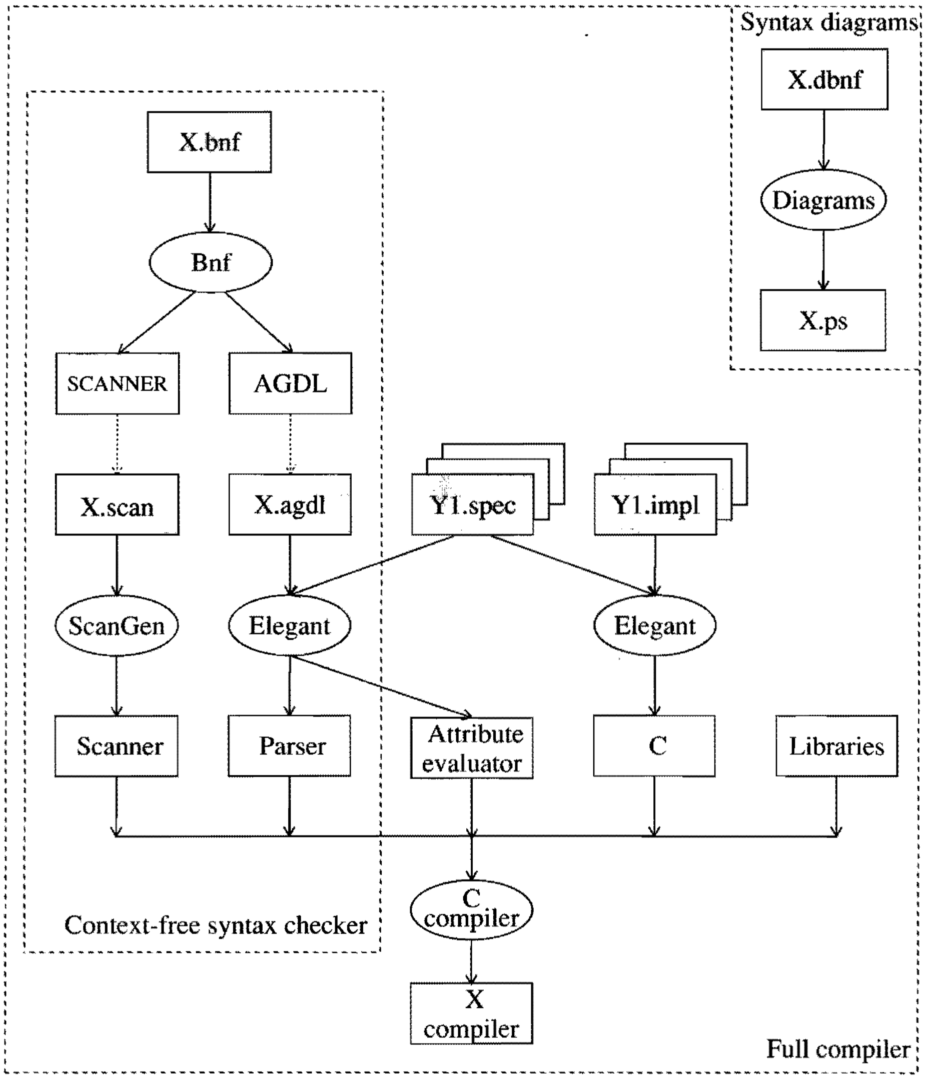
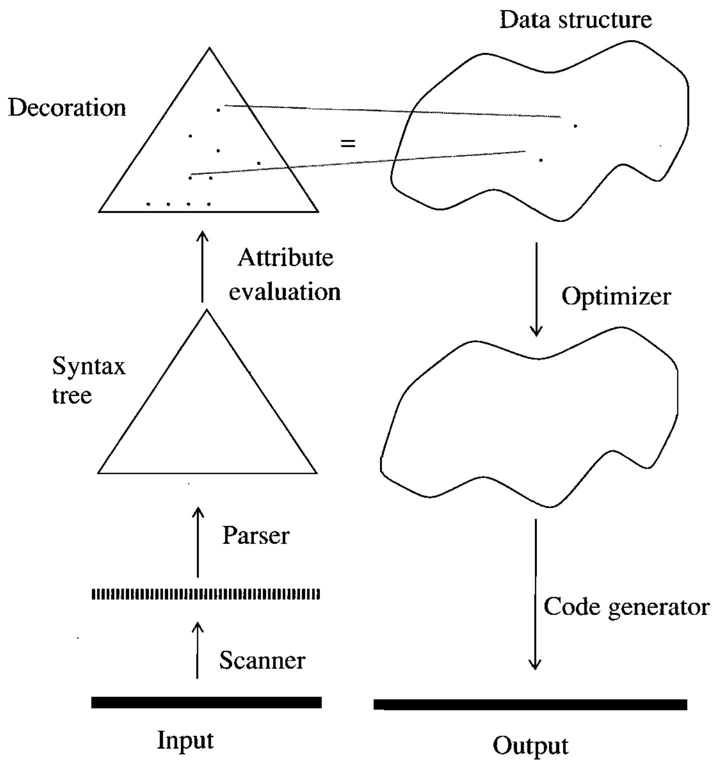


Figure 1.3: The structure of `Elegant` compilers



Compiler = code-generation ∘ optimize ∘ make-data-structure ∘ parse ∘ scan

Figure 1.4: The structure of a compiler

The scanner is generated by `ScanGen`. Its use is to partition the input string into a sequence of words, the so-called terminal symbols. The parser and attribute evaluator (which creates the data structure) are generated by `Elegant` from an attribute grammar. A parser parses the sequence of terminal symbols into a so-called syntax tree. It groups sequences of terminals into composite notions, the so-called non-terminal symbols, according to a grammar. It produces error messages for a syntactically incorrect input string. Subsequently, properties of these non-terminal symbols are computed in the form of so-called attributes. The attribute grammar describes how to compute these properties for each of the non-terminal symbols. These properties should obey certain consistency relations, specified in the attribute grammar in the form of so-called context conditions. When a context condition is not fulfilled, a suitable error message is produced.
The scanner, parser and attribute grammar map an input string onto a data structure which is an abstract model of the input string. Since this data structure is a graph, we call it the abstract syntax graph for the input string. To this graph, certain transformations can be applied which preserve its meaning. The intent of such transformations is to obtain a more efficient representation for the input string and this transformation process is hence called optimization. It is best specified by means of pattern matching techniques which allow the matching on a sub-graph of the data structure. The matching can specify an optimizable structure, which can be transformed into a more efficient one.
Code generation is a process similar to optimization. Again pattern matching is used to recognize certain sub-graphs for which a particular kind of code is generated. `Elegant` offers a general pattern matching mechanism for the construction of optimizers and code-generators. For the latter, a special mechanism, the so-called relations, is available. These relations allow the specification of a pattern for graph matching together with the specification of multiple pieces of code which can be generated independently for a single sub-graph. Thus, a single match can be used to generate different pieces of code and in this way, the consistency of these pieces can be guaranteed easily.
In the next sections of this introduction we will discuss these compiler components and the techniques underlying them in more detail.

1.4.1 Scanning

Conventionally, lexical scanners are expressed by means of *regular expressions* over a set of characters [ASU86]. These regular expressions are implemented by means of different kinds of finite automata, starting with a non-deterministic one, transforming it into a deterministic one and finally minimizing the deterministic automaton. In chapter 5 of this thesis, we concentrate on the expression of scanners in a different way, namely by means of functional programs. Functional programs can be rewritten by algebraic transformations in such a way that the transformations on the scanner functions are similar to the ones on automata. Where automata need at least two formalisms, one to express an automaton (possibly in different formalisms for different classes of automata) and one for the algorithm implementing the execution of the automata, only a single formalism, a functional programming language, is needed by our approach. In this way, automata theory is no longer necessary, which simplifies the treatment of scanning algorithms.

1.4.2 Parsing

Although LALR(1) parsers are dominant in compiler generators, many more different parsing techniques have been developed. These parsers are applicable for different classes of context-free grammars, e.g. a set of recursive functions can be used for parsing LL(1) grammars, stack automata for LR grammars and table-driven parsers for ambiguous grammars. However, until recently, these different techniques were only related weakly. They could be related by means of automata, but this was not the most natural way of representing each of them. With Kruseman Aretz' [Kru88] and Roberts' [Rob88] recursive ascent parsers this state of affairs changed. They were the first to recognize that LR grammars could be parsed by a set of recursive functions as well. Before that, a recursive ascent parser had already been described by Penello [Pen86] as an efficient way to implement an LR automaton, but without recognizing its recursive structure. Similar recent recursive ascent parsers can be found in [HW90] and [Hor93]. `Elegant` implements the Kruseman Aretz parser of [Kru88]. In [LAK92], the recursive ascent technique is extended to parsers for non-deterministic languages.

In this thesis, in chapter 3, we go a step further. We start with a functional program for a non-deterministic top-down parser and apply recursion elimination to it. We use different recursion elimination techniques which are algebraic program transformations that can be applied to a wide class of recursive programs. In a functional language, recursion elimination means the transformation of one of the recursive calls into a tail-recursive call, leaving any other recursive calls unchanged. The result of the application of these transformations to a top-down parser is a number of bottom-up (LR) parsers, among which parsers that are equivalent to conventional automata-based LR(1) and LALR(1) parsers. The conclusion of these transformations is that LR parsers are nothing but the iterative versions of recursive descent parsers and that the application of automata theory has obscured this simple fact for more than twenty years.

The history of tabulating parsers for non-deterministic languages, like the ones of Cocke, Younger and Kasami [You67, Kas65], Earley [Ear70] and Tomita [Tom86], is a similar story. Such parsers use a table to store already accomplished parses in order to avoid performing the same parse twice. In this way, parsing in cubic or even sub-cubic (see [Val75]) time is possible. It took about 20 years before the Earley parser was recognized as a memoized bottom-up parser, e.g. by Norvig [Nor91] and Augusteijn [Aug90c]. Memoization is the technique of constructing so-called memo-functions, functions that store the results of previous invocations together with their corresponding arguments in a table and that deliver such a previous result when supplied with the same arguments as before by retrieving it from the table rather than by recomputing it. By simply postulating that a non-deterministic recursive ascent parser is memoized, variants of the Earley and Tomita parsers are obtained. Where the management of the memo-table is the major complication in the Earley and Tomita algorithms, it becomes an implementation detail once the memoization has been recognized as such.

Less well known than the Earley parser is the Marcus parser [Mar80], a non-deterministic bottom-up parser with non-terminal instead of terminal look-ahead, designed for natural language parsing. For a comprehensible presentation see [Lee92].

Parsing of non-deterministic languages can be performed without tabulation as well. It then requires the use of back-tracking. Of course, this can only be done reasonably when the search space is relatively small, i.e. when the amount of non-determinism of the grammar is limited, which excludes natural language parsing. An example of such a parser is the recursive back-up parser of Koster [Kos74]. In this thesis we show that back-tracking parsers can be obtained from recursive descent and ascent parsers by means of (again) algebraic program transformations. The use of so-called continuations is indispensable here.
Another interesting approach to parsing can be found in [Hut92], where a parser is presented by means of higher order functions. The idea is to pair each operation in a context free grammar, such as choice, sequencing, or repetition (in the case of extended BNF), that combines grammar constructions, with a higher higher order function that combines partial parsers corresponding to the grammar constructions. In this way, a morphism between grammars and parsers is constructed. As in the approach in this thesis, non-deterministic parsers appear to be more naturally expressible than deterministic ones by Huttons approach. Huttons shows that his approach is extendable to lexical scanners and attribute computation as well. Where Hutton aims at implementing a grammar directly in a functional language, in a human readable form, we are more interested in *compiling* a grammar. The compilation may transform the grammar or the functional algorithm to be generated if this happens to be advantageous. It can e.g. increase determinism by using bottom-up parsing and adding look-ahead and need not use higher order functions at all in order to obtain readability of the implementation. Our attitude is that generated code need not be comprehensible, but that the generator should be!

1.4.3 Error recovery

A good parser (generator) should not only provide a general parsing technique and an efficient implementation for it, but also a proper error recovery strategy. The research field of error recovery in parsers suffers from a fundamental problem: on the one hand, error recovery is a sort of non-topic: syntax errors should be repaired by the user and in that respect, just presenting the first error would do the job. The user fixes it and reruns the compiler. On the other hand, users expect error messages to be clear and to the point and appreciate compilers that are capable of presenting the minimum number of error messages for a given program, yet without skipping any syntax errors. Thus, error recovery is an unrewarding topic: one can always do better, yet it is of relative importance since programmers are interested in correct programs, rather than erroneous ones.
Error recovery can be described as the technique to bring the state of a parser and the remainder of the input string into phase, after a syntax error has been detected. Two major techniques are used here: deletion of terminal symbols from the input string, or insertion of terminal or non-terminal symbols that are expected but not present.
A parser generator should thus provide a decent, not necessarily optimal, preferably automatic, error recovery strategy. In recursive descent parsers, such a strategy is relatively simple, see e.g. [KvdSGS80]. One simply passes so-called follower sets as additional parameters to the recursive procedures. Such a set contains the terminal symbols that the

deletion process is not allowed to skip during error recovery.
Until recently, because of the lack of recursive procedures, such a technique was not available for bottom-up parsers. Much more complicated techniques have thus been invented for this case. See e.g. [GR75] as a classical example. With the advent of recursive ascent parsers, this has changed, fortunately. Follower sets can be added to the recursive procedures of such parsers as well and this is the way the `Elegant` recursive ascent parser implements it. In this thesis, we show that the most simple way to do this is to use a recursive ascent parser based on continuations.

1.4.4 Attribute grammars

A compiler needs more than a parser to do its job: syntax analysis is the first step, but after that, the input string has to be translated into a data structure representing that input string for further processing. The formalism of attribute grammars has been designed for the mapping of a string onto a data structure [Knu68, Knu71]. This formalism specifies both the context-free grammar (from which a parser can be derived) as well as properties of language constructions, the so-called attributes, simultaneously. Hence, an attribute grammar can specify a complete compiler front-end. Although attribute grammars have a long history, very few compiler generators have been constructed that exploit their full power. The popular Unix tool called `yacc` only offers minimal support. The GAG tool [KHZ81] supports a traditional attribute grammar formalism, but limits the class of attribute grammars to ordered attribute grammars [Kas80]. For a good overview of a large number of systems based on attribute grammars see [DJL88]. A problem with attribute grammars is that they are a declarative formalism: no ordering on the evaluation of the attributes is specified and hence the generator should make up such an ordering. In [Eng84], a thorough overview of such evaluation techniques is given.
`Elegant` avoids the problem by not computing an ordering on the attributes, but using demand-driven evaluation for their values. In [Aug90a] and this thesis we show how demand-driven evaluation can be implemented efficiently, even so efficiently that the overhead of attribute evaluation is neglectable when compared to the rest of the compilation process. `Elegant` does test, however, the existence of an evaluation order, i.e. the absence of cyclic dependencies. This cyclicity test is intrinsically exponential as shown in [JOR75], but for practical grammars its behavior is usually much more benign and optimizations techniques as in [RS82] and [DJL84] are of great use.
Attribute grammars can be given a formal semantics, as is done in [Knu68], [DJL88] and [Eng84, EF89]. These modelings are unsatisfactory however, due to their complexity. They contain many different notions, among which at least the notions of attribute names, attribute values, inherited and synthesized attributes, semantic rules and semantic conditions. Even in these models, attribute grammars remain an open formalism, using external domains and functions in the definition of their semantics. In this thesis we show that the semantics of attribute grammars can be presented in a substantially simpler way. We do so by using a functional language for the modeling of the external domains and functions. By virtue of the expressiveness of functional languages, we do not need to model attributes at all. Instead of associating all the forementioned different notions with a production

rule, we associate a single so-called rule-function with a production rule. The semantics of an input string with respect to an attribute grammar is defined by composing these rule-functions into composite functions. The composition of these functions requires the ability to use higher order functions and partial parameterization, which make functional languages the formalism of choice for implementing them. The attributes are arguments of the rule-functions, but they do not appear in the semantic model: we only need to compose functions without applying them to actual attributes in defining the semantics. This approach resembles that of Johnsson [Joh87], who also uses a functional language to implement an attribute grammar. The techniques used are very similar to ours, the main difference being that Johnsson implements an attributed parser in a functional language, while we start by defining the semantics, allowing any kind of parser to be used to implement it.
Another approach using functions to implement attributes is found in [Kat84]. There, an individual function is constructed for each attribute that computes the value for that attribute. Here, we construct functions for production rules, not for individual attributes. In the `Elegant` implementation however, attributes are evaluated lazily. This is implemented by constructing a function for each attribute, computing the value for that attribute, just like in [Kat84]. But we regard it as a general technique for the implementation of lazy evaluation, not just for implementing attribute grammars. As a result, our approach is much more general than Katayama's, supporting any non-circular attribute grammar and using result caching for the values of the attributes in order to avoid double evaluation (which is just lazy evaluation of course).

1.4.5 Generalizations of attribute grammars

The success of attribute grammars as an abstraction mechanism in compiler construction has led to different generalizations of attribute grammars.
One such generalization consists of attempts to make attribute grammars more modular. Conventionally, a single attribute grammar describes a complete context-free grammar with all its properties in the form of attributes. This solid description can be modularized in two different ways. The first way decomposes it into modules which contain different sets of production rules, e.g. one module containing the rules for the expressions and the other for the statements of a programming language. This form of modularization disables the static cyclicity check however, unless a second pass is added to the attribute grammar compilation. The second way of decomposing attribute grammars allows one to decompose the set of attributes per production rule into subsets, each of which is defined in a separate module. In this way, conceptually different classes of attributes (such as attributes for implementing scope rule or symbol-table handling) can be implemented in different modules. See e.g. [FMY92] and [RT88] for this approach. Also in this case, the static cyclicity check needs access to the complete grammar and requires the modules to be combined in some way before it can be run.
Another generalization are higher order attribute grammars [VSK89], a formalism which allows the definition of a sequence of cascaded attribute grammars. An attribute in one sub-grammar can then serve as an attributable non-terminal in the next sub-grammar. Such grammars can be used to construct a complete compiler by connecting an attribute grammar

for the front-end to an attribute grammar for the back-end (which describes code generation), with possibly attribute grammars in between. A notion that is similar to higher order attribute grammars is the notion of attribute coupled grammars [Gie88].
Associated with higher order attribute grammars is another generalization of attribute grammars, which we call data structure attribution. Where higher order attribute grammars are based on the identification of non-terminals in one grammar and attribute types in a previous grammar, data structure attribution consists of defining production rules that traverse a data structure in a recursive way, usually selecting the production rules by more general pattern matching rather than by matching on the type of a node in a (syntax) tree. The attributes of the production rules can then form another data structure or be integrated in the traversed data structure. Examples of this form of generalization can be found in [Far92], [Fro92a], [Fro92b] and [Gro92]. `Elegant` offers an abstraction of this form too, the so-called relations. These are treated in chapter 7.
Attribute grammars and all their generalizations can be considered as a restricted form of lazy functional programs. See e.g. [Bir84b] for an example of the encoding of an attribute grammar as a functional program and [Joh87] for a discussion on this topic. One might wonder what the use of attribute grammars is in this respect and whether they could be simply replaced by lazy functional programming. The answer lies in the availability of the static cyclicity check for attribute grammars and the ability to compute an evaluation order for the attributes, something which is not possible in an arbitrary functional program. This offers the prospect of a more efficient implementation of attribute grammars. In an attribute grammar, the data structure to be traversed, be it a syntax tree or a more general data structure, is separated from its properties that are computed in the form of attributes. This separation is conceptually simple and allows for an optimized implementation.
This separation has another advantage that is exploited by the next generalization, called incremental attribute evaluation. It allows for the interactive modification of the syntax tree (or other data structure) and the subsequent recomputation of the affected attributes. This technique is particularly useful for the construction of syntax directed editors and (more generally) structure editors. See e.g. [KS87], [BvZ89] and [RT88] for this generalization.

1.5 Functional programming

The abstraction mechanisms of `Elegant`, although an imperative language, have been influenced to a large extent by functional programming languages. These languages offer powerful abstraction mechanisms, such as polymorphic and higher order functions, pattern matching and list-comprehensions. In addition to these abstraction mechanisms, they are clean, in the sense that due to the absence of side-effects, they allow the definition of algebraic identities over the set of their expressions. These identities can be used to rewrite expressions just as in algebra, e.g. by applying distributive laws or exploiting associativity. In this thesis, we make extensive use of these so-called *algebraic program transformations*. For a good introduction to functional programming see Bird and Wadler [BW88]. Hughes in [Hug89] shows the power of the different abstraction mechanisms.
A major draw-back of functional languages is their relatively inefficient implementation.

See [PMN88] for a number of problems that are not (easily) expressible in functional languages with the same order of complexity as in imperative languages.

1.5.1 Algebraic program transformations

Due to both the absence of side-effects and to their orthogonal design (especially reflected in the fact that functions are first-class citizens), functional programs lend themselves to *algebraic program transformations*, i.e. the re-writing of one program into another by the application of algebraic rewrite rules. Some simple examples of such rewrite rules are the rule that states that function composition is associative:

$$f \circ (g \circ h) = (f \circ g) \circ h$$

and the rule that states that the map function (that applies its function-valued first argument to all the elements of its list-valued second argument) distributes over function composition:

$$\text{map } (f \circ g) = \text{map } f \circ \text{map } g$$

In this thesis we use such rules very frequently in the derivation of algorithms from their specification and of complicated algorithms from more simple ones. In chapter 2 we present the algebraic identities that we use throughout this thesis.
These transformations were introduced by Backus [Bac78], Burstall and Darlington [BD77] and explored more sytematically by Bird [Bir80, Bir84a, BH87] and Meertens [Mee83] and are sometimes called the Bird-Meertens formalism. In [Bac88], a good introduction to this 'formalism' can be found. In [SS90], a very clear treatment of the limits of these transformations in the presence of non-determinism can be found.

1.5.2 The category theoretical approach

After the introduction of algebraic program transformations, they soon became the subject of mathematical investigation, see e.g. [Spi89, Mal89, Fok92b]. The mathematical treatment of the program transformations is founded in category theory [BW90]. This approach has yielded a large number of very general laws (as category theory always does). On the one hand, these laws give a lot of insight in the structure of programming languages and data types, on the other hand, they can not be treated as a practical substitute to the earlier Bird-Meertens approach. The category theoretical approach first abstracts away from a concrete program, then applies some general laws and then transforms back to a concrete program. The original Bird-Meertens approach rewrites programs in the programming language domain, thus yielding valid programs along the way of a derivation, which are all executable. Although this approach is less general, it is applicable by a programmer that is not trained in category theory, that is, the 'average' programmer.
The category theoretical approach has another draw-back, which is that it is limited by its own tendency to abstract. This abstraction tries to get rid of concrete variables as much as possible, dealing with functions only, which are composed, mapped, lifted, transformed by morphisms into other functions, etc. But where one can get rid of a single variable easily (namely by λ-abstraction), the multiple occurrence of the same variable in a formula

gives more problems. To overcome this, different kinds of combinators are introduced, i.e. functions that have the purpose of duplicating values and shifting them around. We believe that variable duplication arises frequently in real algorithms and programs, and thus that the more practical approach is not to abstract from concrete variables.
For these reasons, we regret that the original approach seems to be abandoned by its originators and that the major line of research currently lies in the category theoretical field. In this thesis, we explicitly limit ourselves to the earlier approach, and rewrite valid programs into new valid ones until we reach the required result, maintaining a moderate level of abstraction in order not to scare away the average programmer.

1.5.3 Pattern matching

One of the powerful and very practical abstraction mechanisms of functional languages is the pattern matching paradigm. It is very elegant, especially in combination with function declaration. It allows the specification of a function by defining a sequence of partial functions, which each have a limited domain which is defined by a pattern over the arguments of the function. Take for example the definition of the already mentioned map function over lists. It is defined separately for empty lists and non-empty lists and thus together for all lists.

```
map f [ ] = [ ]
map f (head:tail) = f head:map f tail
```

Pattern matching can be compiled in an efficient way, see e.g. [Aug85], [BGJ89], [Oph89], [Lav88] and especially [Wad87] for a treatment of the techniques that can be employed. In chapter 6, we show that a pattern can be defined as a *sub-type*, given a suitable type system. This allows us to formalize a rule as a partial function, and the corresponding total function as the union of the corresponding partial functions. This definition can be rewritten by means of algebraic program transformations into an improved implementation for pattern matching. This implementation is the one that is used in `Elegant`, where it is compiled into C-code that inspects each value at most once (and only when necessary) in the selection of the proper rule for a function.

1.6 Recursion elimination

One of the major applications of algebraic program transformations in this thesis is the transformation of recursion into iteration, also known as *recursion elimination.* A good overview of this area can be found in [Boi92]. Other contributions to this subject can be found in [AK82], [Gor65], [HK92] and [PP76].
In chapter 3 we derive (again by means of algebraic program transformations) a family of very general rules for recursion elimination, that are applicable to so-called recursive descent functions, a certain class of recursive functions. Recursive descent parsers are among the functions in this class. By means of the different recursion elimination rules (which are expressed as higher order functions), a recursive descent parser is rewritten into

a recursive ascent one. One of the resulting parsing functions is an LR parser, expressed as a recursive function. The parse stack has disappeared (it has become the recursive stack) and the parse function is a function of grammar rules, rather than of a parse-table that is constructed by auxiliary means. Once familiar with recursive ascent parsers, one sees that an LR automaton is nothing but a low level implementation of recursion.
In [Aug92a], we presented another application of the same recursion eliminating program transformations, namely the derivation of a binary heap traversal function. In this thesis we show it again as an example.

1.7 Typing

Chapter 6 is devoted to typing in both functional and imperative languages. When comparing modern functional and imperative type systems, several important differences become apparent.
The first is that functional languages have a tendency to make the type system *implicit* and use a type inference algorithm to let a compiler deduce the typing from a program that possibly does not contain any typing of variables or functions at all, see [Hin69], [Mil78]. This has the advantage that simple educational programs become very readable, but the disadvantage that large scale programs become less readable due to lack of type information (which may optionally be added by the user however) and that compilation becomes slow due to the type inference performed by the compiler.
However, explicit strong typing seems the way the imperative world has chosen. These explicit type systems usually supply so-called type extensions as introduced by Wirth in its Oberon language [Wir88b, Wir88c, Wir88a] and taken over by the Modula-3 designers [CDG+88, CDJ+89]. Type extensions provide a form of linear inheritance without the usual object-oriented methods and can be seen as a generalization of Pascal's variant records. A type is a record, defining a number of named fields and a sub-type may add a number of fields to those of its super-type.
This brings us to the second important difference between functional and imperative type systems. Although functional type systems contain record types, they offer a very limited form of inheritance. They only allow variants of a type up to one level of sub-typing, like the Pascal variant types, calling such types 'sum-of-product types'. In such a type the fields are *unnamed*, which requires them to be bound to names in pattern matching and which hinders the treatment of patterns as sub-types and has several other disadvantages that we discuss in chapter 6. Nevertheless, attempts have been made to integrate sub-typing with functional programming [FP91], [MMM91], [Wan91] and in algebraic specification languages, [FGJM85].
Object-oriented languages offer a more extended form of type extensions, namely inheritance. Inheritance is a mixture of sub-typing and code reuse. These two concepts are not easily merged into a single mechanism, which is reflected by the fact that the object-oriented community has difficulty with sub-typing as well, confusing inheritance and sub-typing. See e.g. [Coo89] and [CHC90] for an illustration of this. In this thesis, we do not exploit the object-oriented approach, especially for the reason that it can not be combined easily with

pattern matching. Pattern matching causes the selecting of a partial function by matching all the actual arguments, while object-oriented message selection is performed on the type of the first argument (the destination) only. Both mechanisms have advantages: matching multiple arguments is more general, but can not easily be combined with sub-typing over module boundaries, a feature that is essential to the object-oriented programming style.
In chapter 6, we show that sub-typing in the form of type extensions can be smoothly integrated with functional programming and that it gives a clear semantics to patterns as sub-types as a reward. `Elegant` implements such a type system for the reasons that it is both very expressive due to the sub-typing ability and pattern-based function definition and efficient due to the implementation techniques for sub-typing and pattern matching.
A third important distinction between modern functional and imperative type systems is the presence of *polymorphism* in functional languages, which is the ability to declare a single function for many different types by a single definition (in contrast to overloading, which is the ability to declare many functions with the same name for many different types). For a good treatment of polymorphism, see [CW85] and [Car87]. For a proposal to integrate overloading with polymorphism in functional languages, see [WB89].
It is very unclear why this powerful abstraction mechanism has not (yet) become popular in imperative programming. A simple form of polymorphism can be found in object-oriented programming and some imperative languages that allow parameterized types or modules, where generic operations on such types or in such modules can be defined. Usually, it is not possible, however, to even define e.g. a simple polymorphic function like the function composition operator. For a way to integrate type inference with sub-typing see [FM90]. For combining polymorphism with object-oriented programming, see [GB89].
In chapter 6, we explain how `Elegant` combines sub-typing with general polymorphism and overloading.

1.8 Level of abstraction

Given the fact that memo functions date from 1968 [Mic68] and that recursion is even older, one may wonder why fairly simple abstractions, like memo functions and recursive ascent functions, have been recognized so late in parsing theory, one of the oldest and most thoroughly investigated fields of computer science. It seems likely that a strong pre-occupation with imperative programming using data structures like arrays, stacks and finite automata has hindered the recognition of abstractions like memo functions and recursive implementations.
Memoization has been rediscovered many times and several interesting papers about it have been published, of which we mention Richard Birds [Bir80] and [Bir84a], the papers [Aug90c], [Lee92] and [Nor91] that describe memoized Earley parsers, the more general papers on memoization [Hil76], [Hug85] and [War92] and, of course, the original paper of Michie [Mic68].
Once one has seen imperative algorithms explicitly manipulating stacks or tables instead of using recursion or memoization, one becomes suspicious when encountering a stack or a table. Examples other than the automaton-based LR parsers or the Earley and Tomita

parsers are then soon found. The Tarjan algorithm [Tar72] which computes the strongly connected components of a graph in linear time and which is conventionally described by means of a stack of sets of nodes can be described much more clearly in a recursive way. A very simple example is the Warshall algorithm [War62] that computes the shortest paths in a graph (or equivalently: the transitive closure of a relation). This algorithm maintains an array of distances between nodes, which appears to be an implementation of the memo-table of a memoized function. Let the number of nodes be n, and let them be numbered $1 \ldots n$. Let the direct distances along the edges be given by the function D, such that D j k is the direct distance from j to k (which equals ∞ when no edge between j and k exists). Then the length of the shortest path can be inductively defined as follows. Let d i j k be the distance between j and k along paths containing only intermediate nodes $m \leq i-1$. Then the length of the shortest path with intermediate nodes $m \leq i$ is either the length of the path not including i, or a path via i. Thus the following recursive definition can be given:

```
d 0 j k = D j k
d i j k = min (d (i-1) j k) (d (i-1) j i + d (i-1) i k),   i > 0
```

Now assume that the function d is memoized. Then the maximum number of invocations for this function equals the total amount of possible different arguments, which equals n^3. Each invocation takes $O(1)$ time, yielding as a total complexity $O(n^3)$, which equals that of the Warshal algorithm. The space complexity of the memoized function is $O(n^3)$, which is larger than the $O(n^2)$ of the Warshall algorithm. This is caused by the fact that the $(i-1)$-th row in the memo-table can be overlaid by the i-th row in an imperative version.
Another abstraction mechanism that is offered by functional and some imperative languages (like `Elegant`) and that is often explicitly implemented over and over again is that of pattern matching. As mentioned, this allows the definition of a function as the union of several partial functions. This has the great advantage that a programmer only needs to specify the different *cases* and not the *case analysis*. Especially in a compiler back-end, this has the advantage that one can specify a general rule and later on add more specific ones to describe optimized code generation, with pattern matching on the special properties of the construction that can be translated in an optimized way. No modification of the existing code need take place, only addition of the new rules. Moreover, pattern matching can be translated into case analysis by a compiler much more efficiently than it can be hand-coded.

1.9 Contents of this thesis

We conclude this introduction by summarizing the contents of the different chapters.

- Chapter 1 is this introduction.

- Chapter 2 is a brief introduction to functional programming and algebraic program transformations. It lists a number of algebraic identities that will be used in the transformations throughout this thesis.

- Chapter 3 describes a family of recursion removal transformations and shows how these transformations can transform a recursive descent parser into a recursive ascent one. A family of such parsers is derived in this way and it is explained how they are related. We show how the Earley and Tomita parsers can be obtained by memoization.

- Chapter 4 models attribute grammars by associating higher order functions with the production rules of a context-free grammar. It explains how these higher order functions can be implemented efficiently in combination with an LL(1) recursive descent parser. We present a new class of attribute grammars, the so-called pseudo circular attribute grammars (PC-AG), that form a generalization of the well known class of non-circular attribute grammars (NC-AG) and give a general scheme for the construction of an attribute grammar for a wide class of programming languages.

- Chapter 5 expresses lexical scanners as functional programs and presents algebraic program transformations as an alternative to finite automata in the derivation of such scanning functions. As a reward, attribution of lexical scanners is trivial.

- Chapter 6 discusses both functional and imperative type systems and shows how they are combined in `Elegant`. It derives an efficient implementation of pattern matching by means of algebraic program transformations.

- Chapter 7 presents a generalization of attribute grammars, the so-called attribute functions and explains how pattern matching can replace a parser in the selection of attributed rules. This generalization is particularly useful in the specification of code generation and extensively used by `Elegant` itself.

- Chapter 8 concludes with an evaluation of `Elegant` as a formalism and of its efficiency: the imperative programming language `Elegant` aims to combine many abstraction mechanisms from functional languages with the efficiency of imperative languages. Moreover, it offers support for the implementation of compilers, itself being one of these.

Chapter 2

Notations

2.1 Introduction

In this chapter we introduce the notations that we will use in this thesis. These notations form a simple functional programming language. Functional languages allow the transformation of programs by means of algebraic identities, due to the referential transparency that these languages exhibit. We present a large number of such identities that we will be using in this thesis.
We denote a sequence of transformation steps by successive expressions, separated by = signs, which are labeled by the identity being applied, as in the following example:

$(a+b).(a-b)$
  = (distributivity of . over −)
$(a+b).a-(a+b).b$
  = (distributivity of . over +)
$a.a+a.b-(a.b+b.b)$
  = (distributivity of − over +)
$a.a+a.b-a.b-b.b$
  = (definition of −)
$a.a-b.b$
  = (definition of $x^2 = x.x$)
a^2-b^2

Our transformations will exhibit this same structure, but instead of transforming algebraic expressions over numbers, we will transform functional programs.
Section 2.6 contains a brief introduction to functional programming that can be skipped by those readers that are familiar with it.
We will be frequently using set-valued functions in this thesis. These functions can be seen as a way to implement relations. The technique of using set-valued functions and alternative ways to represent relations is discussed in section 2.7.

2.2 Notations and definitions

The notation we use in this thesis for functional programs resembles that of Haskell [HWA+92] and other modern functional languages. Function application is denoted by juxtaposition of the function and its argument, e.g. 'f a' meaning f applied to a. It has a higher priority than any operator and it is left associative. We use parentheses whenever appropriate. Pattern matching is used in defining functions. See section 6.3 for a formal semantics of pattern matching.
Our basic data types are (possibly infinite) trees, lists, bags and sets. Of these, lists and sets are the most important ones. As observed in [Boo79], these data types are strongly related, as the following definition shows.

Definition 2.1 (boom-hierarchy)

Let A^* denote the set of trees, lists, bags or sets over A. Then A^* is defined as the smallest set satisfying:

$$\begin{array}{ll} & \{\} \in A^* \\ \mathsf{a} \in A & \Rightarrow \{\mathsf{a}\} \in A^* \\ \mathsf{x} \in A^* \wedge \mathsf{y} \in A^* & \Rightarrow \mathsf{x} +\!\!+ \mathsf{y} \in A^* \end{array}$$

with

$$\{\} +\!\!+ \mathsf{x} = \mathsf{x} +\!\!+ \{\} = \mathsf{x}$$

□

In general this definition defines the set of binary trees over A, denoted by *Tree*(A). If $+\!\!+$ is associative however, A^* is the set of lists over A, denoted by *List*(A). Moreover, if $+\!\!+$ is also commutative, then A^* is the set of bags over A, denoted by *Bag*(A). If in addition $+\!\!+$ is also idempotent, then A^* is the set of sets over A, denoted by *Set*(A). When $+\!\!+$ is associative, we abbreviate $\{x_1\} +\!\!+ \ldots +\!\!+ \{x_n\}$ by $\{x_1, \ldots, x_n\}$. Thus, this expression can denote a list, a bag as well as a set of elements. It will be clear from the context which of these types is actually meant.
Tuples are denoted by $(\mathsf{x}, \mathsf{y}, \ldots)$. Sometimes we distinguish between the one element tuple (a) and a itself.
If $\otimes$ is an operator of type $A \times A \mapsto A$, then we define the operator '/' of type $(A \times A \mapsto A) \times A^* \mapsto A$ by:

$$\begin{array}{lcl} \otimes/\{\mathsf{a}\} & = & \mathsf{a} \\ \otimes/(\mathsf{x} +\!\!+ \mathsf{y}) & = & (\otimes/\mathsf{x}) \otimes (\otimes/\mathsf{y}) \end{array}$$

If $\otimes$ has left and right neutral elements e_l and e_r, such that $e_l = e_r$, then we define:

$$\otimes/\{\} = \mathsf{e}$$

When $\otimes$ is a binary operator, we use the following partial parameterizations (also called the section notation):

$$((\otimes)\ x\ y) = (x \otimes y) \qquad \text{(op-0)} \quad (2.2)$$
$$(\otimes\ y)\ x = (x \otimes y) \qquad \text{(op-r)} \quad (2.3)$$
$$(x\ \otimes)\ y = (x \otimes y) \qquad \text{(op-l)} \quad (2.4)$$

We define the following basic functions:

$$\mathsf{I}\ x = x \qquad \text{(I)} \quad (2.5)$$

$$(f \circ g)\ x = f\ (g\ x) \qquad \text{(compose)} \quad (2.6)$$

$$\begin{aligned} f^0 &= \mathsf{I} \\ f^n &= f \circ f^{n-1}, n \geq 1 \end{aligned} \qquad \text{(f-power)} \quad (2.7)$$

$$a : x = \{a\} +\!\!+ x \qquad \text{(cons)} \quad (2.8)$$

$$\mathsf{hd}\ (a : x) = a \qquad \text{(hd)} \quad (2.9)$$
$$\mathsf{tl}\ (a : x) = x \qquad \text{(tl)} \quad (2.10)$$

$$\mathsf{single}\ a = \{a\} \qquad \text{(single)} \quad (2.11)$$

$$\begin{aligned} \mathsf{map}\ f\ \{\} &= \{\} \\ \mathsf{map}\ f\ \{a\} &= \{f\ a\} \\ \mathsf{map}\ f\ (x +\!\!+ y) &= \mathsf{map}\ f\ x +\!\!+ \mathsf{map}\ f\ y \end{aligned} \qquad \text{(map)} \quad (2.12)$$

$$\mathsf{concat} = +\!\!+/ \qquad \text{(concat)} \quad (2.13)$$

$$\begin{aligned} \#\ \{\} &= 0 \\ \#\ \{a\} &= 1 \\ \#\ (x +\!\!+ y) &= \#\ x + \#\ y \end{aligned} \qquad \text{(size)} \quad (2.14)$$

$$\begin{aligned} f \circ_1 g &= f \circ g \\ f \circ_n g &= (f\ \circ\ ) \circ_{n-1} g,\ n \geq 2 \end{aligned} \qquad \text{(compose-n)} \quad (2.15)$$

The last definition (2.15) can be stated more comprehensibly as

$$(f \circ_n g)\ x_1\ \ldots\ x_n = f\ (g\ x_1\ \ldots\ x_n)$$

2.3 Comprehensions

A well known mathematical notation is the set comprehension notation, e.g.

$$\mathit{triples} = \{(a,b,c) \mid \exists a,b,c \in \mathbf{N}, a^2 + b^2 = c^2\}$$

This notation was introduced as a programming language construction in [Tur82] and has later on been incorporated in language like Miranda [Tur86] and Haskell [HWA+92]. Since there is no reason to limit the use of this notation to functional programming languages, `Elegant` also contains a comprehension notation.

For the notation to be useful for a programming language, it must be given a precise semantics and modified into a form that is computable. Particularly, the existential and universal quantifications must be removed. In this thesis, we will often start with a specification, using comprehensions which contain existential quantification and transform this specification into a form free of such quantifiers which is *effectively computable*. This approach has two disadvantages. On the one hand, it is less clear which are the dummy variables, bound by the quantifiers. We solve this problem by postulating that every variable introduced in a pattern (see below) which is not yet defined in the environment is a variable that iterates over a given set. On the other hand, it is less clear which of our expressions should be interpreted mathematically and which one as programs. In effect, all should be treated as *programs*, but some are not effectively computable due to the quantifiers they contain. The reader is warned here not to read these expressions as mathematically expressions in the traditional sense. They are programs, but ones that can be manipulated algebraically, like mathematical expressions.

An advantage of the comprehension notation for programming is that it can be generalized from a set-notation onto a bag or list-notation. Definition (2.1) gives the basis for this generalization. In programming languages, comprehensions are usually restricted to lists for efficiency reasons, but there is no technical reason for this. The comprehension notation in `Elegant` goes further and can be used to range over arbitrary types that can be destructed into elements (like sets, lists or files) and to deliver elements arbitrary types that can be constructed (like sets, lists, files, or side-effects). In section 8.2.2 we present this general form of comprehensions.

So what are these comprehensions? List, bag and set comprehensions provide a concise notation for list, bag and set-valued expressions. We define its syntax by the following EBNF rules:

$$\begin{array}{lcl}
\langle\textit{comprehension}\rangle & ::= & \{\ \langle\textit{expression}\rangle\ [\ \mid\ \langle\textit{qualifiers}\rangle\ ]\ \} \\
\langle\textit{qualifiers}\rangle & ::= & \langle\textit{qualifier}\rangle \\
 & \mid & \langle\textit{qualifiers}\rangle\ ,\ \langle\textit{qualifiers}\rangle \\
\langle\textit{qualifier}\rangle & ::= & \langle\textit{boolean-expression}\rangle \\
 & \mid & \langle\textit{pattern}\rangle \in \langle\textit{expression}\rangle \\
 & \mid & \langle\textit{pattern}\rangle \leftarrow \langle\textit{expression}\rangle
\end{array}$$

The Pythagorean triples example above becomes:

$$\mathsf{triples} = \{(\mathsf{a},\mathsf{b},\mathsf{c}) \mid \mathsf{a} \in 1.., \ \mathsf{b} \in 1..(\mathsf{a}-1), \ \mathsf{c} \in (\mathsf{a}-\mathsf{b})..(\mathsf{a}+\mathsf{b}), \ \mathsf{c}^2 = \mathsf{a}^2 + \mathsf{b}^2\}$$

The definitions of the .. operator are given below, but we believe that the user can guess them here.

The semantics of the comprehensions is as follows.

- The expression $\{\mathsf{e}\}$, where e is of type A, denotes the singleton instance $\{\mathsf{e}\}$ of A^*.

- A qualifier that is a boolean expression acts like a filter that removes the elements not obeying the boolean expression.

  $$\{\mathsf{e} \mid \mathsf{b}\} = \textbf{if } \mathsf{b} \textbf{ then } \{\mathsf{e}\} \textbf{ else } \{\} \textbf{ fi} \qquad \text{(bool-set)} \quad (2.16)$$

  An example of this qualifier is the following expression that selects the even elements of a set (list or bag):

  $$\{\mathsf{a} \mid \mathsf{a} \in \mathsf{x},\ \mathsf{even\ a}\}$$

- A qualifier of the form *pattern* $\in$ *expression* can be regarded as a generalization of the boolean form of a qualifier.

  A pattern is a restricted expression that may contain fresh variables, which are bound by matching the pattern to a value. Variables bound by a pattern in a *qualifier* extend their scope to the *expression* as well as the subsequent *qualifiers*.

  When the pattern does not contain fresh variables, i.e. when it is a constant, it is equivalent to the boolean expression performing an element test.

  The semantics of this form is defined by:

  $$\begin{aligned}
  \{\mathsf{e} \mid \mathsf{p} \in \{\}\} &= \{\} && \text{(map-set-a)} \quad (2.17a)\\
  \{\mathsf{e} \mid \mathsf{p} \in \mathsf{x} +\!\!+ \mathsf{y}\} &= \{\mathsf{e} \mid \mathsf{p} \in \mathsf{x}\} +\!\!+ \{\mathsf{e} \mid \mathsf{p} \in \mathsf{y}\} && \text{(map-set-b)} \quad (2.17b)\\
  \{\mathsf{e} \mid \mathsf{p} \in \{\mathsf{a}\}\} &= \{\mathsf{e}\ \textbf{where}\ \mathsf{p} = \mathsf{a}\}, \text{when } \mathsf{p} \text{ matches } \mathsf{a} && \text{(map-set-c)} \quad (2.17c)\\
  \{\mathsf{e} \mid \mathsf{p} \in \{\mathsf{a}\}\} &= \{\}, \text{otherwise} && \text{(map-set-d)} \quad (2.17d)
  \end{aligned}$$

  In this definition, matching means that a substitution for the fresh variables in p can be found, and the where-part binds these variables to the values of this substitution.

  When p is a single fresh variable, say x, it always matches and we have the identity:

  $$\{\mathsf{e}_1 \mid \mathsf{x} \in \mathsf{e}_2\} = \mathsf{map}\ (\lambda \mathsf{x}.\mathsf{e}_1)\ \mathsf{e}_2 \qquad \text{(map-set)} \quad (2.18)$$

  An example of this qualifier is the following expression that maps a set (list or bag) x onto the set (list or bag) containing the squares of the elements of x:

  $$\{\mathsf{a}^2 \mid \mathsf{a} \in \mathsf{x}\}$$

  Another example is the definition of triples above.

  An example of a filtering pattern is the following, filters selects the first elements of those pairs x which second element equals true.

  $$\{\mathsf{a} \mid (\mathsf{a}, \mathsf{true}) \in \mathsf{x}\}$$

- A qualifier of the form *pattern* $\leftarrow$ *expression* can be regarded as a shorthand for the qualifier *pattern* $\in$ $\{$*expression*$\}$. Hence, it may introduce fresh variables and bind these. It has the semantics:

$$\{e_1 \mid p \leftarrow e_2\} \;=\; \{e_1 \mid p \in \{e_2\}\} \qquad \text{(arrow-set)} \quad (2.19)$$

An example of this qualifier is, which combines all three forms of qualifiers is:

$$\text{triples} \;=\; \{(a,b,c) \mid a \in 1.., \; b \in 1..(a-1), \; c \leftarrow \sqrt{a^2+b^2}, \; \text{entier } c\}$$

- Finally, the semantics of the combination of two qualifiers q_1 and q_2 is given by:

$$\{e \mid q_1, \; q_2\} \;=\; \text{concat} \;\; \{\{e \mid q_2\} \mid q_1\} \qquad \text{(double-set)} \quad (2.20)$$

The definition (2.20) is meaningful because it can be proven that:

$$\begin{aligned} &\{e \mid q_1, q_2, q_3\} \\ =\;& \text{concat } \{\{e \mid q_3\} \mid q_1, q_2\} \\ =\;& \text{concat } \{\{e \mid q_2, q_3\} \mid q_1\} \end{aligned} \qquad \text{(set-assoc)} \quad (2.21)$$

which is based on identity (2.48) below and proven in section 2.7.
Another example of the comprehension notation is the function prod that computes the cartesion product of the sets x and y:

$$\text{prod } x \; y \;=\; \{(a,b) \mid a \in x, \; b \in y\}$$

Another example is the function that computes the list of all prime numbers:

$$\begin{aligned} &\text{primes} \;=\; \text{sieve } (2..) \\ &\text{sieve } (a : x) \;=\; a \;:\; \text{sieve } \{b \mid b \in x, \; b \bmod a \neq 0\} \end{aligned}$$

An example of this use of patterns as filters is the definition of set intersection below, where the first qualifier $a \in x$ binds a to the elements of x, while the second qualifier $a \in y$ checks whether a is an element y, by virtue of the fact that a is already defined in the context of this qualifier.

$$x \cap y \;=\; \{a \mid a \in x, \; a \in y\}$$

It is equivalent to the definition:

$$x \cap y \;=\; \{a \mid a \in x, \; b \in y, \; a = b\}$$

The comprehension notation can be seen as a shorthand notation for the application of functions like map, concat and single. This notation can be extended for other such triples satisfying suitable laws. Such a triple of functions (together with a domain) is called a *monad*. Monads have a surprisingly wide area of application, as they appear to be useful as abstractions from iteration, state changing, exception handling, non-deterministic choice, continuations, parsing, call-by-value vs. call-by-name semantics and others. In this thesis, we do not use them since we favor a more explicit treatment of these subjects. Currently, they are popularized by Wadler [Wad90, Wad92], where this notations is called monad-comprehensions and which forms a good introduction to them.

2.4 More useful functions

Some more useful functions are defined:

$$\text{filter f x} = \{\text{a} \mid \text{a} \in \text{x}, \text{f a}\} \qquad \text{(filter)} \quad (2.22)$$

$$\text{split f s} = (\text{filter f s},\ \text{filter} (\neg \circ \text{f})\ \text{s}) \qquad \text{(split)} \quad (2.23)$$

$$\text{first (a, b)} = \text{a} \qquad \text{(first)} \quad (2.24)$$
$$\text{second(a, b)} = \text{b} \qquad \text{(second)} \quad (2.25)$$

In the following definition we make use of overlapping patterns. In such a case we assume that the textually first matching rule is applied. We are aware of the fact that this makes our formalism non-algebraic, when identities are not treated with care. I.e. one may not simply substitute a matching rule, but should use a semantics that subtracts the textually preceding cases from a rule. Such a semantics is given in section 6.3 for the `Elegant` pattern matching mechanism. See e.g. [Ken90] for pitfalls in this fields.

$$\text{zip (a : x, b : y)} = \text{(a, b) : zip (x, y)}$$
$$\text{zip (x, y)} = \{\} \qquad \text{(zip)} \quad (2.26)$$

$$\text{a}\bullet\text{b} = \text{zip (a, b)} \qquad \text{(bullet)} \quad (2.27)$$

$$\text{firsts} = \text{map first} \qquad \text{(firsts)} \quad (2.28)$$
$$\text{seconds} = \text{map second} \qquad \text{(seconds)} \quad (2.29)$$

$$\text{iterate f x} = \text{x} : \text{map f (iterate f x)} \qquad \text{(iterate)} \quad (2.30)$$

$$\text{forever} = \text{iterate I} \qquad \text{(forever)} \quad (2.31)$$

$$\text{a..b} = \textbf{if } \text{a} > \text{b } \textbf{then } \{\} \textbf{ else } \text{a} : ((\text{a} + 1)..\text{b}) \textbf{ fi} \qquad \text{(upto)} \quad (2.32)$$
$$\text{a..} = \text{iterate (+1) a} \qquad \text{(from)} \quad (2.33)$$

$$\text{a} \cap \text{b} = \{\text{x} \mid \text{x} \in \text{a},\ \text{x} \in \text{b}\} \qquad \text{(list-intersect)} \quad (2.34)$$
$$\text{a} - \text{b} = \{\text{x} \mid \text{x} \in \text{a},\ \text{x} \notin \text{b}\} \qquad \text{(list-minus)} \quad (2.35)$$
$$\text{a} \subseteq \text{b} = (\text{a} \cap \text{b} = \text{a}) \qquad \text{(list-subset)} \quad (2.36)$$
$$\text{a} / \text{b} = \text{c, such that a} = \text{c} +\!\!+ \text{b} \qquad \text{(list-right-div)} \quad (2.37)$$
$$\text{b} \backslash \text{a} = \text{c, such that a} = \text{b} +\!\!+ \text{c} \qquad \text{(list-left-div)} \quad (2.38)$$

Some of these operations are well-defined on e.g. both lists and sets, like (2.34) to (2.36). They are even well-defined when one argument is a set and the other is a list. Program transformations involving these identities are valid as long as the expressions are well-defined, regardless of the exact typing of the arguments. Thus, we will write $\text{l} \subseteq \text{s}$ where l is a list and s is a set, meaning that all elements of l are also elements of s.

2.5 Identities

With respect to the functions defined above, several useful identities can be observed. We list some of them, without proving them explicitly. Some identities are given both in plain and list-comprehension notation. As usual, we assume that no name clashes are introduced when we apply these identities in transforming programs.

$$\text{map } (f \circ g) = \text{map } f \circ \text{map } g \qquad \text{(map-map)} \quad (2.40)$$
$$\text{map } f \, \{e \mid q\} = \{f\ e \mid q\} \qquad \text{(map-set-2)} \quad (2.40a)$$
$$\{f\ (g\ a) \mid a \in x\} = \{f\ b \mid b \in \{g\ a \mid a \in x\}\} \qquad \text{(map-map-list)} \quad (2.40b)$$

$$\text{first} \circ \text{split } f = \text{filter } f \qquad \text{(first-split)} \quad (2.41)$$
$$\text{second} \circ \text{split } f = \text{filter } (\neg \circ f) \qquad \text{(second-split)} \quad (2.42)$$

$$\{f\ a \mid a \in x, c\ a\} = (\text{map } f \circ \text{filter } c)\ x \qquad \text{(map-filter)} \quad (2.43)$$

$$\text{iterate } f\ x = x : \text{iterate } f\ (f\ x) \qquad \text{(iterate-alt)} \quad (2.44)$$
$$\text{map } g \circ \text{iterate } (f \circ g) = \text{iterate } (g \circ f) \circ g \qquad \text{(map-iterate-compose)} \quad (2.45)$$
$$\text{iterate } (f \circ g)\ x = x : (\text{map } f \circ \text{iterate } (g \circ f) \circ g)\ x \qquad \text{(iterate-compose)} \quad (2.46)$$

$$f\ (x +\!\!+ y) = f\ x +\!\!+ f\ y \Rightarrow f \circ \text{concat} = \text{concat} \circ \text{map } f \qquad \text{(f-concat)} \quad (2.47)$$

If both f and g distribute over $+\!\!+$, then so does $f \circ g$. Identity (2.47) implies:

$$\text{concat}^2 = \text{concat} \circ \text{map concat} \qquad \text{(concat-concat)} \quad (2.48)$$
$$\text{map } f \circ \text{concat} = \text{concat} \circ \text{map } (\text{map } f) \qquad \text{(map-concat)} \quad (2.49a)$$
$$\{f\ a \mid a \in \text{concat } x\} = \{f\ a \mid b \in x, a \in b\} \qquad \text{(map-concat-list)} \quad (2.49b)$$
$$\{f\ a \mid a \in \text{concat } x\} = \text{concat } \{\text{map } f\ a \mid a \in x\} \qquad \text{(map-concat-list-2)} \quad (2.49c)$$

$$\text{filter } f \circ \text{concat} = \text{concat} \circ \text{map } (\text{filter } f) \qquad \text{(filter-concat)} \quad (2.50a)$$
$$\{a \mid a \in \text{concat } x, f\ a\} = \{a \mid b \in x, a \in b, f\ a\} \qquad \text{(filter-concat-list)} \quad (2.50b)$$

$$\text{concat} \circ \text{map } (\text{map } f \circ g)$$
$$= \text{map } f \circ \text{concat} \circ \text{map } g \qquad \text{(concat-map-map)} \quad (2.51a)$$
$$\{f\ b \mid a \in x, b \in g\ a\}$$
$$= \{f\ b \mid b \in \text{concat } \{g\ a \mid a \in x\}\} \qquad \text{(concat-map-map-list)} \quad (2.51b)$$

2.6 Functional programming

In this section we present some programming styles which occur frequently in functional programming, but are not common in imperative settings. Readers that are familiar with functional programming can skip this section. The functional programming styles differ in abstraction level, from explicit recursion to the use of higher order functions. We illustrate the styles by a number of examples in order to give the reader some idea of it.

2.6.1 Explicit recursion

A function that computes the factorial of a natural number can be expressed by *explicit recursion*:

$$\mathsf{fac}\ \mathsf{n}\ =\ \textbf{if}\ \mathsf{n} = 0\ \textbf{then}\ 1\ \textbf{else}\ \mathsf{n} * \mathsf{fac}\ (\mathsf{n}-1)\ \textbf{fi}$$

2.6.2 Comprehensions

List, bag or set valued expressions are often more conveniently denoted by means of *comprehensions* than by explicit recursion. The quicksort function is easily defined in this way:

$$\begin{aligned}\mathsf{sort}\ \{\} &= \{\}\\ \mathsf{sort}\ (\{\mathsf{a}\} \mathbin{+\!\!+} \mathsf{l}) &= \mathsf{sort}\ \{\mathsf{x} \mid \mathsf{x} \in \mathsf{l},\ \mathsf{x} < \mathsf{a}\} \mathbin{+\!\!+} \{\mathsf{a}\} \mathbin{+\!\!+} \mathsf{sort}\ \{\mathsf{x} \mid \mathsf{x} \in \mathsf{l},\ \mathsf{x} \geq \mathsf{a}\}\end{aligned}$$

Also the set of all Pythagorian triples can be expressed smoothly in this way as a one-liner, as we have seen above:

$$\mathsf{triples}\ =\ \{(\mathsf{a},\mathsf{b},\mathsf{c}) \mid \mathsf{a} \in 1..,\ \mathsf{b} \in 1..(\mathsf{a}-1),\ \mathsf{c} \in \mathsf{a}-\mathsf{b}..\mathsf{a}+\mathsf{b},\ \mathsf{c}^2 = \mathsf{a}^2 + \mathsf{b}^2\}$$

2.6.3 Higher order functions

A *higher order function* is a function that takes another function as one of its arguments. The factorial function can be expressed smoothly by means of a higher order function:

$$\mathsf{fac}\ \mathsf{n}\ =\ (*)/(1..\mathsf{n})$$

The function / takes the multiplication function as its left argument. Functional programming makes abundant use of higher order functions. The functions map, /, filter, split and iterate that we encountered before are all higher order functions.
Another good example of the use of higher order functions is the following variant of quicksort.

$$\begin{aligned}\mathsf{sort}\ &= \{\}\\ \mathsf{sort}\ (\{\mathsf{a}\} \mathbin{+\!\!+} \mathsf{l}) &= \mathsf{sort}\ \mathsf{small} \mathbin{+\!\!+} \{\mathsf{a}\} \mathbin{+\!\!+} \mathsf{sort}\ \mathsf{large}\\ &\quad \textbf{where}\ (\mathsf{small}, \mathsf{large}) = \mathsf{split}\ (< \mathsf{a})\ \mathsf{l}\end{aligned}$$

2.6.4 Lazy evaluation

The Pythagorian triples function makes use of *lazy evaluation*. The list (1..) is computed lazily, that is, it is expanded as more and more elements of it are needed in the computation. Also the result, i.e. the list triples, is computed lazily. If only its first argument is evaluated, i.e. the triple $(4,3,5)$, the list 1.. is computed upto its fourth element.

2.6.5 Local definitions

The use of *local definitions* in where-parts is also something specific to the functional programming world. The n-queens problem below demonstrates its use, although it leans even more heavily on comprehensions. An n×n chess board must be filled with n queens such that no two queens attack each other. The queens are placed on the first, second, etc. column. Each column is represented by the number of the row on which the queen is placed. (Of course, it is no use placing two queens on one column.) A partial solution, consisting of the first m queens being placed, is represented by the sequence of the row numbers of these m queens and is computed by the function q m.

```
queens n = q n
where q 0 = {{}}
      q m = {board ++ {r} | board ∈ q (m − 1), r ∈ 1..n, safe r board}
      where safe r board = ∧/{r ≠ s | s ∈ board}∧
                           ∧/{m − i ≠ |r − s| | (s, i) ∈ board•(1..(m − 1))}
```

Staying with chess, the knight's tour, where a knight must traverse the whole chess board, visiting each square exactly once, is also conveniently expressed using comprehensions:

```
knight n (x, y) = tour {(x, y)}
where tour t = if #t = n² then {t}
               else concat {tour ((x′, y′) : t) | (x, y) ← hd t,
                                                  (dx, dy) ∈ moves,
                                                  (x′, y′) ← (x + dx, y + dy),
                                                  x′ > 0, y′ > 0, x′ ≤ n, y′ ≤ n,
                                                  (x′, y′) ∉ t}
               fi
moves = {(1,2), (2,1), (−1,2), (2,−1), (1,−2), (−2,1), (−1,−2), (−2,−1)}
```

2.6.6 Example of different styles

We summarize the different styles by giving a number of different definitions for the list of all powers of x.

```
powers x = {xⁿ | n ∈ 0..}                          (p1) (2.52)
powers x = f 0 where f n = xⁿ : f (n + 1)          (p2) (2.53)
powers x = {1} ++ map (x*) (powers x)             (p3) (2.54)
powers x = p where p = {1} ++ map (x*) p          (p4) (2.55)
powers x = p where p = {1} ++ {x * y | y ∈ p}     (p5) (2.56)
powers x = iterate (x*) 1                          (p6) (2.57)
```

Observe the use of the higher order functions map and iterate and the partial parameterization in the expression (x*).

The first version (2.52) can be seen as a specification. The second one (2.53) makes use of explicit recursion and is less obvious. The third (2.54) uses a higher order function, partial parameterization and explicit recursion. The next (2.55) is a simple variant of (2.54), but more efficient on most implementations. The fifth (2.56) uses both laziness and comprehensions, while the last (2.57) uses a single higher order function. It is the most abstract of them, if we define that expression e_1 is more abstract than e_2 when either e_1 contains no less higher order functions than e_2 or less symbols.
We can use these examples to demonstrate the techniques of program transformation by deriving these definitions from (2.52). The definitions and identities above can be used to rewrite an algorithm algebraically:

Derivation 2.58

powers x
  = (definition (2.52))
$\{x^n \mid n \in 0..\}$ (2.52)
  = (definition (2.33) and (2.30))
$\{x^n \mid n \in \{0\} +\!\!+ 1..\}$
  = (identity (2.49*c*))
$\{x^0\} +\!\!+ \{x^n \mid n \in 1..\}$
  = (mathematics)
$\{1\} +\!\!+ \{x * x^n \mid n \in 0..\}$
  = (definition (2.12) and (2.52))
$\{1\} +\!\!+$ map (x∗) (powers x) (2.54)
  = (introduce p = powers x)
p **where** p = $\{1\} +\!\!+$ map (x∗) p (2.55)
  = (definition (2.12))
p **where** p = $\{1\} +\!\!+ \{x * y \mid y \in p\}$ (2.56)

□

Furthermore, (2.54) is easily rewritten into (2.57) by the definition of iterate (2.30), which leaves us with the derivation of (2.53).

Derivation 2.59

powers x
  = (definition (2.52))
$\{x^n \mid n \in 0..\}$ (2.52)
  = (introduce the more general function f)
f 0 **where** f n = $\{x^i \mid i \in n..\}$
  = (definition (2.33) and identity (2.49*c*))
f 0 **where** f n = $\{x^n\} +\!\!+ \{x^i \mid i \in (n+1)..\}$
  = (definition of f)
f 0 **where** f n = $\{x^n\} +\!\!+$ f (n + 1) (2.53)

□

These derivations show the power of the program transformations which allow the algebraic manipulation of functional programs, giving rise to simple and elegant program derivations and correctness preserving transformations. If e.g. version (2.52) is taken as a specification, the few lines above constitute the proof of correctness of all the other forms. The conciseness of this proof shows the power of the algebraic approach. Other notations and proof methods, e.g. imperative programs with while loops in stead of comprehensions combined with weakest precondition reasoning, require much more lengthy proofs.

2.7 Relations and back-tracking

The following expression

$$\{x_n \mid x_1 \in f_1\ a,\ x_2 \in f_2\ x_1,\ \ldots,\ x_n \in f_n\ x_{n-1}\ \} \tag{2.60}$$

can be interpreted as a set comprehension notation computing solutions x_n for a given 'problem' a. When we treat this expression more operationally, assuming lazy evaluation, this same expression can be interpreted as a notation for solving the problem a by means of back-tracking. Assuming lazy evaluation, each time a new result x_n is required, more intermediate elements x_i are computed and larger parts of each $f_i\ x_{i-1}$ are computed.
Thus, the problem is solved by computing sub-solutions x_1 with f_1 and for each of these sub-solutions deriving more refined solutions x_2 with f_2, etc., until finally the problem a is solved with solutions x_n. When $f_i\ x_{i-1}$ happens to yield $\{\}$, another x_{i-1} must be attempted, and hence the whole process can be seen as a back-tracking process. An excellent treatment of this notation for back-tracking can be found in [Wad85].
Here we present a more abstract notation for the same expression. We make use of the *continuation operator* (functor) ' ; ' with type:

$$(\ ;\ ) :: (T \mapsto U^*) \times (U \mapsto V^*) \mapsto (T \mapsto V^*)$$

that is defined by:

$$f\ ;\ g\ =\ \text{concat} \circ \text{map}\ g \circ f \qquad \text{(continue)} \tag{2.61}$$

The operational interpretation of this operator is as follows: when the function $f\ ;\ g$ is applied to a an argument x, first f is applied to x, delivering a set (bag, list, tree) of values. To each of these values, g is applied and the results of these applications of g are flattened with the function concat. Hence the operator applies first f and then g to the results of f.

2.7.1 Properties of ;

It is easily proved that the operator ; is associative:

Derivation 2.62

$(f \; ; \; g) \; ; \; h$
 = (definition of ;)
$\text{concat} \circ \text{map } h \circ \text{concat} \circ \text{map } g \circ f$
 = (by (2.49*a*))
$\text{concat} \circ \text{concat} \circ \text{map (map } h) \circ \text{map } g \circ f$
 = (by (2.48))
$\text{concat} \circ \text{map concat} \circ \text{map (map } h) \circ \text{map } g \circ f$
 = (by (2.40) twice)
$\text{concat} \circ \text{map (concat} \circ \text{map } h \circ g) \circ f$
 = (definition of ;)
$\text{concat} \circ \text{map } (g \; ; \; h) \circ f$
 = (definition of ;)
$f \; ; \; (g \; ; \; h)$

□

The function single is the neutral element of ; :

$\text{single} \; ; \; f = f$ (single-left) (2.63)
$f \; ; \; \text{single} = f$ (single-right) (2.64)

The operator ; can be used in expressing back-tracking. This is based on the following identity:

Derivation 2.65

$\{y \mid x \in f\ a,\ y \in g\ x\}$
 = (by (2.20))
$\text{concat } \{g\ x \mid x \in f\ a\}$
 = (by (2.18))
$(\text{concat} \circ \text{map } g)\ \{x \mid x \in f\ a\}$
 = (by (2.18), f = I)
$(\text{concat} \circ \text{map } g \circ f)\ a$
 = (definition of ;)
$(f \; ; \; g)\ a$

□

Using this identity, we can prove by induction that

$\{x_n \mid x_1 \in f_1\ a,\ x_2 \in f_2\ x_1,\ \ldots,\ x_n \in f_n\ x_{n-1}\}$
$= (f_1 \; ; \; f_2 \; ; \; \ldots \; ; \; f_n)\ a$ (set-continue) (2.66)

2.7.2 Relations

The expression (2.60) can also be interpreted as a way of solving certain relations. Let R be a relation, then we can associate a set-valued function $\mathcal{F}_R$ with R which is defined by:

$$\mathcal{F}_R\ x = \{y \mid \exists y : x\ R\ y\} \tag{2.67}$$

We can define the composition of two relations by:

$$x\ (R \circ S)\ y = \exists z : x\ R\ z \wedge z\ S\ y \tag{2.68}$$

Expression (2.60) can then be considered as the set of solutions to the expression:

Derivation 2.69

$a\ R_1\ x_1 \wedge x_1\ R_2\ x_2 \wedge \ldots \wedge x_{n-1}\ R_n\ x_n$
  $=$ (relation composition (2.68))
$a\ (R_1 \circ R_2 \circ \ldots \circ R_n)\ x_n$
  $=$ (identity (2.67))
$x_n \in \mathcal{F}_{R_1 \circ R_2 \circ \ldots \circ R_n}\ a$
  $=$ (proven below)
$x_n \in (\mathcal{F}_{R_1} ; \mathcal{F}_{R_2} ; \ldots ; \mathcal{F}_{R_n})\ a$
□

The last equivalence is proven as follows:

Derivation 2.70

$\mathcal{F}_{R \circ S}\ x$
  $=$ ((2.67) and (2.68))
$\{z \mid \exists y : x\ R\ y,\ y\ S\ z\}$
  $=$ ((2.20))
$\text{concat}\ \{\{z \mid y\ S\ z\} \mid \exists y : x\ R\ y\}$
  $=$ ((2.67))
$\text{concat}\ \{\mathcal{F}_S\ y \mid \exists y : x\ R\ y\}$
  $=$ ((2.67))
$\text{concat}\ \{\mathcal{F}_S\ y \mid y \in \mathcal{F}_R\ x\}$
  $=$ ((2.18))
$(\text{concat} \circ \text{map}\ \mathcal{F}_S \circ \mathcal{F}_R)\ x$
  $=$ ((2.61))
$(\mathcal{F}_R ; \mathcal{F}_S)\ x$
□

Summarizing:

$$\mathcal{F}_{R \circ S} = \mathcal{F}_R ; \mathcal{F}_S \tag{2.71}$$

We will be frequently using set valued functions when presenting parsing algorithms in chapter 3. These algorithms could have been presented by means of relations instead and calculations performed on these relations in the same fashion as [BvdW92]. Since we are interested in parsing algorithms as functions, as we want to actually implement them ,and since we want to investigate for which classes of grammars these functions become single-valued, we will not be using the relational approach in this thesis. The ; operator allows us to compose set-valued functions in the same way as their corresponding relations. As a consequence, the algorithms expressed by means of the continuation operator ; turn out to be the most concise and abstract ones appearing in this thesis.

2.7.3 Bunches

There is another alternative to the use of relations, namely *bunch-valued* functions [SH92]. A bunch is a set with the additional property that function application distributes over bunch construction, that is (using the already overloaded notation for sets for bunches as well):

$$\begin{aligned} \mathsf{f}\,\{\} &= \{\} \\ \mathsf{f}\,\{\mathsf{a}\} &= \mathsf{f\,a} \\ \mathsf{f}\,(\mathsf{x} \mathbin{+\!\!+} \mathsf{y}) &= \mathsf{f\,x} \mathbin{+\!\!+} \mathsf{f\,y} \end{aligned}$$

Bunch valued functions can be interpreted as non-deterministic functions. They have one major draw-back, which is illustrated by the following example. Let $\mathsf{square}\ \mathsf{x} = \mathsf{x} * \mathsf{x}$ be a squaring function. Then, we can apply this function to a bunch:

$$\begin{aligned} &\mathsf{square}\ \{2,3\} \\ = &\ \mathsf{square}\ 2 \mathbin{+\!\!+} \mathsf{square3} \\ = &\ \{4,9\} \end{aligned}$$

Thus, the *unfoldability principle*, which states that function application is defined by substituting the actual arguments for the formal arguments in the function body, is violated by bunches, as it would yield for our example:

$$\begin{aligned} &\mathsf{square}\ \{2,3\} \\ = &\ \{2,3\} * \{2,3\} \\ = &\ 2*2 \mathbin{+\!\!+} 2*3 \mathbin{+\!\!+} 3*2 \mathbin{+\!\!+} 3*3 \\ = &\ \{4,6,9\} \end{aligned}$$

In [SS90] an excellent treatment of this and similar problems can be found. In [Lee93] a presentation of parsing by means of bunch valued functions can be found. Many of the expressions in our chapter 3 can be found in bunch form in Leermakers' book. They are more concise in bunch notation, but since we are calculating by means of program transformations on parsing functions, we do not want to lose the unfoldability principle and hence we will not be using bunches.

