Reinhard Wilhelm
Helmut Seidl
Sebastian Hack

Compiler Design

Syntactic and Semantic Analysis

Springer

Compiler Design

Reinhard Wilhelm · Helmut Seidl ·
Sebastian Hack

Compiler Design

Syntactic and Semantic Analysis

Springer

Helmut Seidl
Fakultät für Informatik
Technische Universität München
Garching, Germany

Reinhard Wilhelm, Sebastian Hack
FB Informatik
Universität des Saarlandes
Saarbrücken, Germany

ISBN 978-3-642-17539-8
ISBN 978-3-642-17540-4 (eBook)
DOI 10.1007/978-3-642-17540-4

Library of Congress Control Number: 2013938370

Printed on acid-free paper.

Springer is part of Springer Science+Business Media
www.springer.com

To Margret, Hannah, Eva, and Barbara
R. W.

To Kerstin and Anna
H. S.

To Kerstin, Charlotte, and Marlene
S. H.

Preface

Compilers for high-level programming languages are large and complex software systems. They have, however, several distinct properties by which they favorably differ from most other software systems. Their semantics is (almost) well defined. Ideally, completely formal or at least precise descriptions exist both of the source and the target languages. Often, additional descriptions are provided of the interfaces to the operating system, to programming environments, to other compilers, and to program libraries.

The task of compilation can be naturally decomposed into subtasks. This decomposition results in a modular structure which, by the way, is also reflected in the structure of most books about compilers.

As early as the 1950s, it was observed that implementing application systems directly as machine code is both difficult and error-prone, and results in programs which become outdated as quickly as the computers for which they have been developed. High-level machine-independent programming languages, on the other hand, immediately made it mandatory to provide compilers, which are able to automatically translate high-level programs into low-level machine code.

Accordingly, the various subtasks of compilation have been subject to intensive research. For the subtasks of lexical and syntactic analysis of programs, concepts like regular expressions, finite automata, context-free grammars and pushdown automata have been borrowed from the theory of automata and formal languages and adapted to the particular needs of compilers. These developments have been extremely successful. In the case of syntactic analysis, they led to fully automatic techniques for generating the required components solely from corresponding specifications, i.e., context-free grammars. Analogous automatic generation techniques would be desirable for further components of compilers as well, and have, to a certain extent, also been developed.

The current book does not attempt to be a cookbook for compiler writers. Accordingly, there are no recipes such as “in order to construct a compiler from source language X into target language Y, take …”. Instead, our presentation elaborates on the fundamental aspects such as the technical concepts, the specification formalisms for compiler components and methods how to systematically derive implementations. Ideally, this approach may result in fully automatic generator tools.

The book is written for students of computer science. Some knowledge about an object-oriented programming language such as JAVA and very basic principles of a functional language such as OCAML or SML are required. Knowledge about formal languages or automata is useful, but is not mandatory as the corresponding background is provided within the presentation.

Organization of the Book

For the new edition of the book *Wilhelm/Maurer: Compiler Construction*, we decided to distribute the contents over several volumes. The first volume *Wilhelm/Seidl: Compiler Design – Virtual Machines* speaks about *what* a compiler does, i.e., which correspondence it realizes between source program and target program. This is exemplified for an imperative, a functional, a logic-based and an object-oriented language. For each language, a corresponding *virtual* machine (in earlier editions also called *abstract* machine) is designed, and implementations are provided for each construct of the programming language by means of the instructions of the virtual machine.

The subsequent three volumes then describe *how* the compilation process is structured and organized. Each corresponds to one phase of compilation, namely to *syntactic and semantic* analysis, to *program transformation*, and to *code generation*. The present volume is dedicated to the analysis phase, realized by the *front-end* of the compiler. We also briefly discuss how the compilation process as a whole can be organized into phases, which tasks should be assigned to different phases, and which techniques can be used for their implementation.

Further material related to this book is provided on the webpage: http://www2.informatik.tu-_muenchen.de/~seidl/compilers/

Acknowledgement

Besides the helpers from former editions, we would like to thank Michael Jacobs and Jörg Herter for carefully proofreading the chapter on syntactic analysis. When revising the description of the recursive descent parser, we also were supported by Christoph Mallon and his invaluable experience with practical parser implementations.

We hope that our readers will enjoy the present volume and that our book may encourage them to realize compilers of their own for their favorite programming languages.

Saarbrücken and München, August 2012

Reinhard Wilhelm, Helmut Seidl
and Sebastian Hack

Contents

1 The Structure of Compilers

Our series of books treats the compilation of higher programming languages into the machine languages of virtual or real computers. Such compilers are large, complex software systems. Realizing large and complex software systems is a difficult task. What is so special about compilers such that they can even be implemented as a project accompanying a compiler course? One reason is that the big task can be naturally decomposed into subtasks which have clearly defined functionalities and clean interfaces between them. Another reason is automation: several components of compilers need not be programmed by hand, but can be directly generated from specifications by means of standard tools.

The general architecture of a compiler, to be described in the following, is a *conceptual* structure of the process. It identifies the subtasks of compiling a *source* language into a *target* language and defines interfaces between the components realizing the subtasks. The concrete architecture of the compiler is then derived from this conceptual structure. Several components might be combined if the realized subtasks allow this. On the other hand, a component may also be split into several components if the realized subtask is very complex.

A first attempt to structure a compiler decomposes the compiler into three components executing three consecutive phases:

1. The *analysis phase*, realized by the *front-end*. It determines the syntactic structure of the source program and checks whether the static semantic constraints are satisfied. The latter contain the type constraints in languages with static type systems.
2. The *optimization* and *transformation* phase, performed by the *middle-end*. The syntactically analyzed and semantically checked program is transformed by *semantics-preserving* transformations. These transformations mostly aim at improving the efficiency of the program by reducing the execution time, the memory consumption, or the consumed energy. These transformations are independent of the target architecture and mostly also independent of the source language.
3. The *code generation and the machine-dependent optimization* phase, performed by the *back-end*. The program is now translated into an equivalent program

R. Wilhelm, H. Seidl, S. Hack, *Compiler Design*, DOI 10.1007/978-3-642-17540-4_1,

of the target language. Machine-dependent optimizations may be performed, which exploit peculiarities of the target architecture.

This organization splits the task of compilation into a first phase, which depends on the source language, a third phase, which depends only on the target architecture, and a second phase, which is mostly independent of both. This separation of concerns allows one to readily adapt compilers to new source languages as well as to new target architectures.

The following sections present these phases in more detail, decompose them further, and show them working on a small running example. This book describes the analysis phase of the compiler. The transformation phase is presented in much detail in the volume *Analysis and Transformation*. The volume *Code Generation and Machine-Oriented Optimization* covers code generation for a target machine.

1.1 Compilation Subtasks

Figure 1.1 shows the conceptual structure of a compiler. Compilation is decomposed into a sequence of phases. The analysis phase is further split into subtasks, as this volume is concerned with the analysis phase. Each component realizing such a subtask receives a representation of the program as input and delivers another representation as output. The format of the output representation may be different, e.g., when translating a symbol sequence into a tree, or it may be the same. In the latter case, the representation will in general be augmented with newly computed information. The subtasks are represented by boxes labeled with the names of the subtasks and maybe with the name of the module realizing this subtask.

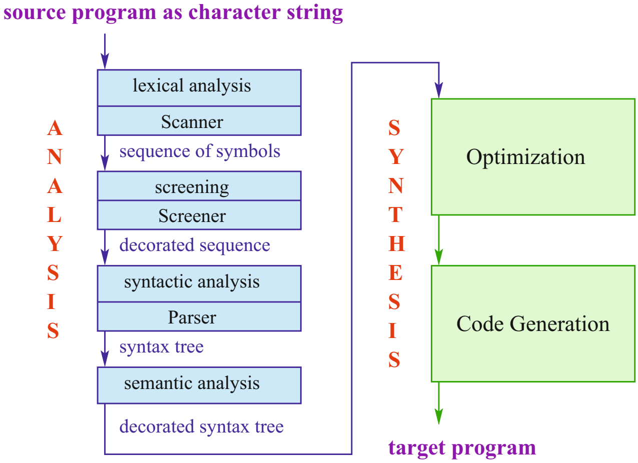


Fig. 1.1 Structure of a compiler together with the program representations during the analysis phase

We now walk through the sequence of subtasks step by step, characterize their job, and describe the change in program representation. As a running example we consider the following program fragment:

$$\begin{array}{l}\mathsf{int}\ a, b;\\ a = 42;\\ b = a * a - 7;\end{array}$$

where $' ='$ denotes the assignment operator.