Chapter 3

Functional parsers

3.1 Introduction

In this chapter we present a spectrum of different parsers, expressed as functional programs. We start with a derivation of a simple top-down deterministic parser and derive several other parsers from it by exploiting algebraic program transformation. Deterministic parsers are identified later on as special cases of non-deterministic ones. It may come as a surprise that we start with non-deterministic parsers, but recent developments in parsing theory [LAK92] indicate that non-deterministic parsers are easily described as functional programs.
We investigate the derivation of several non-deterministic and deterministic functional parsing algorithms by means of algebraic program transformations [Bir89, Mee83, Bac88]. We identify a class of recursive functions which we call *recursive descent* functions. For a recursive descent function a transformation into a so-called *recursive ascent* function can be defined. This transformation RA1 only influences the termination conditions for the function, leaving the rest of the semantics unchanged. It is a transformation that transforms recursion into iteration, starting from a sub-problem that has a simple solution (meaning that its computation does not need the recursive function under consideration) and constructing solutions for larger and larger problems out of it until the final problem has been solved.
We present a simple non-deterministic top-down (recursive descent) parser and show how it can be transformed into a bottom-up (recursive ascent) parser. Subsequently, four more transformations from descent to ascent are presented: RA2, SRA1, DRA1 and DRA2. The latter two are defined on *doubly recursive descent* functions, a class of functions that contains the parsing algorithms under consideration. With the use of these transformations, several parsing algorithms are derived, among which the Leermakers parser [LAK92].
A further program transformation is presented, which transforms a back-tracking function into a function that realizes this back-tracking by means of *continuations*. This transformation can also be applied to a top-down parser, resulting in a variant of Koster's recursive back-up parsers [Kos74].
The non-deterministic parsing algorithms behave deterministically for certain classes of

context-free grammars. The descent and ascent parsers can be modified in these cases to deterministic LL(1), LR(1) or LALR(1) equivalents by the addition of look-ahead information. The resulting deterministic parsers can be extended with error recovery facilities.
In general, the non-deterministic parsers show exponential complexity, which can be reduced to cubic complexity with the aid of a technique called *memoization* [Hug85, Mic68, Nor91, War92]. This technique consists of the caching of function results, such that when a function is called multiple times with identical elements, the corresponding result need be computed only once. In this way, variants of the Earley and Tomita parsers are obtained [Ear70, Tom86].
At the end of this chapter we give an overview in a diagram of the different parsers that we have presented and show the relationships between these parsing functions. Since each of the different parsing functions is recursive, it is possible to make a new choice between the different parsing techniques at each recursive call. Thus, combinations of LL(1), LALR(1), back-tracking, memoized and other parsing algorithms can be achieved.
From all the parsers presented, three have been implemented in the `Elegant` system: a recursive descent LL(1) parser, a recursive ascent LALR(1) parser and a recursive backup parser based on continuations.

3.2 Preliminaries

In this section we define the notions of a context-free grammar and derivation and present and prove the correctness of a simple functional top-down parser.
Definition 3.1 (def:CFG)

A *context-free grammar* G is a tuple (V_T, V_N, P, S), where

- V_T is a set of *terminal symbols*
- V_N is a set of *non-terminal symbols*,
  $V_T \cap V_N = \{\}$
- $V = V_T \cup V_N$ is the set of *all symbols*
- V^* is the set of *sequences of symbols*
- $P \subset V_N \times V^*$ is the set of *production rules*
- $S \in V_N$ is the *start symbol*

□
We use the following notation for typical elements of each of the following sets.

set	typical element
V_T	x, y, z
V_N	A, B, ...
V	X, Y, Z
V^*	$\alpha, \beta, \ldots$
V_T^*	ω, i, j, k, l
V^0	ε, the empty sequence

Warning: We are sloppy with the notation for singleton elements of V^*. We write X *instead of* $\{X\}$. *We will also write* $\alpha\beta$ *instead of* $\alpha \mathbin{+\!\!+} \beta$.

Definition 3.2 (def:derives)

We define the relation $\rightarrow$ (*directly derives*) on V^* by $\alpha A \beta \rightarrow \alpha\gamma\beta \equiv (A, \gamma) \in P$.

The transitive and reflexive closure of $\rightarrow$ is denoted by $\overset{*}{\rightarrow}$, as usual. When $\alpha \overset{*}{\rightarrow} \beta$ we say that α *derives* β.

□

Note that $A \rightarrow \alpha \equiv (A, \alpha) \in P$. We often use this notation to state that a production rule is an element of P.

Warning: We will also use the notation $A \rightarrow \alpha$ *for the qualifier* $(A, \alpha) \in P$, *as in the following function* rhs *that maps a non-terminal onto the set of right-hand sides of its production rules:*

$$\text{rhs } A = \{\alpha \mid A \rightarrow \alpha\} \tag{3.3}$$

The following notations will prove useful in this chapter:

Definition 3.4 (def:derives-div)

$\alpha \overset{*}{\rightarrow} \beta/\gamma \equiv \exists\delta : \beta = \delta\gamma \wedge \alpha \overset{*}{\rightarrow} \delta$

□

Definition 3.5 (def:left-derives)

We define the relation $\rightarrow_L$ (*directly left derives*) on V^* by $A\beta \rightarrow_L \gamma\beta \equiv (A, \gamma) \in P$. We say that $A\beta$ *directly left derives* $\gamma\beta$ when $(A, \gamma) \in P$. We use the notation $\alpha \rightarrow_L \beta$ when α directly left derives β.

When $\alpha \overset{*}{\rightarrow}_L \beta$ we say that α *left derives* β.

Observe that the $\alpha \overset{*}{\rightarrow}_L \beta$ requires that $\alpha = A\gamma$. We will be using this relation in the derivation of bottom-up parsers, especially in the definition of a so-called *state*.

□

The meaning of a context-free grammar is given by the language that it generates:

Definition 3.6 (def:CFG-lang)

The *language* L_G generated by a context-free grammar $G = (V_T, V_N, P, S)$ is defined as

$$L_G = \{\omega \mid \exists\omega : S \overset{*}{\rightarrow} \omega\}$$

□

A *recognizer* is an algorithm that determines, given a context-free grammar G and an input string $\omega \in V_T^*$, whether $\omega \in L_G$.
A *parser* does not only determine whether a given input string belongs to the language, but also why, i.e. it constructs the structure of the derivation in the form of a so-called derivation tree and thus provides a constructive proof for the fact that the input string belongs to the language.
It might come as a surprise that we are not interested in these derivation trees at all. In this chapter we restrict ourselves to the recognition problem. In chapter 4 we extend the recognizers presented here to parsers for attribute grammars, again without considering derivation trees explicitly. We are rather sloppy with the word *parser*. We often use it where actually a recognizer is meant, hoping no confusion will arise.
We specify the following simple parser function:

Specification 3.7 (spec:TD-nd)

$$\mathsf{parse} :: V^* \times V_T^* \mapsto Set(V_T^*)$$
$$\mathsf{parse}\ \alpha\ \mathsf{i} = \{\mathsf{j} \mid \exists \mathsf{j} : \alpha \overset{*}{\rightarrow} \mathsf{i/j}\}$$

□

We can rewrite the recognition problem in the following way:

$\omega \in L_G$
    = (definition of L_G)
$S \overset{*}{\rightarrow} \omega$
    = (specification (3.7))
$\varepsilon \in \mathsf{parse}\ S\ \omega$

In the implementation of our parsers, we will distinguish four different cases for the argument α.

- $\alpha = \varepsilon$
- $\alpha = \mathsf{x}$
- $\alpha = \mathsf{A}$
- $\alpha = \beta\gamma$, assuming $\beta \neq \varepsilon, \gamma \neq \varepsilon$

The parser is implemented recursively as follows:

- parse ε i
  = (specification (3.7))
  $\{j \mid \exists j : \varepsilon \overset{*}{\rightarrow} i/j\}$
  = (definition (3.4))
  $\{i\}$
- parse $\{x\}$ i
  = (specification (3.7))
  $\{j \mid \exists j : \{x\} \overset{*}{\rightarrow} i/j\}$
  = (definition (3.4))
  $\{j \mid x : j \leftarrow i\}$
- parse $\{A\}$ i
  = (specification (3.7) and $i, j \in V_T^*$)
  $\{j \mid \exists j : A \overset{*}{\rightarrow} i/j\}$
  = (definition (3.2))
  $\{j \mid A \rightarrow \alpha,\ \alpha \overset{*}{\rightarrow} i/j\}$
  = (specification of parse)
  $\{j \mid A \rightarrow \alpha,\ j \in \text{parse } \alpha\ i\}$
- parse $(\alpha\beta)$ i
  = (specification (3.7))
  $\{k \mid \exists k : \alpha\beta \overset{*}{\rightarrow} i/k\}$
  = (definition of $\overset{*}{\rightarrow}$)
  $\{k \mid \exists j, k : \alpha \overset{*}{\rightarrow} i/j,\ \beta \overset{*}{\rightarrow} j/k\}$
  = (specification (3.7))
  $\{k \mid j \in \text{parse } \alpha\ i,\ k \in \text{parse } \beta\ j\}$

Summarizing the algorithm gives, abbreviating $\{X\}$ by X:

Implementation 3.8 (of specification (3.7)) (TD-nd)

$$\begin{aligned}
\text{parse } \varepsilon\ i &= \{i\} \\
\text{parse } x\ i &= \{j \mid x : j \leftarrow i\} \\
\text{parse } A\ i &= \{j \mid A \rightarrow \alpha,\ j \in \text{parse } \alpha\ i\} \\
\text{parse } (\alpha\beta)\ i &= \{k \mid j \in \text{parse } \alpha\ i,\ k \in \text{parse } \beta\ j\}
\end{aligned}$$

□

3.3 Non-deterministic parsers

3.3.1 Introduction

In this section, a simple program transformation that transforms a so-called *recursive descent* function into a *recursive ascent* function is defined. This transformation can transform a recursive descent parser into a recursive ascent parser. Moreover, several variants of the transformation scheme can be derived, each resulting in a different recursive ascent parser. Also, different recursive descent parsers can be used as a starting point, giving rise to even more ascent parsers, among which the Leermakers parser [LAK92].

3.3.2 Recursive descent algorithms

We define two domains S and T. The set T is a finite set of so-called *problems* with typical elements α, β. The set S is a finite set of *solutions* with typical elements x, y. We assume that $T = T_1 \cup T_2$. T_1 denotes the set of so-called *trivial* problems, where T_2 is the set of *non-trivial* problems. These sets are not necessarily disjoint.
A trivial problem α can be solved by a function t, which returns the set of solutions for a problem. For a problem which is not trivial, it returns the empty set. A non-trivial problem α can be reduced by a function sub to a set of direct sub-problems with elements denoted by β. This set of sub-problems is the set of possible sub-problems of α. Each sub-problem can be solved separately, yielding a set of solutions. Out of each of these solutions for β, solutions for α can be constructed. Thus, the set of sub-problems does not form an 'and'-wise splitting of α, but an 'or'-wise one, and should be interpreted as the *set of possible sub-problems* of α. The function sub delivers this set of all direct sub-problems β for a given problem α.
Thus, these functions obey the following properties:

$$\begin{array}{ll} \mathsf{t} & :: T \mapsto Set(S) \\ \mathsf{sub} & :: T \mapsto Set(T) \end{array}$$

$$\begin{array}{ll} \mathsf{t}\,\alpha = \{\}, & \alpha \in T \backslash T_1 \\ \mathsf{sub}\,\alpha = \{\}, & \alpha \in T \backslash T_2 \end{array}$$

Let the function h be a function that combines a problem α, a direct sub-problem β of α and a solution x for β into a set of solution for α . It has type:

$$\mathsf{h} :: T_2 \times T \times S \mapsto Set(S)$$

Observe that this function h is partial: it is only defined for arguments $\alpha, \beta, \mathsf{x}$ with $\beta \in \mathsf{sub}\ \alpha$ and $\mathsf{x} \in \mathsf{p}\ \beta$. In the sequel we will apply h only to arguments obeying these restrictions, and define it only for such arguments.
Given these functions t, sub and h, we can define a function p that computes the set of solutions for a given problem α:
Let p be a function that, when applied to a problem α, delivers the set of solutions of α.

$$\mathsf{p} :: T \mapsto Set(S)$$

$$\mathsf{p}\,\alpha = \mathsf{t}\,\alpha \mathbin{+\!\!+} \mathsf{concat}\ \{\mathsf{h}\ \alpha\ \beta\ \mathsf{x} \mid \beta \in \mathsf{sub}\ \alpha,\ \mathsf{x} \in \mathsf{p}\ \beta\}$$

A function written in this form is called a *recursive descent function.*
This form is the general recursive descent form that we will use in the sequel. The first part $t\ \alpha$ solves a trivial problem α, the second part reduces α to direct sub-problems β,

recursively solves each of the sub-problems β and solves α by applying h. In order to remain more general, we have assumed that h produces sets of solutions. This anticipates specifically the situation where no solution exists for α, given a solution for β, in which case h produces the empty set. The function p is actually not a real function, but a *function scheme* since it contains the free variables t, sub and h. We can transform this scheme into a real function RD, which is the following higher order function:

Definition 3.9 (RD)

$$\mathsf{RD} :: (T \mapsto Set(S)) \times (T \mapsto Set(T)) \times (T_2 \times T \times S \mapsto Set(S)) \mapsto (T \mapsto Set(S))$$

$$\begin{aligned} &\mathsf{RD\ t\ sub\ h} &&= \mathsf{p} \\ &\textbf{where}\ \mathsf{p}\ \alpha &&= \mathsf{t}\ \alpha \mathbin{+\!\!+} \mathsf{concat}\ \{\mathsf{h}\ \alpha\ \beta\ \mathsf{x} \mid \beta \in \mathsf{sub}\ \alpha,\ \mathsf{x} \in \mathsf{p}\ \beta\} \end{aligned}$$

□

Example 3.10: Recursive descent factorials

A simple recursive descent problem is the computation of factorials. The straightforward definition

$$\begin{aligned} &\mathsf{fac}\ 0 &&= 1 \\ &\mathsf{fac}\ \mathsf{n} &&= \mathsf{n} * \mathsf{fac}\ (\mathsf{n}-1) \end{aligned}$$

can be rewritten into the recursive descent form, now delivering a singleton set of result values:

$$\begin{aligned} \mathsf{fac}\ \mathsf{n} = {} & \{1 \mid \mathsf{n} = 0\} \mathbin{+\!\!+} \\ & \mathsf{concat}\ \{\mathsf{n} * \mathsf{f} \mid \mathsf{n}' \in \{\mathsf{n}-1 \mid \mathsf{n} > 0\},\ \mathsf{f} \in \mathsf{fac}\ \mathsf{n}'\} \end{aligned}$$

Here, the functions t, sub and h are expressed as:

$$\begin{aligned} &\mathsf{t\ n} &&= \{1 \mid \mathsf{n} = 0\} \\ &\mathsf{sub\ n} &&= \{\mathsf{n}-1 \mid \mathsf{n} > 0\} \\ &\mathsf{h\ n\ n'\ f} &&= \{\mathsf{n} * \mathsf{f}\} \end{aligned}$$

□

Example 3.11: Construction of a binary heap

In [Sch92] a derivation of a binary heap construction algorithm is presented, making use of program inversion techniques. In this example, we present an alternative derivation, resulting in a recursive descent version, using algebraic program transformation techniques in a purely functional setting. In section 3.22 we transform this recursive descent solution into a version that is comparable with the solution of [Sch92].

Problem specification

We define a binary heap by:

$\langle\rangle$	is a heap,
$\langle \mathsf{l}, \mathsf{m}, \mathsf{r}\rangle$	is a heap when m is an integer, l, r are heaps and $\mathsf{l} \geq \mathsf{m}$ and $\mathsf{r} \geq \mathsf{m}$, with the relation $\geq$ defined on heap $\times$ integer: $\langle\rangle \geq \mathsf{m} \equiv \mathsf{true}$; $\langle \mathsf{l}', \mathsf{n}, \mathsf{r}'\rangle \geq \mathsf{m} \equiv \mathsf{n} \geq \mathsf{m}$

The in-order traversal of a heap is defined by the following function in from heaps to sequences of integers:

$$\begin{aligned}\mathsf{in}\ \langle\rangle &= \varepsilon\\ \mathsf{in}\ \langle \mathsf{l}, \mathsf{m}, \mathsf{r}\rangle &= \mathsf{in}\ \mathsf{l} +\!\!+ \{\mathsf{m}\} +\!\!+ \mathsf{in}\ \mathsf{r}\end{aligned}$$

For notational convenience, we define the following relation over $\mathbf{N}^* \times \mathbf{N}$.

$$\begin{aligned}&\varepsilon \leq \mathsf{m} \equiv \mathsf{true}\\ &\{\mathsf{n}\} +\!\!+ \mathsf{s} \leq \mathsf{m} \equiv \mathsf{n} \leq \mathsf{m}\end{aligned}$$

The problem is to find an efficient implementation of the function heaps (from $\mathbf{N}^*$ onto a set of heaps), defined by:

$$\mathsf{heaps}\ \mathsf{s} \ = \ \mathsf{in}^{-1}\ \mathsf{s} \ = \ \{\mathsf{h} \mid \exists \mathsf{h} : \mathsf{s} = \mathsf{in}\ \mathsf{h}\}$$

We will transform this specification of heaps into recursive descent form.

Problem generalization

Instead of rewriting the function heaps, we rewrite a more general version f of this function. We chose for a left-to-right traversal of the input string. This means that when we are constructing a right heap r, we have already encountered the top-element m above r. As r is bounded by that top element, it can be advantageous to pass that top-element to f, since it will limit the size of r. Since r is bounded, only part of the input string can be converted into r. Hence, the function f will consume part of the input string and return a heap and the remainder of the input string.

$$\mathsf{f}\ \mathsf{m}\ \mathsf{s} \ = \ \{(\mathsf{h}, \mathsf{t}) \mid \exists (\mathsf{h}, \mathsf{t}) : \mathsf{s} \ = \ \mathsf{in}\ \mathsf{h} +\!\!+ \mathsf{t},\ \mathsf{h} \geq \mathsf{m}\}$$

We can express the function heaps in terms of f in the following way:

Derivation 3.12

$$\begin{aligned}&\mathsf{heaps}\ \mathsf{s}\\ &\quad = \text{(definition of heaps and f)}\\ &\{\mathsf{h} \mid (\mathsf{h}, \mathsf{t}) \in \mathsf{f}\ (-\infty)\ \mathsf{s},\ \mathsf{t} \ = \ \varepsilon\}\\ &\quad = \text{(definition of } \leq\text{)}\\ &\{\mathsf{h} \mid (\mathsf{h}, \mathsf{t}) \in \mathsf{f}\ (-\infty)\ \mathsf{s},\ \mathsf{t} \leq -\infty\}\end{aligned}$$

□

A recursive descent definition

We can rewrite the definition of f into a recursive form by algebraic program transformations:

Derivation 3.13

f m s
  = (definition of f)
$\{(h,t) \mid \exists(h,t) : s = \text{in } h \mathbin{+\!\!+} t,\ h \geq m\}$
  = (h is either empty or not)
$\{(\langle\rangle,t) \mid \exists t : s = \text{in } \langle\rangle \mathbin{+\!\!+} t,\ \langle\rangle \geq m\} \mathbin{+\!\!+}$
$\{(\langle l,a,r\rangle,w) \mid \exists l,a,r,w : s = \text{in } \langle l,a,r\rangle \mathbin{+\!\!+} w,\ l \geq a,\ r \geq a,\ \langle l,a,r\rangle \geq m\}$
  = (definition of in, heap and ≥)
$\{(\langle\rangle,s)\} \mathbin{+\!\!+}$
$\{(\langle l,a,r\rangle,w) \mid \exists l,a,r,w : s = \text{in } \langle l,a,r\rangle \mathbin{+\!\!+} w,\ l \geq a,\ r \geq a,\ a \geq m\}$
  = (definition of in)
$\{(\langle\rangle,s)\} \mathbin{+\!\!+}$
$\{(\langle l,a,r\rangle,w) \mid \exists l,a,r,t,u,w : s = t \mathbin{+\!\!+} \{a\} \mathbin{+\!\!+} u \mathbin{+\!\!+} w,\ t = \text{in } l,\ u = \text{in } r,\ l \geq a,\ r \geq a,\ a \geq m\}$
  = (definition of f)
$\{(\langle\rangle,s)\} \mathbin{+\!\!+}$
$\{(\langle l,a,r\rangle,w) \mid \exists l,a,t,u,w : s = t \mathbin{+\!\!+} \{a\} \mathbin{+\!\!+} u \mathbin{+\!\!+} w,\ t = \text{in } l,\ l \geq a,\ a \geq m,\ (r,w) \in f\ a\ (u \mathbin{+\!\!+} w)\}$
  = (definition of f again, substitute $v = u \mathbin{+\!\!+} w$)
$\{(\langle\rangle,s)\} \mathbin{+\!\!+}$
$\{(\langle l,a,r\rangle,w) \mid (l, \{a\} \mathbin{+\!\!+} v) \in f\ m\ s,\ l \geq a,\ a \geq m,\ (r,w) \in f\ a\ v\}$
□

Observe that an application of f as defined by the last equation is non-computable (although it is quantifier free), as f m s calls f m s again. Nevertheless, it is an equation that defines f and which we will transform further into a form that is a computable implementation for f.

The definition that we have obtained for f can be brought into the recursive descent form by choosing:

Definition 3.14 (RD-heap)

$t\ (m,s) = \{(\langle\rangle,s)\}$
$\text{sub}\ (m,s) = \{(m,s)\}$
$h\ (m,s)\ (m,s)\ (l,u) = \{(\langle l,a,r\rangle,w) \mid a : v \leftarrow u,\ l \geq a,\ a \geq m,\ (r,w) \in f\ (a,v)\}$
□

We leave the proof as an exercise to the reader. Observe that h is only partially defined, namely for problem, sub-problem, solution triples, as explained above.

□

Observe that in this example t, sub anf h do not *characterize* f, since h still contains a recursive call to f. Nevertheless, we will be able to apply some very interesting program transformations to recursive descent functions, as our next section illustrates.

3.3.3 Recursive ascent scheme RA1

The recursive descent scheme can lead to non-termination when a problem α is a direct or indirect sub-problem of itself. This non-termination can in certain cases be avoided by transforming a recursive descent algorithm into a so-called *recursive ascent* algorithm. This transformation replaces the recursive call to p by iteration.

The idea is to compute the transitive and reflexive closure of the sub-problem relation for given α. The set of all sub-problems of α is denoted by $\overline{\alpha}$. From this set we can select the set of trivial problems. These trivial problems are solved first. Then for each solution x and solved problem β, which has a direct ancestor γ in the sub-problem relation, the problem γ can be solved. In this way, solutions for smaller problems, starting with trivial ones, can be used to construct solutions for larger problems until finally α has been solved iteratively.

It is important to understand that we replace only one recursive call to p by iteration. Since we have assumed nothing about h, it is perfectly possible that h calls p, as in our binary heap example. This is the case for functions that do contain more than one recursive call: only one of these can be replaced by iteration (at least without explicit manipulation of a recursion stack).

We define $\overline{\alpha}$, *the set of all sub-problems of* α, which is an element of *Set*(T), as the smallest set satisfying:

Definition 3.15 (all-sub)

$\overline{\alpha} :: Set(T)$
$\overline{\alpha} \;=\; \alpha : \{\gamma \mid \exists\beta,\gamma : \beta \in \overline{\alpha},\; \gamma \in \mathsf{sub}\;\beta\}$

□

The set $\overline{\alpha}$ is finite, since T is.

Another useful notion is the set of *super-problems* of a problem β with respect to a problem α, which is defined as:

$$\mathsf{sup}\;\alpha\;\beta \;=\; \{\gamma \mid \gamma \in \overline{\alpha},\; \beta \in \mathsf{sub}\;\gamma\} \qquad \text{(sup)} \quad (3.16)$$

We can now express p in another way, making use of a function q, defined local to p with typing

$$\mathsf{q} \;:: \overline{\alpha} \times S \rightarrow Set(S)$$

The recursive ascent version of p, which we denote by RA1, is defined as follows:

Definition 3.17 (RA1)

```
RA1 t sub h = p
where
p α = concat {q β x | β ∈ ᾱ, x ∈ t β}
where
q β x = {x | β = α} ++
          concat {q γ y  | γ ∈ sup α β, y ∈ h γ β x}
```

□

Observe that q has been defined locally to p. As a consequence, the argument α of p appears as a free variable in the definition of q. When we manipulate q in the sequel, the reader should keep in mind that α is a constant within q.

3.3.4 Correctness proof of the ascent algorithm (RD = RA1)

In this section we prove the equivalence of the descent and ascent versions of p, that is, we proof RD t sub h = RA1 t sub h and thus RD = RA1. We make use of a relation $\Rightarrow$ over $(T \times S) \times (T \times S)$ that is defined by:

$$(\alpha, y) \Rightarrow (\beta, x) \equiv (x \in p\ \beta \wedge \beta \in sub\ \alpha \wedge y \in h\ \alpha\ \beta\ x)$$

Observe that $(\alpha, y) \Rightarrow (\beta, x)$ implies that $y \in p\ \alpha$. This relation is very useful in specifying q:

Specification 3.18 (spec:RA1-q)

$$y \in q\ \beta\ x \equiv (\alpha, y) \overset{*}{\Rightarrow} (\beta, x)$$

□

This definition states that if β is a (direct or indirect) sub-problem of α and x is a solution for β, then q β x computes all solutions y for α that are consistent with β and x.
The relation $\Rightarrow$ is also useful for an alternative definition of p. Define the sets $\mathcal{P}$ and $\mathcal{T}$ as follows:

$$(\alpha, y) \in \mathcal{P} \equiv y \in p\ \alpha$$
$$(\alpha, y) \in \mathcal{T} \equiv y \in t\ \alpha$$

When $\Rightarrow$ is an arbitrary relation over $U \times U$ and $S \subseteq U$, we define $\Rightarrow \circ S$ by:

$$\Rightarrow \circ S = \{x \mid \exists x \in U,\ y \in S\ :\ x \Rightarrow y\}$$

Using this notation we can rewrite the recursive descent definition of P (3.9) into the equation:

$$\mathcal{P} = \mathcal{T} \cup \Rightarrow \circ \mathcal{P}$$

This equation has as a least fixed point (see [BvdW92]):

$\mathcal{P} = \overset{*}{\Rightarrow} \circ \mathcal{T}$

This step has transformed a recursive definition for $\mathcal{P}$ into an iterative one, which approximates $\mathcal{P}$ from below, starting from $\mathcal{T}$. Since S and T are finite, this approximation process terminates.
Using the relation $\Rightarrow$ and the sets $\mathcal{P}$ and $\mathcal{T}$, we can rewrite the recursive descent definition of $\mathsf{p}\ \alpha$ into an iterative one:

Derivation 3.19

$\mathsf{p}\ \alpha$
  $=$ (definition of $\mathcal{P}$)
$\{\mathsf{y} \mid (\alpha, \mathsf{y}) \in \mathcal{P}\}$
  $=$ (least solution for $\mathcal{P}$)
$\{\mathsf{y} \mid (\alpha, \mathsf{y}) \in \overset{*}{\Rightarrow} \circ \mathcal{T}\}$
  $=$ (definition of $\overset{*}{\Rightarrow} \circ \mathcal{T}$)
$\{\mathsf{y} \mid \exists (\beta, \mathsf{x}) \in \mathcal{T} : (\alpha, \mathsf{y}) \overset{*}{\Rightarrow} (\beta, \mathsf{x})\}$
  $=$ (definition of $\mathcal{T}$)
$\{\mathsf{y} \mid \beta \in \overline{\alpha},\ \mathsf{x} \in \mathsf{t}\ \alpha,\ (\alpha, \mathsf{y}) \overset{*}{\Rightarrow} (\beta, \mathsf{x})\}$
  $=$ (specification (3.18))
$\{\mathsf{y} \mid \beta \in \overline{\alpha},\ \mathsf{x} \in \mathsf{t}\ \alpha,\ \mathsf{y} \in \mathsf{q}\ \beta\ \mathsf{x}\}$
  $=$ (identity (2.20))
$\mathsf{concat}\ \{\mathsf{q}\ \beta\ \mathsf{x} \mid \beta \in \overline{\alpha},\ \mathsf{x} \in \mathsf{t}\ \beta\}$
□

where we rewrite q, assuming that $\beta \in \overline{\alpha},\ \mathsf{x} \in \mathsf{p}\ \beta$:

Derivation 3.20

$\mathsf{q}\ \beta\ \mathsf{x}$
  $=$ (definition of q)
$\{\mathsf{y} \mid (\alpha, \mathsf{y}) \overset{*}{\Rightarrow} (\beta, \mathsf{x})\}$
  $=$ (definition of $\overset{*}{\Rightarrow}$)
$\{\mathsf{x} \mid (\alpha, \mathsf{y}) = (\beta, \mathsf{x})\}$ ++
$\{\mathsf{y} \mid \exists \gamma, \mathsf{z} : (\alpha, \mathsf{y}) \overset{*}{\Rightarrow} (\gamma, \mathsf{z}) \Rightarrow (\beta, \mathsf{x})\}$
  $=$ (definition of $\Rightarrow$ and the assumption $\beta \in \overline{\alpha},\ \mathsf{x} \in \mathsf{p}\ \beta$)
$\{\mathsf{x} \mid \alpha = \beta\}$ ++
$\{\mathsf{y} \mid \exists \gamma : \beta \in \mathsf{sub}\ \gamma,\ \mathsf{z} \in \mathsf{h}\ \gamma\ \beta\ \mathsf{x},\ (\alpha, \mathsf{y}) \overset{*}{\Rightarrow} (\gamma, \mathsf{z})\}$
  $=$ ($(\alpha, \mathsf{y}) \overset{*}{\Rightarrow} (\gamma, \mathsf{z})$ implies $\gamma \in \overline{\alpha}$)
$\{\mathsf{x} \mid \alpha = \beta\}$ ++
$\{\mathsf{y} \mid \gamma \in \overline{\alpha},\ \beta \in \mathsf{sub}\ \gamma,\ \mathsf{z} \in \mathsf{h}\ \gamma\ \beta\ \mathsf{x},\ (\alpha, \mathsf{y}) \overset{*}{\Rightarrow} (\gamma, \mathsf{z})\}$
  $=$ (definition of q and sup)
$\{\mathsf{x} \mid \alpha = \beta\}$ ++
$\{\mathsf{y} \mid \gamma \in \mathsf{sup}\ \alpha\ \beta,\ \mathsf{z} \in \mathsf{h}\ \gamma\ \beta\ \mathsf{x},\ \mathsf{y} \in \mathsf{q}\ \gamma\ \mathsf{z}\}$

$$= \text{(identity (2.20))}$$
$$\{x \mid \alpha = \beta\} + \!\! +$$
$$\text{concat } \{q\ \gamma\ z \mid \gamma \in \text{sup } \alpha\ \beta,\ z \in h\ \gamma\ \beta\ x\}$$

□

Example 3.21: Recursive ascent factorials

The factorials example can be rewritten from recursive descent to recursive ascent by applying the scheme. We recapitulate the recursive descent algorithm:

$$\begin{aligned} \text{fac } n \ =\ & \{1 \mid n = 0\} + \!\! + \\ & \text{concat } \{n * f \mid m \in \{n-1 \mid n > 0\},\ f \in \text{fac } m\} \end{aligned}$$

$$\begin{aligned} t\ n &= \{1 \mid n = 0\} \\ \text{sub } n &= \{n-1 \mid n > 0\} \\ \text{sup } n\ m &= \{m+1 \mid m < n\} \\ h\ n\ m\ f &= \{n * f\} \end{aligned}$$

Substituting these into the general recursive ascent algorithm yields:

$$\begin{aligned} &\text{fac } n \ = \ \text{concat } \{q\ 0\ 1\} \\ &\textbf{where} \\ &q\ m\ f \ = \ \{f \mid m = n\} + \!\! + \text{concat } \{q\ (m+1)\ g \mid m < n,\ g \in \{(m+1) * f\}\} \end{aligned}$$

Of course, this function can be simplified by delivering single results instead of a set of results:

$$\begin{aligned} &\text{fac } n \ = \ q\ 0\ 1 \\ &\textbf{where} \\ &q\ m\ f \ = \ \textbf{if } m = n \textbf{ then } f \textbf{ else } q\ (m+1)\ ((m+1) * f) \textbf{ fi} \end{aligned}$$

The function q is tail-recursive and can be written as a loop in an imperative language. Hence the whole process has transformed recursion into iteration.

□

Example 3.22: A recursive ascent binary heap construction

We recall the recursive descent version of the binary heap algorithm (3.14):

$$\begin{aligned} &t\ (m,s) \ = \ \{(\langle\rangle, s)\} \\ &\text{sub } (m,s) \ = \ \{(m,s)\} \\ &h\ (m,s)\ (m,s)\ (l,u) \ = \ \{(\langle l,a,r\rangle, w) \mid a : v \leftarrow u,\ l \geq a,\ a \geq m, \\ &\qquad\qquad\qquad\qquad\qquad\qquad (r,w) \in f\ (a,v)\} \end{aligned}$$

We can now bring f in the recursive ascent form:

Derivation 3.23

```
f
    = (recursive descent form)
RD t sub h
    = (RD = RA1)
RA1 t sub h
    = (substitute and rewrite using (3.14))
p where
p (m,s) = concat {q (m,s) (⟨⟩,s)}
where q (m,s) (l,u)
      = {(l,u)} ++
        concat {q (m,s) (⟨l,a,r⟩,w) | a : v ← u, l ≥ a, a ≥ m,
                                      (r,w) ∈ p (a,v)}
```
□

The last form can be simplified by observing that all invocations of q have the same first argument (m, s), which can thus be left out. Moreover we can apply Currying, i.e. replacing tuple-valued arguments by separate ones.

```
p m s = q ⟨⟩ s
where q l u = {(l,u)} ++
              concat {q ⟨l,a,r⟩ w | a : v ← u, l ≥ a, a ≥ m,
                                    (r,w) ∈ p a v}
```

By the transformation from descent to ascent, one of the two recursive calls to f (i.e. p) has been removed. This was the recursive call that delivered the left-most sub-tree of a heap and we are left with a recursive call that delivers the right-most sub-tree. We can exploit this fact in strengthening the definition of p in such a way that *maximal* right-most sub-trees are delivered. By this we mean that a longest prefix of the remainder of the input string is consumed. Consuming a smaller prefix makes no sense, since it can never be transformed into a valid heap. The corresponding function p′ is defined as follows, leaving the definition of q unchanged:

Derivation 3.24

```
p′ m s
    = (definition of p′)
{(h,v) | (h,v) ∈ p m s, v ≤ m}
    = (definition of p)
{(h,v) | (h,v) ∈ q ⟨⟩ s, v ≤ m}
    = (definition of q′)
q′ ⟨⟩ s
where q′ l u = {(h,v) | (h,v) ∈ q l u, v ≤ m}
```
□

We can rewrite this definition of q′ by substituting the definition of q into it.

$$\begin{aligned} q'\ l\ u \ = \ & \{(l,u) \mid u \leq m\} \mathbin{+\!\!+} \\ & \text{concat } \{q'\ \langle l,a,r\rangle\ w \mid a : v \leftarrow u,\ l \geq a,\ a \geq m,\ (r,w) \in p\ a\ v\} \end{aligned}$$

We can add the guard $w \leq a$ to the last form of q, since it can be proven that $\neg(w \leq a) \Rightarrow q'\ \langle l,a,r\rangle\ w = \{\}$.

Derivation 3.25

$$\begin{aligned} & q'\ l\ u \\ & \quad = \ (\neg(w \leq a) \Rightarrow q'\ \langle l,a,r\rangle\ w = \{\}) \\ & \{(l,u) \mid u \leq m\} \mathbin{+\!\!+} \\ & \text{concat } \{q'\ \langle l,a,r\rangle\ w \mid a : v \leftarrow u,\ l \geq a,\ a \geq m,\ (r,w) \in p\ a\ v,\ w \leq a\} \\ & \quad = \ (\text{definition of } p') \\ & \{(l,u) \mid u \leq m\} \mathbin{+\!\!+} \\ & \text{concat } \{q'\ \langle l,a,r\rangle\ w \mid a : v \leftarrow u,\ l \geq a,\ a \geq m,\ (r,w) \in p'\ a\ v\} \end{aligned}$$

□

Summarizing this algorithm gives:

Simplification 3.26

$$\begin{aligned} & p'\ m\ s \ = \ q'\ \langle\rangle\ s \\ & \textbf{where} \\ & q'\ l\ u \ = \ \{(l,u) \mid u \leq m\} \mathbin{+\!\!+} \\ & \qquad\qquad \text{concat } \{q'\ \langle l,a,r\rangle\ w \mid a : v \leftarrow u,\ l \geq a,\ a \geq m,\ (r,w) \in p'\ a\ v\} \end{aligned}$$

□

This recursive ascent version can be turned into a deterministic one which returns a single heap, which is an arbitrary element from the set of all possible heaps. This deterministic version can be proved to consume linear time and only constant extra space (over the space required for the heap), as in [Sch92]. See [Aug92a] for more details.

□

3.3.5 Transforming recursive descent into recursive ascent parsing

In this section we show how a very simple non-deterministic recursive descent parsing algorithm can be transformed into a recursive ascent parser.

A simple recursive descent parser

The simple functional parser can be written in the recursive descent form. We repeat its definition (3.8):

$$\begin{aligned} \text{parse } (\varepsilon, i) \ &= \ \{i\} \\ \text{parse } (x, i) \ &= \ \{j \mid x : j \leftarrow i\} \\ \text{parse } (A, i) \ &= \ \{j \mid A \rightarrow \alpha,\ j \in \text{parse } (\alpha, i)\} \\ \text{parse } (\alpha\beta, i) \ &= \ \{k \mid j \in \text{parse } (\alpha, i),\ k \in \text{parse } (\beta, j)\} \end{aligned}$$

This implementation can be rewritten into the general recursive descent form.
The domains become:

$$\begin{aligned} T &= V^* \times V_T^* \\ S &= V_T^* \\ \mathsf{p} &= \mathsf{parse} \end{aligned}$$

Thus, problems are of the form (α, i), a pair of a sequence of symbols and a remainder of the input string. The functions are given (as the reader can verify by substituting the four different forms of parse) by:

Theorem 3.27 (RD-nd)

$$\mathsf{parse}\ (\alpha, i) = \mathsf{t}\ (\alpha, i) \mathbin{+\!\!+} \mathsf{concat}\ \{\mathsf{h}\ (\alpha, i)\ (\gamma, k)\ j \mid (\gamma, k) \in \mathsf{sub}\ (\alpha, i),\ j \in \mathsf{parse}\ (\gamma, k)\}$$

$$\begin{aligned} \mathsf{sub}\ (\varepsilon, i) &= \{\} \\ \mathsf{sub}\ (x, i) &= \{\} \\ \mathsf{sub}\ (A, i) &= \{(\alpha, i) \mid A \to \alpha\} \\ \mathsf{sub}\ (\alpha\beta, i) &= \{(\alpha, i)\} \end{aligned}$$

Note that $(\beta, k) \in \mathsf{sub}\ (\alpha, i)$ implies $k = i$.

$$\begin{aligned} \mathsf{t}\ (\varepsilon, i) &= \{i\} \\ \mathsf{t}\ (x, i) &= \{j \mid x : j \leftarrow i\} \\ \mathsf{t}\ (\alpha, i) &= \{\} \end{aligned}$$

$$\begin{aligned} \mathsf{h}\ (A, i)\ (\alpha, i)\ j &= \{j\} \\ \mathsf{h}\ (\alpha\beta, i)\ (\alpha, i)\ j &= \mathsf{parse}\ (\beta, j) \end{aligned}$$

$$\overline{(\alpha, i)} = \{(\beta, i) \mid \exists \gamma : \alpha \overset{*}{\to}_L \beta\gamma\}$$

$$\begin{aligned} \mathsf{sup}\ (\alpha, i)\ (\beta, i) = &\ \{(A, i) \mid (A, i) \in \overline{(\alpha, i)},\ A \to \beta\} \mathbin{+\!\!+} \\ &\ \{(\beta\gamma, i) \mid (\beta\gamma, i) \in \overline{(\alpha, i)}\} \end{aligned}$$

□

This parser is called on the start symbol S by $\mathsf{parse}\ (S, \omega)$.

The corresponding recursive ascent parser

Given the recursive descent parser, the recursive ascent parser is easily obtained by applying the scheme. First we give the definitions of the relation $\overset{*}{\Rightarrow}$ and q.
The relation $\overset{*}{\Rightarrow}$ can be expressed as:

$$\begin{aligned} &((\alpha, i), k) \overset{*}{\Rightarrow} ((\beta, i), j) \\ = &\ \exists \gamma, k : \alpha \overset{*}{\to}_L \beta\gamma \wedge \beta \overset{*}{\to} i/j \wedge \gamma \overset{*}{\to} j/k \end{aligned}$$

which yields as a specification for q:

$$\mathsf{q}\ (\beta, \mathsf{i})\ \mathsf{j} \ = \ \{\mathsf{k} \mid \exists \gamma, \mathsf{k} : \alpha \overset{*}{\rightarrow}_{\mathsf{L}} \beta\gamma \wedge \beta \overset{*}{\rightarrow} \mathsf{i/j} \wedge \gamma \overset{*}{\rightarrow} \mathsf{j/k}\}$$

Substituting the functions t, sub and h in the general recursive ascent algorithm gives the following recursive ascent parser:

Derivation 3.28

$$\begin{array}{l}
\mathsf{p}\ (\alpha, \mathsf{i}) \\
\qquad = \text{(recursive ascent definition)} \\
\mathsf{concat}\ \{\mathsf{q}\ \beta\ \mathsf{j} \mid \beta \in \overline{(\alpha, \mathsf{i})},\ \mathsf{j} \in \mathsf{t}\ \beta\} \\
\qquad = \text{(definition of t)} \\
\mathsf{concat}\ \{\mathsf{q}\ (\mathsf{x}, \mathsf{i})\ \mathsf{j} \mid (\mathsf{x}, \mathsf{x} : \mathsf{j}) \in \overline{(\alpha, \mathsf{i})}\} \mathbin{+\!\!+} \\
\mathsf{concat}\ \{\mathsf{q}\ (\varepsilon, \mathsf{i})\ \mathsf{i} \mid (\varepsilon, \mathsf{i}) \in \overline{(\alpha, \mathsf{i})}\} \\
\textbf{where} \\
\mathsf{q}\ (\beta, \mathsf{i})\ \mathsf{j} \\
\qquad = \text{(recursive ascent definition)} \\
\{\mathsf{j} \mid (\beta, \mathsf{i}) \ = \ (\alpha, \mathsf{i})\} \mathbin{+\!\!+} \\
\mathsf{concat}\ \{\mathsf{q}\ (\gamma, \mathsf{i})\ \mathsf{k} \mid (\gamma, \mathsf{i}) \in \overline{(\alpha, \mathsf{i})},\ (\beta, \mathsf{i}) \in \mathsf{sub}\ (\gamma, \mathsf{i}),\ \mathsf{k} \in \mathsf{h}\ (\gamma, \mathsf{i})\ (\beta, \mathsf{i})\ \mathsf{j}\} \\
\qquad = \text{(definition of sub)} \\
\{\mathsf{j} \mid \beta \ = \ \alpha\} \mathbin{+\!\!+} \\
\mathsf{concat}\ \{\mathsf{q}\ (\gamma, \mathsf{i})\ \mathsf{k} \mid (\gamma, \mathsf{i}) \in \overline{(\alpha, \mathsf{i})},\ \gamma \ = \ \mathsf{A},\ \mathsf{A} \rightarrow \beta,\ \mathsf{k} \in \mathsf{h}\ (\mathsf{A}, \mathsf{i})\ (\beta, \mathsf{i})\ \mathsf{j}\} \mathbin{+\!\!+} \\
\mathsf{concat}\ \{\mathsf{q}\ (\gamma, \mathsf{i})\ \mathsf{k} \mid (\gamma, \mathsf{i}) \in \overline{(\alpha, \mathsf{i})},\ \beta\delta \leftarrow \gamma,\ \mathsf{k} \in \mathsf{h}\ (\beta\delta, \mathsf{i})\ (\beta, \mathsf{i})\ \mathsf{j}\} \\
\qquad = \text{(definition of h)} \\
\{\mathsf{j} \mid \beta \ = \ \alpha\} \mathbin{+\!\!+} \\
\mathsf{concat}\ \{\mathsf{q}\ (\mathsf{A}, \mathsf{i})\ \mathsf{j} \mid (\mathsf{A}, \mathsf{i}) \in \overline{(\alpha, \mathsf{i})},\ \mathsf{A} \rightarrow \beta\} \mathbin{+\!\!+} \\
\mathsf{concat}\ \{\mathsf{q}\ (\beta\delta, \mathsf{i})\ \mathsf{k} \mid (\beta\delta, \mathsf{i}) \in \overline{(\alpha, \mathsf{i})},\ \mathsf{k} \in \mathsf{p}\ (\delta, \mathsf{j})\}
\end{array}$$

□

This algorithm can be simplified by eliminating the string i from sub-problems, and thus from the elements of $\overline{(\alpha, \mathsf{i})}$:

Definition 3.29 (RA1-nd)

$$\begin{array}{l}
\mathsf{p}\ \alpha\ \mathsf{i} \ = \ \mathsf{concat}\ \{\mathsf{q}\ \mathsf{x}\ \mathsf{j} \mid \mathsf{x} : \mathsf{j} \leftarrow \mathsf{i},\ \mathsf{x} \in \overline{\alpha}\} \mathbin{+\!\!+} \\
\qquad\qquad \mathsf{concat}\ \{\mathsf{q}\ \varepsilon\ \mathsf{i} \mid \varepsilon \in \overline{\alpha}\} \\
\textbf{where}\ \mathsf{q}\ \beta\ \mathsf{j} = \{\mathsf{j} \mid \beta \ = \ \alpha\} \mathbin{+\!\!+} \\
\qquad\qquad\qquad \mathsf{concat}\ \{\mathsf{q}\ \mathsf{A}\ \mathsf{j} \mid \mathsf{A} \in \overline{\alpha},\ \mathsf{A} \rightarrow \beta\} \mathbin{+\!\!+} \\
\qquad\qquad\qquad \mathsf{concat}\ \{\mathsf{q}\ (\beta\gamma)\ \mathsf{k} \mid (\beta\gamma) \in \overline{\alpha},\ \mathsf{k} \in \mathsf{p}\ \gamma\ \mathsf{j}\}
\end{array}$$

□

3.3.6 Another parser transformation

In this section we discuss the transformation of a parser based on *items* (dotted rules). The reason for doing so is that the set of problems T in the previous recursive descent parser is not uniform: it is $V \times V^*$. As a consequence of this, the sub-problem relation must distinguish between the two cases V and V^*. The following parser does not have this problem.

An item-based recursive descent parser

For a given context-free grammar $G = (V_T, V_N, P, S)$ we define the set of items *Items*, with typical elements J, K by:

$$\mathit{Items} \;=\; \{\mathsf{A} \to \alpha.\beta \mid \mathsf{A} \to \alpha\beta\} \mathbin{+\!\!+} \{\mathsf{x} \to .\mathsf{x} \mid \mathsf{x} \in \mathsf{V_T}\} \mathbin{+\!\!+} \{\mathsf{S}' \to \gamma.\mathsf{S}\}$$

where S' is a new symbol not in V and γ an arbitrary element of V^+. We will write $\mathsf{A} \to \alpha.\beta$ for elements of *Items*. Note that this is in conflict with our notational convention, which assumes that the symbol A is an element of V_N, but we hope that no confusion will arise. We use this set *Items* in the definition of the following recursive descent parser parse. This function is different to the one introduced above, since it operates on items. We will be using this form of overloading more frequently in the sequel. The function parse obeys the specification:

Specification 3.30 (spec:ITD-nd)

$$
\begin{array}{lcl}
T & = & \mathit{Items} \times V_T^* \\
S & = & V_T^* \\
\mathsf{parse} & :: & \mathit{Items} \times V_T^* \mapsto \mathit{Set}(V_T^*) \\
\mathsf{parse}\ (\mathsf{A} \to \alpha.\beta)\ \mathsf{i} & = & \{\mathsf{j} \mid \exists \mathsf{j} : \beta \overset{*}{\to} \mathsf{i}/\mathsf{j}\}
\end{array}
$$

□

and which is implemented by:

Implementation 3.31 (of specification (3.30)) (ITD-nd)

$$
\begin{array}{l}
\mathsf{parse}\ ((\mathsf{A} \to \alpha.), \mathsf{i}) \;=\; \{\mathsf{i}\} \\
\mathsf{parse}\ ((\mathsf{x} \to .\mathsf{x}), \mathsf{i}) \;=\; \{\mathsf{j} \mid \mathsf{x} : \mathsf{j} \leftarrow \mathsf{i}\} \\
\mathsf{parse}\ ((\mathsf{A} \to \alpha.\mathsf{X}\beta), \mathsf{i}) \\
= \{\mathsf{k} \mid (\mathsf{X} \to .\gamma) \in \mathit{Items},\ \mathsf{j} \in \mathsf{parse}\ ((\mathsf{X} \to .\gamma), \mathsf{i}),\ \mathsf{k} \in \mathsf{parse}\ ((\mathsf{A} \to \alpha\mathsf{X}.\beta), \mathsf{j})\}
\end{array}
$$

□

We can rewrite this definition in the following recursive descent form, which can be proved by substituting the three cases.

Theorem 3.32 (IRD-nd)

$$
\begin{array}{l}
\mathsf{parse}\ (\mathsf{J}, \mathsf{i}) \;= \\
\mathsf{t}\ (\mathsf{J}, \mathsf{i}) \mathbin{+\!\!+} \mathsf{concat}\ \{\mathsf{h}\ (\mathsf{J}, \mathsf{i})\ (\mathsf{K}, \mathsf{k})\ \mathsf{j} \mid (\mathsf{K}, \mathsf{k}) \in \mathsf{sub}\ (\mathsf{J}, \mathsf{i}),\ \mathsf{j} \in \mathsf{parse}\ (\mathsf{K}, \mathsf{k})\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{t}\ ((\mathsf{A} \to \alpha.), \mathsf{i}) & = & \{\mathsf{i}\} \\
\mathsf{t}\ ((\mathsf{x} \to .\mathsf{x}), \mathsf{i}) & = & \{\mathsf{j} \mid \mathsf{x} : \mathsf{j} \leftarrow \mathsf{i}\} \\
\mathsf{t}\ ((\mathsf{A} \to \alpha.\mathsf{X}\beta), \mathsf{i}) & = & \{\}
\end{array}
$$

$$\begin{aligned}
&\mathsf{sub}\ (A \rightarrow \alpha., i) &&= \{\} \\
&\mathsf{sub}\ (x \rightarrow .x, i) &&= \{\} \\
&\mathsf{sub}\ (A \rightarrow \alpha.X\beta, i) &&= \{(X \rightarrow .\gamma, i) \mid (X \rightarrow .\gamma) \in \mathit{Items}\} \\
&\mathsf{h}\ (A \rightarrow \alpha.X\beta, i)\ (X \rightarrow .\gamma, i)\ j &&= \mathsf{parse}\ ((A \rightarrow \alpha X.\beta), j)
\end{aligned}$$

□

This parser is called on the start symbol S and input string ω by parse $(S' \rightarrow .S, \omega)$.

The corresponding recursive ascent parser

We can transform this new item-based parser into a recursive ascent version. First we give the relation $\Rightarrow$ and the specification for q.

$$((A \rightarrow \alpha.\beta, i), k) \overset{*}{\Rightarrow} ((X \rightarrow .\gamma, i), j)$$
$$\equiv$$
$$\exists \delta : \beta \overset{*}{\rightarrow}_L X\delta \wedge \gamma \overset{*}{\rightarrow} i/j \wedge \delta \overset{*}{\rightarrow} j/k$$

Given the problem $(A \rightarrow \alpha.\beta, i)$, the local function q is specified by:

$$\mathsf{q}\ (X \rightarrow \gamma.\delta, i)\ j \ = \ \{k \mid \exists \mu, k : \beta \overset{*}{\rightarrow}_L X\mu,\ \delta \overset{*}{\rightarrow} i/j,\ \mu \overset{*}{\rightarrow} j/k\}$$

Substituting the functions t, sub and h in the item-based recursive descent algorithm gives the following recursive ascent parser:

Derivation 3.33

$\mathsf{p}\ (J, i)$
 = (recursive ascent definition)
$\mathsf{concat}\ \{\mathsf{q}\ (K, i)\ j \mid (K, i) \in \overline{(J, i)},\ j \in \mathsf{t}\ (K, i)\}$
 = (definition of t (K, i))
$\mathsf{concat}\ \{\mathsf{q}\ (A \rightarrow \alpha., i)\ i \mid (A \rightarrow \alpha., i) \in \overline{(J, i)}\} +\!\!+$
$\mathsf{concat}\ \{\mathsf{q}\ (x \rightarrow .x, i)\ j \mid x : j \leftarrow i,\ (x \rightarrow .x, i) \in \overline{(J, i)}\}$
where
  $\mathsf{q}\ (K, i)\ j$
   = (recursive ascent definition)
  $\{j \mid (K, i) = (J, i)\} +\!\!+$
  $\mathsf{concat}\ \{\mathsf{q}\ (L, i)\ k \mid (L, i) \in \overline{(J, i)},\ (K, i) \in \mathsf{sub}\ (L, i),\ k \in \mathsf{h}\ (L, i)\ (K, i)\ j\}$
   = (definition of sub and h)
  $\{j \mid K = J\} +\!\!+$
  $\mathsf{concat}\ \{\mathsf{q}\ (A \rightarrow \alpha.X\beta, i)\ k \mid (A \rightarrow \alpha.X\beta, i) \in \overline{(J, i)},$
          $(X \rightarrow .\gamma) \leftarrow K,\ k \in \mathsf{p}\ (A \rightarrow \alpha X.\beta, j)\}$

□

Here we have made use of the fact that if $(K, i) \in \mathsf{sub}\ (L, i)$, then K must be of the form $X \rightarrow .\gamma$ due to the definition of sub.

The result can again be simplified by eliminating redundant occurrences of i. We must modify our definition of $\overline{(J, i)}$ to a definition of $\overline{J}$ and adjust some other definitions, but since this amounts only to leaving out occurrences of i, we do not define $\overline{J}$ here. We apply Currying in our simplification in order to leave out redundant parentheses. The simplification of the result of derivation (3.33) thus becomes:

Simplification 3.34

$$\begin{aligned} p\ J\ i\ =\ & \text{concat}\ \{q\ (A \to \alpha.)\ i \mid (A \to \alpha.) \in \overline{J}\} \mathbin{+\!\!+} \\ & \text{concat}\ \{q\ (x \to .x)\ j \mid x : j \leftarrow i,\ (x \to .x) \in \overline{J}\} \end{aligned}$$

where

$$\begin{aligned} & q\ (X \to \gamma.\delta)\ j \\ & = \{j \mid (X \to \gamma.\delta)\ =\ J\} \mathbin{+\!\!+} \\ & \quad \text{concat}\ \{q\ (A \to \alpha.X\beta)\ k \mid (A \to \alpha.X\beta) \in \overline{J},\ \gamma\ =\ \varepsilon,\ k \in p\ (A \to \alpha X.\beta)\ j\} \end{aligned}$$

□

The set $\overline{J}$ can be interpreted as the closure of the item set $\{J\}$ in the classical bottom-up parsing theory.

3.3.7 Another recursive ascent scheme: RA2

In our last example, the item-based recursive ascent parser, we can observe that the second part of q does not make use of all the information stored in the item $X \to \gamma.\delta$. It only uses the symbol X and the emptiness of γ and not the whole item. We can exploit this by modifying the algorithm in such a way that X is passed to q instead of the whole item.
This is more generally described by another way of recursive ascending which we present with its derivation. It is based on considering the transitive, but not reflexive closure of the sub-problem relation. We denote this closure with regard to a problem α by $\overline{\alpha}^+$. Instead of passing problems from $\overline{\alpha}$ to q, we pass problems of $\overline{\alpha}^+$ to q only.

Definition 3.35 (all-sub-plus)

$$\overline{\alpha}^+\ =\ \text{concat}\ \{\text{sub}\ \beta \mid \beta \in \overline{\alpha}\}$$

□

From this definition we can derive the property:

Derivation 3.36

$$\begin{aligned} & \gamma \in \text{sup}\ \alpha\ \beta \\ & \qquad = \ \text{(definition (3.16))} \\ & \beta \in \text{sub}\ \gamma \wedge \gamma \in \overline{\alpha} \\ & \qquad \Rightarrow\ \text{(definition (3.35))} \\ & \beta \in \overline{\alpha}^+ \end{aligned}$$

□

The RA2 definition is given by:

Definition 3.37 (RA2)

```
RA2 t sub h  =  p where p $\alpha$  =  t $\alpha$ ++ concat {q $\beta$ x | $\beta \in \overline{\alpha}^+$, x $\in$ t $\beta$}
where
q $\beta$ x  =  {y | ($\alpha$, y) $\in$ next} ++
            concat {q $\gamma$ y | ($\gamma$, y) $\in$ next, $\gamma \in \overline{\alpha}^+$}
            where next  =  {($\gamma$, y) | $\gamma \in$ sup $\alpha$ $\beta$, y $\in$ h $\gamma$ $\beta$ x}
```
□

The RA2 algorithm can be derived from the RA1 algorithm, thus proving the identity RD = RA1 = RA2. We use q to denote the q from RA1 and q′ for the RA2 version. We define:

$$q'\ \beta\ x\ =\ \text{concat}\ \{q\ \gamma\ y \mid \gamma \in \text{sup}\ \alpha\ \beta,\ y \in h\ \gamma\ \beta\ x\}$$

From this follows by the definition of q in the RA1 scheme (3.17) and derivation (3.36)

$$q\ \beta\ x\ =\ \{x \mid \beta\ =\ \alpha\} +\!\!+ \text{concat}\ \{q'\ \beta\ x \mid \beta \in \overline{\alpha}^+\} \qquad \text{(ra1-ra2)}\quad (3.38)$$

Using this identity, we can rewrite the RA1 definition of p into the RA2 form:

Derivation 3.39

```
p $\alpha$
      =  (RA1 definition)
t $\alpha$ ++ concat {q $\beta$ x | $\beta \in \overline{\alpha}$, x $\in$ t $\beta$}
      =  (identity (3.38))
t $\alpha$ ++
{x | $\alpha \in \overline{\alpha}$, x $\in$ t $\alpha$} ++
concat {q′ $\beta$ x | $\beta \in \overline{\alpha}^+$, x $\in$ t $\beta$}
      =  (set calculus)
t $\alpha$ ++ concat {q′ $\beta$ x | $\beta \in \overline{\alpha}^+$, x $\in$ t $\beta$}
```
□

In the same way, we can rewrite the definition of q′:

Derivation 3.40

```
q′ $\beta$ x
      =  (definition of q′)
concat {q $\gamma$ y | $\gamma \in$ sup $\alpha$ $\beta$, y $\in$ h $\gamma$ $\beta$ x}
      =  (introduce next)
concat {q $\gamma$ y | ($\gamma$, y) $\in$ next}
where next  =  {($\gamma$, y) | $\gamma \in$ sup $\alpha$ $\beta$, y $\in$ h $\gamma$ $\beta$ x}
      =  (apply (3.38))
{y | ($\alpha$, y) $\in$ next} ++
concat {q′ $\gamma$ y | ($\gamma$, y) $\in$ next, $\gamma \in \overline{\alpha}^+$}
where next  =  {($\gamma$, y) | $\gamma \in$ sup $\alpha$ $\beta$, y $\in$ h $\gamma$ $\beta$ x}
```

□

This RA2 scheme can be applied to our item-based parser. In this case, we have to determine whether an item is a member of the set $\{J\}$ or $\overline{J}^+$, given the item is an element of $\overline{J}$. This can easily be determined, given that $A \to \alpha.\beta \in \overline{J}$:

$$\begin{aligned} A \to \alpha.\beta &= J && \equiv \alpha \neq \varepsilon \\ A \to \alpha.\beta &\in \overline{J}^+ && \equiv \alpha = \varepsilon \end{aligned}$$

The resulting recursive ascent parser can thus be expressed as:

```
p J i = {i | (A → α.) ← J} ++
        {j | x : j ← i, J = (x → .x)} ++
        concat {q A i | (A → .) ∈ J̄⁺} ++
        concat {q x j | x : j ← i, (x → .x) ∈ J̄⁺}
where
q X j = {k | (A → α.Xβ, k) ∈ next, α ≠ ε} ++
        {l | (A → .Xβ, k) ∈ next, l ∈ q A k}
        where next = {(A → α.Xβ, k) | (A → α.Xβ) ∈ J̄, k ∈ p (A → αX.β) j}
```

Observing that p is never called with first argument $(x \to .x)$, we can simplify this to

Definition 3.41 (IRA2-nd)

```
p J i = {i | (A → α.) ← J} ++
        concat {q A i | (A → .) ∈ J̄⁺} ++
        concat {q x j | x : j ← i, (x → .x) ∈ J̄⁺}
where
q X j = {k | (A → α.Xβ, k) ∈ next, α ≠ ε} ++
        {l | (A → .Xβ, k) ∈ next, l ∈ q A k}
        where next = {(A → α.Xβ, k) | (A → α.Xβ) ∈ J̄, k ∈ p (A → αX.β) j}
```

□

This parser is called on the start symbol S by p $(S' \to \gamma.S)$ ω.

3.3.8 Solving sets of problems

In this section we present recursive descent and ascent algorithms for solving sets of problems, instead of a single problem. We define a function P that takes a set of problems μ instead of a single problem α. It can not deliver just a set of results, but it has to indicate which result corresponds to which problem in μ. Hence it returns a set of (α, x) pairs, where α is an element of μ and the x is a solution to the problem α. We call the resulting scheme *set recursive descent*, SRD.

Definition 3.42 (SRD)

$\mathsf{SRD} :: T \mapsto Set(S) \times T \mapsto Set(T) \times T_2 \times T \times S \mapsto Set(S) \times Set(T) \mapsto Set(T \times S)$

$$\begin{aligned}\mathsf{SRD\ t\ sub\ h} \;=\; \mathsf{P}\ \textbf{where}\ \mathsf{P}\ \mu \;&=\; \{(\alpha, \mathsf{x}) \mid \alpha \in \mu,\ \mathsf{x} \in \mathsf{p}\ \alpha\}\\ &\textbf{where}\ \mathsf{p} \;=\; \mathsf{RD\ t\ sub\ h}\end{aligned}$$

□

Using this definition we can transform the SRD algorithm into a *set recursive ascent algorithm* SRA1.
We first define some domains:

$\mu,\ \overline{\mu} :: Set(T)$
$P :: Set(T) \mapsto Set(T \times S)$
$Q :: T \times S \mapsto Set(T \times S)$

and the set:

Definition 3.43 (sub-mu-all)

$$\overline{\mu} \;=\; \mathsf{concat}\ \{\overline{\alpha} \mid \alpha \in \mu\}$$

□

and the function

$$\mathsf{sup}\ \mu\ \beta \;=\; \{\gamma \mid \gamma \in \overline{\mu},\ \beta \in \mathsf{sub}\ \gamma\}$$

The recursive ascent functions for SRA1 are given by:

Definition 3.44 (SRA1)

$$\begin{aligned}&\mathsf{SRA1\ t\ sub\ h} \;=\; \mathsf{P}\ \textbf{where}\\ &\mathsf{P}\ \mu \;=\; \mathsf{concat}\ \{\mathsf{Q}\ \alpha\ \mathsf{x} \mid \alpha \in \overline{\mu},\ \mathsf{x} \in \mathsf{t}\ \alpha\}\\ &\textbf{where}\\ &\mathsf{Q}\ \beta\ \mathsf{x} \;=\; \{(\beta, \mathsf{x}) \mid \beta \in \mu\} +\!\!+\\ &\qquad\qquad \mathsf{concat}\ \{\mathsf{Q}\ \gamma\ \mathsf{y} \mid \gamma \in \mathsf{sup}\ \mu\ \beta,\ \mathsf{y} \in \mathsf{h}\ \gamma\ \beta\ \mathsf{x}\}\end{aligned}$$

□

Of course, we are obliged to present the SRD to SRA1 transformation, proving SRD = SRA1. To this end, we make use of the identity RD = RA1 in the definition of p. First we give a specification for Q in terms of the RA1 version of q. In order to do this, we must raise q from a local to the global level, adding the argument α.

$$\mathsf{Q}\ \mu\ \beta\ \mathsf{x} \;=\; \{(\alpha, \mathsf{y}) \mid \alpha \in \mu,\ \beta \in \overline{\alpha},\ \mathsf{y} \in \mathsf{q}\ \alpha\ \beta\ \mathsf{x}\} \qquad \text{(SRA1-Q)} \quad (3.45)$$

Rewriting the definition of P gives:

Derivation 3.46

$\mathsf{P}\ \mu$
  = (definition (3.42))
$\{(\alpha,\mathsf{x}) \mid \alpha \in \mu,\ \mathsf{x} \in \mathsf{p}\ \alpha\}$
  = (RA1 definition of p)
$\{(\alpha,\mathsf{y}) \mid \alpha \in \mu,\ \beta \in \overline{\alpha},\ \mathsf{x} \in \mathsf{t}\ \beta,\ \mathsf{y} \in \mathsf{q}\ \alpha\ \beta\ \mathsf{x}\}$
  = (definition (3.45), (3.43) and (3.15))
$\mathsf{concat}\ \{\mathsf{Q}\ \mu\ \beta\ \mathsf{x} \mid \beta \in \overline{\mu},\ \mathsf{x} \in \mathsf{t}\ \beta\}$

□

The definition of Q can also be rewritten into the required form:

Derivation 3.47

$\mathsf{Q}\ \mu\ \beta\ \mathsf{x}$
  = (definition)
$\{(\alpha,\mathsf{y}) \mid \alpha \in \mu,\ \beta \in \overline{\alpha},\ \mathsf{y} \in \mathsf{q}\ \alpha\ \beta\ \mathsf{x}\}$
  = (RA1 definition of q)
$\{(\alpha,\mathsf{y}) \mid \alpha \in \mu,\ \beta \in \overline{\alpha},\ \beta = \alpha,\ \mathsf{y} \leftarrow \mathsf{x}\} +\!\!+$
$\{(\alpha,\mathsf{z}) \mid \alpha \in \mu,\ \gamma \in \mathsf{sup}\ \alpha\ \beta,\ \mathsf{y} \in \mathsf{h}\ \gamma\ \beta\ \mathsf{x},\ \mathsf{z} \in \mathsf{q}\ \alpha\ \gamma\ \mathsf{y}\}$
  = (definition of Q)
$\{(\beta,\mathsf{x}) \mid \beta \in \mu\} +\!\!+$
$\mathsf{concat}\ \{\mathsf{Q}\ \mu\ \gamma\ \mathsf{y} \mid \gamma \in \mathsf{sup}\ \mu\ \beta,\ \mathsf{y} \in \mathsf{h}\ \gamma\ \beta\ \mathsf{x}\}$

□

We do not present a SRA2 algorithm here but instead go on to the class of doubly recursive programs, very interesting for parsers, which can be derived from the SRA1 and SRA2 algorithms.

3.3.9 Transforming doubly recursive programs

In this section we deal with a special case of recursive descent algorithms, namely the case where h invokes p again in a certain way. We call such an algorithm *doubly recursive descent*. In this section we present a scheme for the transformation of a special class of doubly recursive descent algorithms into recursive ascent algorithms. For this class we assume that h is of the form

$$\mathsf{h}\ \alpha\ \beta\ \mathsf{x} = (\mathsf{g}_1\ \alpha\ \beta\ ;\ \mathsf{p}\ ;\ \mathsf{g}_2\ \alpha\ \beta\ \mathsf{x})\ \mathsf{x}$$

where g_1 and g_2 are assumed to be simple functions, at least with regard to p with typing:

$$\mathsf{g}_1 :: T \times T \times S \mapsto Set(T)$$
$$\mathsf{g}_2 :: T \times T \times S \times S \mapsto Set(S)$$

Just as h, these functions are partial functions.
Both the RA1 and RA2 transformation schemes can be expressed in a special way for such doubly recursive descent algorithms, exploiting the doubly recursive nature.