1.2 Lexical Analysis

The component performing lexical analysis of source programs is often called the *scanner*. This component reads the source program represented as a sequence of characters mostly from a file. It decomposes this sequence of characters into a sequence of lexical units of the programming language. These lexical units are called *symbols*. Typical lexical units are *keywords* such as if, else, while or switch and special characters and character combinations such as $=, ==, ! =, <=, >=, <, >$, (,), [,], { , } or comma and semicolon. These need to be recognized and converted into corresponding internal representations. The same holds for *reserved identifiers* such as names of basic types int, float, double, char, bool or string, etc. Further symbols are identifiers and constants. Examples for identifiers are value42, abc, Myclass, x, while the character sequences 42, 3.14159 and ″HalloWorld!″ represent constants. It is important to note is that there are, in principle, arbitrarily many such symbols. They can, however, be categorized into finitely many symbol *classes*. A symbol class consists of symbols that are equivalent as far as the syntactic structure of programs is concerned. The set of identifiers is an example of such a class. Within this class, there may be subclasses such as type constructors in OCAML or variables in PROLOG, which are written in capital letters. In the class of constants, *int*-constants can be distinguished from floating-point constants and *string*-constants.

The symbols we have considered so far bear semantic interpretations which must be taken into account in code generation. There are, however, also symbols without semantics. Two symbols, for example, need a separator between them if their concatenation would also form a symbol. Such a separator can be a blank, a newline, an indentation or a sequence of such characters. Such *white space* can also be inserted into a program to visualize the structure of the program to human eyes.

Other symbols without meaning for the compiler, but helpful for the human reader, are comments. Comments also can be used by software development tools. Other symbols are *compiler directives* (*pragmas*), which may tell the compiler to include particular libraries or influence the memory management for the program to be compiled. The sequence of symbols for the example program may look as follows:

Int(″*int*″) Sep(″ ″) Id(″*a*″) Com(″,″) Sep(″ ″) Id(″*b*″) Sem(″;″) Sep(″\n″)
Id(″*a*″) Bec(″=″) Intconst(″42″) Sem(″;″) Sep(″\n″)
Id(″*b*″) Bec(″=″) Id(″*a*″) Mop(″∗″) Id(″*a*″) Aop(″−″) Intconst(″7″) Sem(″;″) Sep(″\n″)

To increase readability, the sequence was broken into lines according to the original program structure. Each symbol is represented by its symbol class and the substring representing it in the program. More information may be added such as, e.g., the position of the string in the input.

1.3 The Screener

The scanner delivers a sequence of symbols to the screener. Each symbol is a substring of the program text labeled with its symbol class. It is the task of the screener to further process the symbol sequence. Some symbols are eliminated since they have served their purpose, e.g., as separators. Others are transformed into different representations. More precisely, the screener performs the following actions, specific for different symbol classes:

Separators and comments: Sequences of blanks and newlines serve as separators between symbols. They are of not needed for further processing of the program and can therefore be removed. An exception to this rule are programming languages such as Haskell, where indentation is used to express program nesting. Comments are also not needed later and therefore can be removed.

Pragmas: Compiler directives (pragmas) are not part of the program. They are passed separately on to the compiler.

Other symbols are preserved, while their textual representation may be converted into some internal representation.

Reserved symbols: These symbols have special meanings in the programming language and therefore are preserved by the screener. Examples are begin, end, var, int, etc.

Constants: The sequence of digits as representation of number constants is converted to a binary representation. *String*-constants are stored into an allocated object. In Java implementations, these objects are stored in a dedicated data structure, the *String Pool*. The String Pool is available to the program at run time.

Identifiers: Compilers usually do not work with identifiers represented as string objects. This representation would be too inefficient. Rather, identifiers are coded as unique numbers. The compiler needs to be able to access the external representation of identifiers, though. For this purpose, the identifiers are kept in a data structure, which can be efficiently addressed by their codes.

The screener will produce the following sequence of annotated symbols for our example program:

Int() Id(1) Com() Id(2) Sem()
Id(1) Bec() Intconst(42) Sem()
Id(2) Bec() Id(1) Mop(*Mul*) Id(1) Aop(*Sub*) Intconst(7) Sem()

All separators are removed from the symbol sequence. Semantical values were computed for some of the substrings. The identifiers a and b were coded by the numbers 1 and 2, resp. The sequences of digits for the *int* constants were replaced by their binary values. The internal representations of the symbols Mop and Aop are elements of an appropriate enumeration type.

Scanner and screener are usually combined into one module, which is also called *scanner*. Conceptually, however, they should be kept separate. The task that the scanner, in the restricted meaning of the word, performs can be realized by a finite automaton. The screener, however, may additionally require arbitrary pieces of code.