Scheme 1

In this section we adopt the first transformation scheme RA1 for doubly recursive descent problems.
The RD definition of p can be rewritten into a doubly recursive descent form DRD:

Definition 3.48 (DRD)

$$\begin{aligned}\mathsf{DRD} :: (T \mapsto Set(S))\times \\ (T \mapsto Set(T))\times \\ (T\times T\times S \mapsto Set(()T))\times \\ (T\times T\times S\times S \mapsto Set(S))\times \\ Set(T) \mapsto Set(T\times S)\end{aligned}$$

$\mathsf{DRD\ t\ sub\ g_1\ g_2\ =\ p}$ **where**
$\mathsf{p}\ \alpha\ =\ \mathsf{t}\ \alpha \mathbin{+\!\!\!+} \mathsf{concat}\ \{\mathsf{y} \mid \beta \in \mathsf{sub}\ \alpha,\ \mathsf{x} \in \mathsf{p}\ \beta,\ \mathsf{y} \in (\mathsf{g_1}\ \alpha\ \beta\ ;\ \mathsf{p}\ ;\ \mathsf{g_2}\ \alpha\ \beta \mathsf{x})\ \mathsf{x}\}$
□

We clearly have the identity

$\mathsf{DRD\ t\ sub\ g_1\ g_2\ =\ RD\ t\ sub\ h}$
where $\mathsf{h}\ \alpha\ \beta\ \mathsf{x}\ =\ (\mathsf{g_1}\ \alpha\ \beta\ ;\ \mathsf{p}\ ;\ \mathsf{g_2}\ \alpha\ \beta\mathsf{x})\ \mathsf{x}$

As we will see, doubly recursive ascent algorithms can take advantage of the set-version SRA1 for general recursive ascent algorithms. We make use of two new functions with typing:

$\mathsf{goto} :: Set(T)$
$\mathsf{combine} :: T\times T \mapsto Set(T)$

The doubly recursive ascent functions for the new scheme DRA1 are given by:

Definition 3.49 (DRA1)

$\mathsf{DRA1\ t\ sub\ g_1\ g_2\ =\ P}$ **where**
$\mathsf{P}\ \mu\ =\ \mathsf{concat}\ \{\mathsf{Q}\ \alpha\ \mathsf{x} \mid \alpha \in \overline{\mu},\ \mathsf{x} \in \mathsf{t}\ \alpha\}$
where
$\mathsf{Q}\ \beta\ \mathsf{x}\ =\ \{(\beta,\mathsf{x}) \mid \beta \in \mu\} \mathbin{+\!\!\!+}$
$\qquad \mathsf{concat}\ \{\mathsf{Q}\ \gamma\ \mathsf{z} \mid (\delta,\mathsf{y}) \in \mathsf{P\ goto},\ \gamma \in \mathsf{combine}\ \beta\ \delta,\ \mathsf{z} \in \mathsf{g_2}\ \gamma\ \beta\ \mathsf{x}\ \mathsf{y}\}$
where $\mathsf{goto}\ =\ \mathsf{concat}\ \{\mathsf{g_1}\ \gamma\ \beta\ \mathsf{x} \mid \gamma \in \overline{\mu},\ \beta \in \mathsf{sub}\ \gamma\}$
$\qquad \mathsf{combine}\ \alpha\ \beta\ =\ \{\gamma \mid \gamma \in \mathsf{sup}\ \mu\ \alpha,\ \beta \in \mathsf{g_1}\ \gamma\ \alpha\ \mathsf{x}\}$
□

This recursive ascent algorithm can be interpreted in the following way. Instead of recursively calling p for each solution δ separately, it is more efficient to call P on the whole set of new problems δ which are consistent with β by means of $\mathsf{g_1}$. Of course, P must now return the problem δ that it solved with each solution y for that problem. The function combine β δ constructs the super-problems γ which can be split into consistent (β, δ) problems. Since both β and δ have been solved with solutions x and y respectively, γ has

been solved with solution z, where z is computed with g_2. The result is more deterministic than the previous version, since it groups all new problems $\delta \in g_1\ \gamma\ \beta\ x$ for an already solved problem β into one set goto, rather than solving them one by one.
Of course we are obliged to present the SRA1 to DRA1 transformation. Observe that P does not have to be rewritten, since the DRA1 form is equal to the SRA1 form. The definition of Q can be rewritten into the required form, adding μ to Q for clarity:

Derivation 3.50

$Q\ \mu\ \beta\ x$
  $=$ (SRA1 definition)
$\{(\beta, x) \mid \beta \in \mu\} +\!\!+$
concat $\{Q\ \mu\ \gamma\ y \mid \gamma \in \text{sup}\ \mu\ \beta,\ y \in h\ \gamma\ \beta\ x\}$
  $=$ (definition of h)
$\{(\beta, x) \mid \beta \in \mu\} +\!\!+$
concat $\{Q\ \mu\ \gamma\ y \mid \gamma \in \text{sup}\ \mu\ \beta,\ y \in (g_1\ \gamma\ \beta\ ;\ p\ ;\ g_2\ \gamma\ \beta)\ x\}$
  $=$ (definition of ;)
$\{(\beta, x) \mid \beta \in \mu\} +\!\!+$
concat $\{Q\ \mu\ \gamma\ z \mid \gamma \in \text{sup}\ \mu\ \beta,\ \delta \in g_1\ \gamma\ \beta\ x,\ y \in p\ \delta,\ z \in g_2\ \gamma\ \beta\ y\}$
  $=$ (definition of P)
$\{(\beta, x) \mid \beta \in \mu\} +\!\!+$
concat $\{Q\ \mu\ \gamma\ z \mid (\delta, y) \in P\ (\text{concat}\ \{g_1\ \gamma\ \beta\ x \mid \gamma \in \text{sup}\ \mu\ \beta\}),\ \gamma \in \text{sup}\ \mu\ \beta,$
    $\delta \in g_1\ \gamma\ \beta\ x,\ z \in g_2\ \gamma\ \beta\ x\ y\}$
  $=$ (definition of goto)
$\{(\beta, x) \mid \beta \in \mu\} +\!\!+$
concat $\{Q\ \mu\ \gamma\ z \mid (\delta, y) \in P\ \text{goto},\ \gamma \in \text{sup}\ \mu\ \beta,\ \delta \in g_1\ \gamma\ \beta\ x,\ z \in g_2\ \gamma\ \beta\ x\ y\}$
  $=$ (definition of combine)
$\{(\beta, x) \mid \beta \in \mu\} +\!\!+$
concat $\{Q\ \mu\ \gamma\ z \mid (\delta, y) \in P\ \text{goto},\ \gamma \in \text{combine}\ \beta\ \delta,\ z \in g_2\ \gamma\ \beta\ x\ y\}$

□

Scheme 2

Alternatively, the second scheme RA2 for recursive ascending can also be applied to doubly recursive problems. This yields the following recursive ascent scheme DRA2 which we present without proof. It can, however, be derived from DRA1 in a fashion similar to the derivation of RA2 from RA1.

$\overline{\mu}^{+} = \{\beta \mid \alpha \in \overline{\mu},\ \beta \in \text{sub}\ \alpha\}$

Definition 3.51 (DRA2)

DRA2 t sub $g_1\ g_2$ = P **where**
$P\ \mu = \{(\alpha, x) \mid \alpha \in \mu,\ x \in t\ \alpha\} +\!\!+$
    concat $\{Q\ \alpha\ x \mid \alpha \in \overline{\mu}^{+},\ x \in t\ \alpha\}$

$$\begin{array}{l}
\textbf{where} \\
Q\ \beta\ x = \{(\gamma, y) \mid (\gamma, y) \in \text{next},\ \gamma \in \mu\} \mathbin{+\!\!+} \\
\qquad\quad \text{concat}\ \{Q\ \gamma\ y \mid (\gamma, y) \in \text{next},\ \gamma \in \overline{\mu}^{+}\} \\
\qquad\quad \textbf{where}\ \text{goto} = \text{concat}\ \{g_1\ \gamma\ \beta\ x \mid \gamma \in \overline{\mu},\ \beta \in \text{sub}\ \gamma\} \\
\qquad\qquad\quad \text{next} = \{(\gamma, z) \mid (\delta, y) \in P\ \text{goto}, \gamma \in \text{combine}\ \beta\ \delta, z \in g_2\ \gamma\ \beta\ x\ y\} \\
\qquad\qquad\quad \text{combine}\ \alpha\ \beta = \{\gamma \mid \gamma \in \text{sup}\ \mu\ \alpha,\ \beta \in g_1\ \gamma\ \alpha\ x\}
\end{array}$$

□

Derivation of the Leermakers recursive ascent parser

When we apply this last scheme DRA2 to our item-based parser, we obtain the Leermakers recursive ascent parser [LAK92]. We use μ for elements of *Set*(*Items*) and have omitted redundant occurrences of i. All sub-problems of $(A \to \alpha.\beta, i)$ share the same string i, which can thus be removed from μ and passed as a separate parameter.
Recall that:

$$h\ (A \to \alpha.X\beta)\ (X \to .\gamma)\ j = p\ (A \to \alpha X.\beta, j)$$

which yields for g_1 and g_2:

$$\begin{array}{l}
g_1\ (A \to \alpha.X\beta)\ (X \to .\gamma)\ j = \{(A \to \alpha X.\beta, j)\} \\
g_2\ (A \to \alpha.X\beta)\ (X \to .\gamma)\ j\ k = \{k\}
\end{array}$$

The specification of the Leermakers parser is given by:

Specification 3.52 (spec:Leermakers)

$$\begin{array}{l}
P\ \mu\ i = \{(A \to \alpha.\beta, j) \mid \exists j\ :\ (A \to \alpha.\beta) \in \mu,\ \beta \overset{*}{\to} i/j\} \\
\textbf{where} \\
Q\ X\ j = \{(A \to \alpha.\beta, k) \mid \exists \gamma, k\ :\ (A \to \alpha.\beta) \in \mu,\ \beta \overset{*}{\to}_L X\gamma,\ X \overset{*}{\to} i/j,\ \gamma \overset{*}{\to} j/k\}
\end{array}$$

□

The implementation of the parser becomes (by applying scheme DRA2):

Implementation 3.53 (of specification (3.52)) (Leermakers)

$$\begin{array}{l}
P\ \mu\ i = \{(A \to \alpha., i) \mid (A \to \alpha.) \in \mu\} \mathbin{+\!\!+} \\
\qquad\quad \{(x \to .x, j) \mid x : j \leftarrow i,\ (x \to .x) \in \mu\} \mathbin{+\!\!+} \\
\qquad\quad \text{concat}\ \{Q\ A\ i \mid (A \to .) \in \overline{\mu}^{+}\} \mathbin{+\!\!+} \\
\qquad\quad \text{concat}\ \{Q\ x\ j \mid x : j \leftarrow i,\ (x \to .x) \in \overline{\mu}^{+}\} \\
\textbf{where}\ Q\ X\ j = \{(A \to \alpha.X\beta, k) \mid (A \to \alpha.X\beta, k) \in \text{next},\ \alpha \neq \varepsilon\} \mathbin{+\!\!+} \\
\qquad\qquad\quad \text{concat}\ \{Q\ A\ k \mid (A \to .X\beta, k) \in \text{next}\} \\
\qquad\qquad\quad \textbf{where}\ \text{next} = \{(A \to \alpha.X\beta, k) \mid (A \to \alpha X.\beta, k) \in P\ (\text{goto}\ \mu\ X)\ j\}
\end{array}$$

□

Where goto is defined by:

Definition 3.54 (goto)

$$\text{goto } \mu\ X = \{A \to \alpha X.\beta \mid (A \to \alpha.X\beta) \in \overline{\mu}\}$$

□

Since the function starts with item $S' \to \alpha.S$, $\alpha \in V^+$ and because only items reachable from this one by applying the goto definition iteratively can be elements of μ, we can observe that items $(x \to .x)$ and $(A \to .)$ can never be elements of μ. This allows us to simplify the definition of P. Moreover, the definition of next can be simplified to P goto j, leaving the 'shifting of the dot' to the users of next.

Definition 3.55 (IDRA2-nd)

$$\begin{aligned} \text{P } \mu \text{ i} = {} & \{(A \to \alpha., i) \mid (A \to \alpha.) \in \mu\} +\!\!+ \\ & \text{concat } \{Q\ A\ i \mid (A \to .) \in \overline{\mu}\} +\!\!+ \\ & \text{concat } \{Q\ x\ j \mid x : j \leftarrow i,\ (x \to .x) \in \overline{\mu}\} \\ & \textbf{where } Q\ X\ j = \{(A \to \alpha.X\beta, k) \mid (A \to \alpha X.\beta, k) \in \text{next},\ \alpha \neq \varepsilon\} +\!\!+ \\ & \qquad\qquad\quad \text{concat } \{Q\ A\ k \mid (A \to X.\beta, k) \in \text{next}\} \\ & \qquad\qquad\quad \textbf{where } \text{next} = \text{P (goto } \mu\ X)\ j \end{aligned}$$

□

Which is the Leermakers non-deterministic recursive ascent parser presented in [LAK92]. It is deterministic for exactly the class of LR(0) grammars and thus can be called a non-deterministic LR(0) parser.

3.3.10 Recursive descent parsing using continuations

Using the continuations transformation (2.66), we can derive a continuation-based parser from a plain recursive descent parser. We recapitulate the latter (3.8):

$$\begin{aligned} \text{parse } \varepsilon \text{ i} &= \{i\} \\ \text{parse x i} &= \{j \mid x : j \leftarrow i\} \\ \text{parse A i} &= \{j \mid A \to \alpha,\ j \in \text{parse } \alpha \text{ i}\} \\ \text{parse } \alpha\beta \text{ i} &= \{k \mid j \in \text{parse } \alpha \text{ i},\ k \in \text{parse } \beta \text{ j}\} \end{aligned}$$

The last of these definitions can be expressed by:

$$\text{parse } \alpha\beta \text{ i} = (\text{parse } \alpha \ ;\ \text{parse } \beta)\ \text{i}$$

The continuation-based parser is defined by using:

$$\text{cont} :: V^* \times (V_T^* \mapsto Set(V_T^*)) \mapsto Set(V_T^*))$$

$$\text{cont } \alpha \text{ c} = \text{parse } \alpha\ ;\ \text{c}$$

and thus by the definition of ' ; ' (2.61):

$$\text{cont } \alpha \text{ c i} = \text{concat } \{c\ j \mid j \in \text{parse } \alpha \text{ i}\}$$

Taking the four forms of α and substituting them in the definition of cont gives:

- cont ε c i

  = (definition of cont and parse ε)

  concat {c i}

  = (definition of concat)

  c i
- cont x c i

  = (definition of cont and parse x)

  concat {c j | j $\in$ {j | x : j $\leftarrow$ i}}

  = (discriminate two cases)

  c j, **if** hd i = x **then** c (tl i) **else** {} **fi**
- cont A c i

  = (definition of cont and parse A)

  concat {c j | A $\rightarrow \alpha$, j $\in$ parse α i}

  = (definition of set construction)

  concat^2 {{c j | j $\in$ parse α i} | A $\rightarrow \alpha$}

  = (identity (2.48))

  concat {concat {c j | j $\in$ parse α i} | A $\rightarrow \alpha$}

  = (definition of cont)

  concat {cont α c i | A $\rightarrow \alpha$}
- cont $\alpha\beta$ c i

  = (definition of cont and parse $\alpha\beta$)

  concat {c k | j $\in$ parse α i, k $\in$ parse β j}

  = (definition of set construction)

  concat^2 {{c k | k $\in$ parse β j} | j $\in$ parse α i}

  = (identity (2.48) and definition of cont)

  concat {cont β c j | j $\in$ parse α i}

  = (definition of cont)

  cont α (cont β c) i

Summarizing this gives a recursive definition of cont that is no longer based on parse :

Theorem 3.56 (CTD-nd)

cont ε c = c
cont x c i = **if** hd i = x **then** c (tl i) **else** {} **fi**
cont A c i = concat {cont α c i | A $\rightarrow \alpha$}
cont $\alpha\beta$ = cont $\alpha \circ$ cont β

□

This parser is called on the start symbol S with:

parse S ω

= (single is the right neutral element of ;)

(parse S ; single) ω

= (definition of cont)

cont S single ω

Because of its high abstraction level and consequently simple and elegant definition, we regard this parser as the most beautiful of all. In a much less abstract fashion it is presented in [Kos74] where it is called a *recursive back-up parser.*

3.3.11 Recursive ascent parsing using continuations

Among other things, the Leermakers recursive ascent parser can be transformed into a form using continuations. We define:

Definition 3.57 (der-Leermakers-cont-P)

$$
\begin{array}{l}
S \;=\; Items \times V_T^* \\
C \;=\; S \mapsto Set(S) \\
\mathsf{c} \;::\; C \\
\mathsf{P}' \;::\; Set(Items) \times C \times V_T^* \mapsto Set(S) \\
\mathsf{Q}' \;::\; V \times V_T^* \mapsto Set(S) \\
\\
\mathsf{P}'\ \mu\ \mathsf{c} \;=\; \mathsf{P}\ \mu\ ;\ \mathsf{c} \\
\mathsf{Q}'\ (\mathsf{X},\mathsf{j}) \;=\; (\mathsf{Q}\ \ ;\ \mathsf{c})\ (\mathsf{X},\mathsf{j})
\end{array}
$$

□

Substituting the definition of P and Q from (3.55) gives:

Derivation 3.58

$$
\begin{array}{l}
\mathsf{P}'\ \mu\ \mathsf{c}\ \mathsf{i} \\
\quad = \text{(definition of } \mathsf{P}' \text{ and } ;) \\
(\mathsf{concat} \circ \mathsf{map}\ \mathsf{c})\ (\mathsf{P}\ \mu\ \mathsf{i}) \\
\quad = \text{(substitute P)} \\
(\mathsf{concat} \circ \mathsf{map}\ \mathsf{c})\ (\{(\mathsf{A} \rightarrow \alpha.,\mathsf{i})\ |\ (\mathsf{A} \rightarrow \alpha.) \in \mu\} +\!\!+ \\
\qquad\qquad\qquad \mathsf{concat}\ \{\mathsf{Q}\ (\mathsf{A},\mathsf{i})\ |\ (\mathsf{A} \rightarrow .) \in \overline{\mu}\} +\!\!+ \\
\qquad\qquad\qquad \mathsf{concat}\ \{\mathsf{Q}\ (\mathsf{x},\mathsf{j1})\ |\ \mathsf{x} : \mathsf{j} \leftarrow \mathsf{i},\ (\mathsf{x} \rightarrow .\mathsf{x}) \in \overline{\mu}\}) \\
\quad = \text{(map distributes over } +\!\!+) \\
\mathsf{concat}\ \{\mathsf{c}\ (\mathsf{A} \rightarrow \alpha.,\mathsf{i})\ |\ (\mathsf{A} \rightarrow \alpha.) \in \mu\} +\!\!+ \\
(\mathsf{concat} \circ \mathsf{map}\ \mathsf{c} \circ \mathsf{concat})\ \{\mathsf{Q}\ (\mathsf{A},\mathsf{i})\ |\ (\mathsf{A} \rightarrow .) \in \overline{\mu}\} +\!\!+ \\
(\mathsf{concat} \circ \mathsf{map}\ \mathsf{c} \circ \mathsf{concat})\ \{\mathsf{Q}\ (\mathsf{x},\mathsf{j})\ |\ \mathsf{x} : \mathsf{j} \leftarrow \mathsf{i},\ (\mathsf{x} \rightarrow .\mathsf{x}) \in \overline{\mu}\} \\
\quad = \text{(apply (2.47), } \mathsf{f} = \mathsf{concat} \circ \mathsf{map}\ \mathsf{c}) \\
\mathsf{concat}\ \{\mathsf{c}\ (\mathsf{A} \rightarrow \alpha.,\mathsf{i})\ |\ (\mathsf{A} \rightarrow \alpha.) \in \mu\} +\!\!+ \\
(\mathsf{concat} \circ \mathsf{map}\ (\mathsf{concat} \circ \mathsf{map}\ \mathsf{c}))\ \{\mathsf{Q}\ (\mathsf{A},\mathsf{i})\ |\ (\mathsf{A} \rightarrow .) \in \overline{\mu}\} +\!\!+ \\
(\mathsf{concat} \circ \mathsf{map}\ (\mathsf{concat} \circ \mathsf{map}\ \mathsf{c}))\ \{\mathsf{Q}\ (\mathsf{x},\mathsf{j})\ |\ \mathsf{x} : \mathsf{j} \leftarrow \mathsf{i},\ (\mathsf{x} \rightarrow .\mathsf{x}) \in \overline{\mu}\} \\
\quad = \text{(identity (2.40}a\text{))} \\
\mathsf{concat}\ \{\mathsf{c}\ (\mathsf{A} \rightarrow \alpha.,\mathsf{i})\ |\ (\mathsf{A} \rightarrow \alpha.) \in \mu\} +\!\!+ \\
\mathsf{concat}\ \{(\mathsf{concat} \circ \mathsf{map}\ \mathsf{c})\ \mathsf{Q}\ (\mathsf{A},\mathsf{i})\ |\ (\mathsf{A} \rightarrow .) \in \overline{\mu}\} +\!\!+ \\
\mathsf{concat}\ \{(\mathsf{concat} \circ \mathsf{map}\ \mathsf{c})\ \mathsf{Q}\ (\mathsf{x},\mathsf{j})\ |\ \mathsf{x} : \mathsf{j} \leftarrow \mathsf{i},\ (\mathsf{x} \rightarrow .\mathsf{x}) \in \overline{\mu}\}
\end{array}
$$

```
      = (definition of Q′ and ' ; ')
concat {c (A → α., i)   | (A → α.) ∈ μ} ++
concat {Q′ (A, i)       | (A → .) ∈ μ̄} ++
concat {Q′ (x, j)       | x : j ← i, (x → .x) ∈ μ̄}
```
□

Similarly, we work out Q′:

Derivation 3.59

```
where
Q′ (X, j)
      = (definition of Q′ and Q)
(concat ∘ map c) ({(A → α.Xβ, k) | (A → αX.β, k) ∈ P (goto μ X) j, α ≠ ε} ++
                  concat {Q A k | (A → X.β, k) ∈ P (goto μ X) j})
      = (As above)
concat {c (A → α.Xβ, k) | (A → αX.β, k) ∈ P (goto μ X) j, α ≠ ε} ++
concat {Q′ (A, k) | (A → X.β, k) ∈ P (goto μ X) j}
      = (Introduce d)
concat {d (A → αX.β, k) | (A → αX.β, k) ∈ P (goto μ X) j}
where d (A → αX.β, k)  =  if α ≠ ε then c (A → α.Xβ, k) else Q′ (A, k) fi
      = (identity (2.40a))
(concat ∘ map d) (P (goto μ X) j)
where d (A → αX.β, k)  =  if α ≠ ε then c (A → α.Xβ, k) else Q′ (A, k) fi
      = (definition of P′)
P′ (goto μ X) d j
where d (A → αX.β, k)  =  if α ≠ ε then c (A → α.Xβ, k) else Q′ (A, k) fi
```
□

Summarizing the result of these transformations gives:

Definition 3.60 (CIDRA2-nd)

```
P′ μ c i  =  concat  {c (A → α., i)   | (A → α.) ∈ μ} ++
             concat  {Q′ (A, i)       | (A → .) ∈ μ̄} ++
             concat  {Q′ (x, j)       | x : j ← i, (x → .x) ∈ μ̄}
where
Q′ (X, j)  =  P′ (goto μ X) d j
where d (A → αX.β, k) = if α ≠ ε then c (A → α.Xβ, k) else Q′ (A, k) fi
```
□

The parser is called on the start symbol by:

P $\{S' \rightarrow \alpha.S\}$ ω
= (P $\{S' \rightarrow \alpha.S\}$; single) ω
= P′ $\{S' \rightarrow \alpha.S\}$ single ω

The result is an algorithm exhibiting the elegance of continuations in presenting and implementing back-tracking.

3.4 Termination of non-deterministic parsers

A functional program can be non-terminating only when it contains a recursive function that can call itself (possibly indirectly) with either the same arguments or with an infinite number of different arguments. In our case, assuming finite domains, only the first can be the case. In this section, we investigate the conditions for non-termination for the simple and doubly recursive descent as well as for the RA2 algorithms. Having derived these conditions, they are applied to the simple recursive descent and Leermakers parser respectively.

3.4.1 Termination of recursive descent algorithms

We recall the definition of a recursive descent algorithm:

$$\mathsf{p}\ \alpha\ =\ \mathsf{t}\ \alpha \mathbin{+\!\!+} \mathsf{concat}\ \{\mathsf{h}\ \alpha\ \beta\ \mathsf{x} \mid \beta \in \mathsf{sub}\ \alpha,\ \mathsf{x}\ \in \mathsf{p}\ \beta\}$$

Assuming that h always terminates, this algorithm is non-terminating if p α directly or indirectly calls p α again, that is, if α is a direct or indirect sub-problem of itself. Let the relation $\Rightarrow$ over T denote this sub-problem relation, that is:

$$\alpha \Rightarrow \beta \equiv \beta \in \mathsf{sub}\ \alpha$$

Then the algorithm is cyclic if $\exists\alpha : \alpha \overset{+}{\Rightarrow} \alpha$.

3.4.2 Termination of doubly recursive descent algorithms

We recall the definition of a doubly recursive descent algorithm:

$$\mathsf{p}\ \alpha\ =\ \mathsf{t}\ \alpha \mathbin{+\!\!+} \mathsf{concat}\ \{\mathsf{g}_2\ \alpha\ \beta\ \mathsf{x}\ \mathsf{y}\ \mid \beta \in \mathsf{sub}\ \alpha,\ \mathsf{x}\ \in \mathsf{p}\ \beta,\ \mathsf{y} \in (\mathsf{g}_1\ \alpha\ \beta\ ;\ \mathsf{p})\ \mathsf{x}\}$$

Again, this algorithm is non-terminating if α is a direct or indirect sub-problem of itself, assuming that both g_1 and g_2 are both terminating. The sub-problem relation is a bit more complicated than in the single recursive case:

$$\alpha \Rightarrow \beta\ \equiv\ \exists\gamma,\mathsf{x} : \gamma \in \mathsf{sub}\ \alpha,\ \mathsf{x} \in \mathsf{p}\ \gamma,\ \beta \in \mathsf{g}_1\ \alpha\ \gamma\ \mathsf{x}$$

Then the algorithm is cyclic if $\exists\alpha : \alpha \overset{+}{\Rightarrow} \alpha$.

Termination of a doubly recursive descent parser

The simple doubly recursive descent parser was defined by (definition (3.8), choosing $\#\alpha = 1$ in the last rule):

$$\begin{array}{ll}
\mathsf{parse}\ \varepsilon\ \mathsf{i} & = \{\mathsf{i}\} \\
\mathsf{parse}\ \mathsf{x}\ \mathsf{i} & = \{\mathsf{i} \mid \mathsf{x} : \mathsf{j} \leftarrow \mathsf{i}\} \\
\mathsf{parse}\ \mathsf{A}\ \mathsf{i} & = \{\mathsf{j} \mid \mathsf{A} \rightarrow \alpha,\ \mathsf{j} \in \mathsf{parse}\ \alpha\ \mathsf{i}\} \\
\mathsf{parse}\ \mathsf{X}\beta\ \mathsf{i} & = \{\mathsf{k} \mid \mathsf{j} \in \mathsf{parse}\ \mathsf{X}\ \mathsf{i},\ \mathsf{k} \in \mathsf{parse}\ \beta\ \mathsf{j}\},\ \text{where } \beta \neq \varepsilon
\end{array}$$

Strictly speaking, this parser is not a doubly recursive descent algorithm, but can easily be adjusted by adding a parse ε j qualifier to the third rule. In this case, the sub-problem relation $\Rightarrow$ is defined as the smallest relation over V^* satisfying:

$$\begin{array}{ll} A \Rightarrow \alpha, & \text{if } A \rightarrow \alpha \\ X\beta \Rightarrow X, & \text{if } \beta \neq \varepsilon \\ A\beta \Rightarrow \beta, & \text{if } \beta \neq \varepsilon \wedge A \overset{*}{\rightarrow} \varepsilon \end{array}$$

Cyclicity is obtained if $\exists A : A \overset{+}{\Rightarrow} A$, that is, if $A \overset{+}{\rightarrow} A\alpha$, thus for left recursive grammars. This is easily proved since $A \overset{+}{\Rightarrow} A$ implies $A \Rightarrow \beta_1 B \beta_2 \overset{*}{\Rightarrow} B \Rightarrow \gamma_1 C \gamma_2 \overset{*}{\Rightarrow} \ldots \overset{*}{\Rightarrow} A$. In this case, a derivation $A \rightarrow \beta_1 B \beta_2 \overset{*}{\rightarrow} B\beta \rightarrow \gamma_1 C \gamma_2 \beta \rightarrow \ldots \overset{*}{\rightarrow} A\alpha$ also exists.

3.4.3 Termination of the DRA2 algorithm

We recapitulate the DRA2 recursive ascent algorithm.

```
DRA2 t sub g₁ g₂ = P
where
P μ = {(α, x) | α ∈ μ, x ∈ t α} ++
        concat {Q α x | α ∈ μ̄⁺, x ∈ t α}
where
Q β x = {(γ, y) | (γ, y) ∈ next, γ ∈ μ} ++
         concat {Q γ y | (γ, y) ∈ next, γ ∈ μ̄⁺}
         where goto = concat {g₁ γ β x | γ ∈ μ̄, β ∈ sub γ}
               next = {(γ, z) | (δ, y) ∈ P goto, γ ∈ combine β δ, z ∈ g₂ γ β x y}
               combine α β = {γ | γ ∈ sup μ α, β = g₁ γ α x}
```

The DRA2 algorithm becomes left recursive if either Q or P is cyclic.
We define the relations $\Rightarrow_Q$ and $\Rightarrow_P$ by:

$$\begin{array}{ll} (\beta, x) \Rightarrow_Q (\gamma, y) & \equiv \beta \in \text{sub } \gamma \wedge y \in (g_1\ \gamma\ \beta\ ;\ p\ ;\ g_2\ \alpha\ \beta)\ x \\ \mu \Rightarrow_P \eta & \equiv \exists \alpha \in \bar{\mu}^+, x \in p\ \alpha : \eta = \text{concat } \{g_1\ \gamma\ \alpha\ x \mid \gamma \in \text{sup } \mu\ \alpha\} \end{array}$$

Q is left recursive if:

$$\exists \beta, y : (\beta, y) \overset{+}{\Rightarrow}_Q (\beta, y)$$

P is left recursive if:

$$\exists \mu : \mu \overset{+}{\Rightarrow}_P \mu$$

Termination of the Leermakers parser

Using the general cyclicity conditions for the DRA2 scheme, the cyclicity conditions for the Leermakers parser can be determined.
The parser becomes cyclic with regard to Q if there exists a chain

$$((\mathsf{B} \to .\mathsf{C}\beta, \mathsf{i}), \mathsf{j}) \Rightarrow_{\mathsf{Q}} ((\mathsf{C} \to .\mathsf{D}\gamma, \mathsf{i}), \mathsf{j}) \Rightarrow_{\mathsf{Q}} \ldots \Rightarrow_{\mathsf{Q}} ((\mathsf{B} \to .\mathsf{C}\beta, \mathsf{i}), \mathsf{j})$$

In this relation, all items share the same i because the sub-problem definition implies this, and share the same j, since by cyclicity no progress is made. From this follows (since no progress is made by Q) that $\beta \overset{*}{\to} \varepsilon, \gamma \overset{*}{\to} \varepsilon, \ldots$, and hence $\mathsf{B} \to \mathsf{C} \to \mathsf{D} \to \ldots \to \mathsf{B}$, thus $\mathsf{B} \overset{+}{\to} \mathsf{B}$. That is, it is cyclic for cyclic grammars only, and thus this form of cyclicity will not give many problems in practice.
The other form of cyclicity, P being left recursive, is more problematic. The $\Rightarrow_{\mathsf{P}}$ relation becomes:

$$\begin{aligned}&(\mu, \mathsf{i}) \Rightarrow_{\mathsf{P}} (\eta, \mathsf{j}) \equiv \\ &\qquad \eta \;=\; \{\mathsf{A} \to \alpha\mathsf{X}.\beta \mid \mathsf{A} \to \alpha.\mathsf{X}\beta \in \bar{\mu},\ \mathsf{X} \overset{*}{\to} \mathsf{i}/\mathsf{j}\}\end{aligned}$$

Cyclicity can only occur when i = j and thus when $\mathsf{X} \overset{*}{\to} \varepsilon$. The corresponding relation $\Rightarrow_\varepsilon$ is defined by:

$$\begin{aligned}&\mu \Rightarrow_\varepsilon \eta \equiv \\ &\qquad \eta \;=\; \{\mathsf{A} \to \alpha\mathsf{X}.\beta \mid \mathsf{A} \to \alpha.\mathsf{X}\beta \in \bar{\mu},\ \mathsf{X} \overset{*}{\to} \varepsilon\}\end{aligned}$$

Cyclicity is obtained if $\mu \overset{+}{\Rightarrow}_\varepsilon \mu$ for some μ reachable from the start state. In this case, the grammar contains a symbol A such that $\mathsf{A} \overset{+}{\to} \alpha\mathsf{A}\beta$ and $\alpha \overset{+}{\to} \varepsilon$. The reverse holds too, that is if $\mathsf{A} \overset{+}{\to} \alpha\mathsf{A}\beta$ and $\alpha \overset{+}{\to} \varepsilon$ then the cycle via P occurs. This is the case since $\mathsf{A} \overset{+}{\to} \alpha^{\mathsf{n}}\mathsf{A}\beta^{\mathsf{n}}$, which implies that an arbitrary number of reductions deriving α^{n} can occur before any progress is made. An example grammar in which such cyclicity arises is a grammar with the following production rules:

$$\begin{aligned}&\mathsf{S} \to \mathsf{B}\\ &\mathsf{B} \to \mathsf{AB}\\ &\mathsf{B} \to \mathsf{a}\\ &\mathsf{A} \to \varepsilon\end{aligned}$$

The state $\mu = \{\mathsf{B} \to \mathsf{A}.\mathsf{B}\}$ gives rise to a cycle since P μ i will call itself again after reducing $\mathsf{A} \to \varepsilon$.

3.5 Parsers with look-ahead

In this section we add look-ahead to the simple recursive descent and Leermakers parser. We investigate the conditions under which the resulting parser is deterministic. To guarantee that we always have a head of the input string to inspect, we add the end-of-file marker $\perp$ to the input string. We will never consume $\perp$.

3.5.1 LL(1) recursive descent

We define the set of first symbols with regard to an element α of V^* as the smallest set satisfying:

Definition 3.61 (first-symbols)

$$\begin{aligned}\text{first } \varepsilon &= \{\}\\ \text{first } x\beta &= \{x\}\\ \text{first } A\beta &= \text{concat } \{\text{first } \alpha \mid A \to \alpha\} +\!\!+\\ &\quad\ \text{concat } \{\text{first } \beta \mid A \overset{*}{\to} \varepsilon\}\end{aligned}$$

□

We define the set of follower symbols with regard to an element α of V^* and a set of terminal symbols as:

Definition 3.62 (foll)

$$\text{foll } \alpha\ f = \text{first } \alpha +\!\!+ \text{concat } \{f \mid \alpha \overset{*}{\to} \varepsilon\}$$

□

We define the set of follower symbols of a non-terminal as the smallest set satisfying:

$$B \to \alpha A \beta \Rightarrow \text{foll } \beta\ (\text{foll } B) \subseteq \text{foll } A$$

If A is a non-terminal such that $A \overset{*}{\not\to} \varepsilon$, we assume that $A \to \alpha_1, \ldots, A \to \alpha_k$ are the production rules for A. If $A \overset{*}{\to} \varepsilon$, we assume that $A \to \alpha_1, \ldots, A \to \alpha_k, A \to \alpha_\varepsilon$ are the production rules for A, such that only the last of them derives ε. If more than one production rule for A derives ε, then the grammar is ambiguous.
A recursive descent parser with look-ahead can be defined as follows, assuming that a case expression makes a non-deterministic choice among the open guards:

Definition 3.63 (LL1)

```
parse ε i  =  i
parse x i  =  if hd i = x then (tl i) else error "x expected" i fi
```

If $A \overset{*}{\not\to} \varepsilon$ then:

```
parse A i  =  case x ∈ first α₁    → parse α₁ i
                     ∈ first α₂    → parse α₂ i
                     ...           → ...
                     ∈ first αₖ    → parse αₖ i
                     otherwise     → error "A expected" i
              esac
where x : j ← i
```

If $A \overset{*}{\to} \varepsilon$ then:

```
parse A i = case x ∈ first $\alpha_1$          → parse $\alpha_1$ i
                   ∈ first $\alpha_2$          → parse $\alpha_2$ i
                   ...                 → ...
                   ∈ first $\alpha_k$          → parse $\alpha_k$ i
                   ∈ foll A ++ first $\alpha_\varepsilon$ → parse $\alpha_\varepsilon$ i
                   otherwise           → error "A expected" i
            esac
where x : j ← i
```

Finally:

$$\text{parse } A\alpha \ = \ \text{parse } \alpha \circ \text{parse } A$$

□

(Observe that restricting the result of parse to single elements instead of sets gives a simple definition of parse $A\alpha$.)
When first α_i and first $\alpha_j, i \neq j$ are disjoint and first α_i and first α_ε are disjoint and moreover first α_i and foll A are disjoint whenever $A \overset{*}{\rightarrow} \varepsilon$, this parser is deterministic, since the guards exclude each other and the case expression becomes deterministic.
When, moreover, $A \overset{+}{\not\rightarrow} A\alpha$ for each nonterminal A, the parser is also terminating. These restrictions are exactly the LL(1) restrictions and thus the parser is deterministic for LL(1) grammars.

3.5.2 LALR(1) recursive ascent

Look-ahead can easily be added to the Leermakers parser by restricting the choices for P μ i by adding conditions on x. To this end, we first define the LALR(1) follower set of an item J with regard to a state μ, assuming $J \in \overline{\mu}$, as the smallest set satisfying:

Definition 3.64 (foll-item)

$$\begin{array}{ll} \text{foll } \mu \ (A \rightarrow \alpha.X\beta) & \subseteq \text{foll } (\text{goto } \mu \ X) \ (A \rightarrow \alpha X.\beta) \\ A \rightarrow \alpha.X\beta \in \overline{\mu}, X \rightarrow \gamma & \Rightarrow \ \text{foll } \beta \ (\text{foll } \mu \ (A \rightarrow \alpha.X\beta)) \subseteq \text{foll } \mu \ (X \rightarrow .\gamma) \end{array}$$

□

When $J \notin \overline{\mu}$, we define foll μ J = {}.
The notion of the set of first symbols of a state μ is defined by:

Definition 3.65 (first-state)

$$\text{first } \mu = \{x \mid x \rightarrow .x \in \overline{\mu}\}$$

□

The non-deterministic parser with LALR(1) look-ahead is defined by:

Definition 3.66 (LALR1-nd)

```
P $\mu$ i = $\{(A \to \alpha., i) \mid (A \to \alpha.) \in \mu, x \in$ foll $\mu$ $(A \to \alpha.)\}$ ++
        concat $\{Q\ A\ i \mid (A \to .) \in \overline{\mu}, x \in$ foll $\mu$ $(A \to .)\}$ ++
        concat $\{Q\ x\ j \mid (x \to .x) \in \overline{\mu}\}$
where
x : j $\leftarrow$ i
Q X j = $\{(A \to \alpha.X\beta, k) \mid (A \to \alpha X.\beta, k) \in$ next, $\alpha \neq \varepsilon\}$ ++
        concat $\{Q\ A\ k \mid (A \to X.\beta, k) \in$ next$\}$
   where next = P (goto $\mu$ X) j
```
□

Assuming that the parser is made deterministic in this way, the number of results of P and Q amounts to at most one. Adding a very simple error recovery yields a version that delivers exactly one result:

Definition 3.67 (LALR1)

```
P $\mu$ i = case x $\in$ foll $\mu$ $(A \to \alpha X.)$   $\to (A \to \alpha X., i)$
                $\in$ foll $\mu$ $(A \to .)$     $\to$ Q A i
                $\in$ first $\mu$             $\to$ Q x j
                otherwise           $\to$ error "$\beta$ expected" $(A \to \alpha.\beta, i)$
                                      , with $A \to \alpha.\beta \in \mu$
       esac
where
x : j = i
Q X j = if $\alpha \neq \varepsilon$
        then $(A \to \alpha.X\beta, k)$
        else Q A k
        fi
        where $(A \to \alpha X.\beta, k)$ = P (goto $\mu$ X) j
```
□

The last definition in the where-part simultaneously defines A, α, β and k. This form of definition may be surprising for readers with no experience in the use of modern functional languages.
The error recovery by the insertion of β in the case of an error may lead to problems which are discussed in section 3.6.2. The parser is deterministic if the test sets in the case expression for P are mutually disjoint, that is, for LALR(1) grammars. Notice that the LALR(1) parser is obtained by adding look-ahead *after* the DRA2 transformation.

3.5.3 LR(1) recursive ascent

The Leermakers parser can be extended to a non-deterministic LR(1) parser by adding follower sets to the item-based top-down parser *before* transformation to recursive ascent. The recursive descent algorithm is:

Definition 3.68 (ITDF-nd)

$$\begin{aligned}
&\mathsf{parse}\ (A \to \alpha.)\ f\ i &&= \{i \mid x : j \leftarrow i,\ x \in f\}\\
&\mathsf{parse}\ (x \to .x)\ f\ i &&= \{j \mid x : j \leftarrow i\}\\
&\mathsf{parse}\ (A \to \alpha.X\beta)\ f\ i &&= \{k \mid (X \to .\gamma) \in \mathit{Items},\ j \in \mathsf{parse}\ (X \to .\gamma)\ (\mathsf{foll}\ \beta\ f)\ i,\\
& && \qquad k \in \mathsf{parse}\ (A \to \alpha X.\beta)\ f\ j\}
\end{aligned}$$

□

It is called for the start symbol S by:

$$\mathsf{parse}\ (S' \to \alpha.S)\ \{\bot\}\ (\omega\bot)$$

where $\alpha \neq \varepsilon$ and S' is a new non-terminal symbol. We can rewrite this definition into the recursive descent form:

$$\begin{aligned}
&t\ (A \to \alpha., f, i) &&= \{i \mid x : j \leftarrow i,\ x \in f\}\\
&t\ (x \to .x, f, i) &&= \{j \mid x : j \leftarrow i\}\\
&t\ (A \to \alpha.X\beta, f, i) &&= \{\}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{sub}\ (A \to \alpha.)\ f\ i &&= \{\}\\
&\mathsf{sub}\ (x \to .x)\ f\ i &&= \{\}\\
&\mathsf{sub}\ (A \to \alpha.X\beta)\ f\ i &&= \{(X \to .\gamma, \mathsf{foll}\ \beta\ f, i) \mid (X \to .\gamma) \in \mathit{Items}\}
\end{aligned}$$

$$h\ (A \to \alpha.X\beta, f, i)\ (X \to .\gamma, i)\ j\ =\ \mathsf{parse}\ (A \to \alpha X.\beta)\ f\ j$$

Applying scheme DRA1 gives a version of the Leermakers parser with look-ahead:

Definition 3.69 (LR1-nd)

$$\begin{aligned}
&P\ \mu\ i\ =\ \mathsf{concat}\ \{Q\ (A \to \alpha.)\ i \mid (A \to \alpha., f) \in \overline{\mu},\ x \in f\}\ +\!\!+\\
&\qquad\quad \mathsf{concat}\ \{Q\ (x \to .x)\ j \mid (x \to .x, f) \in \overline{\mu}\}\\
&\mathbf{where}\\
&x : j\ =\ i\\
&Q\ (X \to \gamma.\delta)\ j = \{(X \to \gamma.\delta, j) \mid \gamma \neq \varepsilon\}\ +\!\!+\\
&\qquad\qquad\qquad \mathsf{concat}\ \{Q\ (A \to \alpha.X\beta)\ k \mid (A \to \alpha X.\beta, k) \in \mathsf{next}, \gamma = \varepsilon\}\\
&\qquad\qquad\qquad \mathbf{where}\ \mathsf{next}\ =\ P\ (\mathsf{goto}\ \mu\ X)\ j
\end{aligned}$$

□

With goto defined by:

Definition 3.70 (goto-2)

$$\mathsf{goto}\ \mu\ X\ =\ \{(A \to \alpha X.\beta, f) \mid (A \to \alpha.X\beta, f) \in \mu\}$$

□

Depending on the definition of state $\overline{\mu}$, this is either a LR(1) parser or not. When we define $\overline{\mu}$ in the straightforward way it is not, since equal items may be equipped with different follower sets in the same state. In a LR(1) parser each item appears only once in a state and the follower sets are unified for each item. This is allowed since it can be proven (by induction over *Items*) that $\mathsf{parse}\ J\ f\ i\ +\!\!+\ \mathsf{parse}\ J\ g\ i = \mathsf{parse}\ J\ (f +\!\!+ g)\ i$ and since all problems in $\overline{\mu}$ share the same string i, and i can hence be separated from the state. The resulting $\overline{\mu}$ is the minimal set obeying:

$$\begin{aligned}\bar{\mu} &= \mu \mathbin{+\!\!+} \mathsf{concat}\ \{(X \to .\gamma, f) \mid (A \to \alpha.X\beta, f') \in \bar{\mu}, X \to .\gamma \in \mathit{Items},\\ &\qquad\qquad f = \mathsf{concat}\ \{\mathsf{foll}\ \beta'\ f'' \mid (A' \to \alpha'.X\beta', f'') \in \bar{\mu}\}\}\end{aligned}$$

Whenever the resulting parser is deterministic we call the grammar LR(1). In this case, a single result can be obtained instead of a set of results. We assume that the set of completed items in $\bar{\mu}$ is $\{(A_1 \to \alpha_1., f_1), \ldots, (A_n \to \alpha_n., f_n)\}$ and define first as $\{x \mid (x \to .x, f) \in \bar{\mu}\}$. A deterministic version of the Leermakers parser thus becomes:

Definition 3.71 (LR1)

```
P μ i =
case x ∈ f₁          → Q (A₁ → α₁.) i
       ...           → ...
     ∈ fₙ            → Q (Aₙ → αₙ.) i
     ∈ first         → Q (x → .x) j
     otherwise       → error "β expected" (A → α.β, i), where (A → α.β, f) ∈ μ
esac
where
x : j ← i
Q (X → γ.δ) j = if γ ≠ ε
                then (X → γ.δ, j)
                else Q (A → α.Xβ) k
                fi
                where (A → αX.β, k) = P (goto μ X) j
```

□

3.5.4 The Kruseman recursive ascent parser

The deterministic LR(1) parser that was derived in the previous section can easily be expressed in an imperative language as well. Both P and Q can be merged into a single function since the tail-recursiveness of Q can be transformed into a loop. The string i can become a global variable that is modified (shortened) in the case of shift and can hence be removed as an argument as well as from the result. The remaining result is an item $A \to \alpha.\beta$. From this item, only A and the emptiness of α are used (by Q). Thus we can modify the result by returning A and $\#\alpha$ only, again in two global variables Symbol and pop respectively. The resulting function for a state μ then becomes:

Definition 3.72 (LR1-imp)

```
P_μ =
case x ∈ f₁          → symbol := A₁; pop := #α₁; Q
       ...           → ...
     ∈ fₙ            → symbol := Aₙ; pop := #αₙ; Q
     ∈ first         → symbol := x; pop := 0; i := j; Q
     otherwise       → print "β expected"; symbol := A; pop := #α; Q
                       , where A → α.β ∈ μ
esac
```

```
where $x : j \leftarrow i$
        $Q$ = if $pop \neq 0$ then return fi
              case symbol of $B_1$   $\rightarrow P_{goto\ \mu\ B_1}$
                              ...
                             $B_n$   $\rightarrow P_{goto\ \mu\ B_n}$
              esac
              $pop := pop - 1; Q;$
```
□

When the tail-recursive local function Q is transformed into a while-loop within the body of P, this is exactly the recursive ascent parser of Kruseman [Kru88].

3.6 Error recovery in deterministic parsers

In this section we extend the error recovery already introduced in the previous section and show that this error recovery does not introduce non-termination. The purpose of an error recovery strategy is to find as many syntax errors in the input text as possible. Two basic mechanisms are available: deletion of unexpected terminal symbols and insertion of symbols that are expected but not encountered. It is the art of a good error recovery strategy to apply these two mechanisms in such a way that deletion of symbols does not extend beyond a point where the parse may be continued. We make use of an extra parameter, a set of terminal symbols, that represents the set of 'useful' symbols, i.e. symbols for which a correct parse may be resumed.

3.6.1 LL(1) recursive descent

We can add deletion and synchronization to the deterministic LL(1) recursive descent parser. We must synchronize before making a choice between production rules and before recognizing a terminal symbol. A follower set is passed as an extra parameter to prevent the deletion of symbols that can be recognized by the continuation of earlier calls.

Definition 3.73 (LL1-E)

```
parse $\varepsilon$ f i = i
parse x f i = if hd $i'$ = x then tl $i'$ else error "x expected" $i'$ fi
              where $i'$ = delete (f ++ {x}) i
parse A f i = case x $\in$ first $\alpha_1$    $\rightarrow$ parse $\alpha_1$ f $i'$
                     ...              $\rightarrow$ ...
                     $\in$ first $\alpha_k$    $\rightarrow$ parse $\alpha_k$ f $i'$
                     $A \overset{*}{\rightarrow} \varepsilon$         $\rightarrow$ parse $\alpha_\varepsilon$ f $i'$
                     otherwise     $\rightarrow$ error "A expected" $i'$
              esac
              where $i'$ = delete (f ++ first A) i
                    x = hd $i'$
```

$$\begin{array}{lcl} \mathsf{parse}\ (A\alpha)\ f\ i & = & \mathsf{parse}\ \alpha\ f\ (\mathsf{parse}\ A\ f'\ i) \\ & & \textbf{where}\ f' = f \mathbin{+\!\!+} \mathsf{concat}\ \{\mathsf{first}\ X \mid X \in \alpha\} \end{array}$$

□

With delete defined by:

Definition 3.74 (delete)

$$\begin{array}{lcl} \mathsf{delete}\ f\ \varepsilon & = & \varepsilon \\ \mathsf{delete}\ f\ x:j & = & \textbf{if}\ x \in f\ \textbf{then}\ x:j\ \textbf{else}\ \mathsf{delete}\ f\ j\ \textbf{fi} \end{array}$$

□

The parser is entered by a call parse $S\ \{\perp\}\ (\omega\perp)$, where $\perp$ is an end-of-file marker added to the end of the input string.
The insertion of symbols does not introduce non-termination.
Proof

Assume that the following production rules are given

$$\begin{array}{l} S' \to S\perp \\ S \to \alpha_1 A_1 \beta_1 \\ A_{i-1} \to \alpha_i A_i \beta_i \end{array}$$

These rules may give rise to a sequence of recursive calls of parse with subsequent arguments:

$$\begin{array}{l} S' \\ S\perp \\ \alpha_1 A_1 \beta_1 \\ \ldots \\ \alpha_n A_n \beta_n \end{array}$$

Assume without loss of generality that symbol A_n is inserted by a call parse A_n f i, which means that the latter has returned with an error "A_n expected" i expression. We show that necessarily progress is made.

Define

$$\begin{array}{l} \beta_0 = \{\perp\} \\ i' = \mathsf{delete}\ (f \mathbin{+\!\!+} \mathsf{first}\ A_n)\ i \end{array}$$

By induction to n we have

$$f = \mathsf{concat}\ \{\mathsf{first}\ X \mid 0 \le j \le n, X \in \beta_j\}$$

Since A_n was inserted, we have $x \notin \mathsf{first}\ A_n, A_n \overset{*}{\not\to} \varepsilon$ and hence
$\exists k : 0 \le k \le n : x \in \mathsf{concat}\ \{\mathsf{first}\ X \mid X \in \beta_k\}$.

Choose this k maximal. Then $x \notin \mathsf{concat}\ \{\mathsf{first}\ X \mid k < j \le n,\ X \in \beta_j\}$. By induction (to k) it follows that $\beta_n \ldots \beta_{k-1}$ is inserted and that subsequently parse $\beta_k\ i'$ is called. Since $x \in \mathsf{concat}\ \{\mathsf{first}\ X \mid X \in \beta_k\}$, it follows (induction to $\#\beta_k$) that x is recognized and hence progress is made.

□

3.6.2 LR(1) and LALR(1) recursive ascent

In this section we show how a solution that is to a certain extent similar to the LL(1) case with error recovery can be designed for LR(1) and LALR(1). In this design we face a complication. In the LL(1) case, when a symbol is inserted, control is returned to a previous call, thereby synchronizing on the follower set f. The choice of where parsing is resumed is left completely to this previous call.
The problem with bottom-up parsing is that the callee, upon return, chooses an item that is subsequently used by the caller in deciding which state to enter next. But in the case of an error, the callee might not have enough information to decide which item to choose. In this case, it does not even know which is the best callee to return to, a decision that is normally prescribed by the length of α in the return item.
The solution to the problem lies in preventing a return to a callee at all, which can be achieved by using a continuation-based parser. In such a parser, an item that has been recognized is not returned, but passed as an argument to a continuation function. In the case of an error, when no item is recognized, a special *error continuation* is called. Such an error continuation is capable of continuing parsing with a next terminal symbol in a given follower set, just as in the LL(1) case. We only need to design the (error-continuation, follower-set) pair in a consistent way. We use the Leermakers parser with continuations from section 3.3.11 as our starting point. We assume that the grammar is such that no cycles occur in the case without error recovery and we prove that our error recovery is also cycle free, with this assumption.
We define the *directors* of a state as the following set:

Definition 3.75 (def:dir-state)

$$\mathsf{dir}\ \mu \ = \ \mathsf{concat}\ \{\mathsf{first}\ \beta \mid \mathsf{A} \rightarrow \alpha.\beta \in \overline{\mu}\}$$

□

This is the set of symbols that are 'of use' for $\mathsf{P}\ \mu$. Observe that we did not include the followers of completed items $(\mathsf{A} \rightarrow \alpha.)$ in this set. This is because, in the LR(1) case, it can be proven that this set is contained in the follower set F that is passed as an extra argument to P, united with the directors. In the LALR(1) case, with its more clumsy error detection than in the LR(1) case, the set F can be used to restrict dynamically the follower sets, which are too large, to those symbols that are of real use to the caller to which a reduction is made.
By using the proper definition of foll we can distinguish between the LR(1) and the LALR(1) case. The LALR(1) definition of foll has been given above. In the LR(1) case, elements of μ are tuples $(\mathsf{A} \rightarrow \alpha.\mathsf{X}\beta, \mathsf{f})$. By defining $\mathsf{foll}\ \mu\ \mathsf{J} = \mathsf{concat}\ \{\mathsf{f} \mid (\mathsf{J}, \mathsf{f}) \in \overline{\mu}\}$ we can define $\overline{\mu}' = \{\mathsf{J} \mid (\mathsf{J}, \mathsf{f}) \in \overline{\mu}\}$ and use the simplified μ' instead of μ. The expression $\mathsf{foll}\ \mu'\ \mathsf{J}$ can still be used to denote the LR(1) follower set of J in μ. Observe that we can do this since an LR(1) parser groups all follower sets for the same item together. The advatage of this

modification is that the LR(1) and LALR(1) parsing algorithms become identical. The only difference lies in the definition of foll.

We now present the LR(1) or LALR(1) parser with error recovery by subsequently presenting and discussing different parts of it. After that, we recapitulate the complete function. The function P operates on a state μ, a set of synchronization symbols F, a normal and an error continuation c and e and a string i. First it deletes symbols until a suffix string $i' = x : i''$ is obtained such that either $x \in F$ or $x \in \text{dir } \mu$. In the latter case, Q can be called and in the former case, x is either a symbol for which a reduction can take place, i.e. $\exists A \rightarrow \alpha. \in \bar{\mu} : x \in \text{foll } \mu\ (A \rightarrow \alpha.)$, or x is a symbol for which the error continuation can make progress.

```
P μ c e F i = case x ∈ foll μ (A → αX.) → c (A → αX.) i'
                   ∈ foll μ (A → .)    → Q A i'
                   ∈ first μ           → Q x i''
                   otherwise           → e i'
              esac
  where
  i' = delete (F ++ dir μ) i
  x : i'' = i'
```

We now turn to the question of how the follower set F should be extended with a set F′ in the case of a recursive call. Given a call Q X j, the parsing can be resumed with any terminal symbol that can follow X in μ, i.e.

$$F' = \text{concat } \{\text{first } \delta \mid \exists\delta' : A \rightarrow \alpha.\gamma\delta \in \bar{\mu},\ \gamma \overset{*}{\Rightarrow}_L X\delta'\}$$

In the case of an error, the sequence δ' is inserted and parsing is resumed in state a with item $A \rightarrow \alpha\gamma.\delta$.

This state is defined by goto μ γ with:

$$\text{goto } \mu\ \varepsilon = \mu$$
$$\text{goto } \mu\ (X\alpha) = \text{goto } (\text{goto } \mu\ X)\ \alpha$$

The easiest way to incorporate this generalization of goto in our algorithm is by generalizing Q as well. Its first argument is changed from a symbol onto a sequence of symbols.

$$Q\ \beta\ j = P\ (\text{goto } \mu\ \beta)\ c'\ e'\ (F \mathbin{+\!\!+} F')\ j$$

The continuation c′ is the straightforward generalization of the one in section 3.3.11.

$$c'\ (A \rightarrow \alpha\beta.\gamma)\ k = \textbf{if } \alpha \neq \varepsilon \textbf{ then } c\ (A \rightarrow \alpha.\beta\gamma)\ k \textbf{ else } Q\ A\ k \textbf{ fi}$$

The error continuation e′ should be consistent with F′. For each element of F′ it should be possible to resume parsing. We use the notation a | q for denoting hd {a | q}.

```
e' (y : l) = if y ∈ F'    → insert δ' (Q γ yl) | A → α.γδ ∈ μ̄, γ ⇒*_L βδ', y ∈ first δ
                otherwise → e (y : l)
             fi
```

$$\text{insert } \delta\ a = \textbf{if } \delta \overset{*}{\rightarrow} \varepsilon \textbf{ then } a \textbf{ else } \text{error "}\delta \text{ expected" } a \textbf{ fi}$$

It simply finds a proper item in $\bar{\mu}$ for which it can continue. Such an item exists when $y \in F'$.
When we put the pieces of our algorithm together, we obtain the following result.

Definition 3.76 (CIDRA2-E)

```
P μ c e F i = case x ∈ foll μ (A → αX.) → c (A → αX.) i'
                     ∈ foll μ (A → .)   → Q A i'
                     ∈ first μ          → Q x i''
                     otherwise          → e i'
              esac
where
i' = delete (F ++ dir μ) i
xi'' = i'
Q β j = P (goto μ β) c' e' (F ++ F') j
where F' = concat {first δ | ∃δ' : A → α.γδ ∈ μ̄, γ ⇒*_L βδ'}
      c' (A → αβ.γ) k = if α ≠ ε then c (A → α.βγ) k else Q A k fi
      e' (y : l) = if y ∈ F'      → insert δ' (Q γ yl)
                                    | A → α.γδ ∈ μ̄, γ ⇒*_L βδ', y ∈ first δ
                      otherwise → e (y : l)
                   fi
```

□

With the functions insert and goto defined by:

Definition 3.77 (insert)

$\text{insert } \delta\ x = \textbf{if } \delta \overset{*}{\rightarrow} \varepsilon \textbf{ then } x \textbf{ else } \text{error "}\delta \text{ expected" } x \textbf{ fi}$

□

Definition 3.78 (goto-3)

$\text{goto } \mu\ \varepsilon = \mu$
$\text{goto } \mu\ (X\alpha) = \text{goto } (\text{goto } \mu\ X)\ \alpha$

□

We now prove the termination of this algorithm.
Proof

Assume that the version without error recovery terminates. Since deletion does not introduce cycles, non-termination can only be caused by insertion.

When an insertion takes place, parsing continues in a state η with the next input symbol $y \in \eta$. This state could also be reached for a correct input, with the same arguments. In that case, no error recovery takes place, and hence the algorithm makes progress by consuming y in the correct case. Since only the first symbol of the input is inspected, this symbol y is also consumed in the case with error recovery and progress is made.

□

3.6.3 Re-entering the parser

The synchronization mechanism implemented by the function delete in the error recovery mechanisms of the deterministic parsers is rather crude: it skips symbols until a proper follower symbol is encountered. This may imply that e.g. in a Pascal parser, upon encountering an error in a statement sequence in a procedure, all statements are skipped until the `end` symbol of the procedure is encountered. This is problematic for two reasons. First, it is a pity that the remaining statements of the procedure are not analyzed. Second, the deletion might hit upon an `end` symbol not corresponding to the end of the procedure, but e.g. to a local compound statement. Parsing resumes with the attempted recognition of more procedure and function declarations, at a point within the procedure body.
Both problems are solved by allowing the deletion mechanism to call the parser recursively whenever possible. E.g. upon encountering an `if` symbol, the deletion function can call the parser recursively for the parsing of an if-statement, or even a statement sequence. The condition for re-entering the parser upon encountering a symbol x is that there exists a 'unique state' of the parser that handles the symbol x. The conditions for the uniqueness of a state are rather subtle, however, as the example concerning the if-statement versus statement sequence indicates. The former choice is not the best one, since the latter (statement sequence) encloses the former (if-statement).
We derive conditions for the LL(1) and LALR(1) deterministic parsers that define the recursive relation between deletion and parsing.

LL(1)

We start with the definition of the set of states (elements of V^*) that accept a given terminal symbol x:

$$\text{states } x = \{\beta \mid A \rightarrow \alpha\beta,\ x \in \text{first } \beta\}$$

Next we define an ordering relation upon the elements of this set:

$$\alpha \geq \beta = \exists\gamma \in V^* : \alpha \overset{*}{\rightarrow} \beta\gamma$$

The unique maxima of states x, if existent, are now the states of interest:

$$\begin{aligned} \text{max\_states } x &= \{\alpha \mid \alpha \in \text{states } x,\ \beta \in \text{states } x,\ \beta \neq \alpha,\ \alpha \geq \beta,\ \beta \not\geq \alpha\} \\ \text{unique } \alpha\ x &= \text{max\_states } x = \{\alpha\} \end{aligned}$$

The deletion function for the LL(1) parser can now be modified as follows:

```
delete f xi = if $x \in f$                          → xi
                 $\exists\alpha \in V^*$ : unique $\alpha$ x → delete f (parse $\alpha$ f xi)
                 otherwise                  → delete f i
              fi
```

LALR(1)

The technique used for the LALR(1) is similar to that for the LL(1) parser, but now states are elements of *Set(Items)*. We have two options for the states for x: either states μ with $x \in$ first μ or the states η with $\exists \mu : \eta =$ goto μ x. We assume that only states reachable from the start states are taken into account in these and the following definitions. When the former definition gives a unique state for a certain symbol x, so does the latter, but not vice versa, so we prefer the latter.

$$\text{states } x = \{\text{goto } \mu\ x \mid \mu \in \text{reachable},\ x \in \text{first } \mu\}$$

The corresponding ordering relation is the following:

$$\begin{aligned}\mu_0 \geq \mu_n \equiv \exists X_1 \ldots X_n \in V^*, \mu_1, \ldots, \mu_{n-1} :\\ \mu_i = \text{goto } \mu_{i-1}\ X_i, 1 \leq i \leq n, X_1 \ldots X_n \overset{*}{\rightarrow} \varepsilon\end{aligned}$$

This gives as the set of maximal states:

$$\begin{aligned}\text{max\_states } x &= \{\mu \mid \mu \in \text{states } x,\ \eta \in \text{states } x,\ \mu \neq \eta,\ \mu \geq \eta,\ \eta \not\geq \mu\}\\ \text{unique } \mu\ x &= \text{max\_states } x = \{\mu\}\end{aligned}$$

The deletion function for the LALR(1) parser can now be modified. It passes the continuation c as both a normal and error continuation to P. This continuation returns a j such that $\exists (A \rightarrow \alpha.\beta) \in \mu : \beta \overset{*}{\rightarrow} (xi)/j$. The fact that $x \in$ dir μ guarantees that $j \neq xi$ and hence that the deletion function terminates.

```
delete f xi = if x ∈ f              → xi
                 ∃μ : unique μ x → delete f (P μ c c (f ++ dir μ) i)
                                 → delete f i
              fi
where c J j = j
```

Observe that the deterministic bottom-up parser guarantees that $x \in f$, so no recursive call of delete is necessary in this case.
In a Pascal recursive descent parser, the following terminal symbols correspond to unique states and allow resumption of parsing when encountered during deletion:
program, **const**, **type**, **file**, **array**, **record**, **set**, **for**, **to**, **downto**, **goto**, **label**, **if**, **then**, **else**, **repeat**, **until**, **while**, **with**, **not** and a relational operator.

3.7 Memoization of non-deterministic parsers

Non-deterministic parsers are ideal algorithms for memoization. They run in exponential time, but the total number of sub-problems is cubic in the length of the input string. Cubic parsing algorithms have been well known for a long time [Ear70, You67, Kas65, Tom86] and they all rely on memoization. The traditional presentations of these algorithms do not separate the memoization from the function that is memoized: they present an

algorithm where implementation details of the memo table manipulation are intertwined with operations belonging to the algorithm that is used to solved the problem. E.g. a considerable part of [Tom86] is devoted in the design of a memo table structure, called a graph-structured stack, without observing that memoization takes place. In [Nor91], the Earley parser is presented as a memoized bottom-up parser.
In this section, we show that our parsers can be memoized as well, resulting in variants of the Earley and Tomita algorithm, with cubic time behavior.

3.7.1 Recursive descent memoization

We start with a simple recursive descent parser obeying the specification:

Specification 3.79 (spec:MTD-nd)

$$\mathsf{parse}\ \alpha\ \mathsf{i} = \alpha \overset{*}{\rightarrow} \mathsf{i}$$

□

Observe that parse is just another notation for the $\overset{*}{\rightarrow}$ relation.
We will make use of the function splittings obeying the specification:

$$\mathsf{splittings}\ \mathsf{i} = \{(\mathsf{j},\mathsf{k}) \mid \exists \mathsf{j},\mathsf{k} : \mathsf{j} \mathbin{+\!\!+} \mathsf{k} = \mathsf{i}\}$$

We can now trivially implement parse as (leaving the proof to the reader):

Implementation 3.80 (of specification (3.79)) (MTD-nd)

$$\begin{array}{lll}
\mathsf{parse}\ \varepsilon & \mathsf{i} = & \mathsf{i} = \varepsilon \\
\mathsf{parse}\ \mathsf{x} & \mathsf{i} = & \mathsf{i} = \mathsf{x} \\
\mathsf{parse}\ \mathsf{A} & \mathsf{i} = & \vee/\{\mathsf{parse}\ \alpha\ \mathsf{i} \mid \mathsf{A} \rightarrow \alpha\} \\
\mathsf{parse}\ \alpha\beta & \mathsf{i} = & \vee/\{\mathsf{parse}\ \alpha\ \mathsf{j} \wedge \mathsf{parse}\ \beta\ \mathsf{k} \mid (\mathsf{j},\mathsf{k}) \in \mathsf{splittings}\ \mathsf{i}\}, \alpha, \beta \neq \varepsilon
\end{array}$$

□

Assuming that this function is memoized, there are at most $\mathcal{O}(\mathsf{n}^2)$ different calls. The first three rules take constant time, the last one at most $\mathcal{O}(\mathsf{n})$ time, giving as a total $\mathcal{O}(\mathsf{n}^3)$ time for the whole parser. It is as simple as that. Of course, the implementation of the memoization is something that has to be designed as well, but that problem should be separated from the algorithm.
This simple parser does not terminate for left-recursive grammars, as our previous top-down parsers do, so we have a look at memoized bottom-up parsers. First, we start with a variant of our item-based recursive descent parser:

Definition 3.81 (MITD-nd)

$$\begin{array}{lll}
\mathsf{parse}\ (\mathsf{A} \rightarrow \alpha.) & \mathsf{i} = & \mathsf{i} = \varepsilon \\
\mathsf{parse}\ (\mathsf{x} \rightarrow .\mathsf{x}) & \mathsf{i} = & \mathsf{i} = \mathsf{x} \\
\mathsf{parse}\ (\mathsf{A} \rightarrow \alpha.\mathsf{X}\beta) & \mathsf{i} = & \vee/\{\mathsf{parse}\ (\mathsf{X} \rightarrow .\gamma)\ \mathsf{j} \wedge \mathsf{parse}\ (\mathsf{A} \rightarrow \alpha\mathsf{X}.\beta)\ \mathsf{k} \\
& & \qquad \mid \mathsf{X} \rightarrow .\gamma \in \mathit{Items},\ (\mathsf{j},\mathsf{k}) \in \mathsf{splittings}\ \mathsf{i}\}
\end{array}$$

□

3.7.2 Single recursive ascent memoization

We can apply the recursive ascent scheme RA1 to the parser of the previous section (actually we can not, since the function does not return a set, but by representing the empty set by false and an arbitrary non-empty set by true the scheme is applicable again).

$$
\begin{array}{lll}
\mathsf{t}\ (A \to \alpha.) & i = & i = \varepsilon \\
\mathsf{t}\ (x \to .x) & i = & i = x \\
\mathsf{t}\ (A \to \alpha.X\beta) & i = & \mathsf{false} \\
\mathsf{sub}\ (A \to \alpha.) & i = & \{\} \\
\mathsf{sub}\ (x \to .x) & i = & \{\} \\
\mathsf{sub}\ (A \to \alpha.X\beta) & i = & \{(X \to .\gamma, j) \mid X \to .\gamma \in \mathit{Items},\ (j,k) \in \mathsf{splittings}\ i\} \\
\end{array}
$$
$$\mathsf{h}\ (A \to \alpha.X\beta, i)\ (X \to .\gamma, j)\ b\ =\ b \wedge \mathsf{parse}\ (A \to \alpha X.\beta)\ k\ \textbf{where}\ k\ =\ j\backslash i$$

This gives the following recursive ascent (RA1) algorithm, where we have not declared q local to p.

Definition 3.82 (MIRA1-nd)

$$
\begin{array}{l}
p\ J\ i\ =\ \vee/\{q\ J\ (A \to \alpha.)\ i \quad \mid A \to \alpha. \in \bar{J}\} \vee \\
\qquad\qquad \vee/\{q\ J\ (x \to .x)\ j \quad \mid x : j \leftarrow i,\ x \to .x \in \bar{J}\} \\
q\ J\ (X \to \gamma.\delta)\ j\ =\ (X \to \gamma.\delta = J \wedge j = \varepsilon) \vee \\
\qquad\qquad \vee/\{p\ (A \to \alpha X.\beta)\ k \wedge q\ J\ (A \to \alpha.X\beta)\ m \\
\qquad\qquad\quad \mid \gamma = \varepsilon \wedge A \to \alpha.X\beta \in \bar{J} \wedge (k,m) \in \mathsf{splittings}\ j\}
\end{array}
$$
□

There are at most $\mathcal{O}(n^2)$ different invocations of p, each taking unit time and at most $\mathcal{O}(n^2)$ different invocations of q. Each call of q takes at most $\mathcal{O}(n)$ time, resulting in a total time complexity of $\mathcal{O}(n^3)$ again. When we would have declared q local to p, a time complexity of $\mathcal{O}(n^4)$ would have been obtained. This is caused by the fact that q does not depend on i, but each local q would have its own memo table, for each i separately. Here we see that it can be advantageous to apply so-called 'λ-lifting' to a memo function, raising it to a global level and increasing the size of its memo table, thereby allowing for more sharing of results of this function.
Of course, we can apply scheme RA2 as well to our item-based parser. The result of this scheme is:

Definition 3.83 (MIRA2-nd)

$$
\begin{array}{l}
p\ J\ i\ =\ J\ =\ A \to \alpha. \wedge i = \varepsilon\ \vee \\
\vee/\{q\ J\ A\ i \quad \mid A \to . \in \overline{J}^{+}\} \vee \\
\vee/\{q\ J\ x\ j \quad \mid x : j \leftarrow i,\ x \to .x \in \overline{J}^{+}\} \\
q\ J\ X\ j\ =\ \vee/\{p\ (A \to \alpha X.\beta)\ j \mid A \to \alpha.X\beta \in \bar{J},\ \alpha \neq \varepsilon\} \vee \\
\qquad\qquad \vee/\{p\ (A \to X.\beta)\ k \wedge q\ J\ A\ m \mid A \to .X\beta \in \bar{J} \wedge (k,m) \in \mathsf{splittings}\ j\}
\end{array}
$$
□

Again, this parser has cubic time complexity.

3.7.3 Doubly recursive ascent memoization

Doubly recursive ascent memoization is more complex than recursive descent and single recursive ascent memoization. We have two options for the specification:

Specification 3.84 (spec:MIDRA2-nd)

$$P\ \mu\ i \ = \ \{A \to \alpha.\beta \mid A \to \alpha.\beta \in \mu,\ \beta \overset{*}{\to} i\}$$

□

and

Specification 3.85 (spec:MIDRA2-nd-2)

$$P\ \mu\ (A \to \alpha.\beta)\ i \ = \ A \to \alpha.\beta \in \mu \wedge \beta \overset{*}{\to} i$$

□

It is found that the second version degrades to a one item parser again, like the RA1 and RA2 parsers, instead of a parser that operates on a set μ of items, like the DRA1 and DRA2 parsers, so we explore the first specification. A straightforward implementation, based on the Leermakers parser, yields:

Implementation 3.86 (of specification (3.84)) (MIDRA2-nd)

$$\begin{aligned}
P\ \mu\ i \ &= \ \{A \to \alpha. \mid A \to \alpha. \in \mu,\ i = \varepsilon\} +\!\!+ \\
&\quad\ \text{concat}\ \{Q\ \mu\ A\ i \mid A \to . \in \overline{\mu}\} +\!\!+ \\
&\quad\ \text{concat}\ \{Q\ \mu\ x\ j \mid x : j \leftarrow i,\ x \to .x \in \overline{\mu}\} \\
Q\ \mu\ X\ i \ &= \ \{A \to \alpha.X\beta \mid A \to \alpha X.\beta \in P\ (\text{goto}\ \mu\ X)\ i,\ \alpha \neq \varepsilon\} +\!\!+ \\
&\quad\ \text{concat}\ \{Q\ \mu\ A\ k \mid (j,k) \in \text{splittings}\ i,\ A \to X.\beta \in P\ (\text{goto}\ \mu\ X)\ j\}
\end{aligned}$$

□

This parser can be improved further by passing the requirement on the length of α as an argument to P, which then obeys the specification:

Specification 3.87 (spec:MIDRA-nd)

$$P\ \mu\ m\ i \ = \ \{A \to \alpha.\beta \mid A \to \alpha.\beta \in \mu,\ \#\alpha = m,\ \beta \overset{*}{\to} i\}$$

□

and which is implemented by:

Implementation 3.88 (of specification (3.87)) (MIDRA-nd)

$$\begin{aligned}
P\ \mu\ m\ i \ &= \ \{A \to \alpha. \mid A \to \alpha. \in \mu,\ i = \varepsilon,\ \#\alpha = m\} +\!\!+ \\
&\quad\ \text{concat}\ \{Q\ \mu\ A\ m\ i \mid A \to . \in \overline{\mu}\} +\!\!+ \\
&\quad\ \text{concat}\ \{Q\ \mu\ x\ m\ j \mid x : j \leftarrow i,\ x \to .x \in \overline{\mu}\} \\
Q\ \mu\ X\ m\ i \ &= \ \{A \to \alpha.X\beta \mid A \to \alpha X.\beta \in P\ (\text{goto}\ \mu\ X)\ (m+1)\ i\} +\!\!+ \\
&\quad\ \text{concat}\ \{Q\ \mu\ A\ m\ k \mid (j,k) \in \text{splittings}\ i,\ A \to X.\beta \in P\ (\text{goto}\ \mu\ X)\ 1\ j\}
\end{aligned}$$

□

The complexity of this algorithm is dominated by the complexity of the second term of Q. It has $\mathcal{O}(n^2)$ invocations, takes $\mathcal{O}(n)$ recursive calls, each taking linear time by memoization, and each delivering a set of a size dependent only on the size of the grammar. The union of the first and second term takes thus $\mathcal{O}(1)$ time, resulting in (again of course) a $\mathcal{O}(n^3)$ time complexity.

3.8 A hierarchy of parsing algorithms

In this chapter we have derived a large number of parsing algorithms, but the techniques applied in doing so allow for the derivation of far more parsers. The parsers we have derived are depicted in the following diagram. The top-down parser (3.8) and the item-based one (3.31) form the basis of all these algorithms.

The main techniques applied to these parsers are recursive descent to ascent transformation (in the four versions (D)RA(1-2)), continuations for back-tracking, adding look-ahead, adding error recovery and removing non-determinism. These techniques can be applied to many more parsers than we have shown here. The use of continuations can be applied to all non-deterministic parsers, not all versions of ascent parsing have been investigated and the use of look-ahead and error recovery has been investigated for the top-down and Leermakers parsers only. Most transformations we applied contain intermediate expressions which represent also valid parsers, thus giving rise to even more versions.
The parsing algorithm is doubly recursive in its nature and the descent-ascent transformations remove only one of the recursive calls, while the other remains explicit in the algorithm. Since many of the algorithms we presented have the same specification, but a different implementation, this remaining call can invoke any of the different versions, provided that it obeys the correct implementation. Since this new call can in turn make a similar choice, the total amount of possible parsers does not only depend on the grammar, but also on the size of the input string. For example, a certain parser might call a descent version for all its even invocations and an ascent version for all odd invocations. Of course, it is more sensible to let this choice depend on properties of the grammar, e.g. apply top-down parsing to LL(1) states, bottom-up parsing for non-LL(1), but LALR(1) and continuation-based for all other states.
Also parsers for single items and states (set of items) can be combined by observing that a set of items can be attacked by unifying the results for the individual items.

$$\mathsf{P}\ \mu\ \mathsf{i} = \mathsf{concat}\ \{(A \to \alpha.\beta, \mathsf{j}) \mid (A \to \alpha.\beta) \in \mu,\ \mathsf{j} \in \mathsf{parse}\ (A \to \alpha.\beta)\ \mathsf{i}\}$$
$$\mathsf{parse}\ (A \to \alpha.\beta)\ \mathsf{i} = \mathsf{P}\ \{A \to \alpha.\beta\}\ \mathsf{i}$$

Even item-based and production-based parsers can be combined, by using the identities:

$$\mathsf{parse}\ (A \to \alpha.\beta)\ \mathsf{i} = \mathsf{parse}\ \beta\ \mathsf{i}$$
$$\mathsf{parse}\ A\ \mathsf{i} = \mathsf{concat}\ \{\mathsf{parse}\ (A \to .\alpha)\ \mathsf{i} \mid A \to \alpha\}$$

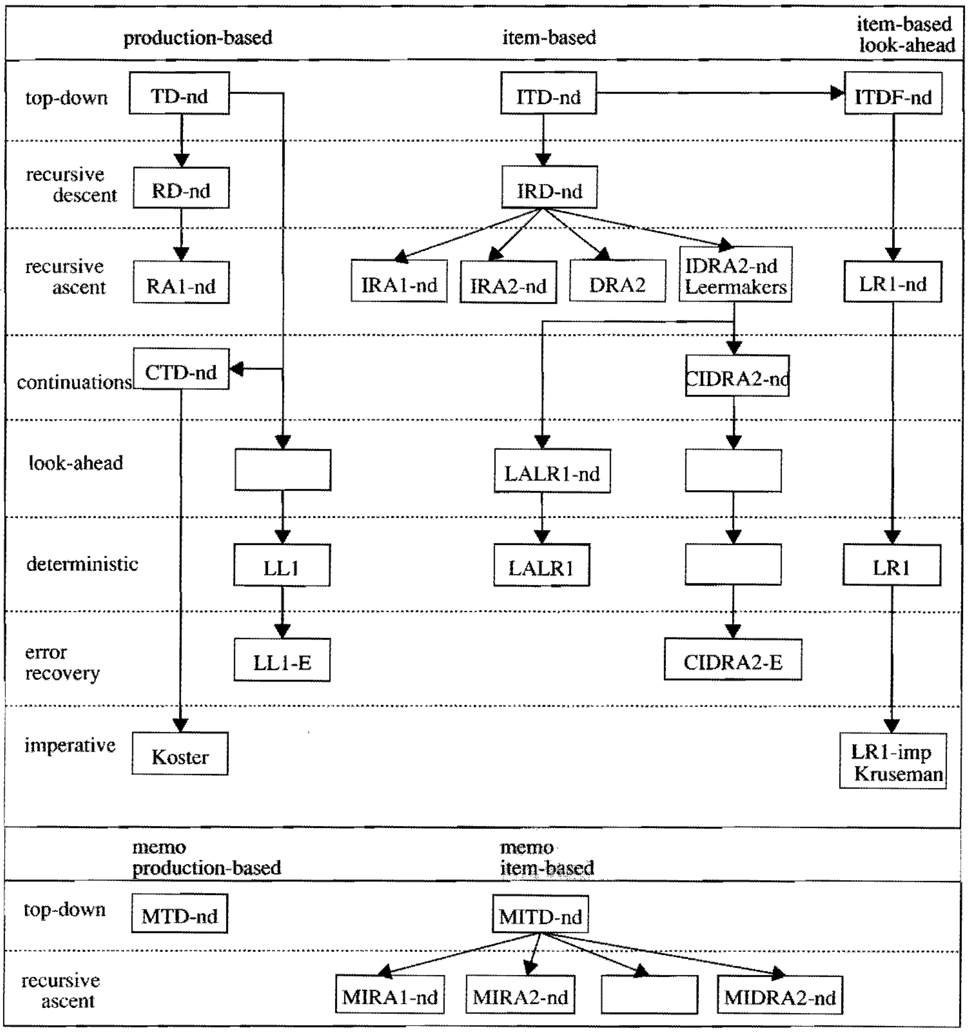


Figure 3.1: A hierarchy of parsing algorithms

Production-based and continuation-based parsing can be combined by using:

$$\text{parse } \alpha \text{ i} = \text{cont } \alpha \text{ single i}$$
$$\text{cont } \alpha \text{ c i} = \text{concat } \{\text{c j} \mid \text{j} \in \text{parse } \alpha \text{ i}\}$$

Thus, virtually all combinations are possible. It would be interesting to investigate what combination yields the best practical parser for a given grammar, e.g. in the sense that the parser is as deterministic as possible (e.g. for a maximum number of states) or is as simple as possible (e.g. has a minimal number of states, or as many LL(1) states as possible). This field has not been explored in this thesis however, but the thesis presents ample opportunity to do so with all the identities it presents. It illustrates the power of functional programming in general and program transformations in particular.

3.9 The Elegant parsers

Of the many parsers presented in this section, three have been implemented in the Elegant system; all are implemented as imperative algorithms, which are generated in the C programming language. In all cases, the first symbol of the remainder of the input string is available in a global variable sym and the next symbol is obtained by calling the function nextsym.

3.9.1 LL(1) recursive descent

The default Elegant parser is an LL(1) recursive descent parser, derived from (3.73). For each nonterminal A with production rules *nfp*$_1 \ldots p_n$, a function Parse_A is generated which calls one of the function parse_p_i, depending on the first symbol of the input string. The function parse$_p$, which is generated for a production rule $p = X_0 \rightarrow X_1 \ldots X_n$, has the following structure:

```
proc parse_p (f : follset);
begin parse_X$_1$(f $\cup$ foll (X$_1$ ... X$_n$) $\varepsilon$);
      ...
      parse_X$_n$(f);
end;
```

The follower sets are computed statically and are represented by a number of bitsets, as large as the number of non-terminals in the grammar. For each terminal symbol x a function parse_x is generated:

```
proc parse_x (f : follset);
begin delete(f);
      if sym = x
      then nextsym();
      else error("x expected");
      fi
end;
```

3.9.2 LALR(1) recursive ascent

On demand, by using a switch, an `Elegant` user can generate an LALR(1) recursive ascent parser. This parser is derived from (3.72), but is an LALR(1) instead of an LR(1) parser. For each state μ a function $\mathsf{parse\_}\mu$ is generated. Since an LALR(1) parser can have a large number of states, the simple ones are omitted and their calls have been replaced by in-line code.
The error recovery is similar to the error recovery for LL(1) parsers: follower sets are added as parameters to the parse functions and are used for synchronization. This technique has one complication, which is the reason why we have presented error recovery with recursive ascent continuation parsing. This complication is that, upon encountering an error and performing an insertion, a parsing function can not decide which item to return, and hence how far to return. Simply returning until a state is reached which can handle the current symbol is not a solution either, since this state does not know which item to insert. Solving this problem would require a non-trivial administration, whose maintenance would mean a considerable overhead, even in the case when no errors occur at all.
In the `Elegant` LALR(1) parser an item is inserted by the state encountering an error. This item is an arbitrary item from that state. Since this insertion can be considered as the addition of an ε-rule to the grammar, it might introduce non-termination of the parser. Indeed, for left-recursive rules, like $\mathsf{A} \rightarrow \mathsf{A}\alpha$, the insertion of α leads to a reduction from state $\mathsf{A} \rightarrow \mathsf{A}.\alpha$ to state $\mathsf{A} \rightarrow .\mathsf{A}\alpha$, which calls the state $\mathsf{A} \rightarrow \mathsf{A}.\alpha$ again, without making any progress. Such potential cycles are detected by the parser generator and extra code for detecting these cycles is inserted in the functions for such states.

3.9.3 Back-tracking recursive descent

The third and most powerful, and hence the most expensive, parser is a continuations back-tracking parser. The back-tracking is not used for non-deterministic parsing but for the deterministic parsing of non LL(1) and LALR(1) grammars. For production rules which are LL(1), the normal LL(1) parser is used, but when a non-LL(1) production rule is encountered, the back-tracking parser is called. This parser returns the parse-function for the selected production rule and the LL(1) parser continues by calling this parse-function. No error recovery is performed by the back-tracking parser. When no production rule is applicable, this might be caused by a wrong choice earlier in the parse, which should be corrected by back-tracking. But it might equally well be an error, which should be detected. This problem is solved by selecting the production rule which is able to parse the maximum amount of terminal symbols, when no correct one is available. Thus a best parse is constructed, which is either correct or, if no correct one exists, maximal. The corresponding parse-function is returned to the LL(1) parser which takes care of the error recovery. The other choices made by the back-tracking parser, are stored in a list. When the LL(1) parser encounters another non-LL(1) procedure, the back-tracking parser first consults this list in order to detect whether this choice has already been made by a previous invocation. If so, it returns the parse-function, if not, it starts back-tracking.
The interface to the scanner becomes more complicated, since terminal symbols must be

preserved. Hence a list of terminal symbols is maintained and the function nextsym either picks one from this list or, when the list is empty, constructs a new one by calling the scanner.
The implementation of back-tracking is taken from Koster [Kos74]. A continuation is represented by a pair (p, c) where p is a parsing function from the continuations parser and c is the continuation of p. These pairs can be allocated on the recursion stack, rather than on the heap, since their life-time corresponds to the life-time of the corresponding stack-frames. The parsing function for the production rule $p = X_0 \to X_1 \ldots X_n$ is thus implemented by (omitting the details concerned with finding the maximum parse in the case of errors):

```
proc parse_p ($c_{n+1}$ : ^cont);
var $c_2, \ldots c_n$ : cont;
begin $c_2$.p := parse_$X_2$;
      $c_2$.c := address($c_3$);
      ...
      $c_n$.p := parse_$X_n$;
      $c_n$.c := $c_{n+1}$;
      parse_$X_1$(address($c_2$));
end;
```

Chapter 4

Attribute grammars as functional programs

4.1 Introduction

Traditionally, the formalization of attribute grammars has not been trivial. Especially the distinction between attributes and attribute values, or the absence of that distinction whre it should have been made, has resulted in involved formalizations, like the ones in [Knu68, Knu71, DJL88, Kas80, Eng84, RS82].

In this chapter we show how attributes can be added to a context free grammar in a simple way. The idea is not to associate all kinds of attributes, semantic rules, semantic conditions and semantic functions with a production rule. Instead, we can associate just a single function with a production rule. We call such a function a *rule function*. This rule function is a higher order function. When supplied with appropriate function valued arguments, it delivers a new function that maps inherited attributes onto synthesized attributes. This new function is called an *attribute function*. The functions that are given as arguments to the rule function are just the attribute functions that correspond to the symbols of the right-hand side of the production rule.

In the previous chapter, we have defined a parser as a function on an input string which is dependent on a context-free grammar. In this chapter, we define an *attributing parser* as a function on an attribute grammar and an input string. For context-free parsers, the explicit construction of a syntax tree was shown to be unnecessary. This happens to be the case for attributing parsers as well. An attributing parser can be described elegantly as a function that composes rule functions to form attribute functions. Such a description does not need to manipulate attributes at all: it merely composes functions over attributes. We nevertheless keep the name 'attribute grammars', in order to link up with a long-standing nomenclature and tradition.

A similar approach to the use of attribute grammars as functional programs can be found in [Joh87]. Johnsson presents a way to implement attributed parsers in a functional language, using a single function-valued synthesized attribute which is equivalent to our attribute functions. Where Johnsson describes a way to implement an attribute in this way, rewriting an arbitrary attribute grammar into a one with function-valued synthesized attributes, we *define* and attribute in this way. This has the advantage that an arbitrary parser can be used to parse the input string and manipulate the rule and attribute function, where the functions in [Joh87] are (slightly) different for each different parser.

After defining attributing parsers and having given an example, we restrict the class of functions that we allow as rule functions. This restriction allows us to perform the static cyclicity check of [Knu68, Knu71] on an attribute grammar. Moreover, when combined with recursive descent LL(1) parsing, it allows for an efficient implementation of attribute grammars using lazy evaluation.

In [Joh87] the possibility to use laziness to enlarge the class of non-circular attribute grammars, which is the most general class described so far, is discussed. Here we show that the class of attribute grammars that is implementable with the technique of rule and attribute functions is indeed more general than the non-circular attribute grammars. This more general class is obtained by taking the non-strictness of functions used in defining attributes into account. We call our more general class *pseudo circular attribute grammars.* `Elegant` supports this more general class.

For our more general class of attribute grammars, a standard attribution scheme can be developed that is to a large extent independent of the programming language modeled by the attribute grammar. The scheme prescribes how to create a data structure that is an abstract representation of the input text and how to relate applied occurrences of symbols (identifiers, operators, etc.) to their defining occurrences. Due to this, attribute-grammar writers can follow this standard scheme instead of applying ad-hoc techniques in circumventing cyclicities caused by a too restricted attribute evaluation mechanism.

The standard attribution scheme, in its turn, allows for an efficient implementation of a symbol-table which is based on side-effects. The scheme guarantees that these side-effects are semantically transparent, so that one need not be concerned about them when writing an attribute grammar in a declarative style, where the order of attribute evaluation is not under the control of the programmer.

4.2 Attribute grammars

In this section we define the notions of an attribute grammar and the semantics of an input string with respect to an attribute grammar. The definition we give here is a non-standard one. Traditionally, attribute grammars are defined by associating sets of inherited and synthesized attributes with production rules and by adding so-called semantic rules to define attributes in terms of each other. As an example of such a style, see e.g. [Knu68, Knu71, DJL88]. Here we will not be concerned with individual attributes. We associate a function with a production rule that maps inherited onto synthesized attributes. Attributes are just parameters and results of such functions, but we need not consider them at all. It suffices to compose functions into new ones.

Thus, an attribute grammar can be defined as an extension of a context-free grammar, by

adding so-called *rule functions* to each of the production rules. A rule function is a higher order function, that takes as its arguments a number of *attribute functions* (corresponding to the symbols of the right-hand side of the production rule) and returns a function, which is the attribute function for the left-hand side. An *attribute function* is related to a derivation for a grammar symbol (terminal or non-terminal). It maps its argument, called an *in-attribute* onto a pair consisting of a boolean value, the *context condition* and a value called the *out-attribute*.
A parser for a context-free grammar can be extended to an attributing parser by letting it compose attribute functions out of rule functions.

Definition 4.1 (def:ag)

An *attribute grammar* AG is an extension of a context-free grammar. It is defined as a tuple (V_T, V_N, P, S, D, F_T), where

- V_T is a set of *terminal symbols.*
- V_N is a set of *non-terminal symbols*, $V_T \cap V_N = \{\}$.
- $V = V_T \cup V_N$ is the set of *all symbols.*
- D is the domain (set) of *attribute values.* D usually is a many sorted set, containing basic attribute values, as well as tuples of attribute values in order to group them. In particular, the empty tuple () is an element of D.
- $F \subseteq D \mapsto \mathsf{Bool} \times D$ is the set of *attribute functions.*
  The argument value of an element of F is called the *in-attribute.* The boolean result value is called the *context condition.* The other result value is called the *out-attribute.*
- $\mathsf{F_T} :: V_T \mapsto (\{()\} \mapsto \{\mathsf{true}\} \times D)$ is the *terminal function.*
  It maps a terminal symbol onto an attribute function, called the *terminal attribute function.* For terminal symbol x, we denote $\mathsf{f_x} = \mathsf{F_T(x)}$. Observe that $\mathsf{f_x} \in \mathsf{F}$.
- $F_R \subseteq F^* \mapsto F$ is the set of *rule-functions.*
- $P \subseteq V_N \times V^* \times F_R$ is the set of *production rules.*
  If $\mathsf{p} = (\mathsf{A}, \alpha, \mathsf{f_p}) \in P$, then $\mathsf{f_p} \in (F^{\#\alpha} \mapsto F)$.
  If $(\mathsf{A}, \alpha, \mathsf{f_1}) \in P$ and $(\mathsf{A}, \alpha, \mathsf{f_2}) \in P$ then $\mathsf{f_1} = \mathsf{f_2}$.
  When $(\mathsf{A}, \alpha, \mathsf{f_p}) \in P$, we often write $\mathsf{f}_{\mathsf{A}\rightarrow\alpha}$ for $\mathsf{f_p}$ and write $\mathsf{A} \rightarrow \alpha$ to state that $\exists \mathsf{f} : (\mathsf{A}, \alpha, \mathsf{f}) \in P$.
- $S \in V_N$ is the *start symbol.*

□

With each attribute grammar AG an underlying CFG G can be associated by leaving out the details about attributes, rule functions and attribute functions.
For an attribute grammar the concept of a derivation is defined as follows:

Definition 4.2 (def:ag-derives)

With respect to an attribute grammar $AG = (V_T, V_N, P, S, D, F_T)$, we define the relation $\rightarrow$ (*derives*) on $(V^* \times F^*) \times (V^* \times F^*)$ by:

$(\alpha A\beta, \mathsf{fHg}) \rightarrow (\alpha\gamma\beta, \mathsf{fhg}) \equiv (A, \gamma, f_{A\rightarrow\gamma}) \in P$, $(H = f_{A\rightarrow\gamma}\ h) \wedge (\#\gamma = \#h) \wedge (\#\alpha = \#f) \wedge (\#\beta = \#g)$. The relation $\overset{*}{\rightarrow}$ denotes the reflexive and transitive closure of $\rightarrow$ over $(V^* \times F^*)$. We restrict ourselves to only those pairs (α, f) with $\#\alpha = \#f$.

□

Let $E\ i = \{f_x \mid x \in i\}$, i.e. the sequence of terminal attribute functions corresponding to i, then we abbreviate $(\alpha, f) \overset{*}{\rightarrow} (i, E\ i)$ by $(\alpha, f) \overset{*}{\rightarrow} i$, with $i \in V_T^*$.

Definition 4.3 (def:ag-lang)

The *language* $\mathcal{L}_{AG}$ of an attribute grammar $AG = (V_T, V_N, P, S, D, F_T)$ is defined as:

$$\mathcal{L}_{AG} = \{\omega \mid \exists f \in F, \sigma \in D : (S, f) \overset{*}{\rightarrow} \omega,\ (\mathsf{true}, \sigma) = f\ ()\}$$

The *semantics* of an input string $\omega \in V_T^*$ is defined by:

$$S_\omega = \{\sigma \mid \exists f \in F, \sigma \in D : (S, f) \overset{*}{\rightarrow} \omega,\ (\mathsf{true}, \sigma) = f\ ()\}$$

□

An *attributing parser* is an algorithm that determines, when given an attribute grammar *AG* and an input string ω, the semantics of ω.

4.2.1 Example

We illustrate our definition of an attribute grammar by a simple example, which is a classical one for attribute grammars, see e.g [Knu68]. This example describes a grammar that maps a string representing a binary fractional number onto the value of that fractional number. For example, the string ".1011" is mapped onto the number 0.6875 (in decimal representation). The attribute grammar for this example passes a position i down to each digit d, such that d contributes to the final value by $d.2^{-i}$

The attribute grammar is given by the following definitions:

$$
\begin{array}{lll}
V_T & = & \{'.', '0', '1'\} \\
V_N & = & \{\mathsf{number, frac, digit}\} \\
S & = & \mathsf{number} \\
D & = & \mathbf{R} \cup \{()\} \\
P & = & \{(\mathsf{number},\ \ \{'.', \mathsf{frac}\}, f_1) \\
 & & \ \ (\mathsf{frac},\ \ \{\mathsf{digit}\}, f_2) \\
 & & \ \ (\mathsf{frac},\ \ \{\mathsf{digit}, \mathsf{frac}\}, f_3) \\
 & & \ \ (\mathsf{digit},\ \ \{'0'\}, f_4) \\
 & & \ \ (\mathsf{digit},\ \ \{'1'\}, f_5)\} \\
f_{'.'}\ () & = & (\mathsf{true}, ()) \\
f_{'0'}\ () & = & (\mathsf{true}, 0) \\
f_{'1'}\ () & = & (\mathsf{true}, 1)
\end{array}
$$

$$
\begin{aligned}
f_1\ g_1\ g_2\ () &= (c_1 \wedge c_2,\ v_2)\\
&\quad \textbf{where } (c_1, v_1) = g_1\ ()\\
&\qquad\qquad (c_2, v_2) = g_2\ 1\\
f_2\ g_1\ i &= (c_1,\ v_1)\\
&\quad \textbf{where } (c_1, v_1) = g_1\ i\\
f_3\ g_1\ g_2\ i &= (c_1 \wedge c_2,\ v_1 + v_2)\\
&\quad \textbf{where } (c_1, v_1) = g_1\ i\\
&\qquad\qquad (c_2, v_2) = g_2\ (i+1)\\
f_4\ g_1\ i &= (c_1,\ v_1.2^{-i})\\
&\quad \textbf{where } (c_1, v_1) = g_1\ ()\\
f_5\ g_1\ i &= (c_1,\ v_1.2^{-i})\\
&\quad \textbf{where } (c_1, v_1) = g_1\ ()
\end{aligned}
$$

This simple example can be extended by adding the context condition that the number of digits may be at most 10, which requires modification of f_4 and f_5:

$$
\begin{aligned}
f_4\ g_1\ i &= (c_1 \wedge i \leq 10,\ v_1.2^{-i})\\
&\quad \textbf{where } (c_1, v_1) = g_1\ ()\\
f_5 &= f_4
\end{aligned}
$$

4.3 Attributing parsers

4.3.1 A recursive descent attributing parser

In this section we present a simple functional attributed recursive descent parser. It obeys the specification:

Specification 4.4 (spec:RD-ag)

$$\text{parse} :: V^* \times V_T^* \mapsto Set(F^* \times V_T^*)$$

$$\text{parse}\ \alpha\ i = \{(f, j) \mid \exists f, j\ :\ (\alpha, f) \overset{*}{\rightarrow} i/j\}$$

□

It is implemented recursively as follows:

- parse ε i

  $= \text{(specification)}$

  $\{(f, j) \mid \exists f, j\ :\ (\varepsilon, f) \overset{*}{\rightarrow} i/j\}$

  $= (\text{definition of } \overset{*}{\rightarrow})$

  $\{(f, j) \mid \exists f, j\ :\ i = j \wedge (\varepsilon, f) \overset{*}{\rightarrow} (i/j, \{\})\}$

  $= (\text{definition of } \overset{*}{\rightarrow} \text{ and substitute i for j and } \{\} \text{ for f})$

  $\{(\{\}, i)\}$

- parse x i
  = (specification, observe that we abbreviate $\{x\}$ by x)
  $\{(f,j) \mid \exists f,j \ : \ (x,f) \overset{*}{\rightarrow} i/j\}$
  = (definition of $\overset{*}{\rightarrow}$)
  $\{(f_x,j) \mid x : j \leftarrow i\}$
- parse A i
  = (specification)
  $\{(f,j) \mid \exists f,j \ : \ (A,f) \overset{*}{\rightarrow} i/j\}$
  = (definition of $\overset{*}{\rightarrow}$)
  $\{(f,j) \mid \exists f,f_\alpha,j \ : \ (A,f) \rightarrow (\alpha,f_\alpha),\ (\alpha,f_\alpha) \overset{*}{\rightarrow} i/j\}$
  = (specification of parse and definition of $\rightarrow$)
  $\{(f_{A\rightarrow\alpha}\ f_\alpha,j) \mid A \rightarrow \alpha,\ (f_\alpha,j) \in \text{parse}\ \alpha\ i\}$
- parse $(\alpha\beta)$ i
  = (specification)
  $\{(f,j) \mid \exists f,j \ : \ (\alpha\beta,f) \overset{*}{\rightarrow} i/j\}$
  = (definition of $\overset{*}{\rightarrow}$)
  $\{(f_\alpha +\!\!+ f_\beta,k) \mid \exists j,k,f_\alpha,f_\beta : (\alpha,f_\alpha) \overset{*}{\rightarrow} i/j,\ (\beta,f_\beta) \overset{*}{\rightarrow} j/k\}$
  = (specification of parse)
  $\{(f_\alpha +\!\!+ f_\beta,k) \mid (f_\alpha,j) \in \text{parse}\ \alpha\ i,\ (f_\beta,k) \in \text{parse}\ \beta\ j\}$

Summarizing the algorithm gives:

Implementation 4.5 (of specification (4.4)) (RD-ag)

$$
\begin{array}{lcl}
\text{parse}\ \varepsilon\ i & = & \{(\{\},i)\} \\
\text{parse}\ x\ i & = & \{(f_x,j) \mid x : j \overset{i}{\rightarrow}\} \\
\text{parse}\ A\ i & = & \{(f_{A\rightarrow\alpha}\ f_\alpha,j) \mid A \rightarrow \alpha,\ (f_\alpha,j) \in \text{parse}\ \alpha\ i\} \\
\text{parse}\ (\alpha\beta)\ i & = & \{(f_\alpha +\!\!+ f_\beta,k) \mid (f_\alpha,j) \in \text{parse}\ \alpha\ i,\ (f_\beta,k) \in \text{parse}\ \beta\ j\}
\end{array}
$$
□

The semantics of an input string are given by the expression:

$\{\sigma \mid \exists f \in F, \sigma \in D \ : \ (S,f) \overset{*}{\rightarrow} \omega,\ (\text{true},\sigma)\ =\ f\ ()\}$
  = (specification of parse)
$\{\sigma \mid (f,\varepsilon) \in \text{parse}\ S\ \omega,\ (\text{true},\sigma) \overset{f}{\rightarrow}\ ()\}$

This definition of an attribute grammar and the semantics of an input string is remarkably simple when compared to traditional presentations like [Eng84, DJL88] which do not formally define the semantics of an input string or the language generated by an attribute grammar. These traditional presentations rely on the notions of decorated syntax trees and dependency graphs, concepts that are not needed in the functional approach. In section (4.4) we present a normal form for rule functions that can easily be mapped onto the more traditional approaches and where inherited and synthesized attributes appear as arguments and results of attribute functions.

4.3.2 Attributed Leermakers parser

Not only recursive descent parsers can be generalized to attributing parsers, but recursive ascent parsers as well. An attributed version of the Leermakers parser obeys the specification:

Specification 4.6 (spec:Leermakers-ag-P)

$$\mathsf{P}\ \mu\ \mathsf{i} = \{(A \to \alpha.\beta, \mathsf{f}, \mathsf{j}) \mid \exists \mathsf{f}, \mathsf{j}\ :\ A \to \alpha.\beta \in \mu,\ (\beta, \mathsf{f}) \overset{*}{\to} \mathsf{i}/\mathsf{j}\}$$

□

and is implemented as:

Implementation 4.7 (of specification (4.6)) (Leermakers-ag-P)

$$\begin{aligned}\mathsf{P}\ \mu\ \mathsf{i} = {} & \{(A \to \alpha., \{\}, \mathsf{i}) \mid A \to \alpha. \in \mu\} +\!\!+ \\ & \mathsf{concat}\ \{\mathsf{Q}\ A\ \mathsf{f}_{A\to\varepsilon}\ \mathsf{i} \mid A \to . \in \overline{\mu}\} +\!\!+ \\ & \mathsf{concat}\ \{\mathsf{Q}\ x\ \mathsf{f}_x\ \mathsf{j} \mid x : \mathsf{j} \leftarrow \mathsf{i},\ x \to .x \in \overline{\mu}\}\end{aligned}$$

□

Here, Q is defined local to P and it obeys the specification:

Specification 4.8 (spec:Leermakers-ag-Q)

$$\mathsf{Q}\ X\ \mathsf{f}_X\ \mathsf{j} = \{(A \to \alpha.\beta, \mathsf{f}_\beta, \mathsf{k}) \mid \exists \mathsf{f}_\beta, \mathsf{j}, \mathsf{k}\ :\ A \to \alpha.\beta \in \mu,\ (X, \mathsf{f}_X) \overset{*}{\to} \mathsf{i}/\mathsf{j},\ (\beta, \mathsf{f}_\beta) \overset{*}{\to} \mathsf{j}/\mathsf{k}\}$$

□

while it is implemented by:

Implementation 4.9 (of specification (4.8)) (Leermakers-ag-Q)

$$\begin{aligned}\mathsf{Q}\ X\ \mathsf{f}_X\ \mathsf{j} = {} & \{(A \to \alpha.X\beta, \mathsf{f}_{X\beta}, \mathsf{k}) \mid (A \to \alpha X.\beta, \mathsf{f}_\beta, \mathsf{k}) \in \mathsf{next},\ \alpha \neq \varepsilon\} +\!\!+ \\ & \mathsf{concat}\ \{\mathsf{Q}\ A\ (\mathsf{f}_{A\to X\beta}\ \mathsf{f}_{X\beta})\ \mathsf{k} \mid (A \to X.\beta, \mathsf{f}_\beta, \mathsf{k}) \in \mathsf{next} \\ & \qquad\qquad\qquad\qquad \textbf{where}\ \mathsf{f}_{X\beta} = \{\mathsf{f}_X\} +\!\!+ \mathsf{f}_\beta\} \\ & \textbf{where}\ \mathsf{goto} = \{A \to \alpha X.\beta \mid A \to \alpha.X\beta \in \overline{\mu}\} \\ & \qquad\quad \mathsf{next} = \mathsf{P}\ \mathsf{goto}\ \mathsf{j}\end{aligned}$$

□

4.3.3 Attributed continuation parser

An attributed continuation parser can be designed as well. Again, it is a fine example of abstraction. It is specified by:

Specification 4.10 (spec:cont-ag)

$$\mathsf{cont} \in \bigcup_{k\geq 0} V^k \times (F^k \times V_T^* \mapsto Set(D)) \times V_T^* \mapsto Set(D)$$

$$\mathsf{cont}\ \alpha\ \mathsf{c}\ \mathsf{i} = \mathsf{concat}\ \{V \mid \exists \mathsf{j}, \mathsf{f}_\alpha\ :\ (\alpha, \mathsf{f}_\alpha) \overset{*}{\to} \mathsf{i}/\mathsf{j}$$

□

Its implementation is given by:

Implementation 4.11 (of specification (4.10)) (cont-ag)

$$\begin{aligned}
&\text{cont } \varepsilon \text{ c i} &&= \text{c i}\\
&\text{cont x c i} &&= \textbf{if} \text{ hd i} = \text{x}\textbf{then} \text{ c } f_x \text{ (tl i) } \textbf{else} \ \{\} \ \textbf{fi}\\
&\text{cont A c i} &&= \text{concat } \{\text{cont } \alpha \text{ (c } \circ_{|\alpha|} f_{A\to\alpha}) \text{ i} \mid A \to \alpha\}\\
&\text{cont } \alpha\beta \text{ c i} &&= (\text{cont } \alpha \circ \text{cont } \beta \circ \text{c}) \text{ i}
\end{aligned}$$

□

where the operator $\circ_k$ is defined by definition (2.15) and can be defined in another way by:

$$(f \circ_k g)\, h_1 \dots h_k = f\,(g\, h_1\, \dots\, h_k)$$

This parser is called on the start symbol in the following way:

$$\text{cont } S \text{ d } 0 \ \textbf{where} \text{ d } f_S \text{ j} = \{\sigma \mid \text{j} = \varepsilon,\ (\text{true}, \sigma) \leftarrow f_S\ ()\}$$

4.4 Normal form rule functions

In this section we restrict the form of the rule functions. As already mentioned, a rule function $f_{X_0 \to X_1 \dots X_n}$ maps a list L of n attribute functions onto a new attribute function. This latter, when supplied with an in-attribute a, yields a pair (c_0, e_s). Here, c_0 is a boolean value, called the *context condition* and e_s is called the *out-attribute*. The restriction in form dictates a special way to compute the elements of this pair. They are obtained by applying the elements of L to arguments (derived from a and some intermediate values). c_0 is required to be the conjunction of the n context conditions that are obtained in this way and some additional rule condition e_c. e_s is an expression over a and some intermediate results, among others the n out-attributes yielded by the elements of L. More formally expressed, the resulting restricted form of a rule function has type $f_{X_0 \to X_1 \dots X_n} :: F^n \times D \mapsto (\text{Bool} \times D)$ and is implemented as follows:

$$\begin{aligned}
&f_{X_0 \to X_1 \dots X_n}\ f_1 \dots f_n\ a = (c_0, e_s)\\
&\textbf{where}\ c_0 = e_c \wedge c_1 \wedge \dots \wedge c_n\\
&\qquad (c_1, s_1) = f_1\ e_1\\
&\qquad \dots\\
&\qquad (c_n, s_n) = f_n\ e_n
\end{aligned}$$

Here the e's stand for arbitrary expressions over $a, s_1, \dots s_n$. We silently assume that all expressions are well-formed, e.g. the type of e_1 corresponds to the argument type of f_1. This restriction in form allows for the static cyclicity check on attribute grammars [Knu68, Knu71]. The example of section 4.2.1 is in normal form.

We can compare this form of a rule function with the more traditional definition of an attributed production rule. The variable a can be viewed as the inherited attribute of the left-hand side[1]. The synthesized attributes of the right-hand side are represented by the variables s_i. The other attributes are less clearly present. The left-synthesized and right-inherited attributes are not distinguishable as variables. Instead, their defining expressions e_i and e_s are directly passed as arguments (results) of the corresponding attribute and rule functions. Thus this form is more general than the traditional forms of attribute grammars [Knu68, Knu71, Eng84, DJL88]. It is easy however to restrict the normal form even further to obtain a one-to-one correspondence between variables and traditional attributes. Since it does not gain us anything, we do not impose this additional restriction here. The notation of the `Elegant` attribute grammars closely corresponds to the normal form of rule functions. See section 4.8 and [Aug92b] for more details.

4.5 LL(1) parsing and normal form rule functions

We can combine the normal form of a rule function and the corresponding function for the parsing of a production rule iofn an LL(1) recursive descent parser into a single function. First we present a form of the recursive descent parse parse (4.5) for the deterministic case. This deterministic parser delivers exactly one result. We extend its domain from symbols to the union of symbols and production rules $(V_N \times V^*)$ for convenience.

$$\mathsf{parse} :: (V \cup (V_N \times V^*) \times V_T^* \mapsto F \times V_T^*$$

$$\mathsf{parse\ x\ i} = \textbf{if}\ \mathsf{hd\ i} = \mathsf{tl\ i} \leftarrow \mathsf{i}\ \textbf{then}\ (\mathsf{f_x, j})\ \textbf{else}\ \mathsf{error}\ \text{"x expected"}\ (\mathsf{f_x, i})\ \textbf{fi}$$

```
parse A i = case x ∈ foll $\alpha_1$ (foll A) → parse (A → $\alpha_1$) i
                 ...                      → ...
                 otherwise                → error "A expected" ($\perp$, i)
            esac
where x : j = i
```

```
parse ($X_0 \to X_1 \ldots X_n$) $i_0$ = ($f_{X_0 \to X_1 \ldots X_n} f_1 \ldots f_n, i_n$)
where ($f_1, i_1$) = parse $X_1$ $i_0$
      ...
      ($f_n, i_n$) = parse $X_n$ $i_{n-1}$
```

This deterministic parser still composes attribute functions. We can modify it however, by explicitly passing an inherited attribute to the parse function as an additional argument. The corresponding synthesized attribute is, together with the context condition, delivered as an additional result. We denote this new attributed parser by parse_attr and define it by (using ϕ to denote an element from $(V_N \times V^*) \cup V$):

$$\mathsf{parse\_attr} :: ((V_N \times V^*) \cup V) \times D \times V_T^* \mapsto V_T^* \times \mathsf{Bool} \times D$$

[1]In practice, an attribute function will operate on more than one attribute. In that case, the parameter a can be a tuple and the result e_s can be a tuple-valued expression. Also the variables s_i may be replaced by tuples of variables in this case.

```
parse_attr $\phi$ a i = (j, c, b)
where (f, j) = parse $\phi$ i
      (c, b) = f a
```

Substituting the different rules for parse in this definition yields:

```
parse_attr x a i = if hd i = x ← i
                   then (tl i, true, v)
                   else error "x expected"  (i, false, v)
                   fi
                   where (true, v) = $f_x$ a

parse_attr A a i = case x ∈ foll $\alpha_1$ (foll A) → parse (A → $\alpha_1$) a i
                        ...                        → ...
                        otherwise                  → error "A expected"  (i, false, ⊥)
                   esac
where x : j = i

parse_attr ($X_0$ → $X_1 \dots X_n$) a $i_0$ = ($i_n$, $c_0$, s)
where (g, $i_n$) = parse ($X_0$ → $X_1 \dots X_n$) $i_0$
      ($c_0$, s) = g a
   = (definition of parse)
parse_attr ($X_0$ → $X_1 \dots X_n$) a $i_0$ = ($i_n$, $c_0$, s)
where ($f_1$, $i_1$) = parse $X_1$ $i_0$
      ...
      ($f_n$, $i_n$) = parse $X_n$ $i_{n-1}$
      ($c_0$, s) = $f_{X_0 \to X_1 \dots X_n} f_1 \dots f_n$ a
   = (Normal form for $f_{X_0 \to X_1 \dots X_n}$)
parse_attr ($X_0$ → $X_1 \dots X_n$) a $i_0$ = ($i_n$, $c_0$, $e_s$)
where ($f_1$, $i_1$) = parse $X_1$ $i_0$
      ($c_1$, $s_1$) = $f_1$ $e_1$
      ...
      ($f_n$, $i_n$) = parse $X_n$ $i_{n-1}$
      ($c_n$, $s_n$) = $f_n$ $e_n$
      $c_0$ = $e_c \wedge c_1 \wedge \dots \wedge c_n$
   = (Definition of parse_attr)
parse_attr ($X_0$ → $X_1 \dots X_n$) a $i_0$ = ($i_n$, $c_0$, $e_s$)
where $c_0$ = $e_c \wedge c_1 \wedge \dots \wedge c_n$
      ($i_1$, $c_1$, $s_1$) = parse_attr $X_1$ $e_1$ $i_0$
      ...
      ($i_n$, $c_n$, $s_n$) = parse_attr $X_n$ $e_n$ $i_{n-1}$
```

The function parse_attr has now been defined independently of parse and the use of the definition of normal form rule functions has enabled us to get rid of function composition (actually: partial parameterization) for attribute functions. The resulting function parse_attr can be implemented efficiently in an imperative language. When attributes are mutually dependent in the where-part, they can be implemented with the use of lazy (demand-driven) implementation. In section 4.9 we present such an implementation.

4.6 Pseudo circular attribute grammars

The normal form of rule grammars corresponds to the attribute grammars of [Knu68, Knu71]. Each production rule is associated with one rule function. Such a rule function can deliver an attribute function that usually operates on a tuple a, the *inherited* attributes and produces a result e_s, which usually is a tuple of *synthesized* attributes. Observe that we do not require synthesized attributes to have a name. They can be declared as local attributes, but this is by no means mandatory. Moreover, the rule specifies a *rule condition* e_c. Each derivation of a symbol of the right-hand side of a production rule corresponds to an attribute function f_i. Such a function operates on e_i, its *inherited* attributes, which again need not be named, and produces its *synthesized* attributes s_i and its context condition c_i. The latter is only propagated.
Our treatment of attributes as tuple elements implies that these elements must be independently computable in order to obtain the power of traditional attribute grammars. Fortunately, lazy functional languages allow such a treatment of attributes and in the sequel we assume such an implementation.
The restricted normal form of rule functions allows us to take over the static cyclicity check of [Knu68, Knu71], since a production rule with its attribute function can be transformed into a traditional attributed production rule. Without the restriction to the normal form, e.g. by allowing the attribute function arguments to be mixed with attributes, such a static check would not be possible. Lazy functional languages offer an extra dimension of computability to attribute grammars. Where in [Knu68, Knu71], defining expressions for attributes are assumed to be strict in the attributes they contain as free variables, lazy functional languages do not impose such a restriction. If the amount of strictness of these expressions (or an approximation to that) is known, the cyclicity check can be transformed in order to take this into account. When b occurs in a defining expression for a, but such that the value of b is not needed for the computation of the value of a, the dependency of a onto b can be removed from the dependency relation. Johnsson [Joh87] exploits laziness in the same way when implementing attribute grammars in a functional language.
An example of such a non-strict dependency is the declaration of an identifier. A declaration can be modeled as an (ident, scope, meaning) triple, which models the binding of an identifier to a meaning in a certain scope. We describe this formally in section 4.7. A declaration is stored in a symbol-table, thereby extending this symbol-table with this new declaration. The actual value of meaning is irrelevant for the *storing* of the declaration. When this storing is denoted as new_table = store old_table (ident, scope, meaning), then the function store need not evaluate meaning in order to construct a new symbol-table: it can store the meaning *unevaluated* in the new table. The actual evaluation of meaning is performed when an applied occurrence of the identifier requires the retrieval

of this meaning from the symbol-table. Thus a non-strict dependency is introduced here, which can be discarded in the cyclicity check.
Of course, any attribute grammar for a programming language is full of declarations of identifiers and the dependency relation can be reduced drastically by the knowledge of non-strictness. It is exactly this property that allows us to present a standard attribution scheme for programming languages in section 4.7. This scheme prescribes the attribution of production rules for programming languages with nested scopes in a simple and elegant way. Without the availability of laziness, the scheme would introduce many cycles and each cycle should be removed in a way that is specific to the programming language under consideration. This is what frequently happens when using compiler generators based on ordered attribute grammars [KHZ81]. There, the programmer must program 'around' cycles, thereby introducing extra attributes, corrupting the structure of his attribute grammar and paying a performance price for the allocation and evaluation of these extra attributes.
In [Eng84] and [DJL88], hierarchies of attribute grammar evaluators are presented, based on the class of grammars for which the evaluator can compute the values of the attributes. Some evaluators compute attribute values at parse time and are thus very restricted in the attribute dependencies that can be allowed (e.g. L-attributed grammars). Others compute statically an evaluation order (e.g. ordered AGs), but this is too restricted as well, since the order on the attribute dependencies is a partial order and turning this partial order into a total one may introduce artificial cycles. The most powerful class described is the class of non-circular attribute grammars, which simply requires that the partial order is acyclic. But even this most powerful class of attribute grammars that is distinguished in e.g. [Eng84, DJL88] does not exploit laziness.
The confinement to statically computable evaluation orders is, however, overly restrictive. The evaluation order can also be determined dynamically and, by means of lazy evaluation, even in such a way that it is never determined explicitly, but just occurs as a result of the demand driven nature of lazy evaluation. Nevertheless, it remains possible to guarantee statically that every possible evaluation order is a-cyclic.
Therefore we introduce a new class of attribute grammars, for which we propose the name *pseudo circular attribute grammars* (PC-AG), which is the most powerful class one can obtain using an attribute evaluator that computes the attributes in an effective way. We do not give a formal definition here. Such a definition is a simple modification of the definition of non-circular attribute grammars, obtained by taking the non-strictness of semantic functions into account and thus removing certain dependencies. Johnsson in [Joh87] has observed that laziness increases the class of attribute evaluators.
Of course, one can make even more powerful evaluators by resorting to back-tracking, unification or approximation of fixed points, but each of these techniques has a considerable performance price.
As the reader might have guessed, `Elegant` allows non-strict attribute dependencies and its evaluator falls into the PC-AG class. We describe the `Elegant` attribute grammar formalism in section 4.8.

4.7 Standard attribution scheme

4.7.1 Introduction

Pseudo circular attribute grammars allow the definition of a generic and simple scheme for the construction of an attribute grammar for a programming language. The application of this scheme results in the construction of an attribute grammar that maps the input text onto a data structure representing the static semantics of the input text. If this static semantics does not exist, the scheme describes the detection of certain semantic errors and the generation of corresponding error messages.
The scheme has three major advantages. First, an attribute grammar writer does not have to reinvent the wheel over and over again. Second, attribute grammars written according to the scheme can be understood by someone who knows the scheme, but has no specific knowledge about the grammar or even the programming language under consideration. Thirdly, the scheme allows for a generic and highly efficient implementation of the symbol-table in a compiler.
The symbol-table manipulation forms an important part of the scheme forms. A symbol-table is a data-structure that is used to store defining occurrences of identifiers as declarations and to retrieve them in the case of applied occurrences of identifiers. We present a scheme for this which is very general, in the sense that the majority of the programming languages can be covered by it. To obtain this generality we will not *define* certain of the functions that we introduce below, but merely *categorize* them by giving certain relations that they must obey. These relations leave freedom in the choice of the actual functions, a freedom which is filled in in different ways by different programming languages. Examples of these differences are found e.g. in different scope rules.
[Aug90b] describes the ways in which the scheme fails for an attribute grammar for ISO Pascal. In all these cases, design flaws in the Pascal language definition appeared the cause of these failures. This experiment suggests that a failure of the scheme is an indication for an ill-defined language construct.

4.7.2 Symbol-table handling

In this section we define the notions of scope, binding and symbol-table. We do this in a general way, in order to be able to model the majority of programming languages. Some operations will not be defined exactly, but their behavior is characterized by inequalities, thereby defining properties of these, since the precise definition would be programming language dependent. Throughout this section we use the following Pascal program as an example:

```
program pascal;

var x : boolean;
    x : char;
```

```
procedure p;
var x : integer;
begin x := 3;
end;

begin
end.
```

For a given programming language, we define the following domains.

- C a finite set of *language constructs.*
  A language construct is something that we do not define formally. We can not, since it is dependent on the source language and we need not, since we are only interested in the fact that C is a finite set. A language construct corresponds to an abstraction modeled by a programming language and it usually corresponds to a non-terminal in a context-free grammar for the language. Examples are Pascal statements, expressions, an if-statement, a procedure, etc. We are not always able to identify a language construct with a non-terminal, since a language may be described by many different context-free grammars. One might say however, that in the 'most natural' context-free grammar for a language, the set of language constructs is identical to the set of non-terminals. A construct is usually composed of a number of sub-constructs.
  In our example, let $\mathsf{x}_1, \mathsf{x}_2, \mathsf{x}_3$ denote the constructs representing the three variable declarations respectively, p_1 the procedure declaration, main the main program and block the procedure block. The construct x_1 represents the properties of the first global variable `x`, like its type.

- *Id* the set of *identifiers.*
  An identifier is an object that can be *bound* to a language construct in a declaration. Within a certain region of a program, it can then be used to denote this construct. In this sense, an identifier is a syntactical pointer, with an allocation operation (the declaration, also-called the defining occurrence) and de-referencing operations, the usage (also-called the applied occurrence).

- *Ns* is the set of *name-spaces.*
  The set of declarations occurring in a program text can be partitioned into independent sub-sets, called name-spaces. Although Pascal has only two name-spaces (for regular identifiers and labels), the first name-space could be partitioned further into a name-space for types and a name-space for other objects (variables, procedures, etc.). We denote the two name-spaces of Pascal by $\mathsf{n}_{\mathsf{lab}}$ and n_{id}.

- S is the set of *scopes.*
  A scope is an (abstract) region of a program text that limits the visibility of declarations contained in it. Examples are a block and a record type definition. Usually a scope corresponds to some language construct. As we will be dealing with nested scopes only, where a scope is included within another scope (called the surrounding

scope) we can model a scope as a list of constructs, the head being the deepest, i.e. most narrow, one. Thus, the last element of this list always corresponds to the complete program text. We model S by defining $S = List(C)$.
In our example, we can identify three different scopes, which we denote by $\mathsf{s0} = \{\mathsf{main}\}$, $\mathsf{s1} = \{\mathsf{p_1, main}\}$, $\mathsf{s2} = \{\mathsf{block, p_1, main}\}$.

- B is the set of *bindings*.
A binding is an object representing a declaration. We model it by defining $B = Id \times Ns \times S \times C$. An object $(\mathsf{i, n, s, c})$ represents a declaration for identifier i within name-space n, appearing in scope s and binding construct c. The bindings in the example are denoted by:

$$\begin{aligned} b_1 &= ('x', n_{id}, s_0, x_1) \\ b_2 &= ('x', n_{id}, s_0, x_2) \\ b_3 &= ('p', n_{id}, s_0, p_1) \\ b_4 &= ('x', n_{id}, s_2, x_3) \end{aligned}$$

- T is the set of *symbol-tables*.
A symbol-table is an object that contains a sequence of bindings. Hence we model it by defining $T = List(B)$. Observe that T can not be $Set(B)$, since the order of declarations matters.
A symbol-table representing all bindings of our example is $A = \{b_1, b_2, b_3, b_4\}$.

For the domains introduced above, we define some useful operations, that we characterize by a number of equalities and in-equalities and which model the interplay between the domains. These characterizations are designed in such a way that additional properties can be added, depending on the properties of the programming language under consideration. Here we present only those properties that are common to the vast majority of programming languages.
On scopes, we define the partial order relations $<$, $\leq$ and the (in)comparability relations $\sim$, $\not\sim$.

Definition 4.12 (def:ag-scope-rel)

The relations $<, \leq, \sim, \not\sim$ on $S \times S$ are defined by:

$$\begin{aligned} s_1 < s_2 &\equiv \exists s_3 \in S, s_3 \neq \{\}, s_1 = s_3 \mathbin{+\!\!+} s_2 \\ s_1 \leq s_2 &\equiv s_1 < s_2 \vee s_1 = s_2 \\ s_1 \sim s_2 &\equiv s_1 \leq s_2 \vee s_2 \leq s_1 \\ s_1 \not\sim s_2 &\equiv \neg(s_1 \sim s_2) \end{aligned}$$

□

In our example, we have the relations $s_2 < s_1 < s_0$ and e.g. $s_2 \sim s_0$.
Furthermore, we assume the definition of a *visibility relation* $\sqsubseteq$ on scopes, that defines when the declarations in one scope are visible in another scope. This means that $s' \sqsubseteq s \equiv$ 'a declaration $(\mathsf{i, n, s, c})$ is visible in s'. This relation is programming language dependent, but we require that it lies in between the relations $=$ and $\leq$, which we can denote, maybe slightly cryptically by:

$= \subseteq \sqsubseteq \subseteq \leq$

That is, declarations in a scope are always visible in that same scope ($= \subseteq \sqsubseteq$) and declarations are at most visible in a sub-scope ($\sqsubseteq \subseteq \leq$). In Pascal, the relations $\leq$ and $\sqsubseteq$ are the same.
Furthermore, we define the notion of a *projection of a symbol-table.*
Definition 4.13 (def:ag-symtab-project)
The projection of a symbol-table t on an (identifier, name-space, scope) triple $(\mathsf{i}, \mathsf{n}, \mathsf{s})$, denoted by $\mathsf{t}\backslash(\mathsf{i}, \mathsf{n}, \mathsf{s})$, is defined by:

$$\begin{array}{lcl} \backslash & :: & T \times (Id \times Ns \times S) \mapsto T \\ \mathsf{t}\backslash(\mathsf{i},\mathsf{n},\mathsf{s}) & = & \{(\mathsf{i},\mathsf{n},\mathsf{s}',\mathsf{c}) \mid (\mathsf{i},\mathsf{n},\mathsf{s}',\mathsf{c}) \in \mathsf{t}, \mathsf{s}' \sim \mathsf{s}\} \end{array}$$

□
In our example, we have e.g. $\mathsf{A}\backslash('\mathsf{x}', \mathsf{n}_{\mathsf{id}}, \mathsf{s}_2) = \{\mathsf{b}_1, \mathsf{b}_2, \mathsf{b}_4\}$.
The fact that we assume that this operation induces a partition on symbol-tables is reflected by the definition of the following equivalence relation on symbol-tables.
Definition 4.14 (def:ag-symtab-eq)
The equivalence of two symbol-tables t_1 and t_2, denoted by $\mathsf{t}_1 \sim \mathsf{t}_2$, is defined by:

$$\begin{array}{l} \sim \subseteq T \times T \\ \mathsf{t}_1 \sim \mathsf{t}_2 \equiv \forall \mathsf{i} \in Id, \mathsf{n} \in Ns, \mathsf{s}_1, \mathsf{s}_2 \in S, \mathsf{s}_1 \sim \mathsf{s}_2 : \mathsf{t}_1\backslash(\mathsf{i},\mathsf{n},\mathsf{s}_1) = \mathsf{t}_2\backslash(\mathsf{i},\mathsf{n},\mathsf{s}_2) \end{array}$$

□
Below we also use this projection operation on a *set* of bindings, rather than on a list. It obeys the same definition as the list version above.
Having defined all these relations, we are ready to introduce the two functions retrieve and doubles. The latter maps a symbol-table onto a set of bindings, the ones that are *double declarations* in that symbol-table. When A is the symbol-table containing all bindings of a program, doubles $\mathsf{A} \neq \{\}$ implies that the program is erroneous due to double declarations. The function retrieve retrieves a set of bindings from a symbol-table for a given identifier, name-space and scope.
These functions obey the following properties, which are not definitions of these functions, but relations which they should satisfy to let our scheme work. These relations leave space for different scope rules to be implemented by our scheme.

$$\begin{array}{ll} \mathsf{retrieve} & : T \times (Id \times Ns \times S) \mapsto Set(B) \\ \mathsf{doubles} & : T \mapsto Set(B) \end{array}$$

$$\begin{array}{lcl} \mathsf{retrieve\ t\ (i,n,s)} & \subseteq & \{(\mathsf{i},\mathsf{n},\mathsf{s}',\mathsf{c}) \mid \exists \mathsf{s}',\mathsf{c} \ : \ (\mathsf{i},\mathsf{n},\mathsf{s}',\mathsf{c}) \in \mathsf{t},\ \mathsf{s} \sqsubseteq \mathsf{s}'\} \\ \mathsf{doubles}\ \{\} & = & \{\} \\ \mathsf{doubles}\ \{\mathsf{b}\} & = & \{\} \\ \multicolumn{3}{l}{\mathsf{doubles\ t1} + \mathsf{doubles\ t}_2 \subseteq \mathsf{doubles}\ (\mathsf{t}_1 + \mathsf{t}_2) \subseteq \mathsf{t}_1 + \mathsf{t}_2} \\ (\mathsf{doubles\ t})\backslash(\mathsf{i},\mathsf{n},\mathsf{s}) & = & \mathsf{doubles}\ (\mathsf{t}\backslash(\mathsf{i},\mathsf{n},\mathsf{s})) \\ \mathsf{retrieve\ t\ (i,n,s)} \cap \mathsf{doubles\ t} & = & \{\} \end{array}$$

The last property states that double declarations are not retrievable and vice versa. The subset relation in the characterization of retrieve is used to model the fact that not all bindings obeying the $\sqsubseteq$ relation are retrievable, e.g. by hiding due to local redeclarations. With regard to our Pascal example, the following identities are consistent with the above mentioned properties:

$$\begin{aligned}
&\mathsf{doubles}\ A = \{b_2\}\\
&\mathsf{retrieve}\ A\ ('x', n_{id}, s_2) = \{b_4\}\\
&\mathsf{retrieve}\ A\ ('x', n_{id}, s_0) = \{b_1\}
\end{aligned}$$

The following additional properties can easily be proved:

$$\begin{aligned}
&\mathsf{retrieve}\ t\ (i,n,s) \subseteq t \backslash (i,n,s)\\
&\mathsf{retrieve}\ t\ (i,n,s) = \mathsf{retrieve}\ (t \backslash (i,n,s))\ (i,n,s)\\
&\mathsf{retrieve}\ t = \mathsf{retrieve}\ (t - \mathsf{doubles}\ t)\\
&\mathsf{doubles}\ t \subseteq t\\
&t_1 \sim t_2 \Rightarrow \mathsf{retrieve}\ t_1 = \mathsf{retrieve}\ t_2\\
&t_1 \sim t_2 \Rightarrow \mathsf{doubles}\ t_1 = \mathsf{doubles}\ t_2
\end{aligned}$$

The last property implies that retrieve and doubles respect the equivalence relation $\sim$ on name-spaces. As we will see later, this property allows us to implement a symbol-table by its partitioning with regard to to the $\backslash$ operation, e.g. one symbol-table per (identifier, name-space) pair.
We suppose some more properties of these functions which should hold for a programming language. The first property states that when two declarations for the same identifier, name-space and scope are added, the latter of these declarations will certainly be a double declaration.

$$\{(i,n,s,c) \mid (i,n,s,c') \in t_1, (i,n,s,c) \in t_2\} \subseteq \mathsf{doubles}\ (t_1 \mathbin{+\!\!+} t_2)$$

When a declaration (i,n,s,c) is retrievable from symbol-table t while it is no longer retrievable when another declaration (i,n,s',c') is added to t, that is the first is removed from the set of valid declarations in favor of the latter, we require that the latter is declared for a larger scope than the former. This means that a local declaration can never make a global one invalid (although it may hide it in the local scope). This can be illustrated by an example assuming a Pascal-like language, where variables can be used before their declaration and where variables may not be redeclared in a more local scope.

```
procedure p ();
var x : boolean;
begin ... end;

var x : integer;
```

Here it is debatable which of the two declarations, the local, textually first one, or the global, textually later one, should be marked as a double declaration. Our scheme chooses the local one, which we formalize by:

$$(\mathsf{i},\mathsf{n},\mathsf{s},\mathsf{c}) \in \textsf{retrieve t } (\mathsf{i},\mathsf{n},\mathsf{s}'') \wedge (\mathsf{i},\mathsf{n},\mathsf{s},\mathsf{c}) \in \textsf{doubles } (\mathsf{t} \mathbin{+\!\!+} \{(\mathsf{i},\mathsf{n},\mathsf{s}',\mathsf{c}')\}) \Rightarrow \mathsf{s} < \mathsf{s}'$$

A global declaration for a scope s can only be invisible in a local scope s', when it is hidden by a more local declaration for a scope s'' in between s and s'. Formalized:

$$\begin{array}{l}(\mathsf{i},\mathsf{n},\mathsf{s},\mathsf{c}) \in \textsf{retrieve t } (\mathsf{i},\mathsf{n},\mathsf{s}) \wedge \mathsf{s}' < \mathsf{s} \wedge (\mathsf{i},\mathsf{n},\mathsf{s},\mathsf{c}) \notin \textsf{retrieve t } (\mathsf{i},\mathsf{n},\mathsf{s}') \\ \Rightarrow \exists (\mathsf{i},\mathsf{n},\mathsf{s}'',\mathsf{c}') \in \textsf{retrieve t } (\mathsf{i},\mathsf{n},\mathsf{s}') \wedge \mathsf{s}' \leq \mathsf{s}'' < \mathsf{s}\end{array}$$

For a particular programming language, additional properties over these general ones may be formulated in the same vein. The fact that these properties are formalized so easily when a symbol-table is treated as a list of declarations shows the power of this approach. We need not, however, *implement* a symbol-table in this way. E.g. a set of lists of declarations for different (identifier,name-space) pairs would also suffice. In order to be able to do so we define three more functions to abstract from a particular symbol-table implementation.

$$\begin{array}{lcl}\textsf{store } (\mathsf{i},\mathsf{n},\mathsf{s},\mathsf{c}) & = & \{(\mathsf{i},\mathsf{n},\mathsf{s},\mathsf{c})\} \\ \textsf{double t b} & = & \mathsf{b} \in \textsf{doubles t} \\ \textsf{push s c} & = & \{\mathsf{c}\} \mathbin{+\!\!+} \mathsf{s}\end{array}$$

4.8 `Elegant` attribute grammars

The notation of the `Elegant` attribute grammars closely corresponds to the normal form of rule functions. We briefly introduce this notation here and the interested reader is referred to [Aug92b, Jan93] for more details. In appendix A syntax diagrams describing the full `Elegant` syntax are listed. An `Elegant` attributed production rule defines the underlying context free production rule, the in-attributes (assuming a to be a tuple), the out-attributes (assuming $\mathsf{e_s}$ to be a tuple as well), local attributes and some rule conditions that are conjuncted to form $\mathsf{e_c}$. Each rule condition also lists an appropriate error message or warning. All attributes are typed. An EBNF syntax for the `Elegant` production rules is the following:

```
⟨root-rule⟩         ::= ROOT ⟨production-rule⟩ [ ⟨global-section⟩ ]
⟨production-rule⟩   ::= ⟨nonterminal-identifier⟩
                        ( [ IN ⟨attributes⟩ ] [ OUT ⟨expression-list⟩ ] )
                        -> { ⟨right-element⟩ }
                        [ ⟨context-conditions⟩ ]
                        [ ⟨local-section⟩ ]
⟨right-element⟩     ::= ⟨symbol⟩ ( [ IN ⟨expression-list⟩ ] [ OUT ⟨attributes⟩ ] )
⟨attributes⟩        ::= ⟨attribute-list⟩ : ⟨type-identifier⟩
                        { , ⟨attribute-list⟩ : ⟨type-identifier⟩ }
⟨attribute-list⟩    ::= ⟨attribute-identifier⟩ { , ⟨attribute-identifier⟩ }
⟨context-checks⟩    ::= CHECKS { IF ⟨check⟩ { ELSIF ⟨check⟩ } }
⟨check⟩             ::= ⟨boolean-expression⟩
                        THEN [ WARNING ] ⟨message-expression⟩
⟨global-section⟩    ::= GLOBAL { ⟨local-attribute⟩ }
⟨local-section⟩     ::= LOCAL { ⟨local-attribute⟩ }
⟨local-attribute⟩   ::= ⟨attribute-identifier⟩ : ⟨type-identifier⟩ = ⟨expression⟩
⟨symbol⟩            ::= ⟨terminal-identifier⟩ | ⟨nonterminal-identifier⟩
```

As an extension to traditional attribute grammars, global attributes have been added, which must be defined in the (unique) production rule for the start symbol. Attributes are evaluated lazily and may be passed unevaluated as an argument to a non-strict function, thus supporting the PC-AG class. After a successful parse, the context conditions are evaluated in textual order. When no syntax or context errors occur, the out-attributes of the start production rule are evaluated from left to right. Then the compiler terminates.
A simple example, taken from a Pascal attribute grammar, is the following production rule that defines an if-statement.

Example 4.15: Pascal conditional statement

```
IfStat (IN s : Scope OUT ifstat)
-> IFsym        ()
   Condition    (IN s OUT cond : Expr)
   THENsym      ()
   Statement    (IN s OUT then : Stat)
   OPTElsePart  (IN s OUT else : Stat)
LOCAL
  ifstat : Stat = CreateIfStat (cond, then, else)
;
```

□

As a more complicated example , we list the attribute grammar of section 4.2.1 in `Elegant` format.

Example 4.16: Fractions attribute grammar

```
number (OUT f) -> dotsym () frac (IN 1 OUT f : Float) ;
```

```
frac (IN i : Int OUT f)   -> digit (IN i OUT f : Float) ;
frac (IN i : Int OUT d+f) -> digit (IN i OUT d : Float)
                             frac (IN i+1 OUT f : Float) ;

digit (IN i : Int OUT v / 2^(i))
CHECKS IF i > 10 THEN "Too many digits"
-> zero (OUT v : Int) ;

digit (IN i : Int OUT v / 2^(i))
CHECKS IF i > 10 THEN "Too many digits"
-> one (OUT v : Int) ;
```

□

4.8.1 The attribution scheme

We can now present a generic scheme that describes how to write an attribute grammar for a given programming language. All examples are given in the Elegant formalism. Elegant allows the start symbol of an attribute grammar to define so called *global* variables. These are in scope in every production rule, as if they had been passed as in-attributes to each production rule. The scheme consists of two parts. The first part, which is a trivial one, introduces attributes which model the construction of the language constructs, the 'abstract syntax tree', which in our case is a (potentially cyclic) graph. The other part introduces attributes which implement the scope rules and symbol-table construction.

- The symbol-table that contains all bindings is represented by a global attribute called AllBindings of type SymbolTable. It is defined by the start symbol. It plays a prime rôle in the attribution scheme. It holds all bindings, both valid and invalid ones. The scheme we present is such that all applications of the retrieve and double functions are applied to AllBindings. This means that we never have to worry about whether a binding has already been added to a symbol-table or not. Moreover, determining the set of double bindings is most generally done on the set of all bindings, rather than on some intermediate symbol-table, storing only part of the bindings occurring in a program.
  Example 4.17: Pascal start production rule
  ```
  Start ()
  -> Program (OUT t : SymbolTable, p : Program)
  GLOBAL AllBindings : SymbolTable = t
  ;
  ```
  □
- Each language construct T is mapped onto an Elegant type T. As for proper context-free grammars, $T \in V_N$, this defines a mapping between non-terminals in the context-free grammars and types in the abstract syntax graph. Elegant offers sub-typing which is used in modeling different variants of language constructs. E.g. both an assignment and if-statement are statements, which is modeled by defining an Elegant type Stat with sub-types AssignStat and IfStat. The sub-constructs of a language construct T are modeled by fields of the type T which refer

to the sub-constructs.

- For each construct T_0 with sub-constructs $T_1 \ldots T_n$, corresponding to type T_0 with fields of types $T_1 \ldots T_n$, a function $\texttt{Create}_{T_0}$ :: $T_1 \times \ldots \times T_n \mapsto T_0$ is declared. See, for example, the `CreateIfStat` in the local-part below.
- Each production rule covering a language construct of type T has an out-attribute of type T representing that construct. Usually the sub-constructs are covered by the right-hand side of the production. If this is not the case, the corresponding sub-constructs must be passed as in-attributes to the production rule (this case can arise when the grammar is corrupted, in order to adjust it to the needs of the compiler generator (make it LL(1), LALR(1) or whatever).
  If the construct is of a type T as mentioned above, it is defined by `construct` = `Create`T (`sub`$_1$ ... `sub`$_n$). Here `sub`$_1$ ... `sub`$_n$ are the attributes representing the (semantics of) sub-constructs.
  Example 4.18: Pascal conditional statement
```
IfStat (OUT ifstat)
-> IFsym () Condition (OUT cond : Expr)
   THENsym () Statement (OUT then : Stat)
   OPTElsePart (OUT else : Stat)
LOCAL ifstat : Stat = CreateIfStat (cond, then, else) ;
```
  □
- Each production rule covering a language construct that can contain the applied occurrence of an identifier must have access to the scope in which it appears in order to be able to determine the construct bound to the identifier. Hence it has an in-attribute `s` of type `Scope` representing this scope.
  Example 4.19: Pascal conditional statement again
```
IfStat (IN s : Scope OUT ifstat)
-> IFsym () Condition (IN s OUT cond : Expr)
   THENsym () Statement (IN s OUT then : Stat)
   OPTElsePart (IN s OUT else : Stat)
LOCAL ifstat : Stat = CreateIfStat (cond, then, else) ;
```
  □
- A production rule representing the applied occurrence of an identifier `id` for name-space `N` has an out-attribute `construct`, representing the construct bound to that identifier. This attribute is defined by

```
LOCAL
  binding : Binding = RetrieveN (AllBindings, id, s)
  construct : C = GetC (binding)
```

Here `id` represents the identifier and C the type of the construct. Moreover, two context checks must be applied to check whether the identifier was bound at all (`binding` ≠ `NIL`), and to check whether the construct has the required properties for this applied occurrence (`construct` ≠ `NIL`).

Example 4.20: Pascal left hand side of assignment
```
Variable (IN s : Scope OUT var) -> Identsym (OUT id : Ident)
```

```
CHECKS
  IF binding = NIL THEN "Identifier not declared"
  ELSIF  var = NIL THEN "Variable expected"
LOCAL
  binding : IdentBinding = RetrieveIdent (AllBindings, id, s)
  var     : Variable     = GetVariable (binding)
;
```

□

- Each production rule representing a declaration which binds a construct `construct` of type C must create a binding and the symbol-table for it. Hence it has an out-attribute `t` that is defined by:

```
LOCAL
  binding : Binding = CBinding (id, s, construct)
  t : SymbolTable = Store (binding)
```

Since this can create a double declaration, a context check must also be added.

Example 4.21: Pascal constant declaration

```
ConstDeclaration (IN s : Scope OUT t, binding)
-> Identsym (OUT id : Ident)
   equalssym ()
   Constant (IN s OUT expr : Expr)
CHECKS
  IF Double (AllBindings, binding)
  THEN "Identifier already declared"
LOCAL
  binding : IdentBinding = CreateConstBinding (id, s, expr)
  t       : SymbolTable  = Store (binding)
;
```

□

- Each production rule corresponding to a construct that can contain (possibly in its sub-constructs) declarations must add the symbol-tables representing these declarations (with ++) and pass the result as an out-attribute. This is illustrated in the next example below.
- Each production rule corresponding to a construct that opens a new scope must push that construct onto its scope.

Example 4.22: Pascal procedure declaration

```
ProcDeclaration (IN s : Scope OUT t, binding)
-> PROCEDUREsym () Identsym (OUT id : Ident)
   OPTFormalParameters
         (IN s1 OUT t1 : SymbolTable, formals : List(Binding))
   semicolonsym ()
   Block (IN s1 OUT t2 : SymbolTable, block : Block)
   semicolonsym ()
CHECKS
  IF Double (AllBindings, binding)
  THEN "Identifier already declared"
  ELSIF formals # NIL AND
        HasBeenDeclaredForward (AllBindings, binding)
  THEN "Formal parameters not allowed"
LOCAL
  proc    : Proc          = CreateProc (id, formals, block)
  binding : IdentBinding  = CreateProcBinding (id, s, proc)
  s1      : Scope         = Push (s, proc)
  t       : SymbolTable   = Store (binding) ++ t1 ++ t2
;
```

□

The scope s describes the scope in which the procedure appears. The scope s1 describes the scope *within* the procedure, s1 is passed to the formal parameters and block of the procedure. These sub-constructions pass out the symbol-tables t1 and t2 that contain their respective declarations. The symbol t contains all declarations of the procedure, including that of the procedure itself.

- A scope may be extracted from a construct rather than be obtained as an attribute. This may only be done for the retrieval of identifiers visible in that scope, not for the storing of bindings. An example is the field selection in Pascal, presented below, where the scope of the record type is used to retrieve the field.

Example 4.23: Pascal field selection

```
Variable (IN s : Scope OUT var)
-> Variable (IN s OUT record : Variable) dotsym ()
   Identsym (OUT id : Ident)
CHECKS
IF ~IsRecord (type) THEN "Non-record type in field selection"
ELSIF field = NIL THEN "Field not declared"
LOCAL
  type    : Type          = TypeOf (record)
  binding : IdentBinding
          = RetrieveIdent (AllBindings, id, ScopeOf (type))
  field   : Field         = GetField (binding)
  var     : Variable      = CreateFieldVariable (record, field)
;
```

□

4.8.2 Pseudo circular grammars and lazy evaluation

The scheme above usually results in a grammar whose attributes are circularly defined, thus, in an ill-defined grammar. The 'cycles' introduced by the attribution scheme are all of the same form and can be illustrated by the example for the Pascal procedure declaration. AllBindings depends on t, which depends on binding, which depends on proc, which depends on formals and block. The block is likely to contain applied occurrences of identifiers, and hence depends on AllBindings via Retrieve.

We can observe, however, that although binding takes proc as one of its arguments of its creating function, the value of binding (that is a reference to the data-structure representing that binding) does not depend on the value of proc. We say that the function CreateProcBinding is *non-strict* in that argument. With a suitable implementation (like Elegant supplies), the non-strict arguments of a function can be passed *unevaluated*

(lazy) to it. In this way, CreateProcBinding can create a ProcBinding containing a reference to an unevaluated Proc, thereby cutting the cycle.
The cycle can be cut at other places as well. The function *CreateProc* is also non-strict in its arguments. Cutting the dependencies for those arguments that depend on declarations, in this case formals and block, also removes the cycle. The latter place for cutting the cycle is more favorable to overloading in a language and therefore we adopt it as a standard in our scheme.
The example is thus modified by explicitly indicating lazy evaluation:

Example 4.24: Pascal procedure declaration

```
ProcDeclaration (IN s : Scope OUT t, binding)
-> PROCEDUREsym () Identsym (OUT id : Ident)
   OPTFormalParameters
                 (IN s1 OUT t1 : SymbolTable, formals : List(Binding))
   semicolonsym ()
   Block (IN s1 OUT t2 : SymbolTable, block : Block)
   semicolonsym ()
CHECKS
IF Double (AllBindings,binding)
THEN "Identifier already declared"
ELSIF (formals # NIL) AND
      HasBeenDeclaredForward (binding, AllBindings)
THEN "Formal parameters not allowed"
LOCAL
  proc    : Proc          = CreateProc (id, LAZY(formals), LAZY(block))
  binding : IdentBinding  = CreateProcBinding (id, s, proc)
  s1      : Scope         = Push (s, proc)
  t       : SymbolTable   = Store (binding) ++ t1 ++ t2
;
```

□

Since this cycle-cutting results in the creation of data structures containing unevaluated attributes, the typing of these data structures must reflect this. To this end, the type of the fields formals and block in a Proc are so-called *lazy* types, representing this non-evaluatedness. The Elegant prelude defines the parameterized type Lazy(T) for this purpose.
For the class of attribute grammars supporting using our scheme and cycle-cutting technique, we did propose the term *pseudo circular* attribute grammar (PC–AG), which means that the circularity disappears when the non-strictness of the defining functions for the attributes is taken into account. It can be shown that NC–AG $\subset$ PC–AG. NC–AG stands for non-circular AG [DJL88], the most general class of attribute grammars described so far. An important distinction between these two classes is that a pseudo circular attribute grammar, using a non-lazy functional language for its attribute definitions, is capable of constructing a cyclic abstract syntax graph, in contrast to a non-circular attribute grammar, which can at best create a DAG.
Elegant will report the 'real' cycles, so a user can design its attribute grammar in two steps: first use the scheme presented in this section and then cut the reported cycles in the way explained here.
Although lazy evaluation poses some overhead on the evaluation of each attribute when

compared to ordered attribute evaluation, it has two advantages concerning its efficiency:

- When an attribute value is computed, its defining expression must be evaluated. Since a single synthesized attribute of a non-terminal symbol is defined by different expressions in the different production rules for the non-terminal, the proper defining expression must be selected first. Lazy evaluation offers the ability to bind the expression to the attribute at parse time, that is, when the proper production rule is selected. Systems which do not exploit lazy evaluation often need to interpret a data structure in order to decide which expression to evaluate for the attribute. This interpretation can impose a considerable overhead on attribute evaluation when occurring frequently.
- Far less attributes need to be evaluated, since the attribution scheme leads to such concise attribute grammars. In fact, we believe these grammars to be minimal, since the attribution scheme only introduces attributes which are necessary to map the syntactical representation of the program onto an abstract syntax graph and does not introduce attributes whose sole function is to avoid 'cyclic' dependencies.

Since no separate attributes are needed to avoid the occurrence of cycles, pseudo circular attribute grammars are not just more concise than ordered ones, but also much more clear, since the cycle avoiding attributes obscure the attribute grammar and add unnecessary 'noise' to the AG, deteriorating its readability.

4.8.3 Overloading

In the case of overloading (e.g. of functions) the scheme has to be generalized. In order to see this, assume that procedure overloading has to be supported. In this case, the well-formedness of a procedure declaration (e.g. the fact whether it is a double declaration or not, and hence its storing in the symbol-table) depends on the types of the procedure arguments, and hence on the corresponding type declarations. Consequently it depends on AllBindings, thereby creating a cycle. This is caused by the fact that the `Store` operation is no longer non-strict in the meaning of the declaration to be stored. It evaluates certain components of the meaning (e.g. the type of the meaning) in order to store the declaration. This cycle can be avoided by introducing more than one symbol-table to retrieve from. If in this way an ordering on symbol-tables can be obtained, such that bindings in the i-th symbol-table can only be overloaded by using information from the j-th symbol-table, $j < i$, no cycle occurs.
In the example of procedure overloading, an additional symbol-table AllTypeBindings can be created to remove the cycle. Thus, AllTypeBindings is located before AllBindings in the ordering relation. Although this may seem cumbersome, one should keep in mind that all cycles are detected statically and can be removed after the completion of the attribute grammar, when they are reported by `Elegant`.

4.8.4 Semantic cycles

To complete our discussion of various forms of cycles we present a third form of cycles in this section that can not be cut. We call these cycles *semantic cycles* because they either correspond to semantic errors in the program text or to some semantic value that needs to be computed by the compiler, but which is defined by some fixed point giving rise to a cyclic attribution. These cases can be illustrated by the example of a cyclic constant declaration, say `CONST x = x + 1;` in a Pascal-like language. If the compiler needs the *value* of the constant (e.g. to perform constant folding), it is in trouble, since this value is undefined. Such undefined semantic constructs correspond to a cycle in the grammar. Such a cycle can not be cut, since the error can not be removed: it is simply there.
Instead of cutting it, we can *detect* it and in the mean time prevent the occurrence of cyclic attribute evaluation. The technique used here depends on side-effects. Each construct that may be ill-defined in this way, is encapsulated in a special type, called `Cautious` that is able to detect the cycle. This type is predefined in the `Elegant` prelude as follows:

```
SPEC UNIT CautiousPrelude

TYPE CautiousState = (UnEval | UnderEval | Evaluated | Cyclic)

     Cautious(X) = ROOT,   value : Lazy(X),
                           state : CautiousState = UnEval

RULES

Cautious:[x : Lazy(?T)] : Cautious(T)
IsCyclic:[x : Cautious(?T)] : Bool
Value:[x : Cautious(?T), error : ?T] : T
```

The production rule for a constant definition may be modified as follows:

```
  ConstDeclaration (IN s : Scope OUT t, binding)
  -> Identsym (OUT id : Ident) equalssym () Constant (IN s OUT expr : Expr)
  CHECKS
    IF Double (AllBindings, binding) THEN "Identifier already declared"
    ELSIF IsCyclic (cautious) THEN "Cyclic constant definition"
  LOCAL
    cautious : Cautious(Expr) = Cautious (LAZY(expr))
    binding : IdentBinding = CreateConstBinding (id, s, cautious)
    t       : SymbolTable  = Store (binding)
  ;
```

The idea is to create a cautious object, storing the unevaluated attribute representing the ill-defined construct and to bind the corresponding declaration not to the construct but to the encapsulated construct.
When evaluating the attribute representing the ill-defined construct, the cycle will retrieve the cautious object and obtain the ill-defined value from it. This evaluation will transform the state of the cautious object from `UnEval` to `UnderEval` and the `value` field is

evaluated. In the case of a cycle, the cautious object will be re-evaluated in a state `UnderEval`. In this case the state becomes `Cyclic` and no further evaluation of the `value` field takes place.
If the evaluation of the `value` field terminates in a state that is not `Cyclic`, the state becomes `Evaluated` and no error has occurred.
Hence, the possible state transitions are:

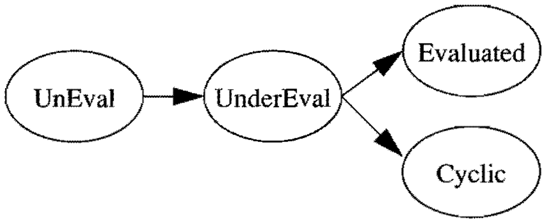


Figure 4.1: Cautious state transitions

4.8.5 Implementing the symbol-table by side-effects

Apart from the properties already explained, the attribution scheme that we presented has another advantage that we explain in this section.
There are four operations defined on a symbol-table, namely store that creates one, $+\!\!+$ that combines them, retrieve that retrieves declarations and double that detects double declarations. The latter two operations operate on AllBindings, that is, on the complete symbol-table that holds all declarations in the entire program. Furthermore, all $+\!\!+$ operations are connected in a tree-like fashion, in such a way that each intermediate symbol-table is used exactly once, namely as an argument of $+\!\!+$. Since there is no other reference to such an intermediate symbol-table, we are free to implement these symbol-tables by means of side-effects upon a single global symbol-table, since this is *semantically transparent*. Thus the $+\!\!+$ operator becomes nothing but the combination of two side-effects into one. And the store operation can be implemented as the side-effect that extends the global symbol-table with a single binding.
This implementation of a single global symbol-table enables the implementation of a very efficient, yet semantically clean symbol-table.
In `Elegant`, so-called *empty* types are supported, that can be used to represent side-effects. Thus the symbol-table type is simply defined as follows, where the `SymTab` type is explained below.

```
TYPE SymbolTable = EMPTY
     SymTab = ROOT
```

A retrieval operation has AllBindings as a strict argument, hence forcing the evaluation

of this symbol-table, which depends on all declarations, thereby evaluating all symbol-table updates before the first retrieval. Furthermore, laziness guarantees that AllBindings is evaluated at most once, namely upon the first retrieval (or double test), thus guaranteeing that each declaration is stored exactly once before the first retrieval. In `Elegant`, this is however only true if AllBindings is of a non-empty type, say of type `SymTab`, since attributes of an empty type can be re-evaluated many times, that is, they are implemented by demand-driven evaluation rather than by lazy evaluation. See section 6.2.10 and 6.5 for a discussion about the different forms of evaluation in `Elegant`.
The start symbol can thus define AllBindings by:

```
GLOBAL AllBindings : SymTab = EvalDeclarations:[LAZY!(t)]
```

where `t` is the symbol-table attribute that represents the side-effect of declaring all bindings, and which is available in the start symbol production rule, as guaranteed by the attribution scheme. The exclamation mark is a hint for the `Elegant` attribute grammar cyclicity checker that although `AllBindings` does only use the *unevaluated* `t`, nevertheless, a dependency between them should be taken into account.
Although the operational semantics of this game may seem complicated and far from straightforward, the scheme guarantees that it is semantically completely equivalent to a side-effect free implementation. In practice however, it is hard to make users not worry about the operational behavior and let them just apply the scheme. Most programmers need an operational model to comfort them!

4.8.6 Implementation scheme for a symbol-table

We now present an implementation technique for a global symbol-table that is updated by side-effects.
Recall that the symbol-table is a mapping from an identifier, name-space and scope onto a set of bindings. Thus, the retrieve operation must project the symbol-table on the identifier, name-space and scope respectively.
The implementation that we present here is perhaps counter-intuitive. The reader might expect to store a list of declarations per scope, thus first projecting on the scope when retrieving a binding. A more efficient implementation is obtained by projecting first on the identifier and name-space and only then on the scope.
A suitable implementation for this is depicted below.
The built-in type Ident is a hashed string, which means that two textually identical identifiers are mapped onto the same pointer, which is an element of the type Ident.
An Ident points to a record, containing the size and the representation of the identifier. Moreover, it contains a field called info, which is capable of storing an arbitrary pointer. Our symbol-table implementation scheme now prescribes that this pointer refers to all bindings for that identifier in the whole program. Thus, within constant time, the projection on the identifier can be performed. In this way, the symbol-table is partitioned into different smaller symbol-tables per identifier, stored at the identifier's info field.
The second projection, on the name-space is also implemented very efficiently. Since we can partition the declarations into different name-spaces and since the name-spaces are

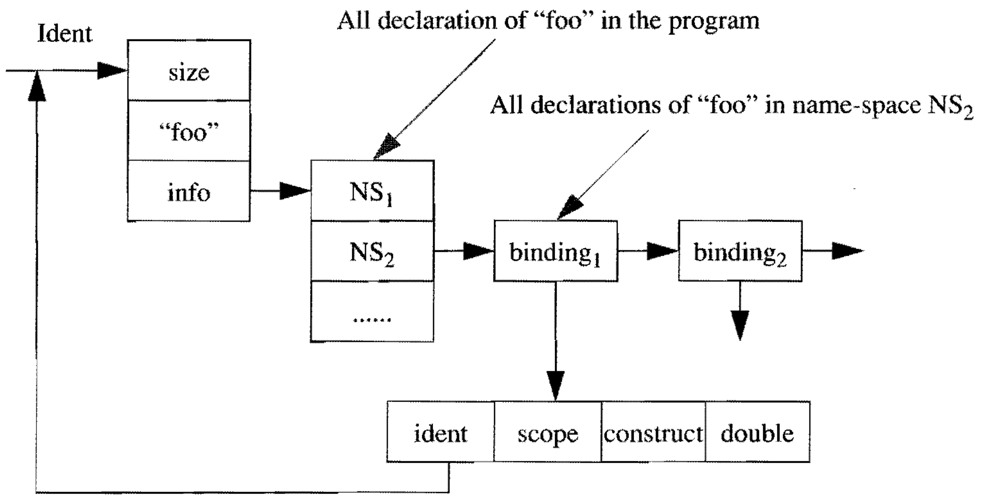

Ident
size
"foo"
info
All declaration of "foo" in the program
All declarations of "foo" in name-space NS2
NS1
NS2
......
binding1
binding2
ident
scope
construct
double

Figure 4.2: Symbol-table structure

statically known (they are determined by the programming language, not by a particular program), and since we have one retrieve operation per name-space, we can use a record with a field per name-space, where each field refers to all declarations for the given identifier and the given name-space. Thus, the second projection can also be performed in constant time. Since the number of name-spaces is usually very limited, this scheme does not introduce a great amount of space overhead. In this way, the symbol-table is partioned further into even smaller symbol-tables per identifier and name-space.
The third and last projection, on the scope, can be performed in at least (or should we say most?) linear time (in the number of declarations for each identifier/name-space pair) by implementing the remaining set of bindings as a linear list. The retrieve operations selects those bindings whose scope component is compatible with the current scope, according to the scope rules of the language under consideration. By using a binary tree instead of a linear list (assuming some ordering on scopes), or by exploiting the structure of the scopes (e.g. first projecting on the current module and then on the current procedure), even more efficient implementations can be designed. In practice however, it is not worth the trouble and a linear list suffices.

4.8.7 Elegant implementation scheme for a symbol-table

In this section we present an implementation for the scheme presented above in `Elegant`. First we list part of the relevant `Elegant` syntax. For a full definition of the `Elegant` language, see [Aug92b, Jan93]. In appendix A syntax diagrams describing the full `Elegant` syntax are listed. Readers not interested in this implementation can savely skip this

section.
`Elegant` type-definitions have the following forms, defining a root-type, a sub-type, an enumerated type, an empty type and an alias type, respectively. The `Elegant` type system is treated in detail in chapter 6.

```
⟨type-section⟩       ::= TYPE { ⟨type-declaration⟩ }

⟨type-declaration⟩   ::= ⟨root-type⟩
                       | ⟨sub-type⟩
                       | ⟨enumerated-type⟩
                       | ⟨empty-type⟩
                       | ⟨alias-type⟩

⟨root-type⟩          ::= [ ABSTRACT | MEMO ]
                         ⟨ident⟩ = ROOT { , ⟨field-declaration⟩ }

⟨sub-type⟩           ::= [ ABSTRACT | MEMO ]
                         ⟨ident⟩ < ⟨record-type-ident⟩ { , ⟨field-declaration⟩ }

⟨enumerated-type⟩    ::= ⟨ident⟩ = ( [ ⟨ident⟩ { | ⟨ident⟩ } ] )

⟨empty-type⟩         ::= ⟨ident⟩ = EMPTY

⟨alias-type⟩         ::= ⟨ident⟩ == ⟨type⟩
```

An `Elegant` function is declared to be a sequence of partial functions, called *rules* which are selected by pattern matching. The patterns are the formal types of the rule parameters, which in general are sub-types of the function corresponding to the rule. When the actual arguments are of these sub-types, the textually first of the matching rules is selected. Apart from matching on argument types, arbitrary boolean conditions over the arguments may be specified.

```
⟨rule-section⟩ ::= RULES { ⟨rule⟩ }

⟨rule⟩         ::= [ ⟨ident⟩ :   | ⟨operator⟩ ]
                   [ ⟨formal-parameters⟩ ] [ :   ⟨type⟩ ]
                   [ CONDITIONS ⟨condition⟩ { , ⟨condition⟩ } ]
                   [ LOCAL { ⟨ident⟩ :   ⟨type⟩ = ⟨expression⟩ } ]
                   -> { ⟨statement⟩ } ;

⟨formal-parameters⟩ ::= ⟨formal-parameter⟩ { , ⟨formal-parameter⟩ }

⟨formal-parameter⟩  ::= [ VAR ] ⟨ident⟩ :   ⟨type⟩
```

The following types in functions implement the symbol-table handling in `Elegant`:

```
TYPE
(****** The symbol-table types ******)

  SymbolTable = EMPTY
  SymTab = ROOT

(* The type NameSpaces is referred to by id.info.
   NS1, etc. are short-hands.
*)

  NS1 == List (NS1_Binding)
  NS2 == List (NS2_Binding)
  .....

  NameSpaces = ROOT,            ns1 : NS1 = NIL,
                                ns2 : NS2 = NIL,
                                .....

(* The bindings are partitioned into different name-spaces.
   Within each name-space, different bindings can be stored,
   for different constructs.
   These are implemented as sub-types of the name-space binding type.
*)
  ABSTRACT
  Binding = ROOT,              id : Ident,
                               scope : Scope,
                               double : Bool = FALSE
  ABSTRACT
  NS1_Binding < Binding
  C1_Binding  < NS1_Binding,   construct : C1
  C2_Binding  < NS1_Binding,   construct : C2
  .....

  ABSTRACT
  NS2_Binding < Binding
  D1_Binding  < NS2_Binding,   construct : D1
  D2_Binding  < NS2_Binding,   construct : D2
  .....


RULES
(******* The symbol-table operations *****)

(* Retrieve and create the name-spaces for an identifier *)

RetrieveNameSpaces:[id : Ident] : NameSpaces -> RETURN (id.info) ;

CreateNameSpaces:[id : Ident] : NameSpaces
-> IF id.info = NIL THEN id.info := NameSpaces:[] FI
   RETURN (id.info) ;

(* The function '++' combines two side-effects and is built-in *)

(* The function 'Store' has one rule for each name-space *)
```

```
Store:[b : NS1_Binding] : SymbolTable
LOCAL n : NameSpaces = CreateNameSpaces:[b.id]
-> n.ns1 :+ b ;

Store:[b : NS2_Binding] : SymbolTable
LOCAL n : NameSpaces = CreateNameSpaces:[b.id]
-> n.ns2 :+ b ;

.....

(* The function ':+' is defined for each name-space.
   It extends the list of bindings with a new one, detecting double
   declarations.
   The types NS1 and NS2 are *not* comparable and we thus have one function
   per name-space, which is implemented by the following three rules.
   We assume that the operator ~~ on scopes implements the scope rules
   and that the operator ~~ on constructs implements overloading.
   In the absence of overloading it defaults to the constant function TRUE.
*)

:+ [VAR l=NIL : NS1, b : NS1_Binding]
-> l := {b} ;

:+ [VAR l : (NS1), b : NS1_Binding]
CONDITIONS b.scope ~~ l.head.scope,
           b.construct ~~ l.head.construct
-> b.double := TRUE ;

:+ [VAR l : (NS1), b : NS_1Binding]
-> l.tail :+ b ;

.....

(* The function 'Retrieve_NSi' is defined for each name-space 'NSi'.
   It retrieves all bindings for the given identifier and the given
   name-space that match the current scope.
   The argument 'all' equals 'AllBindings' and is passed for synchronization
   purposes only!
   We assume that a function '<=' is defined on scopes which implements the
   scope rules.
   In the iteration, first the name-spaces for 'id' are retrieved and matched
   against NIL by means of the typing '(NameSpaces)'.
   From the 'nsi' name-space, those bindings 'b' are retrieved that
   match the scope 's'.
*)

Retrieve_NS1:[all : SymTab, id : Ident, s : Scope] : NS1
-> RETURN ({ b | ns : (NameSpaces) = RetrieveNameSpaces:[id],
                 b : NS1_Binding <- ns.ns1, (s <= b.scope) }) ;

.....

(* The function 'Double' is defined on all bindings.
   It tests a binding for double declaredness.
   The argument 'all' is passed for synchronization only!
*)

Double:[all : SymTab, b : Binding] : Bool -> RETURN (b.double) ;

(* For each construct 'Ci' there is a projection function from a binding
   onto the construct.
*)

GetC1:[b : (C1_Binding)] : C1 -> RETURN (b.construct) ;
GetC1:[b : Binding     ] : C1 -> RETURN (NIL) ;
```

4.9 Implementation of Elegant attribute grammars

In this section we present a scheme for the implementation of an attributed LL(1) parser for a grammar that is in the normal form of section 4.5. We recapitulate part of that parser first, assuming multiple inherited and synthesized attributes per production rule, multiple context conditions $e_{c_1} \ldots e_{c_p}$ and adding local attributes:

$$
\begin{array}{l}
\text{parse\_attr } i_0\ (X_0 \rightarrow X_1 \ldots X_n)\ a_1 \ldots a_{k_0} \ = \ (i_n, c_0, e_1 \ldots e_{m_0}) \\
\textbf{where } c_0 \ = \ e_{c_1} \wedge \ldots \wedge e_{c_p} \wedge c_1 \wedge \ldots \wedge c_n \\
\qquad (i_1, c_1, s_{11} \ldots s_{1k_1}) = \text{parse\_attr } i_0\ X_1\ e_{11} \ldots e_{1m_1} \\
\qquad \ldots \\
\qquad (i_n, c_n, s_{n1} \ldots s_{nk_n}) = \text{parse\_attr } i_{n-1}\ X_n\ e_{n1} \ldots e_{nm_n} \\
\qquad l_1 \ = \ e_{l_1} \\
\qquad \ldots \\
\qquad l_k \ = \ e_{l_k}
\end{array}
$$

Such a function corresponds to the following Elegant production rule (see section 4.8), where we have left out the types for conciseness. With each context condition e_{c_i} an error message m_i is associated.

$$
\begin{array}{l}
X_0\ (\text{IN } a_1 \ldots a_{k_0} \text{ OUT } e_1 \ldots e_{m_0}) \\
\rightarrow X_1\ (\text{IN } e_{11} \ldots e_{1m_1} \text{ OUT } s_{11} \ldots s_{1k_1}) \\
\quad \ldots \\
\quad X_n\ (\text{IN } e_{n1} \ldots e_{nm_n} \text{ OUT } s_{n1} \ldots s_{nk_n}) \\
\quad \textbf{checks if } e_{c1} \textbf{ then } m_1 \\
\qquad\qquad \ldots \\
\qquad\qquad \textbf{if } e_{cp} \textbf{ then } m_p \\
\quad \textbf{local } l_1 \ = \ e_{l_1} \\
\qquad\quad \ldots \\
\qquad\quad l_k \ = \ e_{l_k} \\
;
\end{array}
$$

The parsing function for a non-terminal just calls the parsing function for the appropriate production rule after inspecting the first symbol of the remainder of the input string, passing all inherited and synthesized attributes. We do not repeat it here.

As in [Kat84], we associate a procedure with each attribute. Our scheme is naturally suited for the class of pseudo-circular attribute grammars, where in [Kat84], rather complicated grammar transformations are required to support non-circular attribute grammars.
In our scheme, each of the attributes is be compiled into an instance of a lazy type, that can be defined as follows in a non-lazy Pascal-like language with polymorphic types:

```
type Lazy(T) = record env : address;
                      val : T;
                      eval : proc (Lazy(T)) : T;
               end;
```

An instance of type Lazy(T) is a triple (env, val, eval) and it represents a possibly unevaluated expression e of type T. When the expression has not been evaluated yet, the *environment* env is a pointer to a record containing all the free variables in e. It is represented by the generic pointer type address. A lazy object a can be evaluated by the expression a.eval(a). The *evaluation function* eval computes the value of e, making use of the values of the free variables that it finds in a.env. It then assigns the result to a.val. Since subsequent evaluations need not recompute e, at the end of the evaluation process the identity function id_T is assigned to a.eval:

```
proc id_T (a : Lazy(T)) : T;
begin return (a.val);
end;
```

Thus the evaluation function F_e for expression e is given by:

```
proc F_e (a : Lazy(T)) : T;
var env : Env_e;
begin env := a.env;
      x.val := e[v ← env.v];     for each free variables v in e
      x.eval := id_T;
      return (x.val);
end;
```

and a lazy object is created with the function:

```
proc lazy (e : address, f : (Lazy(T)) : T) : Lazy(T);
var a : Lazy(T);
begin new(a);
      a.env := e;
      a.eval := f;
      return (a);
end;
```

So far for the implementation of lazy evaluation. When compiling the attributed production rules as defined above, we can observe that all attributes for that rule can share the same environment. This environment is defined as follows, leaving out the types which are all lazy ones:

```
type Env = record $a_1 \ldots a_{k_0}$;
                  $s_{11} \ldots s_{nk_n}$;
                  $l_1 \ldots l_k$;
           end;
```

The LL(1) recursive descent parsing function can be extended in the following way with attribute allocation and initialization, assuming that the production rule is called p. It is passed the (unevaluated) inherited attributes as value-parameters and delivers the (unevaluated) synthesized attributes in var-parameters. It first allocates the environment e, subsequently calls the parsing functions for the right-hand side, providing these with the proper inherited attributes and obtaining from them the proper synthesized attributes. Finally it creates its own synthesized attributes and local attributes.

```
proc Parse$_p$ ($a_1 \ldots a_{k_0}$, var out$_1 \ldots$ out$_{m_0}$);
var e : Env;
begin new(e);
      Parse$_{X_1}$ (lazy(e, $F_{e_{11}}$)... lazy(e, $F_{e_{1m_1}}$), e.$s_{11} \ldots$ e.$s_{1k_1}$);
      ...
      Parse$_{X_n}$ (lazy(e, $F_{e_{n1}}$)... lazy(e, $F_{e_{nm_n}}$), e.$s_{n1} \ldots$ e.$s_{nk_n}$);
      out$_1$ := lazy(e, $F_{e_1}$);...; out$_{m_0}$ := lazy(e, $F_{e_{m_0}}$);
      e.$l_1$ := lazy(e, $F_{e_{l_1}}$);...; e.$l_p$ := lazy(e, $F_{e_{l_p}}$);
end;
```

This parser does not construct a syntax tree, nor does it compose attribute functions, it rather constructs a data-flow graph for the attributes, which can be evaluated in a demand-driven fashion.
The parsing function for a non-terminal just calls the parsing function for the appropriate production rule after inspecting the first symbol of the remainder of the input string, passing all inherited and synthesized attributes. We assume that the input string is represented by a global variable i.

```
proc Parse$_A$ ($a_1 \ldots a_{k_0}$, var out$_1 \ldots$ out$_{m_0}$);
var e : Env;
begin if hd (i) $\in$ first($\alpha_1$) then Parse$_{A \to \alpha_1}$ ($a_1 \ldots a_{k_0}$, out$_1 \ldots$ out$_{m_0}$);
      ...
      else if hd (i) $\in$ first($\alpha_n$) then Parse$_{A \to \alpha_n}$ ($a_1 \ldots a_{k_0}$, out$_1 \ldots$ out$_{m_0}$);
      else some error recovery when A $\overset{*}{\not\to}$ $\varepsilon$
      fi

end;
```

The parsing function for a terminal symbol x with attribute a is simple and given by:

```
proc parse$_x$ (var l : Lazy(T));
begin if x = head (i)
      then l := lazy(nil, id$_T$);
           l.val := a;
           i := tail (i);
      else some error recovery
      fi ;
end;
```

4.9.1 Optimizations

The compilation scheme for attributed grammars allows for numerous space and time optimizations. We list a few of these:

- The env and val field of an attribute can be overlaid, since once the value is computed, the environment is no longer needed.
- Only attributes that are used in the definition of other attributes need be stored in the environment.
- Identical attributes may be shared between production rules.
- Instead of allocating attributes separately, they are incorporated as records rather than as pointers in the environment and hence allocated as part of the environment. The pointer is obtained by an up-reference operation, i.e. the `ADR` operation in Modula-2 or the `&` operation in C (Pascal lacks it).
- If an attribute a is defined by a function over only one other attribute b in production rule P, the environment for a is made identical to (a pointer to) the attribute b, rather than the whole environment of P. Hence b need no longer be part of the environment of P on behalf of a, and maybe not at all.
- If an expression e is the only expression which depends on a *local* attribute a in a production rule P, a need not be part of the environment of P, but can be a local variable of F_e of a non-lazy type, whose value is computed before the value of e within F_e. This is not possible when e is not strict in a.

These and other optimizations have been incorporated into `Elegant`, resulting in a very efficient implementation of lazy attribute grammars. The performance is presented and analyzed in chapter 8.3.

Chapter 5

Functional scanners

5.1 Introduction

Since long, regular expressions have been the formalism of choice for the specification of lexical scanners, see e.g. [ASU86]. In a lexical scanner specification, a regular expression over a certain character set is used to define the valid representations of a terminal symbol, each representation being a sequence of characters. A set of regular expressions indexed by the terminal symbols of a (context-free) grammar can be directly mapped onto a non-deterministic finite automaton (NFA) [ASU86]. This automaton can be transformed to make it first deterministic (if possible) and subsequently minimal.
In this chapter, we show that we can associate a recursive function with a lexical scanner specification that recognizes the terminal symbols. We call this function a *scanner*. This recursive function can be transformed into a set of recursive equations that is isomorphic to the corresponding NFA. This set of equations is obtained by transformations that increase the level of abstraction of the expressions under consideration. It is not the correspondence to the NFA that inspires these transformations, but the desire for a higher abstraction level of the form of the equations. The most important of these transformations is (again) the introduction of continuations. The resulting set of equations contains a construction that can be interpreted as a non-deterministic choice. For reasons of efficiency, it is desirable to eliminate this non-determinism. This is done again by means of program transformations. The transformations are inspired by the wish to increase determinism. Minimality is already present in the set of equations and maintained while obtaining determinism.
A major advantage of these techniques over the use of automata is the simplicity of adding attributes. The set of equations is not even modified by the addition of attributes and their evaluation. The result is a smooth integration of an attributed scanner and parser, both expressed as functions in a functional language and both obtained by program transformations.

5.2 Preliminaries

In this section we show how a *scanner specification* can be used to describe the mapping of a sequence of characters onto a sequence of terminal symbols. To this end, such a scanner specification associates a *regular expression* with each terminal symbol. A regular expression generates a language over characters, its *representations*. If a character string is in the language of a regular expression, we say that the regular expression *derives* that character string. These notions are defined formally as follows, assuming that the $*$ and $+$ take precedence over juxtaposition, which takes precedence over $|$.

Definition 5.1 (def:RE)

The set of *regular expressions* RE_C over an alphabet C is the smallest set satisfying:

$$\begin{array}{ll} C \subseteq RE_C & \\ \varepsilon \in RE_C & \\ \alpha, \beta \in RE_C & \Rightarrow \alpha\beta \in RE_C \\ \alpha, \beta \in RE_C & \Rightarrow \alpha \mid \beta \in RE_C \\ \alpha \in RE_C & \Rightarrow [\alpha] \in RE_C \\ \alpha \in RE_C & \Rightarrow \alpha^* \in RE_C \\ \alpha \in RE_C & \Rightarrow \alpha^+ \in RE_C \end{array}$$

□

Here, $\alpha\beta$ denotes α followed by β, $\alpha \mid \beta$ denotes either α or β, $[\alpha]$ denotes zero or one occurrences of α, α^* denotes zero or more occurrences of α and α^+ denotes one or more occurrences of α.

Definition 5.2 (def:scanner-spec)

A *scanner specification* is a tuple (C, T, R) where

- C is a finite *character alphabet*.
- T is a finite *terminal alphabet*, $C \cap T = \{\}$.
- R is a finite set of *scanner rules*, $R \subset T \times RE_C$.
  When $(x, \alpha) \in R$ we write $x \rightarrow \alpha$. We will use this form as a qualifier in comprehensions in the sequel.

□

Definition 5.3 (def:scan-derive)

A regular expression α can *derive* a regular expression β, denoted by $\alpha \overset{*}{\rightarrow} \beta$. This relation is defined as the smallest relation satisfying:

$$\begin{array}{l} \alpha \overset{*}{\rightarrow} \alpha \\ \varepsilon\alpha \overset{*}{\rightarrow} \alpha \\ \alpha\varepsilon \overset{*}{\rightarrow} \alpha \\ \alpha \overset{*}{\rightarrow} \mu, \beta \overset{*}{\rightarrow} \eta \Rightarrow \alpha\beta \overset{*}{\rightarrow} \mu\eta \\ \alpha \mid \beta \overset{*}{\rightarrow} \alpha \end{array}$$

$$\alpha \mid \beta \overset{*}{\rightarrow} \beta$$
$$[\alpha] \overset{*}{\rightarrow} \varepsilon \mid \alpha$$
$$\alpha^* \overset{*}{\rightarrow} \varepsilon \mid \alpha^+$$
$$\alpha^+ \overset{*}{\rightarrow} \alpha(\alpha^*)$$

□

This definition of derivation gives the semantics to the different constructions of regular expressions.

Definition 5.4 (def:RE-lang)

The *language* $\mathcal{L}_\alpha$ of a regular expression α is defined by $\mathcal{L}_\alpha = \{\omega \mid \exists\omega \in C^*, \alpha \overset{*}{\rightarrow} \omega\}$.

□

The language of a context-free grammar consists of a set of strings of terminal symbols. By concatenating elements of the languages of the regular expressions corresponding to these terminal symbols, another language (in terms of characters rather than terminals) for the CFG is obtained. This language is a subset of the language of a scanner specification:

Definition 5.5 (def:scan-lang)

The *language* $\mathcal{L}_S$ of a scanner specification $s = (C, T, R)$ is defined by $\{\omega \mid \exists(t_1,\alpha_1)\ldots(t_n,\alpha_n) \in R, n \geq 0,\ \omega \in C^*\ :\ (\alpha_1\ldots\alpha_n) \overset{*}{\rightarrow} \omega\}$.

□

5.3 Recursive descent scanning

Let $S = (C,R,T)$ be a scanner specification. In the sequel, all definitions will be given relative to this S. A *scanner* is a function that given an input string i returns pairs (t, j), such there exists an α which with $(t,\alpha) \in R$ and αj derives i. It obeys the following specification:

Specification 5.6 (spec:scanner-rd)

$$\text{scanner} :: RE_C \times C^* \mapsto Set(T \times C^*)$$

$$\text{scanner i} = \{(t,j) \mid \exists j\ :\ t \rightarrow \alpha, \alpha j \overset{*}{\rightarrow} i\}$$

□

A straightforward recursive descent implementation can be based on the function scan which obeys the specification:

Specification 5.7 (spec:scan-rd)

$$\text{scan} :: RE_C \times C^* \mapsto Set(C^*)$$

$$\text{scan}\ \alpha\ \text{i} = \{j \mid \exists j\ :\ \alpha j \overset{*}{\rightarrow} i\}$$

□

The implementations of these functions are:

Implementation 5.8 (of specification (5.6)) (scanner-rd)

$$\mathsf{scanner}\ i = \{(t,j) \mid t \rightarrow \alpha,\ j \in \mathsf{scan}\ \alpha\ i\}$$

□

Implementation 5.9 (of specification (5.7)) (scan-rd)

$$\begin{array}{ll}
\mathsf{scan}\ \varepsilon\ i & = \{i\} \\
\mathsf{scan}\ x\ i & = \{j \mid x : j \leftarrow i\} \\
\mathsf{scan}\ (\alpha\beta)\ i & = \{k \mid j \in \mathsf{scan}\ \alpha\ i,\ k \in \mathsf{scan}\ \beta\ j\} \\
\mathsf{scan}\ (\alpha \mid \beta)\ i & = \mathsf{scan}\ \alpha\ i + \mathsf{scan}\ \beta\ i \\
\mathsf{scan}\ [\alpha]\ i & = \{i\} + \mathsf{scan}\ \alpha\ i \\
\mathsf{scan}\ \alpha^*\ i & = \{i\} + \mathsf{scan}\ \alpha^+\ i \\
\mathsf{scan}\ \alpha^+\ i & = \{k \mid j \in \mathsf{scan}\ \alpha\ i,\ k \in \mathsf{scan}\ \alpha^*\ j\}
\end{array}$$

□

All possible scans are given by applying the following function scanall to an input string:

$$\begin{array}{ll}
\mathsf{scanall}\ \varepsilon & = \{\} \\
\mathsf{scanall}\ i & = \{t : \mathsf{scanall}\ j \mid (t,j) \in \mathsf{scanner}\ i\}
\end{array}$$

5.4 Continuation-based scanning

The recursive descent scanner presented above is not yet abstract enough for our purposes. We want to transform a scanner function for a regular expression *as such*, that is, without referring to the input string. A means of obtaining a more abstract form is by rewriting our definition in a continuation-based form. The corresponding function is called Scan and is defined as follows, where A is an arbitrary set:

$$\mathsf{Scan} :: RE_C \times (C^* \mapsto Set(A)) \mapsto Set(A)$$

$$\mathsf{Scan}\ \alpha\ c = \mathsf{scan}\ \alpha\ ;\ c \qquad (5.10)$$

By some rewriting, we obtain the following equations:

$$\begin{array}{ll}
\mathsf{Scan}\ \varepsilon\ c\ i & = c\ i \\
\mathsf{Scan}\ x\ c\ i & = \textbf{if}\ \mathsf{hd}\ i = x\ \textbf{then}\ c\ (\mathsf{tl}\ i)\ \textbf{else}\ \{\}\ \textbf{fi} \\
\mathsf{Scan}\ (\alpha\beta)\ c\ i & = \mathsf{Scan}\ \alpha\ (\mathsf{Scan}\ \beta\ c)\ i \\
\mathsf{Scan}\ (\alpha \mid \beta)\ c\ i & = \mathsf{Scan}\ \alpha\ c\ i + \mathsf{Scan}\ \beta\ c\ i \\
\mathsf{Scan}\ [\alpha]\ c\ i & = c\ i + \mathsf{Scan}\ \alpha\ c\ i \\
\mathsf{Scan}\ \alpha^*\ c\ i & = c\ i + \mathsf{Scan}\ \alpha^+\ c\ i \\
\mathsf{Scan}\ \alpha^+\ c\ i & = \mathsf{Scan}\ \alpha\ (\mathsf{Scan}\ \alpha^*\ c)\ i
\end{array}$$

By introducing the following notations for respectively the union of continuations, character inspection and the terminal continuation:

$$
\begin{array}{lll}
(\mathsf{c} \mid \mathsf{d})\ \mathsf{i} & \equiv & \mathsf{c\ i} +\!\!+ \mathsf{d\ i} \\
(\overset{\mathsf{x}}{\rightarrow} \mathsf{c})\ \mathsf{i} & \equiv & \textbf{if}\ \mathsf{hd\ i} = \mathsf{x}\ \textbf{then}\ \mathsf{c}\ (\mathsf{tl\ i})\ \textbf{else}\ \{\}\ \textbf{fi} \\
(!\mathsf{t})\ \mathsf{i} & \equiv & \{(\mathsf{t},\mathsf{i})\}
\end{array}
$$

we can present the algorithm a little more abstractly by leaving out the parameter i:

$$
\begin{array}{lll}
\mathsf{Scan}\ \varepsilon\ \mathsf{c} & = & \mathsf{c} \\
\mathsf{Scan\ x\ c} & = & \overset{\mathsf{x}}{\rightarrow} \mathsf{c} \\
\mathsf{Scan}\ (\alpha\beta)\ \mathsf{c} & = & \mathsf{Scan}\ \alpha\ (\mathsf{Scan}\ \beta\ \mathsf{c}) \\
\mathsf{Scan}\ (\alpha \mid \beta)\ \mathsf{c} & = & \mathsf{Scan}\ \alpha\ \mathsf{c} \mid \mathsf{Scan}\ \beta\ \mathsf{c} \\
\mathsf{Scan}\ [\alpha]\ \mathsf{c} & = & \mathsf{c} \mid \mathsf{Scan}\ \alpha\ \mathsf{c} \\
\mathsf{Scan}\ \alpha^*\ \mathsf{c} & = & \mathsf{d}\ \textbf{where}\ \mathsf{d} = \mathsf{c} \mid \mathsf{Scan}\ \alpha\ \mathsf{d} \\
\mathsf{Scan}\ \alpha^+\ \mathsf{c} & = & \mathsf{d}\ \textbf{where}\ \mathsf{d} = \mathsf{Scan}\ \alpha\ (\mathsf{c} \mid \mathsf{d})
\end{array}
$$

Assuming that $R = \{(\mathsf{t}_1, \alpha_1), \ldots, (\mathsf{t}_k, \alpha_k)\}$, we can write, substituting $T \times C^*$ for the arbitrary set A:

$$\mathsf{scanner} = \mathsf{Scan}\ \alpha_1\ (!\mathsf{t}_1) \mid \ldots \mid \mathsf{Scan}\ \alpha_k\ (!\mathsf{t}_k) \qquad (5.11)$$

Thus, the terminal continuations $!\mathsf{t}_j$ are passed in at the top and only these are the ones that perform the actual symbol recognition at the bottom. The arbitrary set A allows to do something different at the bottom, something which we will exploit later in this chapter when presenting attributed scanners.

Observe that $\mid$ is commutative, associative and idempotent. Moreover, $\overset{\mathsf{x}}{\rightarrow}$ distributes over $\mid$, i.e.

$$\overset{\mathsf{x}}{\rightarrow} (\mathsf{e}_1 \mid \mathsf{e}_2) = (\overset{\mathsf{x}}{\rightarrow} \mathsf{e}_1) \mid (\overset{\mathsf{x}}{\rightarrow} \mathsf{e}_2)$$

5.5 Deterministic scanning

The definition of scanner in (5.11) can be rewritten by substituting for each expression Scan α c that occurs in the definition, the corresponding definition for Scan, depending on α. Since α decreases in each rewrite step, this process terminates, leaving no instance of Scan behind. The result is a set of equations, where each expression e is of one of the forms !t, $\overset{\mathsf{x}}{\rightarrow}$ e, e | e or d, where d is a name introduced by a where-part. These names can be chosen to be all different and can be turned into global declarations, rather than local ones. The resulting set of equations thus has the following form:

$$
\begin{array}{l}
\mathsf{scanner} = \mathsf{e}_0 \\
\mathsf{d}_1 = \mathsf{e}_1 \\
\ldots \\
\mathsf{d}_n = \mathsf{e}_n
\end{array}
$$

Our goal now is to rewrite this set of equations in such a way that is is maximally deterministic, which we define by:

Definition 5.12 (def:max-determ)

A set of equations of the form defined above is *maximally deterministic* if each equation is of the form:

$$d_i = !t_1 \mid \ldots \mid !t_k \mid \overset{x_1}{\rightarrow} d_{j_1} \mid \ldots \mid \overset{x_m}{\rightarrow} d_{j_m}$$

such that all t_j and x_j are different.

□

The two parts of the right-hand side of this equation can be interpreted as reduce- and shift-actions respectively in automata jargon.

Throughout the transformations applied, we keep this set of equations minimal by never introducing a new definition $d_2 = e$ when a definition $d_1 = e$ already exists. Moreover, a choice $e \mid e$ between two identical expressions e is reduced to e, since $\mid$ is idempotent. Our goal is achieved by applying a number of rewrite steps:

1. First, every expression that is of the form $\overset{x}{\rightarrow} e$, where e is not a name, is rewritten into $\overset{x}{\rightarrow} d$ **where** $d = e$, where d is a new unique name, if e was not already bound to a name of course. Notice that only finitely many of these names are introduced due to the finiteness of the set of equations.

   The result is a similar set of equations, each of the form:

   $$\begin{aligned} d_i = {} & d_1 \mid \ldots \mid d_n \mid !t_1 \mid \ldots \mid !t_k \mid \\ & \overset{x_1}{\rightarrow} d_{j_1} \mid \ldots \mid \overset{x_m}{\rightarrow} d_{j_m} \end{aligned}$$

   This new set of equations can (but need not) be interpreted as a non-deterministic finite state automaton (NFA), where each name d represents a state, expression $\overset{x}{\rightarrow} d$ represents a state transition under consumption of character x, $e \mid e$ represents a non-deterministic choice and $!t$ represents the acceptance of terminal symbol t.

2. We still have to get rid of the d_j elements on the right-hand side in order to obtain maximal determinism. To achieve this, expressions of the form $d_1 \mid d_2$ are replaced by d_3 **where** $d_3 = e_1 \mid e_2$, where e_1 and e_2 are the defining expressions for d_1 and d_2. A definition $d = d \mid e$ can be replaced by the definition $d = e$ due to the idempotency of $\mid$. This leaves at most one d on each right-hand side. This process is finite, since only a finite different number of right-hand sides is possible (modulo renaming of the d's).

3. Every expression that is of the form $(\overset{x}{\rightarrow} d_1) \mid d_2$ (or $!t \mid d_2$) (modulo communicativity and associativity of $\mid$) is rewritten into $(\overset{x}{\rightarrow} d_1) \mid e_2$ (or $!t \mid e_2$), where e_2 is the defining expression for d_2, until no expression of this form remains. This process

corresponds to the computation of the ε-closure of states in a NFA. It terminates since the number of possible right-hand sides is finite, due to the idempotency of |.

As a result, all remaining equations are of the form

$$d_i = !t_1 \mid \ldots \mid !t_k \mid \xrightarrow{x_1} d_{j_1} \mid \ldots \mid \xrightarrow{x_m} d_{j_m}$$

which is the desired form, but which does not guarantee maximal determinism yet, since the x_i need not be different.

4. To achieve this maximal determinism, the expressions are rewritten by applying the identity

$$\xrightarrow{x} d_1 \mid \xrightarrow{x} d_2 = \xrightarrow{x} d_3 \textbf{ where } d_3 = e_1 \mid e_2$$

where e_1, e_2 are the defining expressions for d_1 and d_2. As a result a set of equations is obtained where each equation has the form

$$d_i = !t_1 \mid \ldots \mid !t_k \mid \xrightarrow{x_1} d_{j_1} \mid \ldots \mid \xrightarrow{x_m} d_{j_m}$$

where all t_j and x_j are different.

These resulting equations can be interpreted as a minimal deterministic finite state automaton (MDFA). The determinism has been obtained by the application of the last identity, finiteness is preserved through all transformations. Minimality is obtained by the sharing that is introduced by the binding of expressions to names, where all right-hand sides are different, a property that is maintained by the transformations that are subsequently applied. In automata theory, minimality is obtained by identifying equivalent states, starting at the end of the automaton and proceeding forward.[1] In our derivation, equivalent states (i.e. names bound to the same expression) are identified right from the start and remain identified throughout the transformation process, thus maintaining minimality.

5.5.1 Inherent non-determinism

When for an equation $k > 1$, the scanner is ambiguous since more than one terminal symbol can be recognized. When $k > 0, m > 0$, another kind of conflict occurs, where one symbol is a prefix of another one. Usually, this latter conflict is resolved in favor of the longer symbol, but other choices can be made as well. `Elegant` solves an ambiguity of the first kind by accepting thát symbol t that is specified by a rule $t \to \alpha$ that occurs textually first in the scanner specification. The latter conflict is solved in the same way, except when both symbols have a fixed representation (i.e., its regular expression is a string, like : and :=), in which case the longer one is preferred by default. This can be overruled by an annotation that favors the shortest. Ambiguities of the first kind and rules that become unreachable by this arbitration are reported to the user by means of warnings.

[1]This is equivalent to making the reverse automaton deterministic.

5.6 Attributed scanners

When a scanner must work in conjunction with an attributing parser, the scanner must be able to deliver symbol-attribute pairs, where the attribute is a function of the symbol representation. Such a scanner is a straightforward extension of the non-attributed one.

Definition 5.13 (def:scan-ag-spec)

An *attributed scanner specification* is a tuple (C, T, R, D, F) where

- (C, T, R) is scanner specification.
- D is a set of *attribute values.*
- $\mathsf{F} :: T \times C^* \mapsto D$ is the attribution function.

□

The non-attributed scanner specification is generalized as follows to deal with attributes, substituting $T \times D \times C^*$ for the arbitrary set A.

$$\begin{aligned}
&\mathsf{Ascanner} :: C^* \mapsto Set(T \times D \times C^*)\\
&\mathsf{Ascanner}\ \mathsf{i} = \{(\mathsf{t}, \mathsf{F}\ \mathsf{t}\ (\mathsf{i}/\mathsf{j}), \mathsf{j}) \mid \exists \mathsf{j} : \mathsf{t} \to \alpha,\ \alpha \mathsf{j} \overset{*}{\to} \mathsf{i}\}
\end{aligned}$$

The continuation-based implementation changes as follows:

$$\begin{aligned}
&\mathsf{Ascanner}\ \mathsf{i} = (\mathsf{Scan}\ \alpha_1\ (\mathsf{f}\ \mathsf{t}_1\ \mathsf{i}) \mid \ldots \mid \mathsf{Scan}\ \alpha_k\ (\mathsf{f}\ \mathsf{t}_k\ \mathsf{i}))\ \mathsf{i}\\
&\textbf{where}\ \mathsf{f}\ \mathsf{t}\ \mathsf{i}\ \mathsf{j} = (\mathsf{t}, \mathsf{F}\ \mathsf{t}\ (\mathsf{i}/\mathsf{j}), \mathsf{j})
\end{aligned}$$

Note that the question whether this scanner is deterministic or not is not an issue, since all possible results are returned. The result delivered by a continuation-based scanner is completely hidden in the initial continuations passed by Ascanner. The function Scan is exactly the same in the attributed as in the unattributed case. The fact that the scanning process itself is independent from the attribution is obtained by the abstract form expressed in the continuation-based version. Only the final continuations $\mathsf{f}\ \mathsf{t}_k\ \mathsf{i}$ are affected by the attribution.

Chapter 6

Types and patterns

6.1 Introduction

In this section we discuss different aspects of type systems as they are found in functional and imperative languages. Functional type system offer many advanced abstraction mechanisms, such as parameterized types, polymorphism, laziness and pattern matching. Imperative languages offer sub-typing, object-orientedness and explicit strong typing as major mechanisms. We show how these typing mechanisms from functional and imperative languages have been combined into the `Elegant` type system. It appears that by combining sub-typing and pattern-matching, patterns can be interpreted as sub-types. E.g. the non-empty list can be denoted by a pattern and since the set of non-empty lists is a sub-set of the set of all lists, the non-empty list pattern is a sub-type of the list type. By viewing pattern as sub-types, pattern-matching becomes equivalent to type-analysis. We present a formal definition of the `Elegant` type system and derive from this definition, by means of algebraic program transformations, an efficient mechanism for type analysis, i.e. pattern matching.
In lazy functional languages, laziness is implicit: no distinction is made in the language between unevaluated (lazy) objects and evaluated ones. Lazy objects are much less efficient to implement than evaluated ones and, as a consequence, advanced techniques like strictness analysis have been designed by the functional programming community in order to let a compiler derive this distinction again. In an imperative language, implicit laziness would present a major problem for programmers. Since the order of evaluation is important in an imperative language, the programmer should be able to control this, and implicit laziness with implicit evaluation does not particularly offer this required control. `Elegant` solves these problems by offering explicit laziness. The construction of unevaluated objects and their evaluation is completely under the control of the programmer. Explicit laziness is implemented by offering predefined lazy types, instances of which represent unevaluated objects. Since in an imperative environment the number of times a lazy object is evaluated makes a difference in semantics, different variants of lazy types are supported, which are

distinguished by their ability to remember the result of a previous evaluation. One of these lazy types appears to be useful in the construction of interactive systems.
Some imperative languages offer an empty type, like `void` in C or `VOID` in Algol-68. `Elegant` generalizes this concept by allowing multiple empty types, which can be user-defined and which are discriminated by the type system. This allows the grouping of side-effects into different classes that can not be mixed up. While an empty type has no instances and thus represents a side-effect, a lazy empty type can have instances which represent an unevaluated side-effect, and it indeed forms an important construction that is very frequently used by `Elegant` programmers, especially in code generation, where the unevaluated side-effect represents the writing of code to file.

6.2 Typing in functional languages

In this section we give a brief overview of the most important aspects of type systems in modern functional languages. These aspects are analyzed and discussed. The proposed improvements have been incorporated in the `Elegant` type system. Of these, the sub-typing mechanism which combines naturally with pattern matching as type analysis is the most prominent.

6.2.1 Typing and pattern matching in functional languages

Functional languages like Miranda [Tur86] and Haskell [HWA+92, HF92] have a type system based on *algebraic data types*, also-called *sum of product* types. A type is declared as the disjoint sum of tuples. Each disjunct is identified by a so-called *constructor*. Each element of a tuple is called a *field*. As an example, the following is a definition of a binary tree type over the integers:

```
Tree = LEAF Int | BRANCH Tree Tree
```

The type Tree has two constructors, LEAF and BRANCH, identifying tuples with fields of type Int and of type Tree and Tree respectively. Generally, types can be parameterized, such as the following example defining a binary tree type which is parameterized over the element type A:

```
Tree A = LEAF A | BRANCH (Tree A) (Tree A)
```

Abstract data types can be combined with pattern matching in a very natural way, which is heavily used in functional programming. It allows the programmer to specify a function as the union of several partial functions, so-called *rules*, by restricting an argument of a rule to a single constructor. If rules overlap, certain disambiguation rules are needed to chose between them. In `Elegant` we have chosen the textually first one. For example, the following function reflects a binary tree:

```
reflect (LEAF x) = LEAF x
reflect (BRANCH x y) = BRANCH (reflect y) (reflect x)
```

The first rule applies only to leaves, the second only to branches and the two rules together form a total function of type Tree A $\mapsto$ Tree A.
A function defined by pattern matching need not be total, such as the following function that returns the left branch of a tree. It is not defined for leaves.

```
left (BRANCH x y) = x
```

Functional languages often support *implicit typing*, that is, it is not necessary for a programmer to explicitly state the type of a variable (or function). The compiler can *infer* the (most general) type for each variable by means of a type inference mechanism [Hin69, Mil78], based on unification of type variables. Such an implicit type system allows a compact notation of programs, where the attention can be focused completely on the algorithms. Most programs presented in this thesis are given without any typing information. Such a concise notation is a blessing for educational purposes of small functions and algorithms.
When it comes to software design in the large, the story is rather different. Here, typing information forms an essential part of the documentation of the software and a compiler would spend a considerable amount of time on the inference of types that are not specified by the programmer. The programmer should be able to write down these types easily (otherwise he does not know what he is writing altogether) and in that way both document his programs and speed up the compilation process. Moreover, the theory of type inference is rather subtle and should be re-thought for relatively simple modifications of the type system, like adding sub-typing or overloading.
For these reasons, and to keep things simple, `Elegant` has an explicit type system and we consider explicit typing for the remainder of this chapter, which does not keep us from presenting untyped programs!

6.2.2 Draw-backs of abstract data types

Abstract data types, as they appear in main-stream functional languages, have one important draw-back from the point of view of software engineering: they lack *extensibility*. Although new constructors are added easily, adding a new field is cumbersome since all rules defined by pattern matching must be modified, even if they are not accessing the new field. Changing the order of the fields within a disjunct is even more problematic. Again, all patterns for the constructor must be modified. In combination with implicit typing or when swapping two fields of equal types, it can happen that forgetting to modify a pattern is not noticed at compile time, but will result in an erroneous program.
All these problems stem not from the notion of algebraic data types as such, but from the fact that *fields have no name*. They are only bound to a name in pattern matching and the pattern must list all fields *in the correct order*. We call this practice *positional pattern matching*. From the point of view of software engineering, it is preferable to have named fields, where the names are declared in the type declaration and where patterns use these names in referring to the fields. It is then not necessary to list *all* the fields for a constructor in a pattern, so that adding a field does not require the modification of any associated pattern. It is unclear why such a simple extension has not been included yet in

functional languages. It may be the case that a drive for conciseness is the simultaneous reason for implicit typing and nameless fields. Another reason may be the fact that a function should be self-contained: named fields in patterns can only be interpreted in the context of a type definition, which complicates the application of program transformations. The positional pattern matching of fields has a counterpart in function calls: the binding between formal and positional arguments is positional rather than name-based in most programming languages, both functional and imperative. Exceptions are Modula-3 [CDG+88] and Ada [US 83]. We could present a defense of name-based parameter passing analogous to name-based pattern matching, but things are less clear in the case of parameter passing. One does not want to write down parameter names for simple applications, i.e. common functions of a few arguments, like e.g. addition. In this respect, Modula-3 allows both positional and name-based parameter passing, and even mixed versions. The definition of parameter binding in the mixed case is rather complicated however and experience has shown that named-based binding is rarely used. In some cases, it is practical however, especially when functions with many arguments are involved, most frequently constructor functions, which are presented in section 6.2.4.

6.2.3 Sub-typing

Abstract data types, as they appear in main-stream functional languages, do not support *sub-typing*, although it is nowadays custom to view a type that is the disjoint sum of other types as a super-type with sub-types, the so-called *type extensions* (see e.g. [Wir88b, CDG+88]). An instance of a sub-type is also an instance of its super-types. This removes the distinction between types and constructors, the constructors being mere sub-types of the algebraic data type. It allows the sub-typing of constructors to an arbitrary depth. A sub-type contains the fields of its super-types and may add additional fields to these.
The binary tree example may thus be modified in the following way (using named fields):

```
type Tree A
     Leaf A     < Tree A  =  value : A
     Branch A   < Tree A  =  left : (Tree A)
                             right : (Tree A)
```

This piece of code defines three types, Tree, Leaf and Branch. The latter two are defined as direct sub-types of the former.
If not only leaves hold values, but branches as well, the value field can be moved to the super-type:

```
type Tree A        =  value : A
     Leaf A     < Tree A
     Branch A   < Tree A  =  left : (Tree A)
                             right : (Tree A)
```

Multiple inheritance, where a type can have multiple super-types, can also be examined, but is beyond the scope of this thesis.

Pattern matching in combination with sub-typing can now be interpreted as *type analysis*. A rule is a partial function, defined only for a subset of the total domain, where patterns are used to indicate that subset. The run-time pattern matching process can be interpreted as *type analysis*, analyzing the types of the actual arguments, to select the appropriate rule. In section 6.4 we give an efficient implementation of pattern matching as type analysis. The tree reflection function, based on named fields and sub-typing, now becomes for the first version of the type Tree:

```
reflect (x : Leaf)   =  x
reflect (x : Branch) =  Branch (reflect x.right) (reflect x.left)
```

And for the second version:

```
reflect (x : Leaf)   =  x
reflect (x : Branch) =  Branch (x.value) (reflect x.right) (reflect x.left)
```

The compiler should check that a field access only occurs when the left-hand side has a type allowing that access. Thus, the following function is erroneous, since not every tree has a field left:

```
left (x : Tree)  =  x.left
```

6.2.4 Constructor functions

In most functional languages, when an algebraic data type defines a constructor C, a *constructor function* C is defined *implicitly*. This function is used to create new instances of the type corresponding to the constructor. We have been using it silently already. The arguments of this function are the fields defined for the constructor. Each sub-type has a constructor function associated with it, whereas the super-type has not. This however, can be generalized. The *programmer* should decide whether or not a constructor function is associated with a type or not, regardless of the place of a type in a type hierarchy. A type without a constructor function can not have instances itself and is called *abstract*. This should be the case for the type Tree in our example:

```
type Tree A     = value : A abstract
     Leaf A    < Tree A
     Branch A  < Tree A  =  left : (Tree A)
                            right : (Tree A)
```

The type Tree does not have a constructor associated with it, but the types Leaf and Branch do have constructors.

6.2.5 Predefined fields

Named fields open up another possibility, namely that of fields whose value is determined at construction time, by a fixed expression over other fields. Such fields are (in the absence of side-effects) of course only useful for the increase of efficiency, since their value could easily be recomputed from the other fields, each time it is needed. Of course, a predefined field does not appear as an argument of a constructor function. As an example, a binary tree of integers, where each node stores the sum of the values in the leaves beneath it, is defined as follows:

```
type Tree          = abstract
     Leaf   < Tree = value : Int
                     sum : Int = value
     Branch < Tree = left : Tree
                     right : Tree
                     sum : Int = left.sum + right.sum
```

Such an optimization can be hidden in a function sum:

```
sum (x : Leaf)      = x.value
sum (x : Branch)    = sum x.right + sum x.left
```

that has a more efficient implementation in the case of a pre-defined field:

```
sum (x : Tree) = x.sum
```

This example shows a complication with this approach: the two fields sum are completely unrelated and might even have different types, thereby making the last function erroneous. This can be solved by declaring the field sum for the super-type:

```
type Tree          = sum : Int abstract
     Leaf   < Tree = value : Int
                     sum = value
     Branch < Tree = left : Tree
                     right : Tree
                     sum = left.sum + right.sum
```

Here, the super-type declares the field sum, while the sub-types define the expressions for it. It is not necessary that all sub-types do this. A sub-type not defining a value for such a field simply requires that a value for the field should be given as an argument to the corresponding constructor function.
The absence of the necessity to mention all fields for a constructor in a pattern has another advantage, namely that multiple constructors can be mentioned in a single pattern, as in the following example:

```
sum (x : (Leaf | Branch)) = x.sum
```

In such a case, the fields for the lowest common super-type of the mentioned sub-types (in this case type Tree) are available in the pattern and the defining expression on the right-hand side.

6.2.6 Enumerated types

In [Wir88b], Wirth argues that enumerated types should be abandoned, since their only real purpose would be to serve as a discriminator in variant types and that this construction is better implemented by type extensions. This statement has given rise to a lively discussion about enumerated types [Cas91], [Lin90], [Sak91].

However, given a type system with sub-typing and abstract types, enumerated types come for free. As a matter of fact, this is the way enumerated types are usually mimiced in a functional language offering algebraic data types. An enumerated type `T = (A, B, ...)` can be implemented as an abstract root type `T`, with non-abstract sub-types `A`, `B`, etc., non of which have fields. In this case, assuming no side-effects, the constructor functions are nullary and each type `A`, `B`, etc., has exactly one instance. These instances can be mapped onto integers by an implementation. This is a much better definition than the usual one for enumerated types (e.g. the one of Pascal [BSI82] or Modula-2 [Wir85]), which involves a type and implicitly defined constants for that type, in contrast with the rest of the type system. In our case, enumerated types are a special form of the types definable by the type system. They are easily generalized, e.g. by allowing an arbitrary level of sub-typing, or by allowing fields, provided that the constructor function remains nullary.

In an imperative language, things become somewhat more complicated, since a nullary constructor function need not return the same object upon subsequent invocations. It might e.g. allocate a new object each time it is invoked. This problem can be overcome by *memoizing* constructor functions, guaranteeing that identical invocations deliver identical results.

6.2.7 Type classes

As a response to the class notion in object-oriented languages, where a class not only defines a data-type but also a set of operations for that type, the notion of a *type class* in functional languages was introduced [HWA+92, WB89]. When polymorphic types and functions are available in a language, it is often convenient to assert that a type-valued parameter has certain properties. E.g., in a sort function defined on a domain T, it is convenient to know that $\leq$ is defined on T. If this knowledge is absent, the ordering function must be passed as a parameter to the sort function. Certain built-in functions can be polymorphic and have a dedicated notation, such as {i..j} being shorthand for from_to i j. This particular function can be polymorphic only for domains on which $<, =$ and succ are defined.

A type class allows the programmer to assert that a certain type variable τ belongs to a certain class C, guaranteeing that τ supports the operations declared for the class C. When an actual type t is substituted for τ, the type t must be declared to be an *instance* of C. Such an instance declaration must provide the implementation of the operations required by the class.

In the following example, the class `Eq` supprts the operations == and /=, the class `Ord` extends `Eq` with the operations <, <=, >= and >. The function `sort` assumes that the element type is an instance of the class `Ord`.

Example 6.1: Type classes in Haskell

```
class Eq a where
       (==), (/=) :: a -> a -> Bool

       x /= y                = ~(x == y)

class (Eq a) => Ord a where
       (<), (<=), (>=), (>) :: a -> a -> Bool

       x < y                 = x <= y && x /= y
       x >= y                = y <= x
       x > y                 = y < x

instance Eq Int where
       (==)                  = PrimEqInt      -- Built in

instance Ord Int where
       (<=)                  = PrimLeInt      -- Built in

sort :: (Ord a) => [a] -> [a]

sort []    = []
sort (a:l) = sort [ x | x <- l , x <= a ] ++
                  [a] ++
             sort [ x | x <- l , x > a ]
```

□

`Elegant` does not support a notion of type classes. It is a relatively new language construction that is in our opinion not mature enough to deserve incorporation yet. Moreover, a type class can be mimiced by a parameterized type $Class(T)$ that contains a number of fields which are functions over T, namely those functions that a type class defines. A polymorphic function over type T can then be given explicitly such a class object as an extra parameter. The current Haskell implementations implicitly add such an object, called a dictionary. In this way the problem comes down to either explicitly or implicitly manipulating dictionaries. The following example shows such class types for `Eq` and `Ord`:

Example 6.2: Class types

$$
\begin{array}{lll}
\textbf{type} & \mathsf{Eq(a)} = & (==) :: \mathsf{a} \mapsto \mathsf{a} \mapsto \mathsf{Bool} \\
& & (\neq) :: \mathsf{a} \mapsto \mathsf{a} \mapsto \mathsf{Bool} = \lambda x, y.\ \sim (x == y) \\
& \mathsf{Ord(a)} < \mathsf{Eq(a)} = & (<) :: \mathsf{a} \mapsto \mathsf{a} \mapsto \mathsf{Bool} = \lambda x, y. x \leq y \wedge x \neq y \\
& & (\leq) :: \mathsf{a} \mapsto \mathsf{a} \mapsto \mathsf{Bool} \\
& & (\geq) :: \mathsf{a} \mapsto \mathsf{a} \mapsto \mathsf{Bool} = \lambda x, y. y \leq x \\
& & (>) :: \mathsf{a} \mapsto \mathsf{a} \mapsto \mathsf{Bool} = \lambda x, y. y < x
\end{array}
$$

□

When calling a function like `sort`, the proper class type should be passed as an additional argument. In the Haskell implementation, such an additional argument, called a dictionary, is passed implicitly.

6.2.8 Enabling conditions in rules

Apart from specifying the domain of a rule (i.e. a partial function) by means of patterns, an arbitrary boolean expression, a so-called *guard*, over the arguments of the rule can also be used in many functional languages to specify a restriction of the domain. For example, the following function computes the sum of all positive leaves in a tree:

```
psum (x : Leaf) | x.value < 0   = 0
psum (x : Leaf) | x.value ≥ 0   = x.value
psum (x : Branch)               = psum x.right + psum x.left
```

6.2.9 Explicit lazy typing

In lazy functional languages, no distinction is made between evaluated and unevaluated objects (closures): laziness is implicit. This implicitness has its price in implementation costs. In a naive implementation (like an SKI combinator implementation) every object is packed in a closure and stored in the heap. More advanced implementations, e.g. the one described in [Sme93], use strictness analysis or programmer annotations to avoid packing objects in closures and implement parameter passing on the stack where possible.
Another approach, which is for the programmer more cumbersome but much more efficiently implementable, is to distinguish explicitly between objects and closures. This can be done by distinguishing between a type (e.g. Int) and a *lazy type* (Lazy(Int)). It places on the programmer the burden of explicitly postponing evaluation by the creation of closures. He must also explicitly indicate the laziness of fields in data structures. E.g. the types List(Int), List(Lazy(Int)), Lazy(List(Int)) and Lazy(List(Lazy(Int))) become all distinct. But this is the price to be paid for an efficient implementation as long as strictness analysis within data structures is not routinely done automatically.
Explicit laziness can also be introduced in an imperative language. There, it is not a loss of abstraction, but an enrichment, since it allows the programmer to deal with unevaluated expressions. Of course, the presence of side-effects implies that a later evaluation may deliver another result than an earlier evaluation. It raises the question whether the repeated evaluation of a single closure should deliver identical values or not. A way out of this problem is to put the choice on the programmer again and distinguish between a type Lazy(T) (identical results by repeated evaluations) and a type Closure(T) (possibly distinct results by repeated evaluations). In this case, the type Lazy(T) will be more efficient since there is no need for actually re-evaluating an closure already evaluated.

6.2.10 Side-effects

In a functional language the notion of a *modifiable state* is absent and hence the notion of a *side-effect*. This allows a clean and simple semantics of such a language, at the price of efficiency. But it is not necessary to abandon all side-effects. The following side-effects can be allowed:

- *Modification of part of the state that is never inspected.*
  This allows file output to be treated as a side-effect and be implemented in that way. The introduction of a special type Output is useful for this purpose. Semantically it is a write-only string of characters, but it can be implemented by writing directly to a file, rather than by creating a string.

- *Modification of part of the state that is not inspected after the modification.*
  Since a notion of the order of the computation is not present in functional languages, it is a problem to reason about this kind of side-effect. This only seems feasible if a program is partioned into different parts, where one writes part of the state and does not read it, and the other reads it after the writing, with some proper synchronization mechanism between them.

- *Modification of part of the state that is not inspected before the modification.*
  This is the counterpart of the previous possibility with the same solutions.

- *Modification of part of the state that is only inspected by the modification operation itself.*
  This kind of side-effect has very interesting applications. It means that variables with a reference count of 1 (assuming a reference count garbage collector) may be updated, since the only accessing part of the program is the updating of the variable itself. This kind of side-effect allows the efficient implementation of a symbol-table as explained in section 4.8.5, where each intermediate symbol-table is used only to store the next declaration and only the final symbol-table is really accessed. Such a symbol-table can be implemented as a global variable that is updated by side-effects and access to the final table synchronizes on these side-effects. A typing mechanism that distinguishes objects that are treated in this way could be designed so that such a single reference behavior is enforced by this type system. See e.g. [Wad91] for a proposal in this direction. In [Wad91] special array types are proposed with this behavior which allow a $O(1)$ update implementation, something which is still lacking in functional languages. A rather novel approach to the incorporation of modifiable state in functional languages can be found in [Wad92, Wad90]. It is based on *monads*, a class of abstract data types that can be used to enforce that states are passed in a linear fashion between operations, thereby allowing them to be modified.

- *Automatic recomputation of those objects that depend on the modified variable.*
  In interactive systems based on attribute grammars, like certain syntax directed editors, *incremental evaluation* is used to recompute that part of a data structure that is affected by a user interaction. See e.g. [VSK90, RT88, Vog90]. Such an interaction can be seen as a side-effect on a consistent data structure, thereby making it inconsistent, and followed by a recomputation to restore consistency again. Such a system need not be based on attribute grammars, but can use a more general functional language in which certain variables are modifiable, either by user interaction,

or by the system itself. Some automatic recomputation mechanism takes care of the recomputation of that part of the computation that depends on the affected variables. Such a system would be a very interesting vehicle for the implementation of interactive systems.

`Elegant` supports this automatic recomputation by means of the predefined types Incr(T) and Vincr(T). An Incr(T) object i is a closure and evaluation of such an object performs a re-evaluation when another Incr or Vincr object on which a previous computation did depend has been changed since the previous evaluation of i. Vincr objects are not closures, but bound to a value rather than to an expression. They may be assigned to. This whole process is comparable to the Unix 'make' facility. The Vincr objects are sources that can be modified, the Incr objects are targets that are automatically recomputed when necessary. Instead of maintaining the consistency of a set of files, a data-structure can be kept consistent in this way.

Especially the first kind of side-effect is easy to implement and to deal with for a programmer, since it is semantically equivalent to a write-only string.

6.3 Semantics of typing and pattern matching

In this section we present a formal semantics of the functional type system and pattern matching presented in section 6.2. We show that patterns can be regarded as type expressions denoting sub-types.

6.3.1 Definition of a type system

Definition 6.3 (type-system)

We define a *type system* as a tuple $(T_p, T_u, F, \mathsf{D}, <, I_p, Inst, Abstr)$, such that

- T_p is a finite set of *predefined type names.*
- T_u is a finite set of *user-defined type names*, $T_u \cap T_p = \{\}$.
- $T = T_p \cup T_u$ is the set of *all type names.*
- F is a finite set of *field names.*
- D is a function that maps a user-defined type name onto its *type definition*, $\mathsf{D} \in T_u \mapsto (F \times T)^*$.
- $Abstr \subseteq T_u$ is the set of *abstract types.*
- $<$ is the *sub-type relation* defined on $T_u \times T_u$.
- I_p is a set of sets of *predefined instances.*
- $Inst :: T_p \rightarrow I_p$ is the instance function which associates with each predefined type the set of its instances.
  We write $\mathsf{I_t}$ to denote $Inst(t)$. We assume that $\{\mathsf{I_t} \mid \mathsf{t} \in T_p\}$ is a partition of I_p.
- We assume *linear inheritance*, i.e. $\forall \mathsf{s}, \mathsf{t}, \mathsf{u} \in T_u : \mathsf{s} < \mathsf{t} \wedge \mathsf{s} < \mathsf{u} \Rightarrow \mathsf{t} < \mathsf{u} \vee \mathsf{u} < \mathsf{t} \vee \mathsf{u} = \mathsf{t}$.

- A sub-type may add fields to a super-type:
  $\forall t, u \in T_u : t < u \Rightarrow$
  $(\exists k \geq 0, n \geq k : D(t) = \{f_1, \ldots, f_k, \ldots, f_n\} \wedge D(u) = \{f_1, \ldots, f_k\})$

□
We will often use the word *type* for a type name. Note that D(t) is not defined for predefined types.
The following definition will prove useful in the sequel:
Definition 6.4 (proper-sub-type)
If $t < u$, we call t a *proper sub-type* of u and u a *proper super-type* of t.

□
Definition 6.5 (sub-type)
We define the relation $\leq$ on T as the reflexive and transitive closure of $<$ for elements of T_u. For $t, u \in T_p$, we have $t \leq u \equiv t = u$. The relation $\leq$ is empty otherwise.
If $t \leq u$ we say that t is a *sub-type* of u and that u is a *super-type* of t.

□
Definition 6.6 (root-type)
If $\neg(\exists t, r < t)$, we say that r is a *root-type*. Note that a predefined type is always a root type.
The root type R_t of a type t is the root type u, such that $t \leq u$. It is unique due to the linearity of the subtype relation.

□
Definition 6.7 (compatible-types)
Two types t and u are *comparable* if $t \leq u \vee u \leq t$.

□
Definition 6.8 (siblings)
Two types t and u are *siblings* if $R_t = R_u$.

□
Definition 6.9 (lcs-type)
The *least common super-type* u of two sibling types s and t is the smallest type $u \in T$, such that $s \leq u, t \leq u$. We denote this type by $u = s \sqcup t$.

□

6.3.2 Instances

We continue with presenting definitions. The ones in this section are concerned with instances.

Definition 6.10 (instances)

We define the set of *proto-instances* for the type system $(T_p, T_u, F, D, <, I_p, Inst, Abstr)$ as the smallest set I_0 satisfying $I_0 = I_p \cup (T_u \times (F \times I_0)^*)$. We write $\mathsf{f} = \mathsf{o}$ for elements (f, o) of $F \times I_0$ and $\mathsf{f} : \mathsf{t}$ for elements of $F \times T$.

□

Definition 6.11 (belongs-to)

We define $\sim$ (*belongs to*) as the smallest relation on $I_0 \times T$ which satisfies

$$\begin{aligned} \mathsf{o} \sim \mathsf{t} \equiv\ & \mathsf{t} \in T_u \backslash Abstr \wedge \mathsf{o} = (\mathsf{t}, \{\mathsf{f}_1 = \mathsf{o}_1, \ldots, \mathsf{f}_n = \mathsf{o}_n\}) \\ & \wedge D(\mathsf{t}) = \{\mathsf{f}_1 : \mathsf{t}_1, \ldots, \mathsf{f}_n : \mathsf{t}_n\}, \mathsf{o}_i \sim \mathsf{t}_i, 1 \leq i \leq n \\ & \vee \mathsf{t} \in T_p \wedge \mathsf{o} \in \mathsf{I}_\mathsf{t} \end{aligned}$$

□

Observe that this relation is empty for abstract types.

Definition 6.12 (instances-t)

If t is not predefined, we define I_t, the set of *instances of* t, as $\mathsf{I}_\mathsf{t} = \{\mathsf{o} \mid \exists \mathsf{u} \in T_u\ \mathsf{u} \leq \mathsf{t},\ \mathsf{o} \in \mathsf{I}\ :\ \mathsf{o} \sim \mathsf{u}\}$.

□

Definition 6.13 (wf-inst)

We define the set of *well-formed instances*, or simply the set of *instances* for a type system as the set $I \subseteq I_0$, such that,
$\forall \mathsf{o} \in I : \exists \mathsf{t} \in T : \mathsf{o} \sim \mathsf{t}$.

□

In the sequel we deal with well-formed instances sets only.
We can observe that $\mathsf{u} \leq \mathsf{t} \equiv \mathsf{I}_\mathsf{u} \subseteq \mathsf{I}_\mathsf{t}$.

Definition 6.14 (type-of-instance)

We define $\tau(\mathsf{o}) : I \mapsto T$, the *(dynamic) type of* o by
$\tau((\mathsf{t}, \{\mathsf{f}_1 = \mathsf{o}_1, \ldots, \mathsf{f}_n = \mathsf{o}_n\})) = \mathsf{t}$
$\mathsf{t} \in T_p \wedge \mathsf{o} \in \mathsf{I}_\mathsf{t} \Rightarrow \tau(\mathsf{o}) = \mathsf{t}$

□

Definition 6.15 (field-of-instance)

We define $\mathsf{o.f}\ ::\ I \times F \mapsto I$, the f-*field of* o by: $(\mathsf{t}, \{\mathsf{f}_1 = \mathsf{o}_1, \ldots, \mathsf{f}_n = \mathsf{o}_n\}).\mathsf{f}_i = \mathsf{o}_i, 1 \leq i \leq n$.

□

6.3.3 Constructor functions

With each non-abstract user-defined type t, we associate a *constructor function* C_t. This function creates an instance of t. We will not model predefined fields, which requires a simple extension of the definition of a type system.

Definition 6.16 (constr-function)

Given $t \in T_u \backslash Abstr$, with $D(t) = \{f_1 : t_1, \ldots, f_n : t_n\}$ and objects $o_i \sim t_i, 1 \leq i \leq n$, the *constructor function* C_t is defined as:

$$C_t\ o_1 \ldots o_n \ = \ (t,\ \{f_1 = o_1, \ldots f_n = o_n\})$$

□

6.3.4 Patterns

In this section we define the notion of a pattern and show that it is a generalization of the notion of a type, in the sense that the operations on types extend to patterns.

Definition 6.17 (patterns)

Given a type system $S = (T_p, T_u, F, D, <, I_p, Inst, Abstr)$, we define the set of *patterns* for S as the smallest set P satisfying (writing $f : p$ for elements $(f, p) \in P$):

- $P \subseteq T_u \times (F \times P)^*$
- $\forall p \in P, p = (t, \{f_{i_1} : p_1, \ldots, f_{i_k} : p_k\})$:
  $D(t) = \{f_1 : t_1, \ldots, f_n : t_n\} \wedge \{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\} \ \wedge$
  $(i_j = i_m \Rightarrow j = m) \wedge T(p_j) \leq t_{i_j}$
- $\forall p \in P$, p is finite.

Where $T(p)$, the type of p is defined as

- $T((t, \{f_1 : p_1, \ldots, f_n : p_n\})) = t$

□

Note that patterns for predefined types do not exist.

Definition 6.18 (belongs-to-pattern)

We define the relation $\sim$ (*belongs to*) on $I \times P$ as the smallest relation satisfying:

$$\begin{aligned} o \sim p \equiv p = (t, \{f_{i_1} : p_1, \ldots, f_{i_n} : p_n\}) \wedge \\ o \in I_t \wedge \wedge_{j=1}^{n} (o.f_{i_j} \sim p_j) \end{aligned}$$

□

Definition 6.19 (instances-of-pattern)

Given $\sim$, we define I_p, the *set of instances* of p by $I_p = \{o \mid \exists o \in I \ : \ o \sim p\}$.

□

The partial order $\leq$ on P is defined by $p \leq q \equiv I_p \subseteq I_q$. It is a natural extension of $\leq$ on T, since for a pattern with no fields $(t) \leq (u) \equiv t \leq u$. Note that for any $p, I_p \subseteq I_{T(p)}$ holds. Thus, a pattern can be seen as a *generalization* of a type, and patterns can be treated as types, i.e. patterns can be allowed as types for variables, fields and formal arguments. When patterns are allowed as a type in a field declaration, our notion of a type system must be revised, which is something we will not do here.

6.3.5 Rules and functions

In this section we define the semantics of rules and functions defined by means of pattern matching. We do so by giving a transformation from a rule defined by pattern matching onto a rule defined without pattern matching, without giving semantics for the latter, since this is not the purpose of this section.
A rule denotes a partial function, where patterns and conditions restrict the argument domains. A rule r has the form:

$$f\ (x_1 : p_1)\ldots(x_n : p_n) \mid c_r = e_r$$

Here, f is the function name, x_i an argument name, p_i is a pattern, c_r a condition (boolean expression) and e_r is an expression. The condition and expression have the function and argument names in scope.
The semantics of a rule is given by defining that it is equivalent to the following rule, which is pattern free:

$$\begin{aligned} f\ x_1\ldots x_n \;=\; & \textbf{if}\ x_1 \sim p_1 \wedge \ldots \wedge x_n \sim p_n \wedge c_r \\ & \textbf{then}\ e_r \\ & \textbf{else}\ \bot \\ & \textbf{fi} \end{aligned}$$

We write $[\![p_r]\!]$ to abbreviate $x_1 \sim p_1 \wedge \ldots \wedge x_n \sim p_n$ for rule r.
A *function* f is defined by a sequence R_f of rules, which all have the same name f. The ordering of this sequence is in the case of `Elegant` determined by the textual order of the rules. This order is used in disambiguating the choice between rules with overlapping patterns. We assume that all these rules have the same arity, share equal argument names for corresponding arguments and that corresponding argument and result types are siblings. We define the semantics of R_f by giving a transformation that maps it onto a function defined by a single, pattern-free rule. This single rule is given by:

$$f\ x_1\ldots x_n \;=\; \text{head}\ \{e_r \mid r \in R_f, [\![p_r]\!], c_r\}$$

Suppose that a function f is specified by the following rules:

$$\begin{aligned} f\ p_{11}\ldots p_{1n} \mid c_1 \;&=\; e_1 \\ \ldots \\ f\ p_{m1}\ldots p_{mn} \mid c_n \;&=\; e_m \end{aligned}$$

then it is required for the rules to be consistent that

- $T(p_{ij})$ and $T(p_{kj})$ are siblings.
- $T(e_i)$ and $T(e_j)$ are siblings.

and the type T_f of the function is given by:

$$T_f \;=\; \sqcup_{i=1}^{m} T(p_{i1}) \times \ldots \times \sqcup_{i=1}^{m} T(p_{in}) \;\mapsto\; \sqcup_{i=1}^{m} T(e_i)$$

Thus, the types of the parameters and result are the least common super-types of the types over the different rules. Observe that these need not be root types. As a consequence, a caller of a function may have to check at run-time that the actual arguments of a function are indeed instances of this super-type in the case that the type of an actual argument happens to be a super-type of the corresponding parameter type. If this happens to be the case, a compiler should warn that such a check is included. In turn, the callee must analyze the types of the arguments further to select the proper rule. The checking at the call side may interfere with laziness that requires the unevaluated passing of an argument. In that case, the callee might perform the type checking as part of its pattern matching process.

6.4 Compilation of pattern matching

Since pattern matching plays such an important rôle in functional programming, its efficient implementation is of utmost importance. In this section we show how by means of a series of program transformations, pattern matching can be compiled into case expressions, such that (the type of) each argument or field is inspected at most once. We believe that our presentation is substantially simpler than those given in [Aug85, Wad87, Oph89]. Moreover, by interpreting type constructors as sub-types we avoid the problem of combining variable with constructor patterns that appear in [Aug85, Wad87] as the so-called *mixed rule*, which was improved but not eliminated in [Oph89]. We simply do not distinguish between a variable (super-type) and a constructor (sub-type). The fact that we use named fields rather than the unnamed ones of common functional languages avoids the introduction of new variables and allows us to use more general patterns, as not every field needs to be specified in a pattern.
The essential differences between our approach and [Aug85, Wad87, Oph89] is that the latter base their algorithm on the notion of *failure* of pattern matching, while our pattern matching algorithm delivers a *list of successes*. This style is also advocated by Wadler in [Wad85]. This makes our pattern matching algorithm compositional, in the sense that each rule can be compiled into a case-analysis and that these case-analyses can be combined by the $++$ operator on lists, which is associative and in this particular case even idempotent, though not commutative, since the order of the rules is of importance. Due to the simplicity of our compilation scheme, we are able to prove its correctness by basing it on the semantics of pattern matching.
Since `Elegant` supports explicit laziness, and since we do not allow pattern matching on arguments of a lazy type, we need not worry about a pattern matching algorithm that is fully lazy, i.e. one which only evaluates arguments when it can not be avoided for the selection of a rule. For a treatment of lazy pattern matching, see [Lav88].

6.4.1 Patterns and types

Since a type can be sub-typed and an instance of a sub-type is also an instance of the super-type, a pattern specifying a type matches all instances of that type, and hence also the instances of its sub-types. Thus, a pattern matches not a single type, but a whole set

of (sub-)types in a type hierarchy. We assume that an instance carries its type with it and this type can be inspected by the pattern matching process. In the sequel we show how such pattern matching (which can also be viewed as type analysis) can be implemented efficiently.
The example that we use as an illustration is taken from [Wad87, Oph89]. We assume the following type definition:

```
type List A   = abstract
     Nil A    < List A
     Cons A   < List A = head : A
                         tail : (List A)
```

The following function mappairs takes a function and two lists and applies that function pairwise to the elements of the lists.

```
r1 :   mappairs f (x : Nil) (y : List) = Nil
r2 :   mappairs f (x : List) (y : Nil) = Nil
r3 :   mappairs f (x : Cons) (y : Cons)
        = Cons (f x.head y.head) (mappairs f x.tail y.tail)
```

For example, the call mappairs (+) $\{1,2,3\}$ $\{4,5,6,7\}$ delivers the list $\{5,7,9\}$.

6.4.2 Naive implementation of pattern matching

We show a naive implementation of a function `matching` that delivers the sequence of matching rules, given the actual arguments of a function. This function must obey a well-formedness condition: it may only inspect a field a.f after a has been inspected and it follows from this inspection that a indeed has a type for which field f is defined.
We first define a mapping from the sequence of patterns p_r for rule r onto a boolean expression that tests the matching for r. The enabling condition for a rule r is denoted by c and the expression that forms the body of the rule by e. The mapping is denoted by the circumfix operator $[\![.]\!]$. We let a range over (formal) arguments, p over patterns and f over fields, T over types and s over so-called selectors. A selector is defined as either a formal argument a or a selector indexed by a field, s.f. The selectors form a subset of the set of expressions (that we do not define here further). An expression and hence a selector can be evaluated to an instance (in a given context). We will denote this instance by the expression (selector) and thus extend the operations on instances (like $\sim$ p) to expressions and selectors. For convenience, we write $s \in t$ for $\tau(s) \leq t$. The mapping is specified by:

Specification 6.20 (spec:pattern-match)

- $[\![s_1 : p_1 \ldots s_n : p_n]\!]$ :: $(selector \times P)^* \mapsto$ Bool
  $= [\![s_1 : p_1]\!] \wedge \ldots \wedge [\![s_n : p_n]\!]$
- $[\![s : p]\!]$ :: $selector \times P \mapsto$ Bool
  $= s \sim p$

□

A recursive definition, not making use of the $\sim$ relation, is obtained by rewriting the second part of this definition:

Derivation 6.21

$[\![s : (t, \{f_1 : p_1 \ldots f_n : p_n\})]\!]$
 $=$ (definition of $[\![.]\!]$)
$s \sim (t, \{f_1 : p_1 \ldots f_n : p_n\})$
 $=$ (definition of $\sim$)
$s \in I_t \wedge s.f_1 \sim p_1 \wedge \ldots \wedge s.f_n \sim p_n$
 $=$ (definition of I_T and induction, recall that p_j is finite)
$\tau(s) \leq t \wedge [\![s.f_1 : p_1]\!] \wedge \ldots \wedge [\![s.f_n : p_n]\!]$
 $=$ (convenient notation)
$s \in t \wedge [\![s.f_1 : p_1]\!] \wedge \ldots \wedge [\![s.f_n : p_n]\!]$

□

Summarizing, the mapping is defined by:

Implementation 6.22 (of specification (6.20)) (pattern-match)

$[\![s_1 : p_1 \ldots s_n : p_n]\!]$ $= [\![s_1 : p_1]\!] \wedge \ldots \wedge [\![s_n : p_n]\!]$
for a sequence of patterns
$[\![s : (t, \{f_1 : p_1 \ldots f_n : p_n\})]\!]$
$= s \in t \wedge [\![s.f_1 : p_1]\!] \wedge \ldots \wedge [\![s.f_n : p_n]\!]$
selector with recursive patterns

□

Given this mapping, we can define the sub-sequence of a sequence of rules that match the arguments $a_1 \ldots a_n$. Let R_f be the sequence of rules for function f.

$$\text{matching } R_f\ a_1 \ldots a_n \;=\; \{e_r \mid r \in R_f,\ [\![p_r]\!], c_r\}$$

The arguments $a_1 \ldots a_n$ appear as free variables in p_r, c_r and e_r.
With the aid of this definition, a function call $f\ e_1 \ldots e_n$ can be rewritten. Since the textually first matching rule is applied and since R_f represents the sequence of rules in textual order, this transformation is given by:

$$f\ e_1 \ldots e_n \;=\; \text{head (matching } R_f\ e_1 \ldots e_n)$$

When a more subtle error detection in the case of ill-matching is required, this can be replaced by:

$$f\ e_1 \ldots e_n$$
$$= \text{head (matching } R_f\ e_1 \ldots e_n +\!\!+ \{\text{error "Missing cases for function f"}\ \ \bot\})$$

The definition of matching can be rewritten into two forms for a single rule and for a sequence of rules respectively:

$$\text{matching } \{r\}\ a_1 \ldots a_n \ = \ \textbf{if } [\![p_r]\!] \textbf{ then } \{e_r \mid c_r\} \textbf{ else } \{\} \textbf{ fi}$$

$$\text{matching } (r_1 +\!\!+ r_2)\ a_1 \ldots a_n = \text{matching } e_{r_1}\ a_1 \ldots a_n +\!\!+ \text{matching } r_2\ a_1 \ldots a_n$$

For our example, writing $\{r_1, r_2, r_3\} = R_{\text{mappairs}}$ this compilation delivers:

$$\text{matching } \{r_1\}\ f\ x\ y = \textbf{if } x \in \text{Nil} \wedge y \in \text{List } \textbf{then } \{e_{r_1}\} \textbf{ else } \{\} \textbf{ fi}$$

$$\text{matching } \{r_2\}\ f\ x\ y = \textbf{if } x \in \text{List} \wedge y \in \text{Nil } \textbf{then } \{e_{r_2}\} \textbf{ else } \{\} \textbf{ fi}$$

$$\text{matching } \{r_3\}\ f\ x\ y = \textbf{if } x \in \text{Cons} \wedge y \in \text{Cons } \textbf{then } \{e_{r_3}\} \textbf{ else } \{\} \textbf{ fi}$$

The disadvantage of this simple compilation technique is that arguments and their fields may be inspected multiple times. By means of a number of program transformations we will transform this compilation scheme into a more efficient one.

6.4.3 From if to case expressions

The scheme presented above maps each rule onto an if expression. This if expression is of the form

$$\textbf{if } c_1 \wedge \ldots \wedge c_n \textbf{ then } \{e_r \mid c_r\} \textbf{ else } \{\} \textbf{ fi} \qquad (6.23)$$

where each so-called *test* c_i is of the form $s_i \in t_i$, where s_i is a selector and t_i the union of a number of types. This expression can be rewritten into the expression:

$$\textbf{if } c_1 \textbf{ then } \textbf{ if } c_2 \textbf{ then } \ldots \textbf{ if } c_n \textbf{ then } \{e_r \mid c_r\} \textbf{ else } \{\} \ldots \textbf{ else } \{\} \textbf{ else } \{\} \textbf{ fi}$$

Now we are left with a couple of if expressions of the form

$$\textbf{if } s \in t \textbf{ then } a \textbf{ else } b \textbf{ fi}$$

Such an if expression can be transformed into the *case expression*

```
case s ∈ t      → a
        | Ts\t → b
esac
```

where T_s is the highest type that s can have (this is statically determinable). In general, for a case expression of the form

$$\textbf{case } s \in t_1 \to a_1 \mid \ldots \mid t_n \to a_n \textbf{ esac}$$

it is required that $t_i \cap t_j = \{\}$ if $i \neq j$ and that $\bigcup_{i=1}^{n} t_i = T_s$.
The case expression is equivalent to an if expression

```
case s ∈ t1 → a1 | ... | tn → an esac                    (6.24)
= if s ∈ t1 then a1 else
    ...
    if s ∈ tn then an
    else ⊥
    fi ... fi
```

6.4.4 Rewriting case expressions

For readability we abbreviate the case expression

$$\textbf{case}\ s \in t_1 \rightarrow a_1 \mid \ldots \mid t_n \rightarrow a_n\ \textbf{esac}$$

by

$$\textbf{case}\ s \in t_i \rightarrow a_i\ \textbf{esac}_{\ i \leq n}$$

For the rewriting of case expressions, the following identities are useful:

$$f\ \textbf{case}\ s \in t_i \rightarrow a_i\ \textbf{esac}_{\ i \leq n} = \textbf{case}\ s \in t_i \rightarrow f\ a_i\ \textbf{esac}_{\ i \leq n} \qquad \text{(f-case)} \quad (6.25)$$

$$\begin{aligned}&\textbf{case}\ s \in t_i \rightarrow a_i\ \textbf{esac}_{\ i \leq n} +\!\!+\, e \\ &= \textbf{case}\ s \in t_i \rightarrow a_i +\!\!+\, e\ \textbf{esac}_{\ i \leq n}\end{aligned} \qquad \text{(case expr)} \quad (6.26)$$

$$\begin{aligned}&e +\!\!+\, \textbf{case}\ s \in t_i \rightarrow a_i\ \textbf{esac}_{\ i \leq n} \\ &= \textbf{case}\ s \in t_i \rightarrow e +\!\!+\, a_i\ \textbf{esac}_{\ i \leq n}\end{aligned} \qquad \text{(expr-case)} \quad (6.27)$$

$$\begin{aligned}&\textbf{case}\ s \in t_i \rightarrow a_i\ \textbf{esac}_{\ i \leq m} +\!\!+\, \textbf{case}\ s \in u_j \rightarrow b_j\ \textbf{esac}_{\ j \leq n} \\ &= \textbf{case}\ s \in t_i \cap u_j \rightarrow a_i +\!\!+\, b_j\ \textbf{esac}_{\ i \leq m, j \leq n}\end{aligned} \qquad \text{(case-case)} \quad (6.28)$$

$$\begin{aligned}&\textbf{case}\ s \in t_i \rightarrow \textbf{case}\ s \in u_{ij} \rightarrow a_{ij}\ \textbf{esac}_{\ j \leq n_i}\ \textbf{esac}_{\ i \leq m} \\ &= \textbf{case}\ s \in t_i \cap u_{ij} \rightarrow a_{ij}\ \textbf{esac}_{\ i \leq n, j \leq n_i}\end{aligned} \qquad \text{(case-nest)} \quad (6.29)$$

If $T_s \leq T$ then:

$$\textbf{case}\ s \in T \rightarrow a\ \textbf{esac} = a \qquad \text{(full-case)} \quad (6.30)$$

This rule increases both efficiency and laziness, since the right-hand side does not necessarily evaluate s, while the lhs does suggest such an evaluation.

$$\textbf{case}\ s \in t_1 \rightarrow a \mid \ldots \mid t_n \rightarrow a\ \textbf{esac} = a \qquad \text{(case-eq)} \quad (6.31)$$

This rule increases laziness as well.
Observe that the right-hand side of (6.28) obeys the restrictions $(t_i \cap u_j) \cap (t_k \cap u_l) = \{\}$, for $i \neq k \vee j \neq l$ and $\bigcup_{i=1..m, j=1..n}(t_i \cap u_j) = T_s$, that is, it is a valid case expression. Similarly, the right-hand side of (6.29) is also a valid case expression.

6.4.5 Ordering the selectors

In order to guarantee that each selector is inspected at most once by a case expression, we assume that a total ordering on selectors is defined, such that in a case expression

$$\textbf{case}\ s \in t_1 \rightarrow a_1 \mid \ldots \mid t_n \rightarrow a_n\ \textbf{esac}$$

only selectors $u > s$ are tested within the a_i's, in order to satisfy the well-formedness condition.
The ordering should be chosen such that $s < s.f$, but arbitrary for the rest. After chosing an ordering, a nested case expression can be guaranteed to obey this ordering by chosing the order of $c_1 \ldots c_n$ in (6.23) such that if c_i is of the form $s_i \in t_i$, the implication $i < j \Rightarrow s_i < s_j$ holds. This well-formedness of the corresponding nested case expressions is maintained by our transformations.

6.4.6 Concatenating pattern matching expressions

We use the above identities in rewriting the concatenation of case expressions for the separate rules into a single, nested, case expression for the function. In the resulting case expression, we require that the ordering relation defined on the selectors is maintained, in order to prevent double inspection of a selector.
Thus the expression

$$\textbf{case}\ s \ldots \textbf{esac} \mathbin{+\!\!+} \textbf{case}\ r \ldots \textbf{esac}$$

is rewritten according to rule (6.26) if $s < r$, according to rule (6.27) if $s > r$ and according to rule (6.28) if $s = r$. This transformation maintains the well-formedness condition. We rewrite other concatenations (consisting of a single case expression) according to rule (6.26) or (6.27).
After removing all $+\!\!+$ operations by means of these identities, we are left with a nested case expression, with guarded sequences of rules in the leaves.
The overall structure of the pattern matching implementation after these transformations is a case expression with selector s

$$\textbf{case}\ s \in t_i \rightarrow a_i\ \textbf{esac}\ _{i \le n}$$

such that, if a_i is a case expression itself with selector r, then $s < r$. As a result, each selector s is inspected at most once. Since the first naive pattern matching implementation is well-formed and the transformations preserve this well-formedness, the result is well-formed as well. The transformation of our example thus becomes (assuming $x < y$):

Derivation 6.32

matching $\{r_1, r_2, r_3\}$ f x y
    = (Definition of matching)
$\textbf{if}\ x \in \text{Nil} \wedge y \in \text{List}\ \textbf{then}\ \{e_{r_1}\}\ \textbf{else}\ \{\}\ \textbf{fi}\ +\!\!+$
$\textbf{if}\ x \in \text{List} \wedge y \in \text{Nil}\ \textbf{then}\ \{e_{r_2}\}\ \textbf{else}\ \{\}\ \textbf{fi}\ +\!\!+$
$\textbf{if}\ x \in \text{Cons} \wedge y \in \text{Cons}\ \textbf{then}\ \{e_{r_3}\}\ \textbf{else}\ \{\}\ \textbf{fi}$
    = (nesting if expressions)
$\textbf{if}\ x \in \text{Nil}\ \textbf{then}\ \textbf{if}\ y \in \text{List}\ \textbf{then}\ \{e_{r_1}\}\ \textbf{else}\ \{\}\ \textbf{fi}\ \ \textbf{else}\ \{\}\ \textbf{fi}\ +\!\!+$
$\textbf{if}\ x \in \text{List}\ \textbf{then}\ \textbf{if}\ y \in \text{Nil}\ \textbf{then}\ \{e_{r_2}\}\ \textbf{else}\ \{\}\ \textbf{fi}\ \ \textbf{else}\ \{\}\ \textbf{fi}\ +\!\!+$
$\textbf{if}\ x \in \text{Cons}\ \textbf{then}\ \textbf{if}\ y \in \text{Cons}\ \textbf{then}\ \{e_{r_3}\}\ \textbf{else}\ \{\}\ \textbf{fi}\ \ \textbf{else}\ \{\}\ \textbf{fi}$
    = (introduce case expressions)

```
case x ∈ Nil       → case y ∈ List → {$e_{r_1}$}
                     esac
         | List\Nil → {}
esac
++
case x ∈ List → case y ∈ Nil      → {$e_{r_2}$}
                          | List\Nil → {}
                esac
esac
++
case x ∈ Cons      → case y ∈ Cons      → {$e_{r_3}$}
                               | List\Cons → {}
                     esac
         | List\Cons → {}
esac
      = (rule (6.28), List\Nil = Cons and List\Cons = Nil)
case x ∈ Nil    → case y ∈ List → {$e_{r_1}$}
                  esac
                  ++
                  case y ∈ Nil   → {$e_{r_2}$}
                         | Cons → {}
                  esac
                  ++
                  {}
         | Cons → {}
                  ++
                  case y ∈ Nil   → {$e_{r_2}$}
                         | Cons → {}
                  esac
                  ++
                  case y ∈ Cons → {$e_{r_3}$}
                         | Nil   → {}
                  esac
esac
      = (rule (6.26), (6.27) and (6.28), group case expressions)
case x ∈ Nil    → case y ∈ Nil   → {$e_{r_1}$, $e_{r_2}$}
                         | Cons → {$e_{r_1}$}
                  esac
         | Cons → case y ∈ Nil   → {$e_{r_2}$}
                         | Cons → {$e_{r_3}$}
                  esac
esac
```
□

Being interested in the head of this expression only, we can rewrite this as:

Derivation 6.33

```
head (matching {r1,r2,r3} f x y)
    = (function application distributes over case branches, (6.25))
case x ∈ Nil    → case y ∈ Nil    → e_r1
                          | Cons → e_r1
                  esac
       | Cons → case y ∈ Nil    → e_r2
                          | Cons → e_r3
                  esac
esac
    = (rule (6.31), combine case branches)
case x ∈ Nil    → e_r1
       | Cons → case y ∈ Nil    → e_r2
                          ∈ Cons → e_r3
                  esac
esac
    = (definitions of e_r1, e_r2, e_r3)
case x ∈ Nil    → Nil
       | Cons → case y ∈ Nil    → Nil
                          ∈ Cons → Cons (f x.head y.head)
                                          (mappairs f x.tail y.tail)
                  esac
esac
```
□

The resulting nested case expression inspects each selector (x and y) at most once and is, by definition of a case expression, deterministic in its choices. In this way, pattern matching can be compiled into type analysis that requires execution time linear in the size of the pattern.

6.4.7 Code minimization

Although the scheme presented for compilation of pattern matching does generate code that is linear in execution time, this is not the case for code size. Linearity of both time and size can not be obtained together, but a linear time case analysis is easily minimized in size by sharing common sub-expressions. By replacing, from inside out, common subexpression e by a free variable v and adding a where part **where** v = e to the *outermost* case expression, minimality is obtained. It is exactly the same technique used in minimizing a lexical scanner. Assume, for example, that in the definition of mappairs a guard g x y was present for the first rule. Then the case analysis for this modified example would become:

Derivation 6.34

```
head (matching {r1,r2,r3} f x y)
=
head case x ∈ Nil   → case y ∈ Nil    → if g x y then {e_r1} else {e_r2} fi
                                 | Cons → if g x y then {e_r1} else {} fi
                      esac
              | Cons → case y ∈ Nil    → {e_r2}
                                 | Cons → {e_r3}
                      esac
     esac
     = (distribute head and work out)
case x ∈ Nil   → case y ∈ Nil    → if g x y then e_r1 else e_r2 fi
                            | Cons → if g x y then e_r1
                                   → else error "guard failed"  fi
                 esac
       | Cons → case y ∈ Nil    → e_r2
                            | Cons → e_r3
                 esac
esac
     = (minimization)
case x ∈ Nil   → case y ∈ Nil    → if guard then v1 else v2 fi
                            | Cons → if guard then v1
                                   → else error "guard failed"  fi
                 esac
       | Cons → case y ∈ Nil    → v2
                            | Cons → e_r3
                 esac
esac
where v1  = e_r1
      v2  = e_r2
      guard = g x y
```
□

It can be observed that this whole process of efficiency increasing and minimization strongly resembles the transformations applied to the scanner functions of chapter 5.

6.5 The `Elegant` type system

In this section we present the `Elegant` type system and pattern matching process. The main ingredients have been presented above in the discussion about typing in functional languages. `Elegant` is an imperative language, however, with its own specific type system.

6.5.1 Typing of side-effects

In a functional language and in most imperative languages, types serve as an abstraction of *values* and expressions as a prescription of these values. Hence (in strongly typed

languages) each expression has a unique type. In an imperative language, apart from expressions, so-called *statements* are available. A statement describes not the computation of a value, but of a *side-effect*, the modification of the *state*. In most languages however, although expressions are typed, statements are not. Exceptions are C (with type `void`) and Algol-68 (with type `VOID`), but still only a single predefined and fixed type for statements is available, making this type hardly a way to abstract from side-effects. We feel that, since side-effects play such a prominent rôle in imperative programming, they should be supported much more strongly. The type of a side-effect can be seen as an *empty type* that has no instances. Thus an expression of an empty type can not deliver a value and is hence a statement.
An imperative type system should distinguish between different kinds of side-effects. E.g. writing to a file is a very different side-effect from the modification of a global variable. Also, one might wish to distinguish between modifying global and local variables. The programmer should be able to define his own empty types, just as he is able to define other types. In this way, the compiler has the possibility to discriminate differently typed statements, preventing the mixing up of different side-effects.
It is very well feasible to allow *sub-typing* on side-effects. Such a feature gives the programmer the opportunity to create a hierarchy of empty types. He is thus able to define a general empty type that abstracts from any kind of side-effect, and sub-types that abstract from more specific side-effects.
In the imperative languages that do have an explicit empty type, usually a *coercion* from a non-empty to the empty type is defined, with the semantics of discarding the value of the non-empty type, which is the only sensible semantics when only one kind of side-effect can be distinguished. If side-effects can be treated more subtly, other kinds of coercions come into scope. If e.g. the side-effect of writing an instance to a file is called Output, then the natural coercion of a value to this type Output is the writing of the value to a file, rather than discarding it. In section 6.5.3 the subject of coercions is treated more extensively.
Another issue that is brought up by the availability of side-effects is that of the semantics of equality. Since, in an imperative language, the same expression might deliver different results on subsequent executions, this holds also for the application of a constructor function, even for the nullary one. Subsequent invocations might deliver different results even when supplied with identical arguments since the state may change in between. In `Elegant`, the programmer can decide to memoize the constructor function, thus eliminating this distinction. Since memoization makes use of an equility operation on the arguments of a function, two different notions of equality come up: shallow and deep equality. Shallow equality (pointer equality) is supported in `Elegant` by the = operator. Since deep equality raises nasty questions like 'how deep?' and 'what about cyclic structures?' and 'when are unevaluated objects equal?', this kind of equality is not supported and the user can define his own operation for each of his types. As mentioned above, the notion of equality comes up with the introduction of enumerated types when these are treated as fieldless types. The intuitive semantics here is that each sub-type has exactly one instance, while the super-type has none. Thus, the super-type is an abstract type and the constructor functions for the sub-types are memoized.

6.5.2 Description of the type system

The Elegant type system is a rather straightforward application of the observations made above on functional type systems.
First of all, so-called *record types* can be defined. A record type is either a *root type* or a *sub-type* of another record type. It can define or add a number of *fields* respectively. These are identifier-type pairs. A record type can be *abstract*, in which case it has no constructor function. Fields can be bound to an *initial expression*, indicating the value to be assigned at creation time. When an initial expression is absent, a value for the field must be passed as an argument to the constructor function. After creation, the fields can be modified by assignments. The special value NIL is an instance of all record types. Its malicious behavior is discussed in section 6.5.8. The constructor function returns by default a new object each time it is invoked. It can be memoized however, thereby overruling its default behavior. A warning is issued when a field of a memoized type is modified.
Parameterized record types are also present. They can have an arbitrary number of parameters that can be used in the types of the fields. Parameterized types can be sub-typed in the normal way, but all types in single type hierarchy have the same parameters.
The binary tree example can be defined in Elegant in the following way:
Example 6.35: Binary tree definition

```
ABSTRACT Tree(A) = ROOT,    value : A
Leaf             < Tree
Branch           < Tree,    left, right : Tree(A)
```

□
Enumerated types are restricted record types. The root type is implicitly abstract and the elements are sub-types of the root type. They are enumerated types themselves that can be sub-typed to an arbitrary depth. An enumerated type may not have fields. Thus its constructor function is nullary and always returns the same element, the single instance of a (non-abstract) enumerated type.
Example 6.36: Enumerated types

```
Color  = (Black | White)
ABSTRACT TrueColor < Color
Red    < Color
Yellow < Color
Blue   < Color
```

□
Empty types are restricted enumerated types. An empty type can only be sub-typed by another empty type. It may not have fields and is always abstract. Thus, empty types have no instances and hence serve as an abstraction from side-effects. The following empty root types are predeclared:

- VOID – An arbitrary side-effect.
- Output – Writing to stdout.
- Message – Writing to stderr.

A function without a result type has by default result type `VOID`. When a function has an empty result, all statements in the function body must be coercable to that type, otherwise they must be coercable to `VOID`. The predefined empty types are supported by a number of coercions, see section 6.5.3.
Lazy types are also present. Even lazy empty types are available, which represent unevaluated side-effects. Especially in the generation of code (a write-only side-effect) this is very convenient, since often different but consistent pieces of code have to be generated from a single program construction. These can be computed lazily by a single analysis of the data structure and later evaluated in the proper order to write them to file. When $S < T$, then $Lazy(S) < Lazy(T)$.
Repeated evaluation of a lazy empty type instance should in most cases result in repeated side-effects, and thus in repeated evaluation of the statement within the closure. This is in contrast with the repeated evaluation of lazy non-empty types. It is generalized by the distinction between the types `Lazy(T)` and `Closure(T)`. The first is evaluated at most once, the latter over and over again.
Another lazy type is the type `Incr(T)`, as explained in section 6.2.10. Instances of this type are also closures, but they are only re-evaluated when the values of other `Incr` or `Vincr` objects, on which the previous computation did depend, have been changed in the mean time. The difference between an `Incr` and a `Vincr` object is that the latter is not a closure bound to an expression, but it is bound to a value and it may be assigned to. Such an assignment invalidates all `Incr` objects depending on its value, thereby giving rise to potential re-computation.
The basic types, like `Int`, are all so-called *external types*, i.e. types implemented in C. The user can also define her own external types, which from the point of view of `Elegant` are opaque, i.e., only the name of the type is known, but no other properties. The user can implement the basic operations on such a type in C however and build more complex operations in `Elegant` on them. User-defined external types are rarely used. No subtyping on external types is available.
One-dimensional *array types* are also available: indexing always starts at 0. The array length is not part of the type, but determined at creation (allocation) time. Arrays are not very often used in compilers and thus this kind of type was added at a late stage in the `Elegant` design. For applications other than compiler construction, a more extended array type may be needed in the future. Since array elements can be read and modified, the simultaneously required co-variance and contra-variance of the array type with regard to the element type prohibits a sub-typing relation on arrays.
Function types simply list the types of the argument and the result type (if any). In addition, the keyword `VAR` can be used to indicate that a parameter is a call by reference parameter. The sub-typing relation on function types is defined by contra-variance on the argument and co-variance on the result types. Var-parameters are no-variant of course, since they are simultaneously arguments and results and should thus be simultaneously contra- and co-variant.

6.5.3 Coercions

A coercion from type S to type T is a function of type S $\mapsto$ T that is implicitly applied when an expression of type S occurs in a context requiring (an instance of) type T. Certain coercions are more or less standard, like the coercions between integers and floats. Since they are implicit, when not selected with care, coercions can cause more confusion than benefit. Nevertheless, `Elegant` supports a whole spectrum of coercions, among which the coercions to VOID are the most important.

- An instance x of a type S can be coerced to T when a *nameless* function of type S $\mapsto$ T is visible. The coercion consists of the application of this function to the instance x. These nameless functions are predefined for the built-in types (like Int) to VOID and write their argument to the *current output file*. Usually these functions are used in code generation and overloaded with functions that write out user-defined types. In this way, the formulation of code generation becomes concise and readable. An example is the following code fragment that generates code "`(a|b)`" for integers `a` and `b`:

  Example 6.37: Simple code generation

  ```
  Or:[a : Int, b : Int]
  -> "(" a "|" b ")" ;
  ```

  □

- An expression of type T can be coerced into an expression of type Closure(T), Lazy(T) or Incr(T) by creating a closure for it, capturing the free *non-global* variables in the environment of the closure. Multiple evaluations of the closure result in multiple executions of the statement, where the global variables in it may take different values in different executions. Observe that the use of globals in closures is subtle and thus it is recommended for advanced users only.

- When T can be coerced to VOID it can also be coerced to Lazy(VOID) or Closure(VOID), by performing the two previous coercions in succession.

- When S < T, then S can be coerced to T by the identity function. T is coerced to S by checking the type of the instance to be (a sub-type of) S. If not, the coercion fails and an error message is produced.

- Coercions are not applied transitively in order to avoid cycles and confusion.

6.5.4 Typing of function declaration

Since a function may be specified by a set of rules, where each rule matches a different sub-type, it is needed to require a certain consistency among rules and to define the type of the function associated with these rules. This consistency is the one explained in section 6.3.5.

The pattern matching process may interfere with laziness that requires the unevaluated passing of an argument. This problem has been avoided by not allowing pattern matching on lazy types. When the callee can not find a matching rule, it reports this at run time, reporting the function under consideration and the actual types of the arguments.

6.5.5 Pattern matching in `Elegant`

In `Elegant`, patterns are one of the forms of a type and are allowed everywhere where a type can be written. Pattern matching is implemented in the way presented above. The presence of the overloaded constant NIL which is an instance of many types, for which special pattern matching is available while τ(NIL) is undefined, forms a nasty, but by no means insuperable complication. The type (pattern) T matches NIL and the pattern (T) does not match NIL. In the latter case, fields may be added, resulting in patterns of the form $(T, f_1 : p_1, \ldots)$, where the p_i are types which can be patterns again.

6.5.6 Abstract types

`Elegant` supports abstract types. Recall that an abstract type is simply a type which does not have instances itself. Such a type is easily modeled in our type system by defining $o \sim t = \text{false}$ for an abstract type t.

6.5.7 Polymorphism

`Elegant` supports polymorphic functions. A polymorphic function is a function that can operate on many different types. Polymorphism should not be mixed up with overloading (sometimes called ad-hoc polymorphism), which means that differently typed functions can have the same name. Polymorphism in `Elegant` is based on so-called *generic types*. We can illustrate these types by the following polymorphic function, which swaps the values of two variables.

```
<-> [VAR x : ?T, y : ?T]
LOCAL z : T = x
-> x := y
   y := z ;
```

The type `T` is a generic type, which can be bound to an arbitrary type in a function call. The question marks in the heading indicate that `T` is a free (defining) occurrence of a generic type, while its use in the body is a bound occurrence.
Polymorphism gives rise to two implementation problems. The first is the need to find a consistent substitution for the generic types of a function, given the types of the actual arguments in a call. The presence of inheritance requires the finding of a least general substitution, i.e. a substitution that is consistent, but does not use types too high in a type hierarchy, since this would result in a loss of information. An illustration of this is the following function, that turns an arbitrary object in a single list.

```
Single:[x : ?T] : List(T) -> RETURN ({x}) ;
```

When we have two types, say S and U, such that S < U and we call Single with an instance of S, we want it to return a list of S, not a list of U and hence we need to bind the generic type T to S and not to the more general type U.
The second problem is the implementability of polymorphic functions. We do not want to generate a new, specialized, implementation for each different parameter T. We are not even able to do so (at least not easily), since Elegant does offer separate compilation and a function might be used in a module where its implementation is not known. The alternative is to generate a (hopefully small) number of implementations for each polymorphic function declaration once, such that each call is able to pick its own specialized version from this set. Fortunately this is possible, although not easy. The key idea is to observe that the only operations available for instances of a generic type within a polymorphic function are assignment, equality testing and variable declaration (parameter passing being implicit assignment). The only property of the instance needed for the implementation of these operations is the *size* of the instance. Hence it suffices to generate a separate implementation for each combination of possible sizes of the generic types of a function. Although this is exponential in the number of generic types, it is in practice not too much of a problem, since the number of generic types per function is usually limited to at most three in practical applications and the vast majority of polymorphic functions has only a single generic type. For practical applications, the number of possible sizes of types can be fixed to four, namely size 1 (in bytes) for the C type char, 2 for type short, 4 for type int and pointers and 8 for type double. Hence, the Single function above can thus be compiled into four C functions, say Single_1, Single_2, Single_4 and Single_8. Unfortunately, these sizes are implementation dependent. The only guarantee C gives is the size of type char, which is always one byte. In order to obtain portable code, and to avoid building the sizes of the types into the compiler, a rather peculiar technique has been designed. The C pre-processor, particularly its macro mechanism, is used to *compute* the names of these functions at *C compilation time*. Thus when the Single function is applied to an integer, the C text CAT(Single_,SIZE_Int) is generated for the name of the function. The macro CAT is defined as:

```
#define CAT_(a,b) a##b
#define CAT(a,b) CAT_(a,b)
```

It first expands its arguments a and b and then passes them on to CAT_. This macro glues a and b together. Without the extra indirection, a and b would be glued together *before* expanding them. In a sense, the ## operator treats its arguments non-strict, while macros pass their arguments strict to other macros. By defining

```
#define SIZE_Int 4
```

in some system-dependent library module, the correct, system-dependent, expansion to Single_4 is obtained. This technique is only available in ANSI-C, since the ##-operator is not available in pre-ANSI versions of C. It can be mimiced in non-ANSI C however, by using a non-standardized and non-portable feature of the pre-processor, namely that it expands `a/**/b` to `ab`, after expanding `a` and `b`. Many, but not all C compilers allow this trick, but with the abundant availability of ANSI-C compilers nowadays, there is no need to use this clever but unreliable trick.

6.5.8 Shortcomings

`Elegant` is a language that has grown gradually from a very simple notation for attribute grammars, strongly based on Modula-2, towards a system consisting of many tools, among which `Elegant` is a powerful imperative programming language with an advanced type system. The fact that `Elegant` has had a growing number of users during this development process has led to an upward compatibility requirement that made certain flaws continue to exist although the author had a strong wish to eliminate them.

One deficiency is the presence of the special value NIL that is an instance of most user-defined (non-enumerated) types. This object did sneak in in the early design and sadly never went away again. It represents the null pointer, but since the notion of a pointer is not present in `Elegant`, it is an unpleasant intruder. Unfortunately, it has an efficient implementation (namely the null pointer) which, together with upward compatibility requirements, has kept it alive. From a more abstract point of view, the user should decide for each of his types whether he wants a bottom object or not. If so, he should make a (probably fieldless) sub-type. To ensure that this sub-type has a single instance, he could memoize the type. The implementation should then take care of the efficient compilation of these instances, especially of the efficient use in pattern matching and comparison. Mapping such an instance onto the null pointer is desirable in this respect, but if different types in a type hierarchy have single instances, these instances should be mapped onto different pointers.

Another deficiency is the fact that polymorphic types can be only 1, 2, 4 and 8 bytes large in implementation. Fortunately, this is not a severe restriction, since larger objects can usually be represented by a pointer to them.

Chapter 7

Generalized attribute functions

7.1 Introduction

In chapter 4 we presented attribute functions and an efficient implementation scheme for an attributed recursive descent parser. Since an attribute grammar is such an interesting formalism, allowing the computation of multiple results which may be mutually dependent and even allowing actual arguments to be dependent on the results of the same production rule (i.e. attribute function), it is interesting to investigate to what extent attribute grammars can be generalized while maintaining an efficient implementation. This becomes even more interesting when one realizes that attribute grammars allow a static cyclicity check on the dependency of the attributes [Knu68, Knu71], a property that might also be retained for a more general formalism.
In this chapter we present such a formalism, being a class of recursive functions that has two types of arguments: strict and non-strict. Apart from this, these functions may have non-strict results. By disallowing a strict argument to depend on a non-strict one, a separation between the two is obtained, while the static cyclicity check is preserved. The strict arguments can be used in the selection of a rule for the function, like a production rule is selected by inspection of the input string and like a function rule is selected by pattern matching. The non-strict arguments and results are treated like attributes. They may be mutually dependent, as long as cyclicity is avoided, something which can be checked statically. Strict arguments are not intended to serve as a more efficient version of attributes, ones that can be evaluated at parse-time. Their sole purpose lies in the selection of production rules by other means than by parsing the input text.
Johnsson [Joh87] describes an approach similar to ours. He introduces a new language construct that can be added to a lazy functional language and which can be regarded as a way to incorporate the attribute grammar paradigm into a functional language. Our approach is a bit different. Since attribute grammars can be regarded as limited functional programs, there is no need to *extend* a lazy functional language, but rather a need to *restrict* it, or to be more precise, to exploit the potential of attribute grammars (circularity check,

efficient implementation) in the case of such restricted use. The restriction consists of a strict separation between the strict arguments of a function (which are available for rule selection) and the non-strict arguments and results (which form the attributes). The strict arguments may be used by the non-strict, but not vice versa, since this would make the circularity check incomputable.
Apart from restricting a lazy functional language, one can also extend a non-lazy language with functions accepting non-strict (inherited) arguments and producing non-strict (synthesized) results. Summarizing, the generalized attribute functions can be regarded in three different ways:

- As generalized attribute grammars, using pattern matching instead of parsing for rule selection.
- As restricted lazy functions, separating the strict and non-strict arguments.
- As extended strict functions, adding non-strict arguments and results.

`Elegant` supports a notation for such generalized attribute functions and calls them *relations*. They appear to be extremely useful in the specification of optimization and code-generation in a compiler. The encoding of the `Elegant` back-end makes heavy use of such relations and provides a good example of their usefulness. In this chapter we first show how an attributed LL(1) parser can be generalized to a a generalized attribute function. Next we show how generalized attribute functions are incorporated in `Elegant`and give an example of their use. Then we extend an attributed LL(1) parser with strict attributes and show how an attribute grammar supporting strict attributes can be used to implement an operator precedence parser.

7.2 Generalized attribute functions

We recall the definition of an attributed recursive descent parser:

$$\begin{array}{llll} \text{parse\_attr } A\ i\ a_1 \ldots a_{k_0} = & \textbf{case}\ x \in \text{foll } \alpha_j\ (\text{foll } A) & \rightarrow & \text{parse\_attr } (A \rightarrow \alpha_j)\ i\ a_1 \ldots a_{k_0} \\ & \quad \ldots & \rightarrow & \ldots \\ & \quad \textbf{otherwise} & \rightarrow & \text{error "A expected"} \quad (i, \text{false}, \perp) \\ & \textbf{esac} & & \\ \textbf{where}\ x : j = i & & & \end{array}$$

$$\begin{array}{l} \text{parse\_attr } (X_0 \rightarrow X_1 \ldots X_n)\ i_0\ a_1 \ldots a_{k_0} \ = \ (i_n, c_0, e_1 \ldots e_{m_0}) \\ \textbf{where}\ c_0 \ = \ e_c \wedge c_1 \wedge \ldots \wedge c_n \\ \qquad (i_1, c_1, s_{11} \ldots s_{1k_1}) = \text{parse\_attr } X_1\ i_0\ e_{11} \ldots e_{1m_1} \\ \qquad \ldots \\ \qquad (i_n, c_n, s_{n1} \ldots s_{nk_n}) = \text{parse\_attr } X_n\ i_{n-1}\ e_{n1} \ldots e_{nm_n} \end{array}$$

We can forget about the parsing process and replace the selection of such a rule by the more general process of pattern matching and boolean conditions that was presented in chapter 6. This replaces the (strict) argument i by an arbitrary number of strict arguments on which pattern matching is allowed. Boolean expressions over the strict arguments can also be allowed in the rule selection process. Since the other arguments a_j may depend on the results of the very same rule, the arguments a_j can not be used in the pattern matching process for the rule selection.

We can generalize further by treating the first argument of a parsing function, i.e. a symbol or a production rule, as part of the name of the parse function (since it is known statically) and remove it all together as an argument. In this way we obtain a set of mutually recursive functions over strict and non-strict arguments.

The context conditions are typical for a parser. They lead to error messages regarding the input text and without an input text they can be removed. Another option would be to keep them for more general error messages but we believe that normal error message mechanisms (like writing information to the standard output or error device) are already sufficient for this purpose and hence we remove the context conditions in the generalized attribute functions.

Both local attributes and local variables can be added. The difference between them is that the latter may only depend on strict arguments and not on attributes.

The resulting scheme for a function rule defined according to these rules is the following, where we have added definitions of local variables L_i and attributes l_i to stress the difference between strict and non-strict variables:

```
    $f_0\ p_1 \ldots p_q\ a_1 \ldots a_{k_0} = (e_1 \ldots e_{m_0})$
    **conditions** $d_c$
    **where** $L_1 = d_1$
          $\ldots$
          $L_r = d_r$
    **where** $(s_{11} \ldots s_{1k_1}) = f_1\ d_{11} \ldots d_{1s_1}\ e_{11} \ldots e_{1m_1}$
          $\ldots$
          $(s_{n1} \ldots s_{nk_n}) = f_n\ d_{n1} \ldots e_{ns_n}\ e_{n1} \ldots e_{nm_n}$
          $l_1 = e_{l_1}$
          $\ldots$
          $l_k = e_{l_k}$
```

Here the variables p_j represent strict arguments which may be defined by patterns and the variables a_j represent non-strict arguments (attributes) for which no pattern matching is available. The variables L_j represent strict local variables where l_j denote non-strict local variables (local attributes). The non-strict variables s_{ij} represent the j-th result of f_i. The expressions d_j may depend only on the strict arguments p_j and strict locals $L_k, k < j$. The expressions d_{ij} represent the strict arguments to the attribute function f_i. They may depend only on the strict arguments p_j and strict locals L_k. The boolean expression d_c may only depend on the arguments p_j and is used in the rule selection process as a guard for the rule. The non-strict expressions e_j may contain arbitrary variables, both strict and non-strict.

The formalism could be made even more general by also allowing strict results (results which are returned in evaluated form by a function, in contrast to out-attributes which are

returned unevaluatedly), but these would not in general be usable in the strict expressions since the implementation scheme induces a certain ordering on the computation of the recursive calls for f_1 to f_n. This means that a strict result of such a recursive call is not available in a textually preceding recursive call, giving rise to unpleasant scope rules to avoid these problems. A simpler solution is not to support strict results. In `Elegant` they are nevertheless supported by the var-parameter mechanism that is available for strict arguments. This is a Pascal-like parameter mechanism, which supports a call by reference mechanism for variables.
Since the attributes are strictly separated from the rule selection mechanism, it is possible to select first all the rules and perform all recursive calls, without evaluating a single attribute, and only then evaluate the attributes. This property makes it possible to perform the static cyclity check for attribute dependencies [Knu68, Knu71] also for the class of these generalized attribute functions. A requirement is that the implementation of all these functions is available to the cyclicity checker, or that it must make worst case assumptions for functions that are e.g. implemented in another module.

7.3 Relations in Elegant

`Elegant` supports the generalized attribute functions and calls them *relations*, a name which is historically determined and which is perhaps unfortunate, since they have little in common with mathematical relations. The `Elegant` syntax for relations resembles the syntax for production rules as much as possible. The `Elegant` relation rule corresponding to the rule scheme presented above is the following:

$f_0\ (p_1 \ldots p_q$ IN $a_1 \ldots a_{k_0}$ OUT $e_1 \ldots e_{m_0})$
conditions d_c
local $L_1 = d_1$
$\quad \ldots$
$\quad L_k = d_n$
$\rightarrow f_1\ (d_{11} \ldots d_{1s_1}$ IN $e_{11} \ldots e_{1m_1}$ OUT $s_{11} \ldots s_{1k_1})$
$\quad \ldots$
$\quad f_n\ (d_{n1} \ldots d_{ns_n}$ IN $e_{n1} \ldots e_{nm_n}$ OUT $s_{n1} \ldots s_{nk_n})$
local $l_1 = e_{l_1}$
$\quad \ldots$
$\quad l_k = e_{l_k}$
;

The corresponding syntax is as follows:

```
⟨relation-rule⟩  ::= ⟨identifier⟩
                      ( [ ⟨arguments⟩ ] [ IN ⟨attributes⟩ ]]
                         [ OUT ⟨expression-list⟩ ] )
                     [ CONDITIONS ⟨boolean-expression⟩ ]
                     [ ⟨local-section⟩ ]
                     -> { ⟨right-element⟩ }
                     [ ⟨local-section⟩ ] ;

⟨right-element⟩  ::= ⟨relation-rule-identifier⟩
                      ( [ ⟨expression-list⟩ ] [ IN ⟨expression-list⟩ ]
                         [ OUT ⟨attributes⟩ ] )
                   | ⟨statement⟩
⟨arguments⟩      ::= [VAR] ⟨identifier⟩ : ⟨pattern-type⟩
                     { , [VAR] ⟨identifier⟩ : ⟨pattern-type⟩ }
⟨attributes⟩     ::= ⟨attribute-list⟩ : ⟨type-identifier⟩
                     { , ⟨attribute-list⟩ : ⟨type-identifier⟩ }
⟨attribute-list⟩ ::= ⟨attribute-identifier⟩ { , ⟨attribute-identifier⟩ }
⟨local-section⟩  ::= LOCAL { ⟨local-variable⟩ }
⟨local-variable⟩ ::= ⟨identifier⟩ : ⟨type-identifier⟩ = ⟨expression⟩
```

As can be observed, this syntax is even more general than the scheme, allowing (strict) var-parameters and arbitrary statements between the recursive calls in the right-hand side. The non-strict variables (attributes) in relations are implicitly of a lazy type. Thus, also attributes of empty types are supported. Outside of relations, laziness occurs only explicitly and this requires a special treatment on the boundary of these two sub-formalisms. A relation R may be called, from the outside, by the statement

$$R\ (d_1,\ldots,d_n\ \mathrm{IN}\ e_1,\ldots,e_k\ \mathrm{OUT}\ v_1,\ldots v_n)$$

Here, d_j represents an expression resulting in an argument for a strict parameter, e_j represents an expression that results in a lazy (unevaluated) object that is passed as an argument for a non-strict parameter. The variables v_j must also have lazy types and they store the unevaluated out-attributes of the relation.

During a call of a relation, the strict arguments may be used in the selection of a rule. After this selection, the relations of the right-hand side are called. These relations determine the unevaluated attributes. These attributes may be mutually dependent, as long as cyclicity is avoided. Hence, no such attribute is evaluated during the evaluation of the relations. After the call of a relation, the unevaluated attributes are returned in the variables v_j, as mentioned. They may be evaluated after the call, in any order, at any place that is appropriate. This evaluation of the out-attributes results in the evaluation of other attributes on which the out-attributes depend, and so on. Note that attributes may depend on the strict variables of the relations as well. The latter are also stored in the closures representing the unevaluated attributes.

The implementation techniques used in the compilation of relations are simply the combinations of the techniques for pattern matching, presented in section 6.4 and of the implementation of attribute grammars, presented in 4.9. Hence, we need not elaborate on it here any further.

7.3.1 Example: Elegant code generation

The `Elegant` code generator, which generates C-code, is an excellent example of the power of generalized attribute functions. In general, the compilation of an `Elegant` expression results in a C-expression. But since some `Elegant` expressions are more powerful than C-expressions, extra code may be needed. This extra code may consist of C-statements that should be executed before the expression, extra local variables that should store intermediate results and extra functions or extra global variables that are needed by the expression. Since code generation is implemented by a side-effect, namely writing the code to a file, the attributes representing these pieces of code will have an empty type, in this case the type `VOID`.
The `Elegant` back-end contains a rule for the compilation each kind of expression with the following attributes:

```
E (x : Expr OUT g, f, l, s, e : VOID)
```

It has as a strict argument `x` an object representing the `Elegant` expression and delivers as attributes, i.e. unevaluated, the C-code to generate. This code is delivered by means of 5 out-attributes that represent the C-code for globals, functions, locals, statements and the C-expression itself.
To illustrate this we give the rule for the compilation of an `Elegant` `EVAL`-expression that evaluates a lazy object. Its compilation is `x->eval(x)`, where `x` is the lazy object. The argument of the `EVAL`-expression may be an expression itself that can contain a side-effect and that hence should be evaluated only once. Thus `x` can not be the compilation of this expression, since it occurs twice. The solution lies in the introduction of an extra local variable to store the value. The brackets `(:` and `:)` delimit a statement sequence. The generalized attribute function `NewLocal` delivers a new local variable of type `x.lazy.type` with C-name `v` and C-declaration code `l2`. The resulting code for the C-expression becomes `(x = e, x->eval(x))`, where `x` is the fresh C-variable.

```
E (x : (EvalLazyExpr) OUT g1, f1, l1++l2, s1, e)
-> E (x.lazy OUT g1, f1, l1, s1, e1 : VOID)
   NewLocal (x.lazy.type OUT l2, v : VOID)
LOCAL
  e : VOID = (: "("v"="e1", "v"->eval("v"))" :)
```

Another example is the generation of code for a closure, i.e. for a lazy expression, which we explain below.

```
E (x : (LazyExpr) OUT t++g1, f1, (::), (::), e)
```

```
-> ClosureFunctions (x OUT create, g1, f1 : VOID)
LOCAL
  t : VOID = EnvironmentType:[x.env]
  e : VOID = create " (" EnvironmentArgs:[x.env] ")"
```

For such a lazy expression `LAZY(e)`, a type must be declared that represents the environment of the closure. This type declaration is represented by the attribute `t`. The environment stores all the free, non-global variables of the expression `e`. It has the form:

```
typedef struct RecEnv *Env;
typedef struct RecEnv { T1 x1; ... Tn xn; }
```

Moreover, two functions are declared, represented by `f1`, which have the following form:

```
LazyT CreateClosure (T1 x1, ..., Tn xn)
{ Env env;
  LazyT l;
  env = allocate (sizeof(*env));
  l   = allocate (sizeof(*l));
  env-> x1 = x1 ; ... env->xn = xn;
  l->env = (address)env;
  l->eval = EvalClosure;
  return l;
}

T EvalClosure (LazyT l)
{ Env env = (Env)(l->env);
  l->value = ....;.  /* Defining expression for e, using env->xi */
  l->eval = IdT;     /* Identity function for T,
                        which returns l->value */
  return l->value;
}
```

The first of these, `CreateClosure`, which name is represented by the attribute `create`, creates the closure. It is passed the values of the free variables, allocates the environment, stores the values of the free variables in that environment and stores the latter together with the second function, the evaluation function in the closure. It returns the closure. When the evaluation function is called, it computes the value of `e` using the values of the free variables stored in the environment.
The whole back-end consists of a multitude of these rules, most of them no more complicated than this one. Without the use of generalized attribute functions, the advanced code generation of `Elegant` could not have been implemented in a maintainable way. Or to state it more strongly: the `Elegant` relations could not have been implemented without the aid of `Elegant` relations!!

7.4 Strict arguments in attribute grammars

Once we have added strict variables to attribute functions, there is no reason not to add them to an attribute grammar. We only need to keep them apart from the normal (non-strict) attributes, but that is statically verifiable. A nice extension can be made here. In the case of an LL(1) top-down parser, not only the value of the first symbol of the input stream can be used in the selection process for a production rule, but also the *value of the attribute* of that symbol. `Elegant` supports this by means of the expression `FIRST` that denotes the attribute of the first terminal symbol of a production rule and which is available in the strict part of the attribute grammar. As a restriction, the type of this value must be equal for all the first symbols of a given production rule. This technique can easily be implemented in a top-down LL(1) parser, as our implementation scheme for attribute grammars shows.
We can see a recursive descent parser as a special case of generalized attribute functions, that uses recursion on data-structures. A simple pre-processing operation can be imagined that adds a strict attribute (the first symbol of the remaining input string) and a condition (the matching of the first symbol with the set of directors of the production rule). The result of this pre-processing is again a generalized attribute function. For bottom-up and back-tracking parsers, no such pre-processor is easily imaginable. Such a pre-processor would have to perform recursion to iteration transformation in order to transform the descent structure of the functions traversing the data structure into an ascent structure, which is hard to do without inverting the data structure. Hence we claim that top-down parsers are the most natural parsers for attribute grammars. Operationally explained, an LL(1) recursive descent parser with strict arguments selects its production rules using two different mechanisms. First of all, the first symbol of the input string should be an element of the directors of the production rule. Secondly, the patterns and conditions over the strict arguments and the attribute of the first symbol of the input string should match, respectively be true. Only when all these conditions hold, is the rule selected. As we will see in the next example, the use of strict arguments in attribute grammars can add considerably to its expressiveness. It allows e.g. for the implementation of an operator precedence parser.

7.4.1 Example: operator precedence parsing

In this section we present an operator precedence parser for infix and prefix operators with user-defined priorities. It makes use of generalized attribute functions. More complex examples of operator precedence parsers, also allowing postfix operators and different associativities, are presented in [Aug91]. Since we use it here as an example for generalized attribute functions, we keep it simple in this section.
Assume that a programmer can specify the priority levels of infix and prefix operators in some prologue to his program. Then these levels can be declared at parse time (using strict attributes) and they are available when expressions using these operators are parsed subsequently. Let $\overrightarrow{o}_i$ denote a prefix operator of level i, that is, it binds an expression that contains infix operators of a level $j \leq i$. Let $\overleftrightarrow{o}_i$ denote an infix operator of level i, that is, it binds expressions that contain infix operators of a level $j \leq i$. An operator of a lower

level thus binds more strongly. Let $\perp$ denote an expression not containing any operators (or only ones enclosed in parentheses). Let max denote the maximum (weakest binding) level of all declared operators. Furthermore, let e_i denote an expression that contains no infix operators of level > i. Then we can define a grammar for the expressions that we want to parse:

$$\begin{array}{lll} e_i & = e_i \overset{\leftrightarrow}{o}_i e_i \mid e_{i-1}, & i > 0 \\ e_0 & = \vec{o}_j e_j \mid \perp, & j \geq 0 \end{array}$$

This grammar can be rewritten in the following form, assuming right associativity for infix operators:

$$\begin{array}{lll} e_0 & = \perp \mid \vec{o}_j e_j & \\ e_i & = e_{i-1}[\overset{\leftrightarrow}{o}_i e_i], & i > 0 \end{array}$$

By mapping the subscripts onto strict attributes, this grammar can be implemented by the following attribute grammar. We assume that the function `Level:[o,f]` returns the level of an operator identifier `o` with fixity `f`, the function `Prefix:[o,e]` creates a prefix expression for operator `o` applied to expression `e` and the function `Infix:[l,o,r]` creates an infix expression for operator `o` applied to expression `l` and `r`. We explain it in more detail below.

```
%% Expression -> E_max

Expression (OUT e) -> E (max OUT e : Expr) ;

%% E_0 -> SimpleExpression

E (i=0 OUT e) -> SimpleExpression (OUT e : Expr) ;

%% E_0 -> o_j E_j

E (i=0 OUT e)
-> FindOperator (IN id, Prefix OUT op : Operator)
   Operator (OUT id : Ident)
   E (Level:[FIRST, Prefix] OUT rhs : Expr)
LOCAL e : Expr = Prefix:[op, rhs]
;

%% E_i -> E_(i-1) Infix_i

E (i : Int OUT e) CONDITIONS i > 0
-> E (i-1 OUT lhs : Expr)
   Infix (i IN lhs OUT e : Expr)
;
```

```
%% Infix_i -> o_i E_i

Infix (i : Int IN lhs : Expr OUT e)
CONDITIONS Level:[FIRST, Infix] = i
-> FindOperator (IN id, Infix OUT op : Operator)
   Operator (OUT id : Ident)
   E (i OUT rhs : Expr)
LOCAL e : Expr = Infix:[lhs, op, rhs]
;

%% Infix_i ->

Infix (i : Int IN lhs : Expr OUT lhs) -> ;

%% Retrieve the operator with name 'id' and fixity 'fix'

FindOperator (IN id : Ident, fix : Fixity OUT op)
->
CHECKS
  IF op = NIL THEN "Operator " id " not declared"
LOCAL
  op : Operator = RetrieveOperator:[id, fix]
;
```

The expression FIRST denotes the attribute of the first symbol of the corresponding production rule, in this case an identifier denoting an operator.
The selection of a production rule for the non-terminal E proceeds as follows. First, the last rule for E is selected recursively until i=0. Then either the first or the second rule is selected, depending on the first symbol of the input string being a prefix operator or not. In the latter case, the argument of this prefix operator is parsed recursively. Finally, when the expression is parsed, e.g. by the occurrence of an infix operator on the input string, the recursion returns to surrounding levels, where i>0 and where the rule Infix is applied. This non-terminal generates the empty string, except for the level which equals the precedence level of the infix operator. There, the infix operator is recognized and the right-hand argument of the operator is recursively parsed at the correct level. Note that this mechanism implies that a call at level i can never consume infix operators of a higher level, except as arguments of a prefix operator of a higher level.
The rule FindOperator is an example of a so-called *abstract production rule*. Such a rule produces only the empty string and its purpose lies in the computation of attributes and context checks. In an LL(1) parser non-terminals with only abstract production rules can be added without any problem. In bottom-up parsers these ε-rules tend to introduce shift-reduce conflicts. This is yet another reason to prefer LL(1) grammars when dealing with attribute grammars. About 80% of the code of an attribute grammar is concerned with

the attributes and not with the underlying context-free grammar. Hence it is important to allow as much abstraction (like abstract production rules and strict attributes) as possible for the attributes even at the expense of a slightly corrupted grammar.

Chapter 8

An evaluation of Elegant

8.1 Introduction

In this section we evaluate the abstraction mechanisms and notations `Elegant` offers. We concentrate here on `Elegant` as a programming language, both for general purposes and for attribute grammar specification. We only discuss the more unconventional concepts here. Some features that are traditional in one programming paradigm (like lazy evaluation in functional programming) are not found in another (like lazy evaluation in procedural languages) however. Since `Elegant` attempts to combine the best of both worlds, we discuss such features here in order to be able to address readers that have experience with one of these paradigms only. Many features, like the type system and different parsing techniques, have already been discussed in previous chapters. These features are mentioned, but not discussed here in detail.
We hope that the discussion in this chapter makes clear that functional and imperative programming can be combined in an elegant way. The design and implementation of `El-egant` has been influenced by both worlds, although in this thesis we have stressed the influence of functional programming. This has been done since we have been inspired by the *abstraction capabilities* of functional languages particularly. Most, if not all, of these abstraction capabilities can be incorporated into an imperative language as well, as `Elegant` demonstrates. We believe that the real power of functional languages comes much more from these abstraction capabilities than from the absence of side-effects. However, the clean semantics of functional languages has enabled the use of algebraic program transformations, which have proven to be of essential importance throughout this thesis.

Honesty forces us to admit that these transformations have to a large extent been constructed afterwards. The `Elegant` *parsers, scanner, attribute evaluator and pattern matching have all been implemented before they were derived by means of these transformations. In the field of parsing however, these transformations have yielded a lot of new parsing algorithms which have not been implemented (yet).*

From imperative languages we have borrowed more traditional mechanisms like assignment, procedures, VAR-parameters and the notion of state. The main contribution of the imperative world however is inheritance (sub-typing), which is a powerful abstraction mechanism, born in the object-oriented world programming and not yet present in functional languages. As we have explained, it can be smoothly integrated in functional languages, unifying the concepts of patterns and (sub-)types and giving a very simple and clean semantics to pattern matching, which becomes equivalent to run-time type analysis.
The design of `Elegant` can be characterized as an expedition to the border between abstraction level and efficiency. Starting with a simple idea, which was an efficient implementation technique for lazy evaluation of attributes, `Elegant` has grown into a powerful, yet highly efficient programming language. During this expedition, new abstractions did come up, like polymorphism, which at first seemed unimplementable in the target language (first Modula-2 and later C), but for which implementation techniques were discovered later on. Especially the implementation of polymorphism is far from trivial and on the borderline of what can be expressed in ANSI-C, involving heavy and essential use of the C preprocessor. The borderline between abstraction level and efficiency has moved also by the change from Modula-2 to C as implementation language. Especially the pre-processor with its macro facilities turned out to be invaluable, not only in the implementation of polymorphism, but also in the shifting of many basic operations (like integer addition) from being built into the compiler to being defined in the prelude by a macro name. Last but not least, the availability of faster and faster hardware has contributed to moving the borderline.
`Elegant` is also a compromise between the clean and the·efficient. Hence it offers both side-effects and powerful declarative facilities, like attribute grammars and relations. As the attribution scheme of section 4.8.5 shows, their combination can lead to both clean and efficient coding styles. Of course, their combination can be disastrous as well, but since we aim at the professional and sensible programmer as a user, this has never occurred (yet). The result of this compromise is a very high level programming language that combines the abstraction capabilities of higher order functional languages with the expressive power and efficiency of imperative languages.
Concepts that were desirable but hard to implement, or not efficiently implementable, have been left out. Examples of these are unification and, what is more important, *garbage collection*. Although automatic storage reclamation is a powerful and important facility, its implementation, especially when portable C is used as a target language, is far from trivial and it certainly does not contribute to a better time efficiency. As `Elegant` has been used for the construction of compilers, which run as batch programs, this has not been a problem, due to the abundant availability of memory nowadays. Since `Elegant` is now being used in the construction of interactive systems (using incremental evaluation), which can run for an arbitrary amount of time, a garbage collector would be welcomed and will probably be added in the near future.

8.2 Notational evaluation

As language design is the art of combining incompatible concepts, languages are always a compromise. Elegant is more of a compromise than most languages, as it tries to combine the clean with the efficient. This means that less clean, but very efficient concepts like assignment and the NIL-object have been incorporated. In the process of designing Elegant language design rules have been used and discovered. The standard attribution scheme of section 4.8.1 is the result of such design rules.
The golden rule is to design first the concepts to be expressed by a program, i.e. the set of language constructs. Only after that step, should a syntax for these constructs be designed. This promotes the design of a language supporting the abstractions from the application domain.
Another rule is that a language should not favor the compiler over a programmer. Constructs that are handled by the compiler should be accessible to the user as well. E.g. operators and overloading are present in most languages for built-in types, but seldom available for user-defined operations. Elegant does not contain any built-in type or operator: all of them are defined in the prelude, which is a perfectly legal module that is no more special than user-written modules.
A third important rule says that everything that need not be forbidden should be allowed. Often rather arbitrary restrictions are imposed by a language due to the disregarding of this rule. Examples from Pascal are the forward declaration of procedures as opposed to pointer types and the need to declare constants, types, variables and functions in a strict order.
A fourth important rule says that concepts that are almost the same should be unified whenever possible into a single concept. In Pascal, there is no need to distinguish functions from procedures: they implement almost the same concept, only the presence of a result distinguishes them.
A fifth rule says that a concept should be generalized as far as possible, of course taking efficiency constraints into consideration. E.g. the void types of C and Algol-68 can be generalized to the concept of an empty type.
In the design of Elegant, we have tried to follow each of these rules. As Elegant has grown over the years, being used in projects with their own time-scale and demands, upward compatibility has been of great importance. Nevertheless, we feel that from time to time, a new major release is needed, breaking with flaws or lack of generality of a previous release. Currently, Elegant is in its sixth major release and where possible, (automatic) support in the transition from one release to another has been offered to users.
In this section, we present the less conventional features of Elegant as a programming language and we hope it gives the reader a feeling for the way we have applied the design rules to Elegant. A complete language definition can be found in [Aug92b] and an introduction in [Jan93].

8.2.1 Features already covered

The type system

The `Elegant` type system has been presented in section 6.5. In practice, especially the sub-typing facilities combined with pattern matching are invaluable. Sub-typing matches ideally with the modeling of language constructs that are syntactically modeled by non-terminals with production rules and semantically paired with types and sub-types. The efficient implementation of pattern matching forms one of the corner stones of `Elegant`. It favors the definition of functions by means of rules (partial functions) over explicit case analysis by a programmer. The former is both more comprehensible and efficient than the latter.
Parameterized types and polymorphism have been added only recently. Up to that point, lazy types and arrays were built in the compiler: now they are defined as parameterized types. Especially the parameterized list and set types with their comprehensions, treated in section 8.2.2, and polymorphic higher order functions add to the expressiveness. Before their addition every instance of a list type and every operation on it had to be declared separately. We have the expectation that libraries of polymorphic functions on parameterized data types such as lists, sets, different kind of trees, hash tables, etc. will be constructed, thus opening the door for reuse of software components.
Empty types are not used very frequently. Only the predefined ones, `Message` for writing to `stderr`, `Output` for `stdout` and `VOID` for general side-effects are popular. The lazy forms of these types, especially the type `CODE` (= `Closure(VOID)`), appear to be very useful in code generation. An instance of `CODE` can be used to represent an unevaluated side-effect, normally to produce output code. In this way, one can abstract from code generation, e.g. by the following rule that generates C-code for a variable declaration:

```
Decl:[type : CODE, var : CODE] -> "  " type " " var ";" NL ;
```

or by the next rule that generates an alternation, useful e.g. in the generation of a parameter list. It is applied to a list `l`, a function `f` that is applied to each element of the list, performing a side-effect on each element. These side-effects are alternated by the evaluation of the third argument `a`, of type `CODE`, which represents an unevaluated side-effect.

```
Alternate:[NIL : List(?T), f : [?T], a : CODE] -> ;
Alternate:[l : List(?T), f : [?T], a : CODE]
-> f:[l.head] { EVAL(a) ++ f:[x] | x : T <- l.tail } ;
```

Relations

Relations (generalized attribute functions) are used in the computation of complex properties of data structures, e.g. of data dependency analysis and more frequently in code generation, as explained in section 7.3. It allows the independent generation of different pieces of code for a single data structure element by a single rule. In this way, these related pieces of code (e.g. code for the name generation of a fresh variable, its variable declaration and its initialization) can be kept consistent. Without relations, these pieces of code are generated by multiple traversals of the same data structure (e.g. a traversal to distribute fresh variable names, one to generate code for variable declarations and one for their initialization), which is both less efficient and more error prone, since the multiple operations are usually scattered over the source text and must be kept consistent.

Explicit laziness

`Elegant` offers explicit laziness, in contrast to lazy functional languages that keep lazy evaluation implicit. On the one hand, this is an advantage since it allows a more efficient implementation of non-lazy objects, something which otherwise can be achieved, or better approximated, by extensive and expensive strictness analysis only. On the other hand, it places the burden on the programmer.
Laziness in combination with side-effects is dangerous. The same expression may deliver different results when evaluated at different times. This is something that should be under the control of the programmer, in order to keep it controllable in an environment that allows side-effects. Hence we favor explicit laziness in such an environment. It also gives rise to different forms of laziness, `Lazy`, `Closure` and `Incr` types, which all have their place in practical programming. This is another reason for making laziness explicit: the programmer should be able to choose between the different forms of laziness.
Finally, explicit laziness provides strictness information to the cyclicity checks for attribute grammars, thus allowing the powerful class of pseudo circular attribute grammars.
In relations, implicit laziness is offered for the attributes. As a result, attributes may have an empty type, in contrast to normal variables.

Useful side-effects

In the world of functional programming, side-effects are often despised. Here we present some useful aspects of them, not from a point of view of efficiency, something which is never doubted, but from a notational point of view. We give three examples, all from the field of code generation.
In code generation, one often needs a source of new identifiers. When modeling this in functional languages one needs an extra variable that is e.g. incremented each time a new identifier is needed. This variable must be passed in and out to almost every code generating function, something which is at least an order above the complexity of what should be needed for such a simple task. It obscures functions that would otherwise be much more simple and lucid. Especially in combination with relations, the implementation of this by side-effect is very useful, and it can be encapsulated very simply. This is shown in the next example. The rule `NewVar` allocates a new variable for a given type and initial expression. It delivers the declaration code, initialization code and name of the variable as out-attributes.

```
GLOBAL
  num : Int = 0
```

```
RULES
  NewNum:[] : Int -> num :+ 1 RETURN (num) ;

RELATIONS
  NewVar (IN type, value : VOID OUT decl, init, name) ->
  LOCAL
    i : Int = NewNum:[]
    name : VOID = (: "VAR" i :)
    decl : VOID = (: "  " type " " name ";" NL :)
    init : VOID = (: "  " name " = " value ";" NL :)
  ;
```

The implicit lazy evaluation guarantees that the rhs of the declaration of `i` is evaluated only once, and hence each occurrence of `name` is the same.
Another example is the generation of properly indented code, which with the lack of side-effects would require yet another additional parameter encoding the indentation level.

```
GLOBAL
  indent : Int = 0
  Margin : CODE = { (: " " :) | i : Int <- 1 .. indent }
  NL     : CODE = (: "\n" Margin :)

RULES
  Indent:[n : Int, c : CODE]
  -> indent :+ n
     c
     indent :- n ;
```

When `Indent:[2,c]` is executed, `indent` is increased, subsequently `c` is evaluated and every access to `Margin` or `NL` will reevaluate it, using the increased indentation level. After the evaluation of `c` the indentation level is restored. The evaluation of `Indent:[2,c]` `Indent:[10,c]` produces the same code `c` once with an extra indentation level of 2 and ones with an extra level of 10.
We present a third example, again from code generation. In the generation of target code, say C, one generates a lot of newlines. The newlines must be suppressed sometimes, e.g. in the generation of a macro call. On another occasion, in the generation of debuggable C code, we want to generate
`#line line-number`
for each newline in order to provide the debugger with source code line numbering information. Thus, we need three types of newlines. Without side-effects we would have needed yet another parameter of our already obscured code generating function in order to encode the newline type.

```
GLOBAL
  line_number : Int = 0
  Real_NL   : CODE = "\n"
  No_NL     : CODE = ""
  Debug_NL  : CODE = (: "\n#line " line_number "\n" :)
  NL        : CODE = Real_NL

RULES
  Macro:[c : Code]
  LOCAL nl : CODE = NL
  -> NL := No_NL
     c
     NL := nl ;

  Line:[line : Int, c : CodeS]
  LOCAL nl : CODE = NL
        n : Int = line_number
  -> NL := Debug_NL
     line_number := line
     c
     line_number := n
     NL := nl ;
```

Again, the side-effects are nicely encapsulated, but almost indispensable.
Recently, with the popularization of *monads* [Wad90, Wad92], a technique for dealing with state in functional programming has become available. Through the use of proper higher order function, the passing around of a state, as well as its modification can be hidden. When a state passing monad is implemented as an abstract data type, it can be guaranteed that at any moment at most one reference to a state exists and hence that state modification operations can be implemented destructively. On the one hand, this promises a major performance improvement for functional programs manipulating state, but on the other hand, since monads guarantee the abundant use of higher order functions and the creation of closures, they promise a performance penalty. A way out seems to hide the monads in a system unit that implements the side-effects directly in a semantically transparent way. It is too early yet to draw any conclusions on them in this respect.

8.2.2 Iteration

The elegant notation of comprehensions, which has been used so abundantly throughout this thesis, has inspired us to include a generalization of this notion in `Elegant`. The syntax of this construction in `Elegant` is as follows:

```
⟨iteration⟩    ::= { [ ⟨construction⟩ ] [ ⟨expression-list⟩ ] [ | ⟨iterators⟩ ] }
⟨iterators⟩    ::= ⟨iterator⟩ [ , ⟨iterators⟩ ]
⟨iterator⟩     ::= ⟨typed-ident⟩ { , ⟨typed-ident⟩ }
                   ← ⟨destructable⟩ { , ⟨destructable⟩ }
                 | ⟨typed-ident⟩ = ⟨expression⟩
                 | ⟨condition⟩

⟨typed-ident⟩   ::= ⟨identifier⟩ :  ⟨type⟩
⟨destructable⟩  ::= [ ⟨destruction⟩ ] ⟨expression⟩
⟨construction⟩  ::= / (⟨expression⟩ , ⟨expression⟩)
⟨destruction⟩   ::= / (⟨expression⟩ , ⟨expression⟩)
```

The number of *typed-idents* on the left-hand side of an iteration must equal the number of *destructables* on the right-hand side of an iterator. When multiple destructables are present, they are walked through simultaneously.
When we forget about the optional construction and destruction for a moment (they have a default value and are explained below), this comes close to the comprehensions of functional languages, where the second form of an iterator, $x : T = e$ is equivalent to $x : T \leftarrow \{e\}$. An expression $\{e_1,\dots,e_n\}$ is a shorthand for to $\{e_1\} \mathbin{+\!\!+} \dots \mathbin{+\!\!+} \{e_n\}$, while the rest of the semantics remains the same as in section 2.2, that is to say, for single destructables. Multiple destructables can be defined by the rule:

$$\{e \mid p_1,\dots p_n \in e_1,\dots e_n\} \;=\; \{e \mid (p_1,\dots p_n) \in zip_n\ (e_1,\dots e_n)\}$$

where zip_n is the straightforward generalization of zip to n arguments.
The types that are added to the iterator play an important rôle however. When the iteration $x : T \leftarrow e$ produces elements of type U, with $T \leq U$, only those elements that happen to be an instance of T are selected, while the others are simply skipped over. The type T may be a pattern (recall that patterns are types as well) and in this way a pleasant form of filtering is available.
The construction and destruction generalize the scope of the iteration to much wider classes of types than just lists. They allow iteration over any type whose instances can be destructed and can deliver an instance of any type whose instances can be constructed.
The notion of destruction must be rather general in order to allow e.g. the efficient implementation of iteration over arrays. For lists, a function destructing a list into a head and a tail is sufficient, but for arrays, we need the element, the array and additional information: the current index in the array. We can generalize this to the following information respectively: an element, a destructable object and a *state* storing the additional information. Thus, the general rule for destruction is that for an iterator $x : T \leftarrow e$ a pair of functions (Destruct, State) must be in scope. This pair can be overruled by supplying the destructor explicitly. The general typing of this pair is as follows and explained below:

```
Start     : [Te] : S
Destruct  : [VAR Te, VAR S, VAR RT] : Bool
```

The destruction starts in a certain state of type S, the start state being obtained by Start from the result of e. The function Destruct transforms a (T_e, S) tuple into a (T_e, S, R_T) triple whenever it returns true and to undefined values when returning false. The idea is that it is called for each iteration step again, delivering an element in its third argument, until it returns false. At that point, the iteration terminates. In Elegant, the (Destruct, Start) pair is predefined for the types List, Set, Array, String, Ident and Bitset. They can be overloaded for any user-defined type U to allow extension of the iteration notation to that type U and they can be overruled in every individual iteration by explicitly providing them. The construction operates in a similar fashion. Let E be the root type of $e_1 \ldots e_n$ (it defaults to VOID when $n = 0$) and let U the result type of the iteration. Then the pair (Construct, Null) must be defined with signature:

Null	: U
Construct	: [U, E] : U

The iteration result is accumulated by starting with Null and by adding successive values for e_i by means of Construct. The pair (Construct, Null) can be explicitly provided in the form of a construction. It is predefined for the types List, Bitset and VOID. The last type represents a general side-effect with as start value the empty side-effect and as construction function the sequencing of side-effects. In this way, the iteration can be used as a convenient alternative notation for nested while-loops.
A nice example of the application of iteration as a replacement for loops can be found in section 4.8.7 in the symbol-table implementation and in the following function that returns the value of the first element of an association list that has a certain key (assuming NoValue to be an instance of type Value):

```
Find:[l : List(Pair(Key,Value)), key : Key] : Value
-> { (: RETURN (x.value) :) | x : Pair(Key,Value) <- l, (x.key = key) }
   RETURN (NoValue) ;
```

The computation of the maximum value of a list is easily modeled by using constructors:

```
Max:[l : List(Int)] : Int
-> RETURN ({ /(Max,MinInt) i | i : Int <- l }) ;
```

Pythagorean triples are generated by the expression:

```
{ [a,b,c] | a : Int <- 1 .. n, b : Int <- a .. n, c : Int <- b .. (a+b),
            (c*c = a*a+b*b) }
```

The constructor functions for sets are defined as:

```
Null:[s : ?T] : Set(T) -> RETURN (NIL) ;
Construct:[s : Set(?T), x : ?T] : Set(T) -> RETURN (Insert:[x, s]) ;
```

The destructor functions for arrays are given by:

```
Start:[a : Array(?T)] : Int -> RETURN (0) ;
Destruct:[VAR a : Array(?T), VAR s : Int, VAR x : ?T] : Bool
-> IF s >= #a THEN RETURN (FALSE) FI
   x := a@s
   s :+ 1
   RETURN (TRUE) ;
```

8.2.3 Function objects

One of the cornerstones of functional programming languages is the fact that functions are first-class citizens. They can be declared (bound to a name), passed as arguments and returned as results. Moreover, there exists a denotation for function objects (λ-abstraction), which offers the ability to denote a function without declaring it (giving it a name). This combination of features is very rare in the world of imperative programming. Even the notorious Algol 68 language, well known for the rigorous, if not merciless, application of the orthogonality principle, did not offer first-class functions[1]. `Elegant` offers first-class functions in an imperative setting. The main complication is the storing of the free variables which appear in the body of a function expression in an environment, which is passed as an additional parameter when the function is called. In this respect, function denotations are just parameterized closures and they do not give rise to complications in a language which already offers explicit laziness and closures.

8.2.4 Incremental evaluation

Incremental evaluation is a novel feature that has already been introduced in section 6.2.10. One way to obtain a clean, referential transparent, semantics in the presence of side-effects is to recompute automatically every expression that directly or indirectly depends on the previous value of a modified variable. In this sense, incremental evaluation lies in between `Elegant` lazy types and closures. Lazy objects stabilize after one evaluation and are thus evaluated at most once, closure objects never stabilize and are evaluated over and over again, while incremental objects are stable as long as none of the variables on which their previous computation depended changes its value. In this respect, incremental evaluation is comparable to the Unix `make` facility. To implement incremental evaluation, `Elegant` offers, apart from the Lazy(T) and Closure(T) types, the types Incr(T) and Vincr(T). Instances of the latter of these may be modified by an assignment (they are sources in `make` terms), thereby invalidating instances of the Incr type (the targets), which are subject to (automatic) re-evaluation.
Incremental evaluation is particularly useful in the construction of interactive systems. An interactive system can be viewed as a system that maintains a data-structure by imposing consistency restrictions (invariants) on that data-structure. A user event (pressing a button, moving a scroll-bar) modifies part of the data-structure, making it inconsistent. The system then restores consistency again, thereby updating that part of the screen that became inconsistent. In this respect, incremental evaluation is related to the field of constraint programming, found in some object-oriented languages (see e.g. [Fre90, Hor92, VMGS91, BvZ89]). Viewed in this way, such an interactive system is very naturally modeled by means of incremental evaluation. The user events modify Vincr objects and do not care about restoring consistency. A back-ground process re-evaluates the screen after each user event. This screen is an Incr(VOID) object, representing a re-evaluatable side-effect, namely the up-

[1]Although Algol 68 did allow the returning of a local function as a result, it required that this function it did not access parameters or local variables stored on the stack, which could disappear after the returning of the function. Hence, the facility was not of much use.

dating of the screen. The whole data-structure of the interactive system consists of Incr and Vincr objects, which are defined in terms of each other. Of course, not all invariants can be modeled in this way. The consistency relations must be brought into an effectively computable form, such that each object is defined by means of a (re-evaluatable) expression over other objects. Not all components of the data structure need be of Incr types. Intermediate results may be stored in normal variables. As long as the value of such a variable is not computed in one re-evaluation phase and used in a subsequent one, this does no harm. Also functions used in the computation of Incr objects may have their own local variables that they may modify.
Since the incremental evaluation mechanism is relatively new, there is little experience with it and a good programming style for it is still under development. The first interactive prototype system indicates, however, that it is a valuable and powerful technique that allows for the construction of interactive systems in an untraditional way.

8.2.5 Persistency

For some applications persistency of a data structure is required. In order to meet this demand, `Elegant` offers so-called *graph-io*. An expression WRITE x TO filename writes the object x of, say, type T, together with all the objects it references, directly or indirectly, to a file. The expression READ T FROM filename reads this whole data-structure isomorphically back-in (possibly in another (invocation of the same) program), maintaining sharing and possibly cycles. Of course, not all objects can be written to a file in this way. Function typed objects and unevaluated lazy or closure typed objects (which are function typed objects in disguise, since they contain an unevaluated expression) can not be written. It would probably require an interpreter based system to change this.
Nevertheless it is a powerful mechanism that is particularly useful in passing data-structures between different programs. In order to enhance its applicability, fields in a data-structure can be declared to be write-only or read-only for graph-io, so that an application can have its private fields that are not shared with other applications.

8.2.6 Safety

A high level programming language should not only offer abstraction capabilities, but also *safety*. In the first place, safety means a guarantee that run-time errors will not take place, or will be detected. Detection can take place at compile time, in which case a suitable warning should be given, or at run-time, in which case a suitable (fatal) error must be reported, or both, in which case both messages should be similar. Unfortunately, certain run-time checks are very expensive, for instance the checking on dereferencing NIL on every field access. Such checks, which are performed for each correct construction and not only for the erroneous case and which delay the normal execution of a program, should be generated optionally so that they can be removed once a program has reached maturity.

Raising the level of abstraction

In general, a higher level of abstraction increases the safety of a language. This is caused by the fact that more abstract constructions require less detail and thus offer less chances for errors being made. Less detail implies that a programmer can concentrate more on the essentials of his program, the architecture and the concepts he is dealing with and less on the details of encoding those concepts. As a result, he is liable to make fewer errors. This is particularly true when the language is a problem-specific language. In this case, the concepts of the problem domain appear directly as constructs in the language and the programmer can express them directly, writing them down instead of encoding them. This is e.g. the case in the use of regular expressions for scanner generators and of context-free or attribute grammars for parser generators. A programmer no longer needs to know how to implement regular expressions by automata or how to implement parsers with suitable error recovery. The difference in complexity between an attribute grammar and its implementation is between one and two orders of magnitude and the chance of making errors is thus dramatically decreased by the application of problem-specific languages. This is also caused by the fact that it is easier to make a correct generator than to make each individual application manually correct. A generator can also generate unreadable, incomprehensible but efficient code, as long as the generator itself is comprehensible. Although `Elegant` generates C, the kind of C that it generates is of a form that one would not dare to write by hand. In this respect, C is used as a portable assembly language, rather than as a high level programming language. Fortunately, the quality of C compilers nowadays is such that the extreme programs `Elegant` generates are compiled correctly. In the recent past, this used to be different and especially the quality of the Modula-2 compilers (we generated Modula-2 from 1987 to 1990) was extremely poor on many platforms.

Strong typing

One of the main techniques for the increase of safety is, of course, a good type system. By compile time checks this guarantees that objects of different types, with different representation classes, are not mixed up at run-time, thus preventing errors and run-time type checks are not necessary. A strong static type system can be explicit, like in most imperative languages, or implicit, like in many modern functional languages. As remarked in section 6.2.1, implicit typing is technically much more complicated and much harder to extend, e.g. when adding subtyping, coercions or overloading. When a programmer is writing an implicitly typed function, he usually knows the type (otherwise he would not be able to write it at all), but when he is reading it back, something which is done much more frequently than writing it, he has to infer that type again, thus blurring his thoughts which should be concerned with making a mental model of the function, not with type inference. Hence, when writing the function, i.e. at the moment he knows the type, he should write down that type, thus at the same time documenting his program and speeding up the compiler.

A leak in the type system

Although the type system of Elegant has been designed carefully, it is not completely type-safe, in the sense that it is possible that an object of a super-type T is interpreted as an instance of a sub-type S. This is possible due to the combination of three ingredients, namely pattern-matching, aliasing and side-effects. Due to the absence of side-effects in pure-functional languages, the problem does not appear there. Aliasing frequently takes place in Elegant due to the pointer semantics of the assignment to record-typed variables. The problem is best illustrated by an example. Assume that field f of a type X is declared to have type T. By means of pattern matching, it can be matched locally to be an S, S<T. When the argument a is aliased by a variable of type X, the problem shows up:

```
Leak:[a : (X, f : S)]
LOCAL b : X = ...
      s : S = ...
      t : T = ...
-> b := a              %% a is aliased by b
   b.f := t            %% t is a valid instance for b.f
   s := a.f ;          %% a.f was promised to be an S, but alas
```

An possible solution would be to detect that b is an alias of a, but this is in general undecidable. Another solution would be to interpret a pattern for an argument not as an invariant for a rule (since it isn't, as the example illustrates), but as a precondition, serving its purpose in the selection of the rule only. Since it is extremely convenient to use a pattern as an invariant and since this pathological case is very rare (we have not found a single instance of it in practical applications yet), we have decided to leave the leak in and be pragmatic in this case.

Completeness

A safe language should not only provide protection by consistency checks, e.g. by means of a strong type system, but also by means of completeness checks. The Elegant compiler provides several checks of both categories, apart from strong typing. We list the following:

- **Missing rules.**
  When the rules, which together define a function by means of a sequence of partial functions, do not together constitute a total function, a compile time warning is given, listing the missing cases for the function. When at run-time, arguments are provided in the part of the domain for which the function is undefined, that same message is given, as a fatal run-time error, listing the function name and the missing cases again.

- **Unreachable rules.**
  Apart from missing rules, it is also possible that the set of rules that form a function is overcomplete in the sense that the domain of certain rules is already covered by preceding ones. These superfluous rules are listed as a warning and no code is

generated for them. Both the missing and unreachable rule checks are straightforward to implement giving the pattern matching compilation technique of section 6.4. The semantics of conditions is not taken into account for this check, only their presence.

- **Cyclic relations and attribute grammars.**
On both relations and attribute grammars the standard cyclicity check of [Knu68], [Knu71] is applied, extended with all kinds of optimizations, like the ones presented in [RS82, DJL84]. Although this cyclicity check is intrinsically exponential, the practical performance is very acceptable, up to a few seconds for complex attribute grammars. This check is in practice extremely useful, if not indispensable, since it frequently signals cycles that would otherwise lead to (possible rare) run-time errors that could only be interpreted by either very experienced users, familiar with the attribute evaluation mechanism, or by expensive run-time checks.

- **Initialization at allocation time.**
A cause of hard-to-find errors in many imperative programming languages is the ability to allocate memory without properly initializing it. This happens e.g. in the declaration of variables, or the allocation of a new record on the heap. In `Elegant`, this problem has been overcome by requiring that every variable declaration is accompanied by an initial expression. Every allocation of a record object requires initial expressions for some of the fields in the type declaration and initial expressions for the remaining fields as arguments to the allocation function.

- **Deallocation.**
The ability to deallocate objects explicitly in imperative languages is very error-prone, since it might lead to dangling references. In `Elegant`, no deallocation is available, thus circumventing the problem. A garbage collector would be a proper alternative, reducing memory requirements, but increasing the complexity of the implementation considerably and requiring a non-neglectable run-time overhead.

- **Return in functions.**
One of the other consistency checks `Elegant` performs is a check on the presence of a return statement at all possible end-points of a function. A function that might terminate without delivering a result is detected and not accepted. Moreover, statements after a return statement are unreachable and reported as a warning.

Sub-typing of parameterized types

A problem that is related to the leak described in the previous section is the sub-typing of parameterized types. When e.g. $S < T$, is then $List(S) < List(T)$? The answer is that in a pure-functional language, excluding side-effects, it indeed is, but again, due to the combination of aliasing and side-effects, this is not the case in `Elegant`, as the following example illustrates.

```
LOCAL sl : List(S) = ...
      tl : List(T) = ...
      s : S = ...
      t : T = ...
-> tl := sl              %% Assume S < T => List(S) < List(T)
   tl.head := t          %% t is a valid instance for tl.head
   s := sl.head          %% s is assigned a T, causing a type leak
```

Since this case is much less rare than the previous one, the induced sub-typing for parameterized types is not present in Elegant. This is disappointing, since it would often be convenient to have this relationship. The underlying cause is that fields are both readable (giving rise to co-variance for parameterized types) and assignable (giving rise to contra-variance). A way out would be to distinguish the 2^n different types for all possible combinations of assignability for n fields of a type. This could e.g. be done by allowing assignment only to instances of a type Ref(T), like in Algol 68. A parameterized type is then co-variant, when the types of its fields are. Since the Ref-type would be no-variant, a single assignable field would prohibit a sub-type relation. Although this solution is certainly possible, it would lead to an exponential amount of incomparable variants of a type, and we fear that this explosion would give rise to many more problems than the absence of co-variance for parameterized types. It would, nevertheless, be interesting to experiment with this solution to find out whether our fears are justified or not.

8.2.7 The module system

Elegant has a rather traditional module system. Each module comes in pairs, a specification unit and an implementation unit. The specification unit can list types and the signatures of globals and functions, where the implementation unit provides their implementation.

Scopes

We believe that a module system should obey the property that, when an identifier x is used in a module A while it is declared outside of A, it is always possible to find out in which module the declaration of x resides, using only the text of A. In order to meet this requirement, assuming B is declaring x, A should refer to B.x or explicitly import x from B.

In practice, this requirement needs some relaxation. The Elegant prelude declares many functions, like integer addition, that one does not want to import explicitly. Thus, there exists a so-called *export clause*, which is used for each function exported by the prelude. Such functions need not be imported explicitly, but are implicitly available in each module importing the prelude (which is every module). The export clause places the declaration in a global scope, rather than in a scope local to the module. This export mechanism is also available for users, thus relaxing the requirement.

There is another case where the explicit importing of identifiers is cumbersome and that is the importing of types. It is very inconvenient to import all the types needed, or to use the M.T notation for each type. Therefore, types are always placed in the global scope, as if they had an export clause. This prevents the occurrence of nasty coding standards as e.g. in the case of Modula-3, where an abstract data type A is modeled by a module A with a type with the fixed name `T` and one always has to write e.g. `Stack.T` when referring to a stack type, which is a twistéd way of type naming.

Interface to C

`Elegant` offers a mechanism to interface with C functions and types. One may declare externally implemented types by the phrase `Type FROM C-Module`. In this way, the primitive types like `Int` and `Bool` are defined in the prelude. In general, an external type has no structure, but any field access on it is allowed, giving rise to an untyped expression. It is also possible to import a C module `M` and to refer to variables by the phrase `M.v` or to call functions by means of `M.f(...)`. No property of these C constructs is assumed, and hence no (type) checking is performed. A much better way to interface with C is to define a signature for external functions (or even macros) in an `Elegant` specification unit. For example, one can declare

```
SPEC UNIT UnixProcess

TYPE ExitCode = (Success | Failure)

RULES

Exit:[x : ExitCode]              FUNCTION exit
```

to map the `Elegant` function `UnixProcess.Exit` onto the C function `exit`.
When one wants to pass a C type to a polymorphic function, `Elegant` needs to know the size of the type. This should be declared by means of a macro `SIZE_`T for an external type T that expands to 1, 2, 4 or 8.

Subtyping over module boundaries

An unpleasant limitation of separately compiled modules is the inability to combine efficient pattern matching with extensible types. In object-oriented programming it is common practice to declare a supertype with default operations in one module and sub-types with specific operations in other modules. The pattern matching compilation technique of section 6.4 requires that both all sub-types of a type and all rules for a function are known at compilation time. Thus it is not possible to extend a type or a function later on in another module, but this seems the price to be paid for such an efficient pattern matching implementation. Since pattern matching and its efficient implementation are corner stones of `Elegant` (as well as of most functional languages), we are happy to pay that price.

8.3 Performance evaluation

In this section we present the performance of Elegant and show how the techniques described in this thesis play a crucial rôle in the efficiency of Elegant. We discuss the main efficiency enhancement techniques grouped per compiler component. After that, we present a performance benchmark of the Elegant self-generation.

8.3.1 General

Obtaining a reasonable performance for rather abstract languages like Elegant requires that advanced compilation techniques are used. Although a 'high' level language (C) has been used as a target language, this by no means implies that the target language should be used as it normally is, namely for software engineering and structured programming. The target code need not be structured at all, as long as the generator and its associated input language are. In section 8.3.9 we elaborate on this. Basic operations which are frequently used should be efficiently implemented and no means, even conceptually less acceptable ones, should be avoided to achieve this. In this respect a major contribution to Elegant's efficiency is the use of an efficient memory allocator. First of all, memory is allocated from the operating system (typically Unix) in large chunks, which are re-allocated by the Elegant run-time system. Since Elegant has been used mainly as a tool for creating batch-oriented programs, no memory de-allocation mechanism has been implemented. This allowed us to keep the allocator simple. Many architectures require the alignment of heap-pointers on a long word boundary. This means that rounding of the size of objects (which happen to be in bytes) to multiples of 4 is necessary[2]. This rounding can be avoided if the size is already a multiple of 4, which it is in the vast majority of cases. In the great majority of these cases, this size is C compile-time computable by means of constant expressions, using the sizeof operator. By trusting the C compiler in its job of constant folding, a conditional statement of the form:

```
if (size % BYTESPERWORD == 0) Fast_Allocate (&x, size);
else General_Allocate (&x, size);
```

is transformed into a single call to the appropriate allocation function, where the former allocation function does not need to perform the required alignment. This simple trick gains about 10% in performance.

8.3.2 Scanning

Scanning is a time-consuming task, since it requires the inspection of all characters in the input string one by one. The main speed-up of a scanner comes from the following technique. A scanner needs a token buffer that contains at least the characters of the current terminal symbol. The operating system maintains an input buffer for file-IO. These buffers should coincide to avoid copying of characters from the input buffer to the token buffer. To

[2]Even to a multiple of 8 on some systems when double precision floating point numbers are used.

achieve this, a fairly large input/token buffer is used (10k characters) that is filled by means of a low level system call (Unix `read`). One should never use the standard character based IO (like Unix `getch`) in lexical scanners!
Of much less importance, but still resulting in interesting speedups, is the efficient construction of the scanner itself. Each of the scanning functions ('states' for those who prefer automata) is mapped onto a C-function. These functions call each other under the consumption of one character. Functions that are used only once or which are trivial are in-lined. Tail recursion elimination is applied. When a function can call many different other functions, it selects these by means of a switch (case) statement. Otherwise it uses if-statements. The start state always uses a switch statement under this condition. As a result, the main scanning function consists of a huge switch statement, typically between 5000 and 10,000 lines for moderate languages. To a large extent, this is caused by the in-lining of the 'next character' function.
This rather voluminous main function can be made smaller but slower by not in-lining the 'next character' function and by using a keyword table. Identifiers can be looked up in this table before recognizing them as such. Normally, keywords are detected as separate symbols by the scanner automaton.
A minor contribution to efficient scanners results from an efficient identifier hashing algorithm.

8.3.3 Parsing

Compared to lexical scanning, parsing is of minor importance. It usually requires less than 5% of the total compilation time and hence a rather large saving of 20% in parsing speed would only speed up the total compilation by 1%. Of course, one should not be unnessary inefficient in parsing. The error detection implementation is of relative importance here. Using bitsets in representing follower sets is both time and space efficient, since they can be inspected quickly and do not require heap allocation. A recursive ascent parser can use the same optimization techniques as a lexical scanner, since both are sets of mutually recursive functions. Back-tracking with continuations can use stack-allocation for the continuations due to their stack-wise life time.

8.3.4 Attribute evaluation

The techniques that are used in the efficient implementation of attribute evaluation have already been presented in section 4.9. What is noteworthy here however is the following observation. `Elegant` attribute grammars are usually concise since they do not require the addition of extra attributes for the removal of cyclic definitions. Hence, fewer attributes need to be allocated than for e.g. ordered attribute grammars. Moreover, the demand-driven evaluation technique allows us to assign independently an evaluation function to each out-attribute of different instances of production rules for the same non-terminal symbol at parse-time. As a result, no interpretation of the parse tree is required at attribute evaluation time in finding the proper attribute definition. `Elegant` does not even construct a parse tree: it immediately constructs the attribute dependency graph in the form of lazy attributes!

8.3.5 Pattern matching

As we have seen in section 6.4, each selector is inspected at most once in the selection of the proper rule for a function. This selection, by means of a type-case expression, can be mapped onto a C switch statement by uniquely numbering the types in a type-hierarchy and by giving each object an additional field holding the number of its type. This number is then used as the selector in the switch statement.
The nested case expressions can be mapped onto nested switch statements. If a case expression occurs more than once however, its code is generated separately from the switch statement and the associated case expression jumps towards it by means of a goto statement. In this way, sharing is obtained and the automaton is kept minimal. In the same way, bodies of rules that are used in multiple branches of the nested case expressions are shared. Thus the code size is proportional to the number of different case expressions and the number of rules.

8.3.6 Polymorphism

As explained in section 6.5.7, each polymorphic function is compiled several times, for different sizes of its polymorphic arguments. Each of these compiled functions is as efficient as a corresponding non-polymorphic one would be. Hence, polymorphism does not impose a time penalty, only a space penalty. The latter can become fairly large when functions that are polymorphic in many arguments are used. In practice, only a fraction of the C-functions generated for such polymorphic functions is actually used. Thus, a smart enough linker might get rid of the redundant ones. Unfortunately, such a linker has yet to be found.

8.3.7 Relations

Relations can be seen as the product of pattern matching and attribute evaluation. Hence, they show the same performance as these, which makes them, in combination with their expressive power and static cyclicity check, an attractive sub-formalism.

8.3.8 Iteration

Iteration is mapped onto while loops. The destruction and construction functions for the predefined types (e.g. `List`) are implemented by means of macros and as a result, a list iteration is as efficient as a while loop traversing the list. It is however much more readable, general and concise in its notation.

8.3.9 C as a target language

In this section we discuss the use of the programming language C as a target language for a compiler. Although this language has received a good deal of criticism (see e.g. [PE88, Sak88, Sak92]), it is our impression that it is almost ideally suited as a target language. For this purpose, C is used in a way that is very different from the structured programming style. It is not important that the target code is structured, but that its generator is. One could say that C is a very good portable assembler. In the next sections we discuss particular aspects of C that have been important for the generation of code for `Elegant`.

The preprocessor

The often detested preprocessor has been indispensable in the implementation of polymorphism. It made the efficient implementation of iteration possible. It allowed for the definition of many basic functions in the prelude (like e.g. integer addition) without any performance penalty. Usually, such operations are built into the compiler to achieve this.

An abundance of operators

C offers an extraordinary amount of operators. Although C has been criticized for this [PE88], they were often very convenient, e.g. in the generation of code for an expression where an extra variable had to be allocated and initialized. The initialization can be done in conjunction with the expression itself, by means of the expression composition operator '`,`'. The conditional boolean operators `&&` and `||` are indispensable. The conditional expression `c ? t : e` has been very helpful on many occasions, which brings us to the following point.

Almost an expression language

The fact that C is almost an expression language, e.g. that assignment is an expression rather than a statement, is invaluable. It allows the initialization of extra variables as explained above. The fact that C is not a full expression language, like e.g. Algol 68, is something we have often regretted. Especially the inability to use while and switch statements in expressions has prevented us from allowing the iteration expression everywhere in `Elegant` and from introducing a type case expression rather than a type case statement in `Elegant`. One could say that had C been an expression languages, `Elegant` would have been one!

Function variables

As illustrated by the implementation of lazy evaluation, the use of function valued variables is indispensable for `Elegant`. Although their syntax is obscure, to say the least, a generator does not bother about this syntax. We feel that they would be used much more often by C programmers however, if only they had a better syntax. To experience this, one should try to define a structured type containing a field of a function type requiring a pointer to that same structured type as an argument. This is exactly what is used in the implementation of lazy evaluation.

Switch statement

The switch statement, although it has a completely outdated semantics (allowing, if not encouraging, the falling through from one case branch into the next one), has been indispensable in the implementation of our many 'automata', used in scanning, parsing and pattern matching. A switch statement is by no means unique to C however.

Low level facilities

C offers many low level facilities and very little protection. For software engineering this is disastrous of course, but for code generation it is very convenient. E.g. the trick that transforms an (environment, evaluation function) pair into a (value, identity function) pair for lazy evaluation relies on mixing up different types in unions. Type casting is indispensable in the implementation of polymorphism. Pointer arithmetic is used in the calculation of the sizes of certain objects that need to be allocated.

Return statements

The return statement is very useful in a programming style where a function first detects certain special cases by means of an `IF c THEN s RETURN(e) FI` statement. Although this style is often replaced by pattern matching in `Elegant`, it still appears rather frequently and could not have been compiled without a return statement. The use of a return statement is also very convenient in the compilation of pattern matching. The nested case expressions used there never terminate normally: they terminate by a return statement from a rule body. Hence, several such expressions can be cascaded, connected by jumps, as explained above.

Debugging

C allows the use of so-called line-directives. These are used to map a line in a C source file onto another line, possibly in another file. Such a mapping can be exploited by a debugger, which may thus refer to the other file, e.g. an `Elegant` source file. In this way, it is very easy to obtain a simple source level debugger for `Elegant`. It can be used to set breakpoints in the `Elegant` text and to single step through it. The inspection of variables is more cumbersome however, since they should usually be addressed by C expressions.

Portability and compiler quality

C happens to be very portable. The `Elegant` tools compile onto about 200,000 lines of C code, and we have very seldom encountered a problem in the porting of `Elegant` to other systems. If there are problems, they are usually caused by optimizers, which are easily silenced. At the time when we used to generate Modula-2 our code was practically unportable, due to the low quality of Modula compilers.
The good standardization of C run-time libraries is of lesser importance, since `Elegant` uses its own ones.

8.3.10 Self generation benchmark

In this section we present the results of some measurements of the performance of `Elegant` in generating itself. These measurements have been performed on a sun 4/670 processor, containing two integer and two floating points running at 40MHz and containing 128M memory.
The total compilation time for the sources of `Elegant` is 43.34 seconds, compiling 22,000 lines of source code. As specification units are passed multiple times to the subsequent separate compilations of implementation units, the total amount of input lines processed by `Elegant` is 75,508 lines, meaning a compilation speed of 1740 lines/second. A total amount of 169,019 lines of C code is generated by these compilations. The total amount of heap space allocated is 76.4 Mb, about 1.76 Mb per second, allocated and filled with useful information. The time and space spent is distributed over the different compilation phases in the following way:

phase	time [s]	time [%]	space [Mb]	speed
initialization	0.80	1.8		
scanning	6.91	16.0		11k lines/s
parsing+attribute creation	9.04	20.9	34.7	8.4k lines/s
cycle check	0.95	2.2	3.7	700 rules/s
attribute evaluation	17.53	40.4	29.6	
code generation	8.11	18.7	8.4	21k lines/s
total	43.34	100	76.4	1740 lines/s

One should keep in mind that attribute evaluation time includes the construction of the compiler administration and all context checks. These can not easily be separated, due to the demand-driven evaluation of attributes, which cause some attributes to be evaluated for the purpose of context checks and others later on.
It is interesting to take a closer look at the compilation of the `Elegant` attribute grammar. This has a size of 2878 lines and the total input to the compiler, due to the required specification units, is 4760 lines. This compiles in 4.44 seconds, giving a speed of 1070 lines/second. Code generation produces 42,398 lines of C. As the attribute grammar of `Elegant` is a complex one, mainly due to overloading resolution, the cyclicity check requires a good deal of the compilation time. Remember that this check is intrinsically exponential. For average attribute grammars, the cyclicity check time is less than 5% of the total compilation time. It also allocates a good deal of memory since all potential cycles, i.e. all attribute dependency paths, are stored in order to be able to give a sensible error message in the case of a cycle.

phase	time [s]	time [%]	space [Mb]	speed
initialization	0.02	0.5		
scanning	0.36	8.1		13k lines/s
parsing+attribute creation	0.37	8.3	3.5	13k lines/s
cycle check	0.76	17.2	3.5	540 rules/s
attribute evaluation	1.54	34.7	3.2	
code generation	1.39	31.3	1.7	30k lines/s
total	4.44	100	11.9	1070 lines/s

As these times are not only the times of an `Elegant` compilation but also those of an `Elegant` generated compiler (as `Elegant` is self generating) they are both representative for compiler generation speed as well as for the speed of compilers generated by `Elegant`. As the reader can deduct, this is absolute production quality performance.

8.4 Conclusions

In this thesis we have shown how the design of the compiler generator and programming language `Elegant` has been inspired by the field of functional programming. The abstraction mechanisms found in modern functional languages have influenced the `Elegant` formalism to a large extent. The most important contributions to `Elegant` from functional programming are:

- Parameterized types.
  `Elegant` offers parameterized types with an arbitrary number of parameters.

- Polymorphic functions.
  `Elegant` functions can be polymorphic. The `Elegant` compiler substitutes the lowest common supertypes that are consistent with the types of actual arguments of a function call.

- Higher order functions.
  `Elegant` functions are first-class citizens that may be passed as arguments, returned as results and stored in variables.

- λ-abstraction.
  `Elegant` offers a function valued expression that delivers a parameterized closure, capturing the free non-global variables in an environment.

- Laziness.
  `Elegant` offers three forms of demand-driven evaluation. It does so by offering three distinct, predefined types.

`Lazy(T)`	Instances of this type are evaluated at most once.
`Closure(T)`	Instances of this type are evaluated over and over again.
`Incr(T)`	Instances of this type are evaluated when their value may have changed.

  The fact that lazy types are explicit allows us to distinguish between these different forms and enables the use of laziness in an imperative environment. The use of unevaluated objects of an empty type, i.e. unevaluated side-effects, appears to be indispensable in code generation.

- Comprehensions.
  `Elegant` offers very general comprehensions, based on user-defined destruction and construction of data-objects. In many cases, they offer a convenient and concise alternative to iteration and recursion.

- Pattern matching.
  `Elegant` pattern matching is based on type analysis. By virtue of this foundation, a rule can be seen as a partial function.

Although `Elegant` has been inspired by functional languages, it is an imperative language itself. The reasons for this are twofold. On the one hand, the ability to use side-effects (with care) can enhance efficiency and also readability. On the other hand, most abstraction mechanisms that make functional languages so pleasant to use need not be restricted to functional languages. The `Elegant` language clearly shows that this is the case.
Other important ingredients of `Elegant` are based on the formalism of attribute grammars:

- Attribute grammars.
  Since `Elegant` started as a high level compiler generator tool, it offers attribute grammars as a sub-formalism. The lazy implementation of attribute evaluation has allowed the definition of a very general scheme for the construction of attribute grammars and the support of the very general class of pseudo circular attribute grammars. Strict (compile time computable) attributes may be used to steer the parser, while non-strict attributes can be used in any non-circular fashion.

- Relations.
  The formalism of attribute grammars has been extended in `Elegant` to so-called 'relations'. Instead of a parser, pattern matching on data objects is used in the selection of a production rule. Attributes do not compute properties of the input string, but of the data structure that is traversed by the relations. Especially in code generation, relations are a blessing.

And of course, `Elegant`, being an imperative language, has been inspired by other imperative languages, which is reflected by the following constructions:

- Sub-typing.
  `Elegant` supports sub-typing, in a linear fashion. By regarding patterns as subtypes, pattern matching is easily combined with sub-typing. Enumerated and empty types appear to be special cases of the `Elegant` type system.

- Assignable variables.
  Being an imperative language, `Elegant` supports assignment to variables. Our experience shows that it should be used with care. Most bugs could be traced back to unexpected side-effects to non-local variables. Nevertheless, in some cases side-effects enhance readability and in many cases, enhance efficiency.

- Empty types.
  `Elegant` generalizes the notion of a single empty type to multiple, user definable empty types, which offer the ability to distinguish different classes of side-effects by means of type information.

- Coercions.
  Coercions form an important aspect of `Elegant` programming. Coercions are user-defined and some convenient ones are predefined in the prelude. When used with care, i.e. for representation transformations from one type into another, they are very useful. Especially in code generation they add considerably to the readability of a program.

This combination of ingredients from both functional and imperative programming languages makes `Elegant` a language that combines a high level of abstraction with an efficient implementation. `Elegant` programs are much more concise and structured, better protected, and more reusable than C or C++ programs, yet as efficient as the latter ones. `Elegant` programs are comparable to (typed) functional programs with regard to conciseness, protection and reusability, but much more efficient.
`Elegant` has always lived up to its own expectations: its very first application was its self-generation, and every new version has been self-generated since. It could not have been constructed with the current conciseness and efficiency if it had not been written in `Elegant`. Without `Elegant`, `Elegant` could not have evolved to what it is now!
In the foundation of certain key ingredients of `Elegant`, the use of algebraic program transformations has been of great value:

- They allowed the transformation of recursive descent into recursive ascent parsers. The transformations used for this purpose have brought to light that bottom-up parsers can be regarded as the iterative versions of top-down parsers. This fact can be regarded as an important contribution to parsing theory. It makes the use of automata in the foundation of bottom-up parsers unnecessary. Both top-down and bottom-up parsers can be expressed and analyzed in a functional language.

- They have been used in the expression of lexical scanners in functional languages. The algebraic transformations applied to a functional scanning algorithm appear to be isomorphic to the conventional transformations to finite automata for regular expressions.

- They have been used in the derivation of an efficient pattern matching algorithm based on the view of patterns as sub-types. This algorithm is used by `Elegant` in the compilation of pattern matching and is one of the key ingredients for the efficiency that `Elegant` programs exhibit.

We have been using the algebraic program transformations in the domain of functional programming languages. We have deliberately not lifted it to the field of morphisms and category theory, which seems to be the current fashion. We believe that only a minor loss of generality is suffered when unlifted transformations are applied to real-world problems and that the use of transformations on the programming level brings the practical application of algebraic program transformations much closer to the 'average' programmer than category theory does.
The often rediscovered concept of memoization has been incorporated in `Elegant` in the form of 'memo types', types for which the creation function is memoized. As memoization is transparent in functional languages, a memoized version of a function is equivalent to the non-memoized version, but potentially more efficient. Many imperative algorithms, among which the whole field of dynamic programming, appear to use memo functions in disguise. The Warshall, Earley and Tomita algorithm have been identified as such in this thesis.
This thesis is a reflection of the issues that have come up in the design of the system and language which `Elegant` is. We have discussed the more general and interesting issues

in this thesis, but many more have been studied in the last six years. It has always been a joy and a challenge to encounter new problems. Especially the balance between the clean and the efficient has always been fascinating and both the current design of the `Elegant` language, as well as its implementation are a reflection of the search for this balance. When a certain problem was hard to implement, this was in the vast majority of the cases due to one of the following two reasons: either the problem was too specific and had to be generalized, or the current `Elegant` implementation needed a new or more general abstraction. When one has the comfortable ability to design, modify and implement its own language, the latter case can be dealt with: one simply extends the current language definition. Of course, this has been a problem for users from time to time. Most of the time, they were not waiting for a new release, but rather forced to use one. To relieve these problems, we have tried to keep subsequent versions upward compatible as long as possible and to support the upgrading of source code from one major releases to a next one. Nevertheless, the new abstractions offered by a new release were often warmly welcomed by the users, especially by the more demanding ones, who were aiming at an implementation that used all language features that were useful for it.
For these reasons, I would like to thank the `Elegant` users for their support, inspiration, challenging remarks and their willingness in accepting new releases. Without them, `Ele-gant` would not have been what it is now and this thesis would probably never have been written.

Bibliography

[AK82] J. Arsac and Y. Kodratoff. Some Techniques for Recursion Removal from Recursive Procedures. *ACM Transactions on Programming Languages and Systems*, 4(2):295–322, 1982.

[AM92] Lex Augusteijn and Séan Morrison. Compiling EBNF to `Elegant`'s attribute grammar definition language, release 6. Document RWB-510-re-92242, Philips Research Laboratories, Eindhoven, the Netherlands, November 1992.

[Ame87] Pierre America. POOL-T — a parallel object-oriented language. In Akinori Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 199–220. MIT Press, 1987.

[Ame88] Pierre America. Definition of POOL2, a parallel object-oriented language. ESPRIT Project 415-A Document D0364, Philips Research Laboratories, Eindhoven, the Netherlands, April 1988.

[Ame89] Pierre America. Language definition of pool-x. PRISMA Project Document P0350, Philips Research Laboratories, Eindhoven, the Netherlands, January 1989.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison Wesley, 1986.

[Aug84] L. Augustsson. A compiler for lazy ML. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 218–227, Austin, 1984.

[Aug85] L. Augustsson. Compiling pattern matching. In Jouannaud, editor, *Conference on Functional Programming Languages and Computer Architecture, Nancy*, number 201 in Lecture Notes in Computer Science, pages 368–381. Springer Verlag, Berlin, 1985.

[Aug90a] Lex Augusteijn. The Elegant compiler generator system. In P. Deransart and M. Jourdan, editors, *Attribute Grammars and their Applications*, Lecture Notes in Computer Science 461, pages 238–254. Springer Verlag, Berlin, 1990.

[Aug90b] Lex Augusteijn. Analysis of the programming language pascal by the construction of a compiler front-end. POOMA Project Document M0154, Philips Research Laboratories, Eindhoven, the Netherlands, June 1990.

[Aug90c] Lex Augusteijn. Earley Made Easy. POOMA Project Document M0148, Philips Research Laboratories, Eindhoven, the Netherlands, May 1990.

[Aug91] Lex Augusteijn. Operator precedence parsing with attribute look-ahead. Nat. Lab. Technical Note 259/91, RWR-513-RE-91159, Philips Research Laboratories, Eindhoven, the Netherlands, 1991. To appear.

[Aug92a] Lex Augusteijn. An Alternative Derivation of a Binary Heap Construction Function. In R.S. Bird, C.C. Morgan, and J.C.P. Woodcock, editors, *Mathematics of Program Construction*, number 669 in Lecture Notes in Computer Science, pages 368–374, 1992.

[Aug92b] Lex Augusteijn. Definition of the programming language `Elegant`, release 6. Document RWB-510-re-92244, Philips Research Laboratories, Eindhoven, the Netherlands, November 1992.

[Aug92c] Lex Augusteijn. The `Diagrams` EBNF to PostScript compiler, release 6. Document RWB-510-re-92245, Philips Research Laboratories, Eindhoven, the Netherlands, November 1992.

[Aug92d] Lex Augusteijn. The `Elegant` compiler generator system, release 6. Document RWB-510-re-92241, Philips Research Laboratories, Eindhoven, the Netherlands, November 1992.

[Aug92e] Lex Augusteijn. The `Elegant` library, release 6. Document RWB-510-re-92247, Philips Research Laboratories, Eindhoven, the Netherlands, November 1992.

[Aug92f] Lex Augusteijn. The `Elegant` scanner generator definition language `ScanGen`, release 6. Document RWB-510-RE-92243, Philips Research Laboratories, Eindhoven, the Netherlands, November 1992.

[Bac78] John Backus. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Communications of the ACM*, 21(8):613–641, 1978.

[Bac88] Roland Backhouse. An Exploration of the Bird-Meertens Formalism. Technical Report CS 8810, University of Groningen, PO Box 90, 9700 AV Groningen, the Netherlands, 1988.

[Bar85] H.P. Barendregt. *The Lambda Calculus - Its Syntax and Semantics*. North Holland, 1985.

[BD77] R.M. Burstall and John Darlington. A Transformational System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.

[BGJ89] R.S. Bird, J. Gibbons, and G. Jones. Formal Derivation of a Pattern Matching Algorithm. *Science of Computer Programming*, 12:93–104, 1989.

[BH87] R.S. Bird and John Hughes. The alpha-beta algorithm: An exercise in program transformation. *Information Processing Letters*, 24:53–57, 1987.

[Bir80] R.S. Bird. Tabulation Techniques for Recursive Programs. *Computing Surveys*, 12(4):403–417, 1980.

[Bir84a] R.S. Bird. The Promotion and Accumulation Strategies in Transformational Programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, 1984.

[Bir84b] R.S. Bird. Using Circular Programs to Eliminate Multiple Traversals of Data. *Acta Informatica*, 21:239–250, 1984.

[Bir89] R.S. Bird. Algabraic identities for Program Calculation. *The Computer Journal*, 32(2):122–126, 1989.

[Boi92] Eerke A. Boiten. Improving recursive functions by inverting the order of evaluation. *Science of Computer Programming*, 18:139–179, 1992.

[Boo79] Hendrik Boom. Private communication. IFIP Working Group 2.1. Jablonna, Warschau, 1979.

[BPR88] G. L. Burn, S. L. Peyton Jones, and J.D. Robson. The Spineless G-Machine. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, 1988.

[BSI82] BSI. Specification for computer programming language pascal. Technical Report BS 6192:1982, British Standards Institute, 1982.

[BvdW92] Roland Backhouse and Jaap van der Woude. A Relational Theory of Datatypes. In *Lecture Notes of the STOP 1992 Summerschool on Constructive Algorithmics*, 1992.

[BvZ89] Lee Alton Barford and Bradley T. vander Zanden. Attribute Grammars in Constraint-based Graphics Systems. *Software—Practice and Experience*, 19(4):309–328, 1989.

[BW88] R.S. Bird and P.L. Wadler. *Introduction to Functional Programming*. Prentice-Hall International, Hemel Hempstead, 1988.

[BW90] M. Barr and C. Wells. *Category Theory for Computing Science*. Prentice Hall, 1990.

[Car84] Luca Cardelli. Compiling a Functional Language. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 208–217, Austin, 1984.

[Car87] Luca Cardelli. Basic Polymorphic Typechecking. *Science of Computer Programming*, 8:147–172, 1987.

[Cas91] Mark Cashman. The Benefits of Enumerated Types in Modula-2. *ACM SIGPLAN Notices*, 26(2):35–39, 1991.

[CDG+88] Luca Cardelli, Jim Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 Report. Technical Report 31, Digital Systems Research Center, 1988.

[CDJ+89] Luca Cardelli, Jim Donahue, Mick Jordan, Bill Kalsow, and Greg Nelson. The Modula-3 Type System. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 202–212, 1989.

[CHC90] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritence Is Not Subtyping. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 125–135, 1990.

[Chu41] A. Church. *The Calculi of Lambda Conversion*. Princeton University Press, Princeton, 1941.

[Coo89] W.R. Cook. A Proposal for Making Eiffel Type-safe. *The Computer Journal*, 32(4):305–311, 1989.

[CW85] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[dHO87] Peter den Haan and Hans Oerlemans. Attribute grammar for the SODOM compiler. ESPRIT Project 415-A Document D0182, Philips Research Laboratories, Eindhoven, the Netherlands, Januari 1987.

[DJL84] Pierre Deransart, Martin Jourdan, and Bernard Lorho. Speeding up Circularity Tests for Attribute Grammars. *Acta Informatica*, 21:375–391, 1984.

[DJL88] Pierre Deransart, Martin Jourdan, and Bernard Lorho. *Attribute Grammars*. Number 323 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1988.

[DN66] Ole-Johan Dahl and Kristen Nygaard. SIMULA – An Algol-based Simulation Language. *Communications of the ACM*, 9(9):671–678, 1966.

[Ear70] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[EF89] Joost Engelfriet and Gilberto Filé. Passes, Sweeps, and Visits in Attribute Grammars. *Journal of the ACM*, 36(4):841–869, 1989.

[Eng84] J. Engelfriet. Attribute grammars: Attribute evaluation methods. In B.Lorho, editor, *Methods and Tools For Compiler Construction*, pages 103–138. Cambridge University Press, 1984.

[Far92] Charles Farnum. Pattern-based tree attribution. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 211–222, 1992.

[FGJM85] Kokichi Futatsugi, Joseph A. Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 52–66, 1985.

[FM90] You-Chin Fuh and Prateek Mishra. Type Inference with SubTypes. *Theoretical Computer Science*, 73:155–175, 1990.

[FMY92] R. Farrow, T.J. Marlowe, and D.M. Yellin. Composable Attribute Grammars: Support for Modularity in Translator Design and Implementation. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 223–234, 1992.

[Fok92a] Jeroen Fokker. The Systematic Construction of a One-Combinator Basis for Lamnbda-Terms. *Formal Aspects of Computing*, pages 776–780, 1992.

[Fok92b] Maarten M. Fokkinga. *Law and order in algorithmics*. PhD thesis, Twente University, 1992.

[FP91] Tim Freeman and Frank Pfenning. Refinement Types for ML. *ACM SIGPLAN Notices*, 26(6):268–277, 1991.

[Fre90] Bjorn N. FreeMan-Benson. Kaleidoscope: Mixing Objects, Constraints, and Imperative Programming. In *Proceedings of the 1990 ECOOP/OOPSLA Conference*, pages 77–88, 1990.

[Fro92a] R.A. Frost. Constructing Programs as Executable Attribute Grammars. *The Computer Journal*, 35(4):376–389, 1992.

[Fro92b] Richard A. Frost. Guarded Attribute Grammars. *ACM SIGPLAN Notices*, 27(6):72–75, 1992.

[GB89] Peter Grogono and Anne Bennett. Polymorphism and Type Checking in Object-Oriented Languages. *ACM SIGPLAN Notices*, 24(11):109–115, 1989.

[GE90] Josef Grosch and H. Emmelmann. A Tool Box for Compiler Construction. Technical Report Compiler Generation Report No. 20, GMD Forschungsstelle an der Universität Karlsruhe, 1990.

[Gie88] Robert Giegerich. Composition and Evaluation of Attribute Coupled Grammars. *Acta Informatica*, 25:355–423, 1988.

[GMW79] M.J. Gordon, A.J. Milner, and C.P. Wadsworth. *Edinburgh LCF*. Number 78 in Lecture Notes in Computer Science. Springer Verlag, 1979.

[Gor65] H. Gordon Rice. Recursion and iteration. *Communications of the ACM*, 8(2):114–115, 1965.

[GR75] Susan L. Graham and Steven P. Rhodes. Practical Syntactic Error Recovery. *Communications of the ACM*, 18(11):639–650, 1975.

[Gro92] Josef Grosch. Transformations of Attributed Trees Using Pattern Matching. In U. Kastens and P. Pfahler, editors, *Compiler Construction*, number 641 in Lecture Notes in Computer Science, pages 1–15. Springer Verlag, Berlin, 1992.

[HF92] Paul Hudak and Joseph H. Fasel. A Gentle Introduction to Haskell. *ACM SIGPLAN Notices*, 27(5):Section T, 1992.

[Hil76] J. Hilden. Elimination of Recursive Calls Using a Small Table of "Randomly" Selected Function Values. *Bit*, pages 60–73, 1976.

[Hin69] R. Hindley. The principal type scheme of an object in combinatory logic. *Trans. Am. Math. Soc.*, 146(Dec.):29–60, 1969.

[HK92] Peter G. Harrison and Hessam Khoshnevisan. A new approach to recursion removal. *Theoretical Computer Science*, 93:91–113, 1992.

[Hor92] Bruce Horn. Constraint Patterns As a Basis For Object Oriented Programming. *ACM SIGPLAN Notices*, 27(?):218–219, 1992.

[Hor93] R. Nigel Horspool. Recursive Ascent-Descent Parsing. *Computer Languages*, 18(1):1–15, 1993.

[Hug85] John Hughes. Lazy memo-functions. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 129–146. Springer Verlag, Berlin, 1985.

[Hug89] John Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

[Hut92] Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, 1992.

[HW90] R. Nigel Horspool and Michael Whitney. Even Faster LR Parsing. *Software—Practice and Experience*, 20(6):515–535, 1990.

[HWA$^+$92] Paul Hudak, Philip Wadler, Arvind, Brian Boutel, Jon Fairbairn, Joseph Fasel, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Simon Peyton Jones, Mike Reeve, David Wise, and Jonathan Young. Report on the Programming Language Haskell, A Non-Strict, Purely Functional Language, Version 1.2. *ACM SIGPLAN Notices*, 27(5):Section R, 1992.

[Jan93] Paul Jansen. An introduction to `Elegant`. Document RWB-510-re-93183, Philips Research Laboratories, Eindhoven, the Netherlands, 1993.

[Joh75] S.C.. Johnson. Yacc – yet another compiler compiler. Technical Report Computing Science Technical Report 32, AT & T Bell Laboratories, Murray Hiil, N.J., 1975.

[Joh84] Thomas Johnsson. Efficient compilation of lazy evaluation. In *Proceedings of the ACM Symposium on Compiler Construction*, pages 58–69, Montreal, 1984.

[Joh87] Thomas Johnsson. Attribute Grammars as a Functional Programming Paradigm. In *Functional Programming and Computer Architectures*, number 274 in Lecture Notes in Computer Science, pages 154–173, Berlin, 1987. Springer Verlag.

[JOR75] M. Jazayeri, W.F. Ogden, and W.C. Rounds. The intrinsical exponential complexity of the circularity problem for attribute grammars. *Communications of the ACM*, 18:697–706, 1975.

[Kas65] T. Kasami. An efficient recognition and syntax analysis algorithm for context-free languages. Technical Report AFCRL-65-758, Air Force Cambridge Research Laboratory, 1965.

[Kas80] Uwe Kastens. Ordered Attribute Grammars. *Acta Informatica*, 13:229–256, 1980.

[Kat84] Takuya Katayama. Translation of Attribute Grammars into Procedures. *ACM Transactions on Programming Languages and Systems*, 6(3):345–369, 1984.

[Ken90] Richard Kennaway. The Specificity Rule for Lazy Pattern Matching in Ambiguous Term Rewrite Systems. In N. Jones, editor, *ESOP '90*, number 432 in Lecture Notes in Computer Science, pages 256–270, Berlin, 1990. Springer Verlag.

[KHZ81] Uwe Kastens, Brigitte Hutt, and Erich Zimmermann. *GAG: A Practical Compiler Generator*. Lecture Notes in Computer Science 141. Springer Verlag, Berlin, 1981.

[Knu65] Donald E. Knuth. On the Translation of Languages from Left to Right. *Information and Control*, 8:607–639, 1965.

[Knu68] Donald E. Knuth. Semantics of Context-Free Languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[Knu71] Donald E. Knuth. Semantics of Context-Free Languages (correction). *Mathematical Systems Theory*, 5(1):95–96, 1971.

[Kos74] C.H.A. Koster. A technique for parsing ambiguous languages. Technical Report FB20, TU Berlin, October 1974.

[Kru88] F.E.J. Kruseman Aretz. On a Recursive Ascent Parser. *Information Processing Letters*, 29:201–206, 1988.

[KS87] M.F. Kuiper and S.D. Swierstra. Using attribute grammars to derive efficient functional programs. In *Computing Science in the Netherlands*, pages 39–52, 1987.

[KvdSGS80] F.E.J. Kruseman Artez, J.L.A. van de Snepscheut, H. Grasdijk, and J.M.H. Smeets. SATHE: Some Aspects of an Algol Implementation. *Software—Practice and Experience*, 10:563–573, 1980.

[LAK92] René Leermakers, Lex Augusteijn, and Frans E.J. Kruseman Aretz. A functional LR-parser. *Theoretical Computer Science*, 104:313–323, 1992.

[Lan64] P.J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6:308–320, 1964.

[Lav88] Alain Laville. Implementation of Lazy Pattern Matching Algorithms. In *ESOP '88*, number 300 in Lecture Notes in Computer Science, pages 298–316, 1988.

[Lee92] René Leermakers. Recursive ascent parsing: from Earley to Marcus. *Theoretical Computer Science*, 104:299–312, 1992.

[Lee93] René Leermakers. The use of bunch notation in parsing theory. In *Proceedings of the 3th International Workshop on Parsing Technologies (1993)*, 1993.

[Lin90] C. Lins. Programming Without Enumerations in Oberon. *ACM SIGPLAN Notices*, 25(7):19–27, 1990.

[Mal89] Grant Malcolm. Homomorphisms and Promotability. In J.L.A. van de Snepscheut, editor, *Mathematics of Program Construction*, number 375 in Lecture Notes in Computer Science, pages 335–347, Berlin, 1989. Springer Verlag.

[Mar80] M.P. Marcus. *A Theory of Syntactic Recognition of Natural Language*. PhD thesis, MIT, Cambridge, MA, 1980.

[McC62] J. McCarthy. *The LISP 1.5 Programmers' Manual*. MIT Press, Cambridge, MA, 1962.

[Mee83] Lambert Meertens. Algorithmics: An Algebraic Approach to Algorithm Specification and Construction. Technical report, CWI, October 1983.

[Mic68] D. Michie. "Memo" Functions and Machine Learning. *Nature*, 218:19–22, April 1968.

[Mil78] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[MMM91] John Mitchell, Sigurd Meldal, and Neel Madhav. An extension of Standard ML modules with subtyping and inheritance. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 270–278, 1991.

[Nor91] Peter Norvig. Techniques for Automatic Memoization with Applications to Context-Free Parsing. *Computational Linguistics*, 17:91–98, 1991.

[Oph89] John Ophel. An Improved Mixture Rule For Pattern Matching. *ACM SIGPLAN Notices*, 24(6):91–96, 1989.

[PE88] Ira Pohl and Daniel Edelson. A to Z: C Language Shortcomings. *Computer Languages*, 13(2):51–64, 1988.

[Pen86] Thomas J. Penello. Very Fast LR Parsing. *ACM SIGPLAN Notices*, 21(7):145–151, 1986.

[Pey87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, Berlin, 1987.

[PMN88] Carl G. Ponder, Patrick C. McGeer, and Antony P-C. Ng. Are Applicative Languages Efficient? *ACM SIGPLAN Notices*, 23(6):135–139, 1988.

[PP76] Helmut Partsch and Peter Pepper. A Family of Rules for Recursion Removal. *Information Processing Letters*, 5(6):174–177, 1976.

[Rob88] George H. Roberts. Recursive Ascent: An LR Analog to Recursive Descent. *ACM SIGPLAN Notices*, 23(3):23–29, 1988.

[RS82] Kari-Jouko Räihä and Mikko Saarinen. Testing Attribute Grammars for Circularity. *Acta Informatica*, 17:185–192, 1982.

[RT88] T. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Spinger Verlag, New York, 1988.

[Sak88] Markku Sakkinen. On the darker side of C++. In *ECOOP Proceedings*, pages 162–176, 1988.

[Sak91] Markku Sakkinen. Another defence of enumerated types. *ACM SIGPLAN Notices*, 26(8):37–41, 1991.

[Sak92] Markku Sakkinen. The Darker Side of C++ Revisited. *Structured Programming*, 13:155–177, 1992.

[Sch24] M. Schönfinkel. Über die Bausteine der mathematischen Logik. *Math. Annalen*, 92:305–316, 1924.

[Sch92] Berry Schoenmakers. Inorder Traversal of a Binary Heap and its Inversion in Optimal Time and Space. In R.S. Bird, C.C. Morgan, and J.C.P. Woodcock, editors, *Mathematics of Program Construction*, number 669 in Lecture Notes in Computer Science, pages 291–301, Oxford, 1992.

[SH92] Theodore S.Norvell and Eric C.R. Hehner. Logical Specifications for Functional Programs. In R.S. Bird, C.C. Morgan, and J.C.P. Woodcock, editors, *Mathematics of Program Construction*, number 669 in Lecture Notes in Computer Science, pages 269–290, Oxford, 1992.

[SK84] J.P. Schaap-Kruseman. Handleiding scanner/parser generator. Technical report, Twente University of Technology, September 1984. In Dutch.

[Sme93] Sjaak Smetsers. *Graph Rewriting and Functional Languages.* PhD thesis, Nijmegen University, 1993.

[Spi89] Mike Spivey. A Categorical Approach to the Theory of Lists. In J.L.A. van de Snepscheut, editor, *Mathematics of Program Construction*, number 375 in Lecture Notes in Computer Science, pages 399–408, Berlin, 1989. Springer Verlag.

[SS90] Harald Søndergaard and Peter Sestoft. Referential Transparency, Definiteness and Unfoldability. *Acta Informatica*, 27:505–517, 1990.

[Tar72] R.E. Tarjan. Depth–first search and linear graph algorithms. *SIAM Journal of Computing*, 1:146–160, 1972.

[Tom86] Masaru Tomita. *Efficient Parsing for Natural Language.* Kluwer Academic Publishers, Boston, 1986.

[Tur79] D.A. Turner. A New Implementation Technique for Applicative Languages. *Software—Practice and Experience*, 9:31–49, 1979.

[Tur82] D.A. Turner. Recursive equations as a programming language. In Darlington et al., editor, *Functional Programming and its Applications*, pages 1–28S. Cambridge University Press, 1982.

[Tur86] D.A. Turner. An Overview of Miranda. *ACM SIGPLAN Notices*, 21(12):158–166, 1986.

[US 83] US Department of Defense. Reference Manual for the Ada Programming Language. Technical Report ANSI-MIL-STD-1815A-1983, DOD, Washington DC, 1983.

[Val75] Leslie G. Valiant. General Context-Free Recognition in Less than Cubic Time. *Journal of Computer and System Sciences*, 10:308–315, 1975.

[VMGS91] Bran Vander Zanden, Brad A. Myers, Dario Giuse, and Pedro Szekely. The Importance of Pointer Variables in Constraint Models. In *Proceedings of the UIST Conference*, pages 155–164, 1991.

[Vog90] Harald Vogt. *Higher Order Attribute Grammars.* PhD thesis, Utrecht State University, Utrecht, the Netherlands, 1990.

[VSK89] H.H. Vogt, S.D. Swierstra, and M.F. Kuiper. Higher Order Attribute Grammars. *ACM SIGPLAN Notices*, 21:131–145, 1989. SIGPLAN Conference on Programming Language Design and Implementation.

[VSK90] H. Vogt, D. Swierstra, and M. Kuiper. On the efficient incremental evaluation of higher order attribute grammars. Technical Report RUU=CS-90-36, Utrecht University, December 1990. In Dutch.

[vWMP$^+$76] A. van WijnGaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens, and R.G. Fischer. *Revised Report on the Algorithmic Language Algol 68.* Springer Verlag, Berlin, 1976.

[Wad85] Philip Wadler. How to Replace Failure by a List of Successes. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 113–128, Berlin, 1985. Springer Verlag.

[Wad87] Philip Wadler. Efficient Compilation of Pattern Matching. In Simon L. Peyton Jones, editor, *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[Wad90] Philip Wadler. Comprehending Monads. In *ACM Conference on Lisp and Functional Programming*, 1990.

[Wad91] Philip Wadler. Is there a use for linear logic? In *Conference on Partial Evaluation and Semantics-Based Program Manipulation*, pages 255–273. ACM Press, 1991.

[Wad92] Philip Wadler. The essence of functional programming. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 1–14, 1992.

[Wan91] Mitchell Wand. Type Inference for Record Concatenation and Multiple Inheritance. *Information and Computation*, 93:1–15, 1991.

[War62] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

[War92] David S. Warren. Memoing for Logic Programs. *Communications of the ACM*, 35(1):93–111, 1992.

[WB89] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 60–76, 1989.

[Wir85] Niklaus Wirth. *Programming in Modula-2.* Springer Verlag, Berlin, third, corrected edition, 1985.

[Wir88a] N. Wirth. From Modula to Oberon. *Software—Practice and Experience*, 18(7):661–670, 1988.

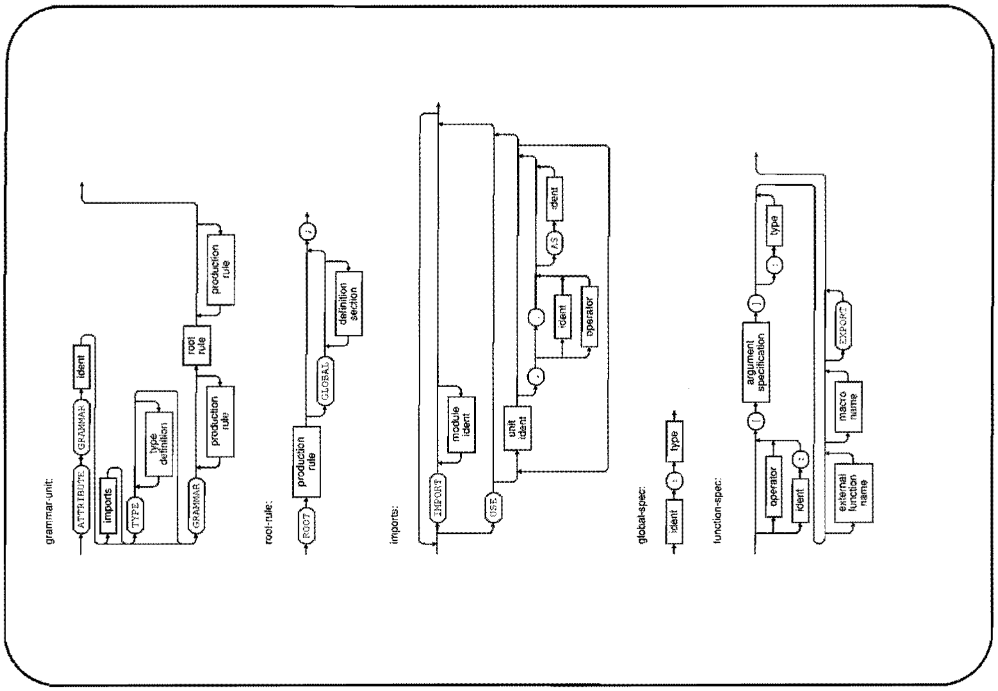
[Wir88b] N. Wirth. The Programming Language Oberon. *Software—Practice and Experience*, 18(7):671–690, 1988.

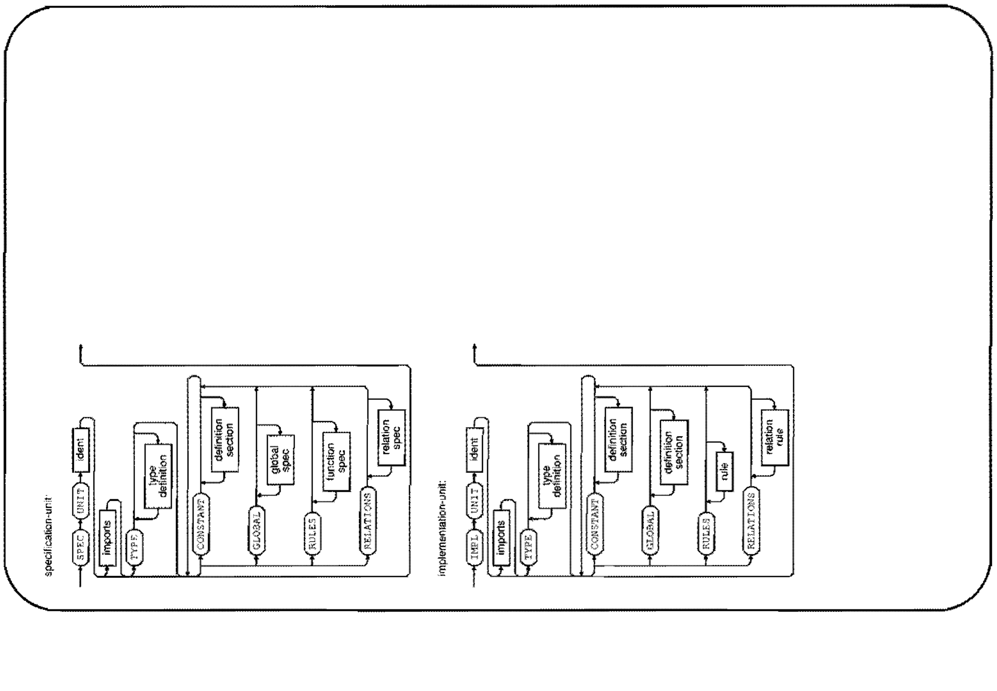
[Wir88c] N. Wirth. Type Extensions. *ACM Transactions on Programming Languages and Systems*, 10(2):204–214, 1988.

[You67] D.H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208, 1967.

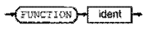
Appendix A

The Elegant syntax

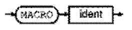




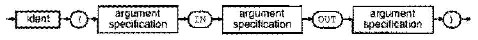
external-function-name:



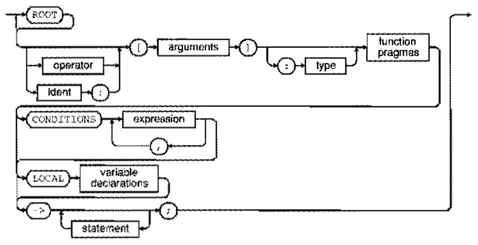
macro-name:



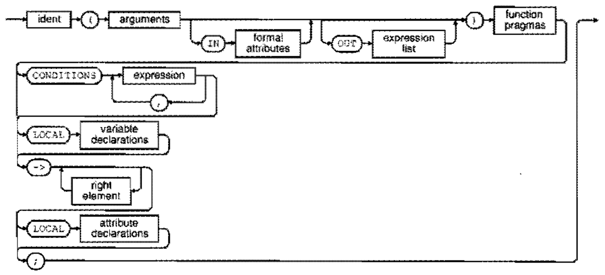
relation-spec:



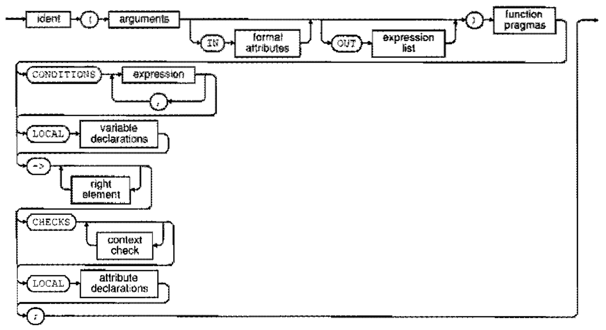
rule:



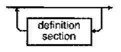
relation-rule:



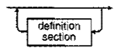
production-rule:



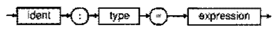
variable-declarations:



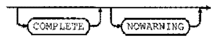
attribute-declarations:



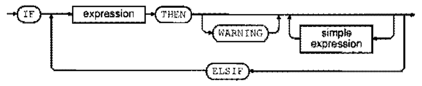
definition-section:

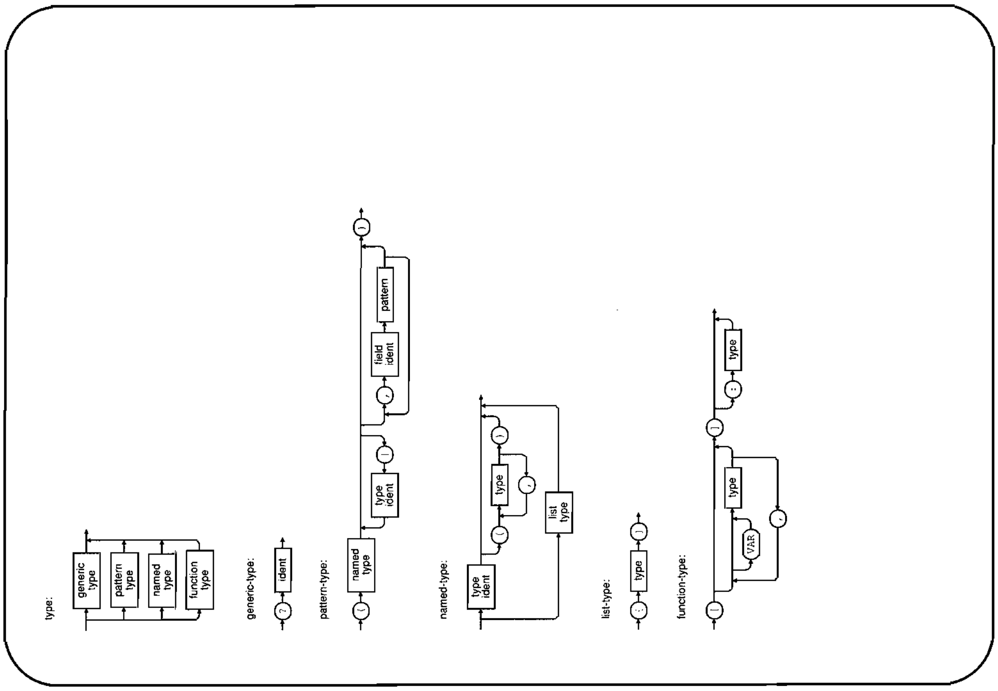


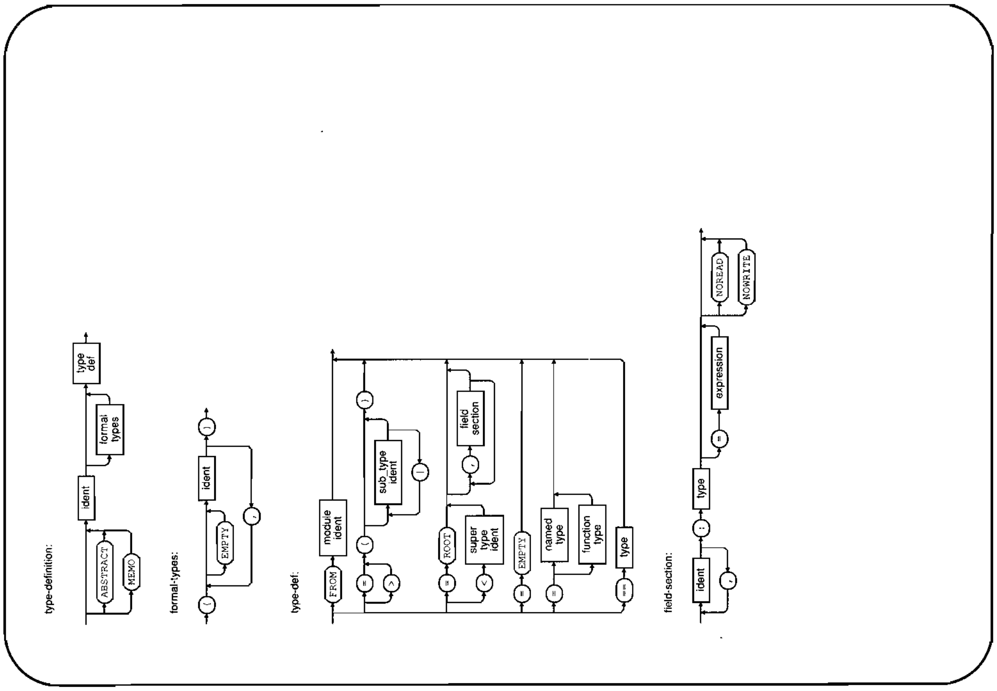
function-pragmas:

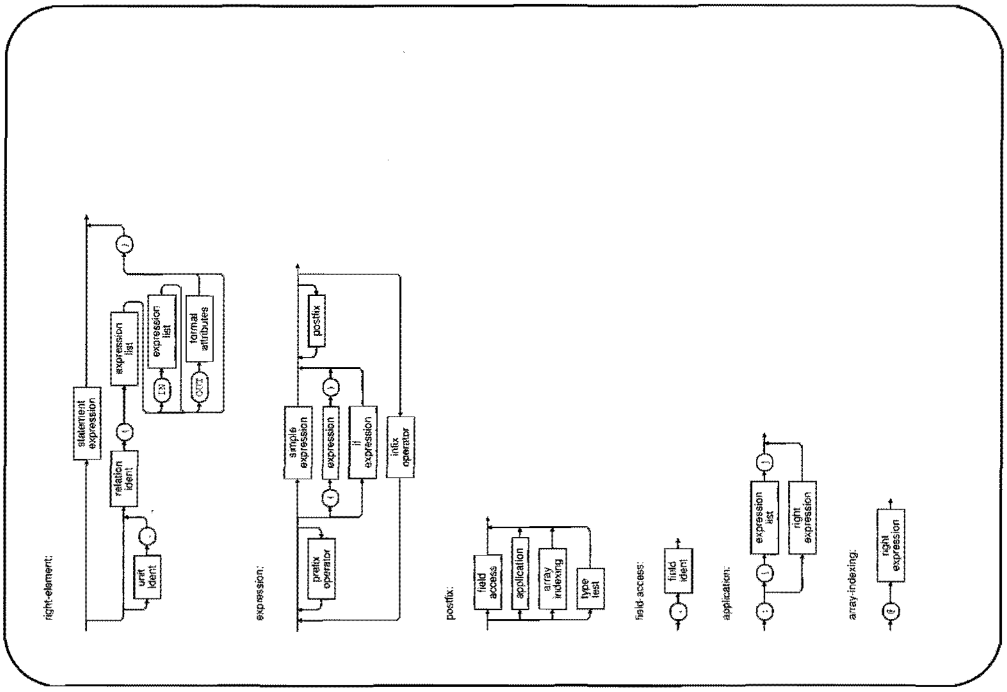


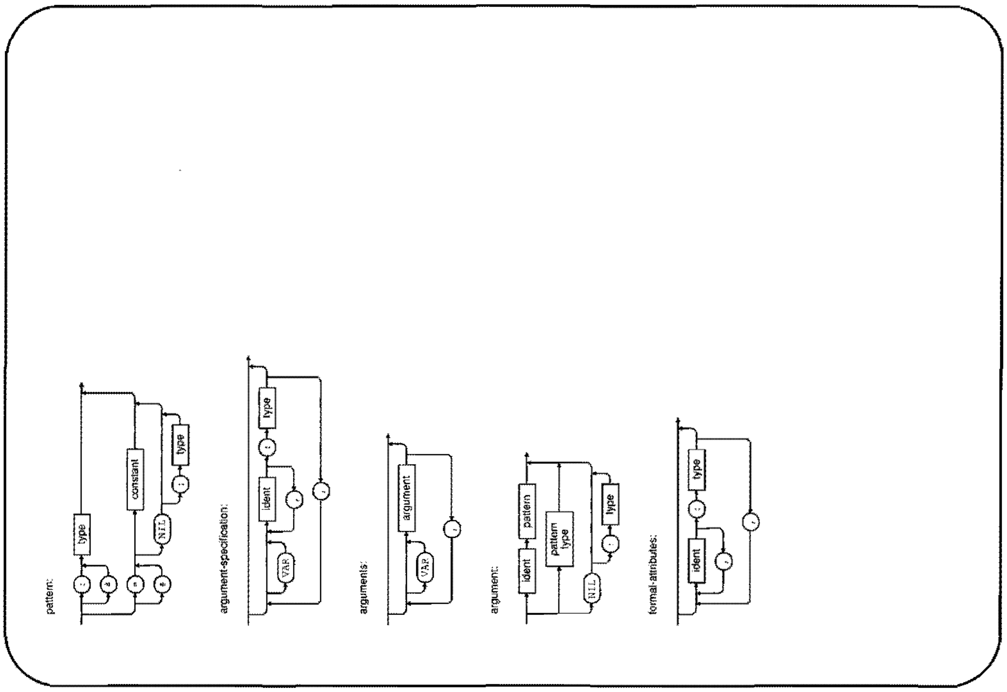
context-check:

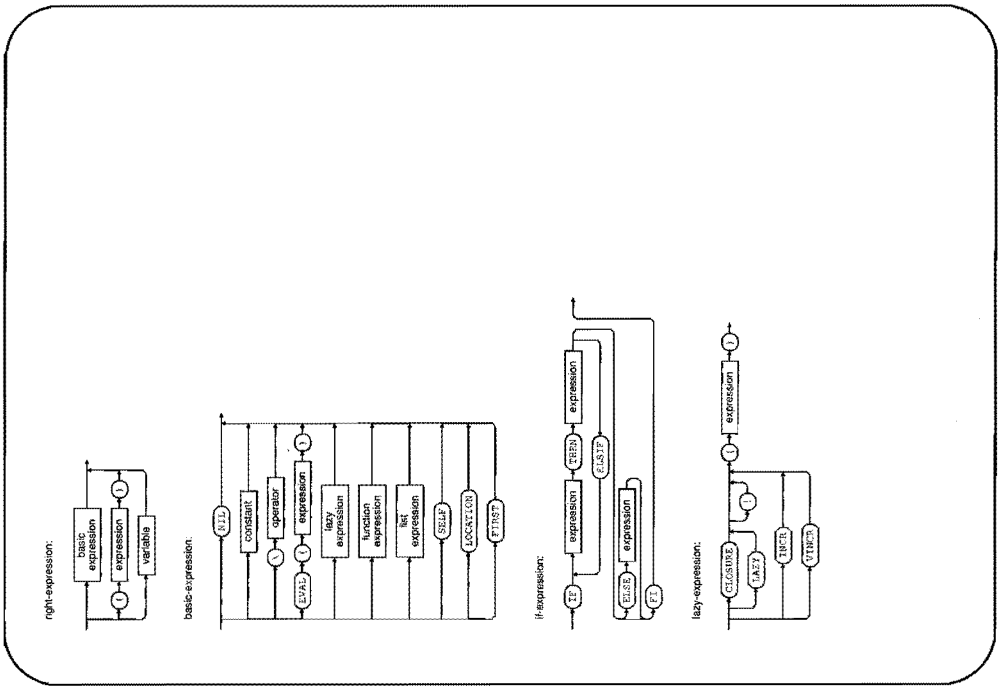


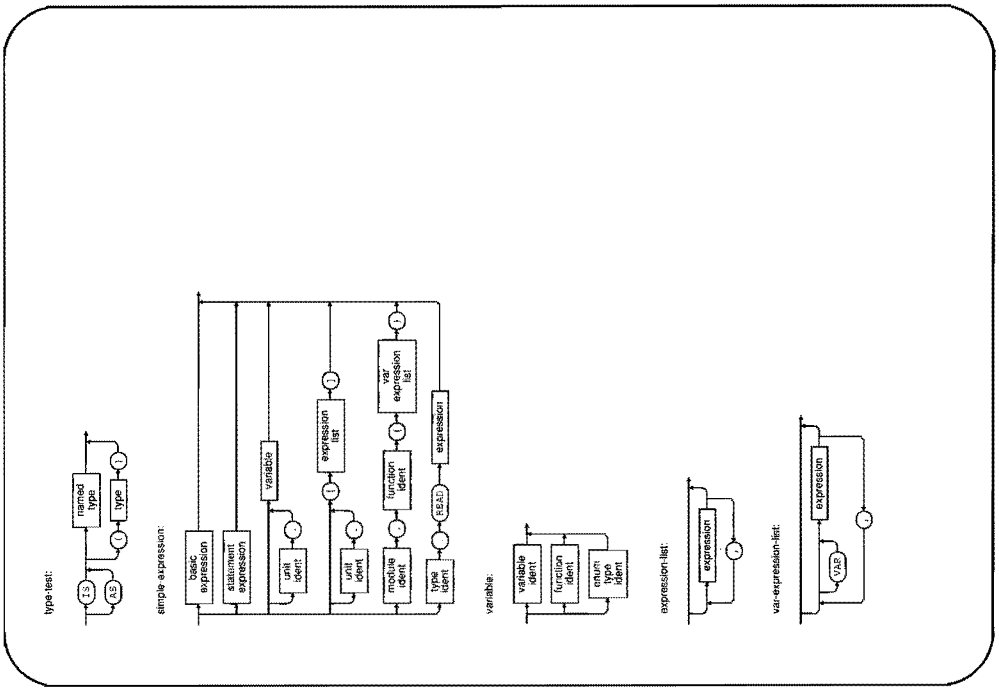

type:
generic type
pattern type
named type
function type
generic-type:
?
ident
pattern-type:
{
named type
type ident
|
,
field ident
pattern
}
named-type:
type ident
(
type
,
)
list type
list-type:
{
type
}
function-type:
[
VAR
type
,
]
:
type


type-definition:
ABSTRACT
MEMO
ident
formal types
type def
formal-types:
(
EMPTY
ident
,
)
type-def:
FROM
module ident
=
>
(
sub_type ident
|
)
=
ROOT
<
super type ident
,
field section
=
EMPTY
=
named type
function type
==
type
field-section:
ident
,
:
type
=
expression
NOREAD
NOWRITE

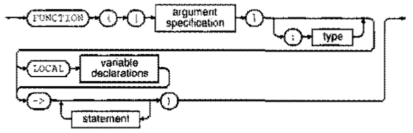

right-element:
unit ident
relation ident
statement expression
expression list
IN
OUT
expression list
formal attributes
expression:
prefix operator
simple expression
expression
if expression
infix operator
postfix
postfix:
field access
application
array indexing
type test
field-access:
field ident
application:
expression list
right expression
array-indexing:
@
right expression


pattern:
type
constant
NIL
type
argument-specification:
VAR
ident
type
arguments:
VAR
argument
argument:
ident
pattern
pattern type
NIL
type
formal-attributes:
ident
type

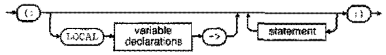

right-expression:
basic expression
(
expression
)
variable
basic-expression:
NIL
constant
\
operator
EVAL
(
expression
)
lazy expression
function expression
list expression
SELF
LOCATION
FIRST
if-expression:
IF
expression
THEN
expression
ELSIF
ELSE
expression
FI
lazy-expression:
CLOSURE
LAZY
INCR
VINCR
!
(
expression
)


type-test:
IS
AS
named type
(
type
)
simple-expression:
basic expression
statement expression
unit ident
.
variable
unit ident
.
(
expression list
)
module ident
.
function ident
(
var expression list
)
type ident
:
READ
expression
variable:
variable ident
function ident
enum type ident
expression-list:
expression
,
var-expression-list:
VAR
expression
,

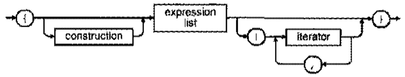
function-expression:



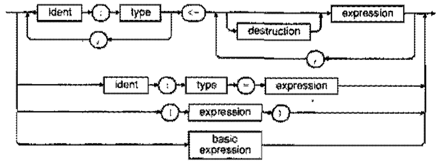
statement-expression:



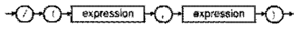
list-expression:



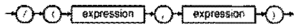
iterator:



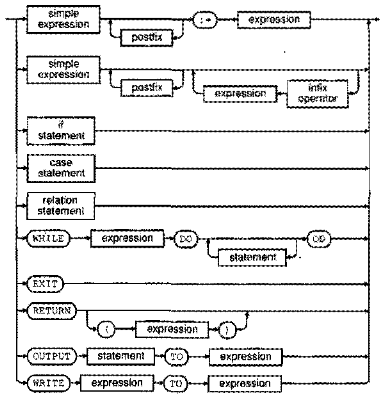
construction:



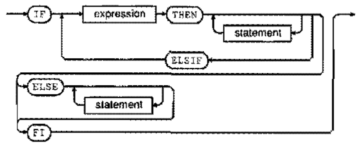
destruction:

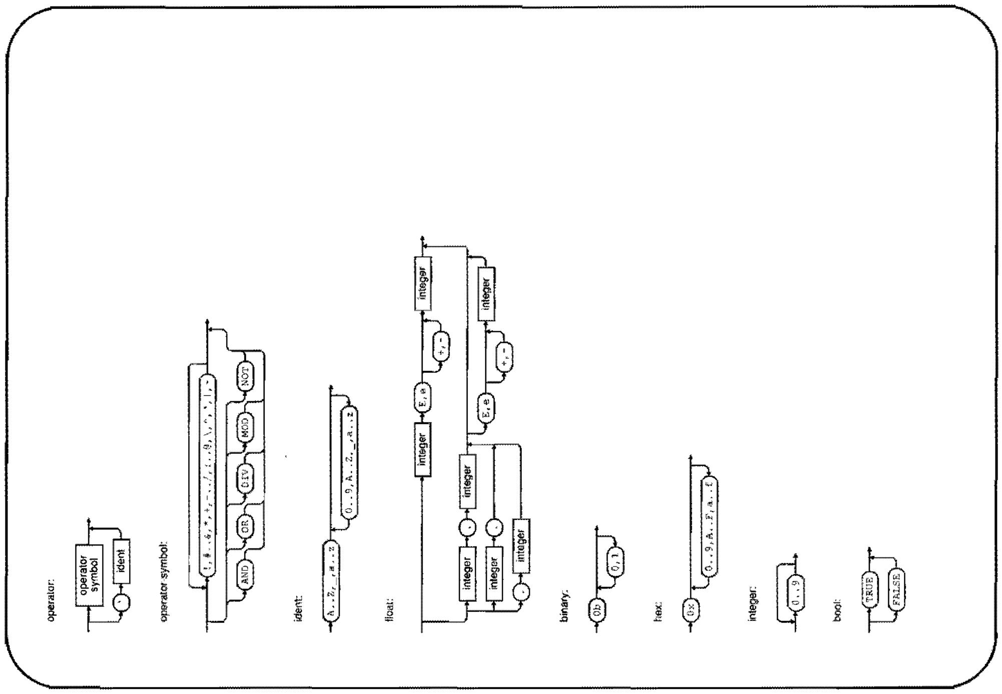


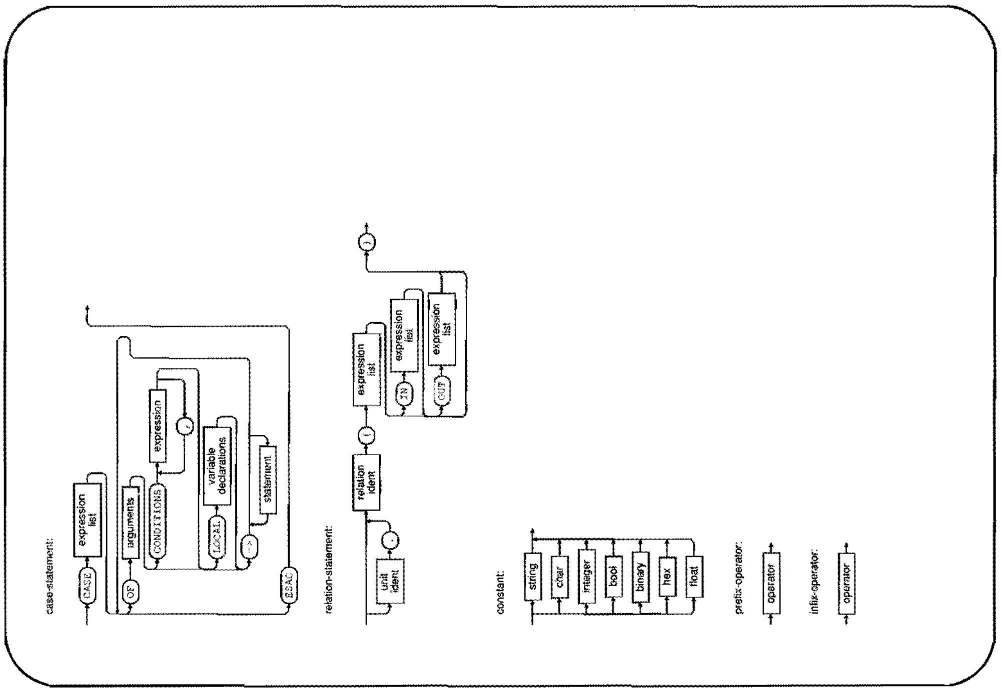
statement:



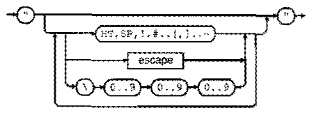
if-statement:



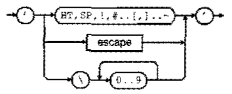

operator:
operator symbol
ident
operator-symbol:
AND
OR
DIV
MOD
NOT
ident:
A..Z,_,a..z
0..9,A..Z,_,a..z
float:
integer
E,e
+,-
binary:
0b
0,1
hex:
0x
0..9,A..F,a..f
integer:
0..9
bool:
TRUE
FALSE


case-statement:
CASE
expression list
OF
arguments
CONDITIONS
expression
LOCAL
variable declarations
->
statement
ESAC
relation-statement:
unit ident
relation ident
expression list
IN
OUT
constant:
string
char
integer
bool
binary
hex
float
prefix-operator:
operator
infix-operator:
operator

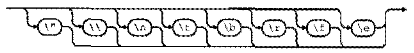
string:



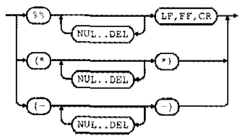
char:



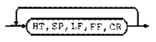
escape:



comment:



separator:



Appendix B

The Elegant Prelude

```
SPEC UNIT Prelude

(* This is the Elegant prelude.
   It contains a host of predefined functions, varying from trivial arithmetic ones
   to polymorphic functions.
   All functions in this prelude are exported which means that they have a system wide
   scope and thus are visible in all units.
   Importing them can not be prevented, nor can they be renamed.
   Alas, some functions, like Head on lists are partial and thus implemented
   as such, implying that their invocation may lead to abortion when arguments out
   of their domain are supplied.
   This unit is a regular Elegant unit, which means that you can create such kind of
   units yourself.
   If you create interesting polymorphic or other functions, please inform the Elegant
   team so that we can include them in this prelude or another library module, thus
   serving the needs of all users.

   For the curious, especially those interested in polymorphism, the unit
   Prelude.impl might be interesting.
*)

IMPORT
  Prelude_C

TYPE

Int             FROM StandardTypes
Float           FROM StandardTypes
Bool            FROM StandardTypes
Char            FROM StandardTypes
Ident           FROM StandardTypes
String          FROM StandardTypes
Bitset          FROM StandardTypes

Array   (X)             FROM StandardTypes
Lazy    (EMPTY X)       FROM StandardTypes
Closure (EMPTY X)       FROM StandardTypes
Incr    (EMPTY X)       FROM StandardTypes
Vincr   (X)             FROM StandardTypes

ReadFile        FROM FileStream
WriteFile       FROM FileStream
```

```
VOID            = EMPTY
Message         = EMPTY
Output          = EMPTY
CODE            = Closure(VOID)

List(X)         = ROOT,         head : X,
                                tail : List(X)

IntList         = {Int}
IdentList       = {Ident}

CONSTANT

MinInt : Int = -1024*1024*2048
MaxInt : Int = -MinInt-1

SP : Char = '\040'
LF : Char = '\012'
CR : Char = '\015'
HT : Char = '\011'
FF : Char = '\014'
SC : Char = '\033'

RULES

[x : Int]     : Message         FUNCTION Messages_FromInt               EXPORT
[x : Bool]    : Message         FUNCTION Messages_FromBool              EXPORT
[x : Char]    : Message         FUNCTION Messages_FromChar              EXPORT
[x : Float]   : Message         FUNCTION Messages_FromFloat             EXPORT
[x : String]  : Message         FUNCTION Messages_FromString            EXPORT
[x : Ident]   : Message         FUNCTION Messages_FromIdent             EXPORT

[x : Int]     : Output          FUNCTION Outputs_FromInt                EXPORT
[x : Bool]    : Output          FUNCTION Outputs_FromBool               EXPORT
[x : Char]    : Output          FUNCTION Outputs_FromChar               EXPORT
[x : Float]   : Output          FUNCTION Outputs_FromFloat              EXPORT
[x : String]  : Output          FUNCTION Outputs_FromString             EXPORT
[x : Ident]   : Output          FUNCTION Outputs_FromIdent              EXPORT

Null:[x : VOID] : VOID                          MACRO _NULL             EXPORT
Construct:[x : VOID, y : VOID] : VOID           MACRO _CONSTRUCT        EXPORT

Null:[x : Output] : Output                      MACRO _NULL             EXPORT
Construct:[x : Output, y : Output] : Output     MACRO _CONSTRUCT        EXPORT

Null:[x : Message] : Message                    MACRO _NULL             EXPORT
Construct:[x : Message, y : Message] : Message  MACRO _CONSTRUCT        EXPORT

Init:[]

{--------- Integer operations --------------}

<  [i : Int, j : Int] : Bool            FUNCTION _F_Lt
                                        MACRO _LT                       EXPORT
<= [i : Int, j : Int] : Bool            FUNCTION _F_Le
                                        MACRO _LE                       EXPORT
>  [i : Int, j : Int] : Bool            FUNCTION _F_Gt
                                        MACRO _GT                       EXPORT
>= [i : Int, j : Int] : Bool            FUNCTION _F_Ge
                                        MACRO _GE                       EXPORT
+  [i : Int, j : Int] : Int             FUNCTION _F_Plus
                                        MACRO _PLUS                     EXPORT
-  [i : Int, j : Int] : Int             FUNCTION _F_Minus
                                        MACRO _MINUS                    EXPORT
-  [i : Int         ] : Int             FUNCTION _F_Neg
```

```
                                        MACRO _NEG                       EXPORT
*  [i : Int, j : Int] : Int             FUNCTION _F_Times
                                        MACRO _TIMES                     EXPORT
/  [i : Int, j : Int] : Int             FUNCTION _F_Div
                                        MACRO _DIV                       EXPORT
DIV [i : Int, j : Int] : Int            FUNCTION _F_Mod
                                        MACRO _DIV                       EXPORT
MOD [i : Int, j : Int] : Int            FUNCTION _F_Mod
                                        MACRO _MOD                       EXPORT
%% exponentiation
** [i : Int, j : Int] : Int                                              EXPORT
:+ [VAR i : Int, j : Int]               MACRO _ASSPLUS                   EXPORT
:- [VAR i : Int, j : Int]               MACRO _ASSMINUS                  EXPORT
:* [VAR i : Int, j : Int]               MACRO _ASSTIMES                  EXPORT
:/ [VAR i : Int, j : Int]               MACRO _ASSDIV                    EXPORT
%% absolute value
#  [i : Int] : Int                                                       EXPORT
Signum:[i : Int] : Int                                                   EXPORT

Max:[i : Int, j : Int] : Int                                             EXPORT
Min:[i : Int, j : Int] : Int                                             EXPORT

Even:[i : Int] : Bool                   FUNCTION _F_Even
                                        MACRO _EVEN                      EXPORT
Odd :[i : Int] : Bool                   FUNCTION _F_Odd
                                        MACRO _ODD                       EXPORT

.. [i : Int, j : Int] : {Int}                                            EXPORT

{--------- Float operations --------------}

CONSTANT
  E  : Float = 2.7182818284590452354
  Pi : Float = 3.14159265358979323846

RULES

<  [i : Float, j : Float] : Bool        FUNCTION _Fl_Lt
                                        MACRO _LT                        EXPORT
<= [i : Float, j : Float] : Bool        FUNCTION _Fl_Le
                                        MACRO _LE                        EXPORT
>  [i : Float, j : Float] : Bool        FUNCTION _Fl_Gt
                                        MACRO _GT                        EXPORT
>= [i : Float, j : Float] : Bool        FUNCTION _Fl_Ge
                                        MACRO _GE                        EXPORT
+  [i : Float, j : Float] : Float       FUNCTION _F_Plus
                                        MACRO _PLUS                      EXPORT
-  [i : Float, j : Float] : Float       FUNCTION _F_Minus
                                        MACRO _MINUS                     EXPORT
-  [i : Float           ] : Float       FUNCTION _F_Neg
                                        MACRO _NEG                       EXPORT
*  [i : Float, j : Float] : Float       FUNCTION _F_Times
                                        MACRO _TIMES                     EXPORT
/  [i : Float, j : Float] : Float       FUNCTION _F_Div
                                        MACRO _DIV                       EXPORT

:+ [VAR i : Float, j : Float]           MACRO _ASSPLUS                   EXPORT
:- [VAR i : Float, j : Float]           MACRO _ASSMINUS                  EXPORT
:* [VAR i : Float, j : Float]           MACRO _ASSTIMES                  EXPORT
:/ [VAR i : Float, j : Float]           MACRO _ASSDIV                    EXPORT

%% absolute value
#  [i : Float] : Float                                                   EXPORT
Signum:[i : Float] : Int                                                 EXPORT
```

```
Max:[i : Float, j : Float] : Float                                EXPORT
Min:[i : Float, j : Float] : Float                                EXPORT

[i : Int] : Float                   FUNCTION _F_Int2Float
                                    MACRO _INT2FL                 EXPORT
Float:[i : Int] : Float             FUNCTION _F_Int2Float
                                    MACRO _INT2FL                 EXPORT
Int:[i : Float] : Int               FUNCTION _F_Float2Int
                                    MACRO _FL2INT                 EXPORT

acos:[x : Float] : Float            FUNCTION acos                 EXPORT
asin:[x : Float] : Float            FUNCTION asin                 EXPORT
atan:[x : Float] : Float            FUNCTION atan                 EXPORT
atan2:[y : Float, x : Float] : Float    FUNCTION atan2            EXPORT
ceil:[x : Float] : Float            FUNCTION ceil                 EXPORT
cos:[x : Float] : Float             FUNCTION ceil                 EXPORT
cosh:[x : Float] : Float            FUNCTION cosh                 EXPORT
exp:[x : Float] : Float             FUNCTION exp                  EXPORT
floor:[x : Float] : Float           FUNCTION floor                EXPORT
fmod:[x : Float, y : Float] : Float     FUNCTION fmod             EXPORT
frexp:[x : Float, VAR exp : Int] : Float FUNCTION frexp           EXPORT
ldexp:[x : Float, exp : Int] : Float    FUNCTION ldexp            EXPORT
log:[x : Float] : Float             FUNCTION log                  EXPORT
log10:[x : Float] : Float           FUNCTION log10                EXPORT
modf:[x : Float, VAR iptr : Float] : Float FUNCTION modf          EXPORT
%% exponentiation
** [x : Float, y : Float] : Float   FUNCTION pow                  EXPORT
sin:[x : Float] : Float             FUNCTION sin                  EXPORT
sinh:[x : Float] : Float            FUNCTION sinh                 EXPORT
sqrt:[x : Float] : Float            FUNCTION sqrt                 EXPORT
tan:[x : Float] : Float             FUNCTION tan                  EXPORT
tanh:[x : Float] : Float            FUNCTION tanh                 EXPORT

GLOBAL
  print_exponent : Bool {- To influence Print:[WriteFile, Float] -}
  print_precision : Int

RULES
Print:[f : WriteFile, x : Float]    FUNCTION PrintFloat    EXPORT

{--------- Bool operations --------------}

%% The following five operations are non-strict in their second argument.

|   [i : Bool, j : Bool] : Bool     FUNCTION _F_Or
                                    MACRO _OR                     EXPORT
OR  [i : Bool, j : Bool] : Bool     FUNCTION _F_Or
                                    MACRO _OR                     EXPORT
&   [i : Bool, j : Bool] : Bool     FUNCTION _F_And
                                    MACRO _AND                    EXPORT
AND[i : Bool, j : Bool] : Bool      FUNCTION _F_And
                                    MACRO _AND                    EXPORT
=>  [i : Bool, j : Bool] : Bool     FUNCTION _F_Implies
                                    MACRO _IMPLIES                EXPORT

~   [i : Bool          ] : Bool     FUNCTION _F_Not
                                    MACRO _NOT                    EXPORT
NOT[i : Bool           ] : Bool     FUNCTION _F_Not
                                    MACRO _NOT                    EXPORT

{--------- Char operations --------------}

<   [i : Char, j : Char] : Bool     FUNCTION _F_Lt
                                    MACRO _LT                     EXPORT
<=  [i : Char, j : Char] : Bool     FUNCTION _F_Le
```

```
                                           MACRO _LE                       EXPORT
>  [i : Char, j : Char] : Bool             FUNCTION _F_Gt
                                           MACRO _GT                       EXPORT
>= [i : Char, j : Char] : Bool             FUNCTION _F_Ge
                                           MACRO _GE                       EXPORT

.. [i : Char, j : Char] : {Char}                                           EXPORT

%% orinal value
#  [i : Char] : Int                        FUNCTION _F_Char2Int
                                           MACRO _CHAR2INT                 EXPORT
%% inverse ordinal
Char:[i : Int] : Char                      FUNCTION _F_Int2Char
                                           MACRO _INT2CHAR                 EXPORT

IsAscii   :[c : Char] : Bool (* 0..127 *)                                  EXPORT
IsControl :[c : Char] : Bool (* 0..31, 127 *)                              EXPORT
IsPrint   :[c : Char] : Bool (* ' ' .. '~' *)                              EXPORT
IsSpace   :[c : Char] : Bool (* SP,HT,CR,LF,BS,FF *)                       EXPORT
IsUpper   :[c : Char] : Bool                                               EXPORT
IsLower   :[c : Char] : Bool                                               EXPORT
IsAlpha   :[c : Char] : Bool                                               EXPORT
IsDigit   :[c : Char] : Bool                                               EXPORT
IsAlphaNum:[c : Char] : Bool                                               EXPORT
IsOctal   :[c : Char] : Bool                                               EXPORT
IsHex     :[c : Char] : Bool                                               EXPORT

ToLower   :[c : Char] : Char                                               EXPORT
ToUpper   :[c : Char] : Char                                               EXPORT

{--------- Bitset operations --------------}

EmptySet:[] : Bitset                       MACRO _EMPTYSET                 EXPORT
%% cardinality
#  [i : Bitset] : Int                      FUNCTION _F_Set_Size            EXPORT
%% i + {j}
+  [i : Bitset, j : Int] : Bitset          FUNCTION _F_Set_Incl            EXPORT
%% i \ {j}
-  [i : Bitset, j : Int] : Bitset          FUNCTION _F_Set_Excl            EXPORT
%% element test
<- [i : Int,    j : Bitset] : Bool         MACRO _IN                       EXPORT
<= [i : Bitset, j : Bitset] : Bool         MACRO SUBSET                    EXPORT
+  [i : Bitset, j : Bitset] : Bitset       MACRO UNION                     EXPORT
-  [i : Bitset, j : Bitset] : Bitset       MACRO DIFFERENCE                EXPORT
*  [i : Bitset, j : Bitset] : Bitset       MACRO INTERSECTION              EXPORT
%% (i+j)-(i*j)
/  [i : Bitset, j : Bitset] : Bitset       MACRO SYMMETRICDIFFERENCE       EXPORT

%% construction
Null:[x : Int] : Bitset                            MACRO _BITSET_NULL      EXPORT
Construct:[l : Bitset, x : Int] : Bitset           MACRO _BITSET_CONSTRUCT EXPORT

%% destruction
Start:[i : Bitset] : Int                   FUNCTION _F_BITSET_START
                                           MACRO _BITSET_START             EXPORT
Destruct:[VAR i : Bitset, VAR s : Int, VAR x : Bool] : Bool
                                           FUNCTION _F_BITSET_DESTRUCT
                                           MACRO _BITSET_DESTRUCT          EXPORT

{--------- Identifier operations --------------}

# [i : Ident] : Int                        FUNCTION Identifier_Length
                                           MACRO    _IDENT_LENGTH          EXPORT

+ [i : Ident, j : Ident] : Ident                                           EXPORT
```

```
Ident:[s : String] : Ident            FUNCTION Identifier_FromString  EXPORT
%% conversion of substring s[l..h]
Ident:[s : String, l : Int, h : Int] : Ident
                                       FUNCTION Identifier_PutInIdTable EXPORT

%% destruction
Start:[i : Ident] : Int                FUNCTION _F_IDENT_START
                                       MACRO _IDENT_START              EXPORT
Destruct:[VAR i : Ident, VAR s : Int, VAR x : Char] : Bool
                                       FUNCTION _F_IDENT_DESTRUCT
                                       MACRO _IDENT_DESTRUCT           EXPORT

{--------- String operations --------------}

%% Allocate a new string with n characters, terminated by a 0 character.
NewString:[n : Int] : String           FUNCTION Strings_New            EXPORT

#  [i : String] : Int                  FUNCTION Strings_Length         EXPORT
+  [i : String, j : String] : String   FUNCTION Strings_Append         EXPORT
+  [i : String, j : Int] : String      FUNCTION Strings_IAppend        EXPORT
%% lexicographical comparisons
== [i : String, j : String] : Bool     FUNCTION _FS_Eq
                                       MACRO _STRING_EQ                EXPORT
<  [i : String, j : String] : Bool     FUNCTION _FS_Lt
                                       MACRO _STRING_LT                EXPORT
>  [i : String, j : String] : Bool     FUNCTION _FS_Gt
                                       MACRO _STRING_GT                EXPORT
<= [i : String, j : String] : Bool     FUNCTION _FS_Le
                                       MACRO _STRING_LE                EXPORT
>= [i : String, j : String] : Bool     FUNCTION _FS_Ge
                                       MACRO _STRING_GE                EXPORT

SubString:[s : String, i : Int, j : Int] : String                      EXPORT

%% Conversions to String
[i : Ident] : String                   FUNCTION Identifier_AsString
                                       MACRO    _IDENT2STRING          EXPORT
String:[i : Ident] : String            FUNCTION Identifier_AsString
                                       MACRO    _IDENT2STRING          EXPORT
String:[int : Int] : String                                            EXPORT

%% destruction
Start:[i : String] : Int               FUNCTION _F_STRING_START
                                       MACRO _STRING_START             EXPORT
Destruct:[VAR i : String, VAR s : Int, VAR x : Char] : Bool
                                       FUNCTION _F_STRING_DESTRUCT
                                       MACRO _STRING_DESTRUCT          EXPORT

{--------- Built-in operations --------------}

(*

For each enumerated type T the conversion functions
  T:[x : Int] : T                                                      EXPORT
  # [x : T] : Int                                                      EXPORT

For each non-abstract node type T the creation function:
  T:[fields] : T                                                       EXPORT

++ [x : VOID, y : VOID] -> x y ;                                       EXPORT

*)

{--------- Generic operations --------------}
```

```
%% shallow comparison
= [x : ?T, y : ?T] : Bool               MACRO _EQ                       EXPORT
# [x : ?T, y : ?T] : Bool               MACRO _NEQ                      EXPORT

%% Identity
I:[x : ?T] : T                                                          EXPORT

%% Function composition
* [f : [?T] : ?U, g : [?S] : ?T] : [S] : U                              EXPORT

%% Swap
<-> [VAR x : ?T, VAR y : ?T]                                            EXPORT

%% Array creation with initial value
Array:[i : Int, x : ?T] : Array(T)      FUNCTION Create_StandardTypes_Array
                                                                        EXPORT
%% destruction
Start:[a : Array(?T)] : Int             FUNCTION _F_ARRAY_START
                                        MACRO _ARRAY_START              EXPORT
Destruct:[VAR a : Array(?T), VAR s : Int, VAR x : ?T] : Bool
                                        FUNCTION _F_ARRAY_DESTRUCT
                                        MACRO _ARRAY_DESTRUCT           EXPORT

%% Lengths
# [l : {?T}] : Int                                                      EXPORT
# [a : Array(?X)] : Int                 MACRO _ARRAY_SIZE               EXPORT

%% Test on emptiness
Null:[l : {?T}] : Bool                                                  EXPORT

%% First and remaining elements
Head:[l : ({?T})] : T                                                   EXPORT
Tail:[l : ({?T})] : {T}                                                 EXPORT

Head:[l : {?T}, b : ?T] : T                                             EXPORT
Tail:[l : {?T}, b : {?T}] : {T}                                         EXPORT

%% Last and remaining elements
Last:[l : ({?T})] : T                                                   EXPORT
Init:[l : ({?T})] : {T}                                                 EXPORT

Last:[l : {?T}, b : ?T] : T                                             EXPORT
Init:[l : {?T}, b : {?T}] : {T}                                         EXPORT

%% First n elements and remainder
Take:[n : Int, l : {?T}] : {T}                                          EXPORT
Drop:[n : Int, l : {?T}] : {T}                                          EXPORT

%% First n elements and remainder
Split:[n : Int, l : {?T}, VAR first : {?T}, last : {?T}]                EXPORT

Reverse:[l : {?T}] : {T}                                                EXPORT

%% i-th element
^ [l : ({?T}), i : Int] : T                                             EXPORT

%% element test
<- [x : ?T, l : {?T}] : Bool                                            EXPORT

%% concatenation
+  [x : ?T, y : {?T}] : {T}                                             EXPORT
+  [l : {?T}, u : ?T] : {T}                                             EXPORT
+  [l : {?T}, m : {?T}] : {T}                                           EXPORT
```

```
%% construction
Null:[x : ?T] : {T}                                 MACRO _LIST_NULL       EXPORT
Construct:[l : {?T}, x : ?T] : {T}                  MACRO _LIST_CONSTRUCT  EXPORT

%% destruction
Start:[a : List(?T)] : List(T)              FUNCTION _F_LIST_START
                                            MACRO _LIST_START              EXPORT
Destruct:[VAR a : List(?T), VAR s : List(?T), VAR x : ?T] : Bool
                                            FUNCTION _F_LIST_DESTRUCT
                                            MACRO _LIST_DESTRUCT           EXPORT


%% concatenation of all elements
+  [l : {{?T}}] : {T}                                                      EXPORT

%% insertion operations with side-effect on first argument
:+ [VAR l : {?T}, u : ?T]                                                  EXPORT
:+ [VAR l : {?T}, m : {?T}]                                                EXPORT

%% matrix transposition
Transpose:[l : {{?T}}] : {{T}}                                             EXPORT

%% apply f to all elements
* [l : {?T}, f : [?T]]                                                     EXPORT
* [l : {?T}, f : [?T] : ?U] : {U}                                          EXPORT

%% filter elements not obeying f
/ [l : {?T}, f : [?T] : Bool] : {T}                                        EXPORT

%% x1 + (x2 + (... xn + a)), where x+y = f:[x,y]
Rec:[l : {?T}, a : ?U, f : [?T, ?U] : ?U] : U                              EXPORT

%% xn + (... (x2 + (x1 + a))...), where x+y = f:[x,y]
Iter:[l : {?T}, a : ?U, f : [?T, ?U] : ?U] : U                             EXPORT

%% conjunction and disjunction of all elements
& [l : {Bool}] : Bool                                                      EXPORT
| [l : {Bool}] : Bool                                                      EXPORT

%% & (p*l)
All:[p : [?T] : Bool, l : {?T}] : Bool                                     EXPORT

%% | (p*l)
Any:[p : [?T] : Bool, l : {?T}] : Bool                                     EXPORT

%% sum and product
+ [l : {Int}] : Int                                                        EXPORT
* [l : {Int}] : Int                                                        EXPORT

+ [l : {Float}] : Float                                                    EXPORT
* [l : {Float}] : Float                                                    EXPORT

%% max and min
Max:[l : {Int}] : Int                                                      EXPORT
Min:[l : {Int}] : Int                                                      EXPORT

Max:[l : ({Float})] : Float                                                EXPORT
Min:[l : ({Float})] : Float                                                EXPORT
```

Samenvatting

Dit proefschrift handelt over het ontwerp van de compilergenerator `Elegant`. Een compiler generator is een computer programma dat vanuit een specificatie een compiler kan genereren. Een compiler is een computer programma dat een gestructureerde invoertekst kan vertalen in een uitvoertekst. Een compiler generator is zelf een compiler welke de specificatie vertaalt in de programmatekst van de gegenereerde compiler. Dit heeft het mogelijk gemaakt om `Elegant` met zichzelf te genereren. Van een compilergenerator wordt verlangd dat deze een krachtig specificatie formalisme vertaalt in een efficiënt programma, een eis waar `Elegant` aan voldoet.
Een compiler bestaat uit een aantal onderdelen, te weten een scanner, een parser, een attribuutevaluator, een optimalisator en een codegenerator. Deze onderdelen kunnen door het `Elegant` systeem geneneerd worden, ieder uit een aparte specificatie, met uitzondering van de parser en attribuutevaluator, welke gezamenlijk worden beschreven in de vorm van een zogenaamde attribuutgrammatica.
De scanner wordt gegenereerd met behulp van een scannergenerator en heeft tot taak de invoertekst te splitsen in een rij symbolen. Deze rij symbolen kan vervolgens ontleed worden door een parser. Daarna berekent de attribuutevaluator eigenschappen van de invoertekst in de vorm van zogenaamde attributen. De attributenwaarden vormen een datastructuur. De vorm van deze datastructuur wordt gedefinieerd met behulp van typeringsregels in de `Elegant` programmeertaal. De optimalisator en codegenerator voeren operaties op deze datastructuur uit welke eveneens beschreven worden in de `Elegant` programmeertaal.

Dit proefschrift beschrijft de invloed die functionele programmeertalen hebben gehad op het ontwerp van `Elegant`. Functionele talen zijn programmeertalen met als belangrijkste eigenschap dat functies een centrale rol vervullen. Functies kunnen worden samengesteld tot nieuwe functies, ze kunnen worden doorgegeven aan functies en worden opgeleverd als functieresultaat. Daarnaast staan functionele talen niet toe dat de waarde van een variable wordt gewijzigd, het zogenaamde neveneffect, in tegenstelling tot imperatieve talen die zo'n neveneffect wel toestaan. Deze laatste beperking maakt het mogelijk om met behulp van algebraïsche regels een functioneel programma te herschrijven in een ander functioneel programma met dezelfde betekenis. Dit herschrijfproces wordt ook wel progammatransformatie genoemd.
De invloed van functionele talen op `Elegant` omvat:

- Het beschrijven van ontleedalgorithmen als functionele programma's. Traditioneel worden ontleedalgorithmen beschreven met behulp van de theorie van stapelautomaten. In hoofdstuk 3 wordt aangetoond dat deze theorie niet nodig is. Met behulp van programmatransformaties zijn vele uit de literauur bekende ontleedalgorithmen af te leiden en worden ook nieuwe ontleedalgorithmen gevonden. Deze aanpak maakt het bovendien mogelijk om de vele verschillende ontleedalgorithmen met elkaar te combineren.

- De evaluatie van attributen volgens de regels van een attribuutgrammatica blijkt eveneens goed te kunnen worden beschreven met behulp van functionele talen. Traditioneel bouwt een ontleedalgorithme tijdens het ontleden een zogenaamde ontleedboom op. Deze ontleedboom beschrijft de structuur van de invoertekst. Daarna wordt deze ontleedboom geanalyseerd en worden eigenschappen ervan in de vorm van attributen berekend. In hoofdstuk 4 van het proefschrift wordt aangetoond dat het niet nodig is de ontleedboom te construeren. In plaats daarvan is het mogelijk om tijdens het ontleden functies die attributen kunnen berekenen samen te stellen tot nieuwe functies. Uiteindelijk wordt er zo één functie geconstrueerd voor een gehele invoertekst. Deze functie wordt vervolgens gebruikt om de attribuutwaarden te berekenen. Voor de uitvoering van deze functie is het noodzakelijk gebruik te maken van zogenaamde "luie evaluatie". Dit is een mechanisme dat attribuutwaarden slechts dan berekent wanneer deze werkelijk noodzakelijk zijn. Dit verklaart de naam `Elegant`, welke een acroniem is voor "Exploiting Lazy Evaluation for the Grammar Attributes of Non-Terminals".

- Scanners worden traditioneel gespecificeerd met behulp van zogenaamde reguliere expressies. Deze reguliere expressies kunnen worden afgebeeld op een eindige automaat. Met behulp van deze automaat kan de invoertekst worden geanalyseerd en gesplitst in symbolen. In hoofdstuk 5 wordt uiteengezet hoe functionele talen het mogelijk maken om scanneralgorithmen te construeren zonder gebruik te maken van automatentheorie. Door een reguliere expressie af te beelden op een functie en de functies voor de onderdelen van samengestelde reguliere expressies samen te stellen tot nieuwe functies kan een scannerfunctie geconstrueerd worden. Door gebruik te maken van programmatransformaties kan deze scanner deterministisch worden gemaakt en minimaal worden gehouden.

- Het typeringssysteem van `Elegant` wordt beschreven in hoodstuk 6 en vormt een combinatie van systemen die in functionele en imperatieve talen worden gevonden. Functionele typeringssystemen omvatten typen welke bestaan uit een aantal varianten. Elk van deze varianten bestaat uit een aantal waarden. Bij een dergelijk typeringssysteem wordt een functie gedefiniëerd door middel van een aantal deeelfuncties. Elke deelfunctie kan met behulp van zogenaamde patronen beschrijven voor welke van de varianten hij gedefiniëerd is. Het blijkt dat imperatieve typesystemen welke subtypering mogelijk maken een generalisatie zijn van functionele typesystemen. In deze generalisatie kan een patroon worden opgevat als een subtype en een deelfunctie als een partiële functie. Het `Elegant` typesystemen maakt deze vorm van typering en

functiebeschrijving mogelijk. Bij toepassing van een functie wordt de bijbehorende deelfunctie geselecteerd door de patronen te passen met de waarden van de actuele functieargumenten. In dit proefschrift wordt een efficiënt algorithme voor dit patroonpassen met behulp van programmatransformaties afgeleid uit de definitie van patronen.
Het `Elegant` typeringssystemen bevat ook typen voor de modellering van luie evaluatie. De aanwezigheid van neveneffekten maakt het mogelijk om drie verschillende luie typen te onderscheiden, welke verschillen in de wijze waarop de waarde van een lui object stabiliseert.

- In hoofdstuk 7 wordt aangetoond dat de regels uit een attribuutgrammatica ook kunnen worden gebruikt om eigenschappen van een datastructuur te berekenen in plaats van eigenschappen van een invoertekst. `Elegant` biedt de mogelijkheid om zulke attribuutregels te gebruiken voor dit doel.

- In hoofdstuk 8 tenslotte worden de `Elegant` programmeertaal en de efficiëntie van de `Elegant` vertaler en door `Elegant` gegenereerde vertalers geëvalueerd. Het blijkt dat de imperatieve `Elegant` programmeertaal dankzij abstractie mechanismen uit functionele talen een zeer rijke en krachtige taal is. Daarnaast zijn zowel `Elegant` zelf als de door `Elegant` gegenereerde vertalers van hoge efficiëntie en blijken geschikt voor het maken van compilers voor professionele toepassingen.

Curriculum vitae

Op 30 november 1957 werd ik geboren in Nijverdal. Na in 1976 het Atheneum B diploma te hebben behaald aan het Christelijk College Nassau Veluwe in Harderwijk, heb ik de studie Elektrotechniek aan de Technische Hogeschool Twente te Enschede gevolgd. Het afstuderen vond plaats bij de vakgroep Informatica onder leiding van prof. A.J.W. Duijvestein met als onderwerp "Automatisering van Homaeopatische Middelenselektie". Na de voltooiing van deze studie in 1982 heb ik mijn vervangende dienstplicht vervuld als wetenschappelijk medewerker aan de Technische Hogeschool Twente met als onderzoeksgebied Functionale Talen.
Sinds 1984 ben ik werkzaam bij het Philips Natuurkundig Laboratorium te Eindhoven. Daar heb ik eerst een onderzoek verricht op het gebied van gedistribueerde garbage collection. Sinds 1987 werk ik aan compiler constructie en ontwerp van formalismen. Onderzoek op het gebied van compiler generatie heeft geleid tot de constructie van de `Elegant` compiler generator en tot dit proefschrift onder leiding van prof. F.E.J Kruseman Aretz.

Stellingen

behorende bij het proefschrift

Functional Programming, Program Transformations and Compiler Construction

van

Lex Augusteijn

Eindhoven
7 oktober 1993

1. Van alle klassen van op attribuutgrammatica's gebaseerde parsers is de klasse van LL(1) parsers de meest natuurlijke gezien de mogelijke generalisatie van ontleden naar het recursief doorlopen van een data structuur.

   [**Lit.**] Hoofdstuk 7 van dit proefschrift.

2. Het feit dat de praktische uitdrukkingskracht van functionele talen in het algemeen groter is dan die van imperatieve talen wordt in hoge mate veroorzaakt door de beschikbaarheid van abstractiemechanismen als lazy evaluation, polymorfie, hogere-ordefuncties en impliciete typering en in veel geringe mate door de afwezigheid van neveneffekten. Derhalve is het zeer wel mogelijk een imperatieve taal met een zelfde mate van uitdrukkingskracht te ontwerpen.

   [**Lit.**] Hoofdstuk 8 van dit proefschrift.

3. Enumeratietypen *zijn* typeëxtensies.

   [**Lit.**] Hoofdstuk 6 van dit proefschrift.

4. Impliciet getypeerde programma's zijn compacter doch moeilijker te onderhouden en trager te vertalen dan expliciet getypeerde programma's.

5. Na de introductie van LR-parsing heeft het meer dan twintig jaar geduurd voordat een recursieve implementatie van een LR parser werd geformuleerd. Na de beschrijving van het Earley-algorithme heeft het bijna twintig jaar geduurd eer dit algorithme als een memo-functie werd beschreven. Na de introductie van lazy functionele talen en hun implementatie met behulp van combinatoren heeft het ongeveer 10 jaar geduurd voordat ontdekt werd dat traditionele compilerbouwtechnieken een veel efficiëntere implementatie voor deze talen geven.
   Uit deze gebeutenissen blijkt dat een breed overzicht van de informatica van groot belang is voor het ontwikkelen van efficiënte technieken.

6. Wanneer in een algorithme een stack wordt gemanipuleerd dient men verdacht te zijn op het bestaan van een recursieve versie van dat algorithme.

7. De problemen die informaticastudenten ondervinden bij het leren programmeren met behulp van recursieve functies zijn meer van psychologische dan van technische aard.

8. Een operating systeem dat slechts interactief bediend kan worden, zoals dat van de Apple Macintosh, is voor eindgebruikers een grote vooruitgang ten opzichte van een textueel operating systeem, zoals Unix, maar voor software ontwikkelaars een grote handicap.

9. Iets ontwikkelen is het tegengestelde van iets ingewikkelds maken.

10. Indien het Philips Natuurkundig Laboratorium goede contacten op informaticagebied met universiteiten wil onderhouden, dan dient zij ervoor te zorgen dat zij een gelijkwaardige gesprekspartner is. Hiervoor is het van belang dat zij eigen informaticaonderzoek van voldoende kaliber verricht.

11. De huidige tendens van het informatica-onderzoek op het Philips Natuurkundig Laboratorium om zich in toenemende mate met de vorm van het ontwikkelproces en in afnemende mate met de inhoud van dat proces bezig te houden leidt tot een toenemende bureaucratie die de wetenschappelijke kwaliteit van het onderzoek aantast.

12. In Nederland worden ouders welhaast gedwongen hun kinderen in te laten enten tegen de bof, mazelen en rode hond daar door de hoge inentingsgraad van de Nederlandse bevolking de besmettingskans gering is geworden en de betreffende ziekten op hogere leeftijd veel grotere complicaties met zich meebrengen dan op lagere leeftijd.