1.4 Syntactic Analysis

The lexical and the syntactic analyses together recognize the syntactic structure of the source program. Lexical analysis realizes the part of this task that can be implemented by means of finite automata. Syntactic analysis recognizes the hierarchical structure of the program, a task that finite automata cannot perform in general. The syntactical structure of the program consists of sequential and hierarchical composition of language constructs. The hierarchical composition corresponds to the *nesting* of language constructs. Programs in an object-oriented programming language like JAVA, for example, consist of class declarations, which may be combined into packages. The declaration of a class may contain declarations of attributes, constructors, and methods. A method consists of a method head and a method body. The latter contains the implementation of the method. Some language constructs may be nested arbitrarily deep. This is, e.g., the case for blocks or arithmetic expressions, where an unbounded number of operator applications can be used to construct expressions of arbitrary sizes and depths. Finite automata are incapable of recognizing such nesting of constructs. Therefore, more powerful specification mechanisms and recognizers are required.

The component used to recognize the syntactic structure is called *parser*. This component should not only recognize the syntactic structure of correct programs. It should also be able to properly deal with syntactically incorrect programs. After all, most programs submitted to a program contain mistakes. Typical syntax errors are spelling errors in keywords, or missing parentheses or separators. The parser should detect these kind of errors, diagnose them, and maybe even try to correct them.

The syntactic structure of programs can be described by *context-free grammars*. From the theory of formal languages and automata it is known that pushdown automata are equivalent to context-free grammars. Parsers are, therefore, based on *pushdown automata*. There exist many different methods for syntactic analysis. Variants of the two most important methods are described in Chap. 3.

Also, the output of the parser may have several different equivalent formats. In our conceptual compiler structure and in Fig. 1.1, we use *parse trees* as output.

1.5 Semantic Analysis

The task of semantic analysis is to determine properties and check conditions that are relevant for the well-formedness of programs according to the rules of the programming language, but that go beyond what can be described by context-free grammars. These conditions can be completely checked on the basis of the program text and are therefore called *static* semantic properties. This phase is, therefore, called semantic analysis. The *dynamic* semantics, in constrast, describes the behavior of programs when they are executed. The attributes *static* and *dynamic* are associated with the *compile time* and the *run time* of programs, respectively. We list some static semantic properties of programs:

- type correctness in strongly typed programming languages like C, PASCAL, JAVA, or OCAML. A prerequisite for type correctness is that all identifiers are declared, either explicitly or implicitly and, possibly, the absence of multiple declarations of the same identifier.
- the existence of a *consistent type association* with all expressions in languages with type polymorphism.

Example 1.5.1 For the program of Fig. 1.2, semantic analysis will collect the declarations of the *decl*-subtree in a map

$$env = \{\mathsf{Id}(1) \mapsto \mathsf{Int}, \mathsf{Id}(2) \mapsto \mathsf{Int}\}$$

This map associates each identifier with its type. Using this map, semantic analysis can check in the *stat*-subtrees whether variables and expressions are used in a type-correct way. For the first assignment, $a = 42;$, it will check whether the left side of the assignment is a variable identifier, and whether the type of the left side is compatible with the type on the right side. In the second assignement, $b = a * a - 7;$, the type of the right side is less obvious. It can be determined from the types of the variable a and the constant 7. Recall that the arithmetic operators are *overloaded* in most programming languges. This means that they stand for the designated operations of several types, for instance, on *int*- as well as on *float*-operands, possibly even of different precision. The type checker then is expected to resolve overloading. In our example, the type checker determines that the multiplication is an *int*-multiplication and the subtraction an *int*-subtraction, both returning values of type int. The result type of the right side of the assignment, therefore, is int. □

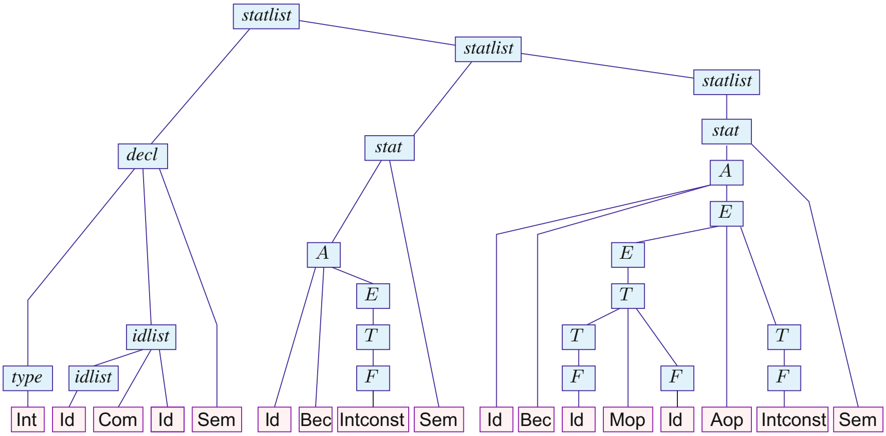


Fig. 1.2 Syntactic analysis of the example program

1.6 Machine-Independent Optimization

Static analyses of the source program may detect potential run-time errors or possibilities for program transformation that may increase the efficiency of the program while preserving the semantics of the program. A *data-flow analysis* or *abstract interpretation* can detect, among others, the following properties of a source program:

- There exists a program path on which a variable would be used without being initialized.
- There exist program parts that cannot be reached or functions that are never called. These superfluous parts need not be compiled.
- A program variable x at a statement in an imperative program has always the same value, c. In this case, variable x can be repaced by the value c in this statement. This analysis would recognize that at each execution of the second assignment, $b = a * a - 7;$, variable a has the value 42. Replacing both occurrences of a by 42 leads to the expression $42 * 42 - 7$, whose value can be evaluated at compile time. This analysis and transformation is called *constant propagation* with *constant folding*.

A major empasis of this phase is on evaluating subexpressions whose value can be determined at compile time. Besides this, the following optimizations can be performed by the compiler:

- *Loop invariant* computations can be moved out of loops. A computation is *loop invariant* if it only depends on variables that do not change their value during the execution of the loop. Such a computation is executed only once instead of in each iteration when it has been moved out of a loop.
- A similar transformation can be applied in the compilation of functional programs to reach the *fully lazy* property. Expressions that only contain variables

bound outside of the function can be moved out of the body of the function and passed to the function in calls with an additional parameter.

These kinds of optimizations are performed by many compilers. They make up the *middle-end* of the compiler. The volume *Compiler Design: Analysis and Transformation* is dedicated to this area.

1.7 Generation of the Target Program

The code generator takes the intermediate representation of the program and generates the target program. A systematic way to translate several types of programming languages to adequate virtual machines is presented in the volume, *Compiler Design: Virtual Machines*. Code generation, as described there, proceeds recursively over the structure of the program. It could, therefore, start directly after syntactic and semantic analysis and work on the decorated syntax tree. The efficiency of the resulting code, though, can be considerably improved if the properties of the target hardware are exploited. The access to values, for example, is more efficient if the values are stored in the registers of the machine. Every processor, on the other hand, has only a limited number of such registers. One task of the code generator is to make good use of this restricted resource. The task to assign registers to variables and intermediate values is called *register allocation*. Another problem to be solved by the code generator is *instruction selection*, i.e., the selection of instruction sequences for expressions and (sequences of) assignments of the source program. Most processor architectures provide multiple possiblities for the translation of a given statement of the source program. From these, a sequence must be selected which is preferable w.r.t. execution time, storage consumption, parallizability, or just program length.

Example 1.7.1 Let us assume that a virtual or concrete target machine has registers $r_1, r_2, \ldots, r_N$ for a (mostly small) N and that it provides, among others, the instructions

Instruction		Meaning
load	r_i, q	$r_i \leftarrow M[q];$
store	q, r_i	$M[q] \leftarrow r_i;$
loadi	r_i, q	$r_i \leftarrow q;$
subi	r_i, r_j, q	$r_i \leftarrow r_j - q;$
mul	r_i, r_j, r_k	$r_i \leftarrow r_j * r_k;$

where q stands for an arbitrary *int*-constant, and $M[\ldots]$ for a memory access. Let us further assume that variables a and b have been assigned the global addresses 1 and 2. A translation of the example program, which stores the values for a and b in the corresponding memeory cells, may result in:

$$\begin{array}{ll}
\mathsf{loadi} & r_1, 42 \\
\mathsf{store} & 1, r_1 \\
\mathsf{mul} & r_2, r_1, r_1 \\
\mathsf{subi} & r_3, r_2, 7 \\
\mathsf{store} & 2, r_3
\end{array}$$

Registers r_1, r_2, and r_3 serve for storing intermediate values during the evaluation of right sides. Registers r_1 and r_3 hold the values of variables a and b, resp. Closer inspection reveals that the compiler may save registers. For instance, register r_1 can be reused for register r_2 since the value in r_1 is no longer needed after the multiplication. Even the result of the instruction subi may be stored in the same register. We, thus, obtain the improved instructions sequence:

$$\begin{array}{ll}
\mathsf{loadi} & r_1, 42 \\
\mathsf{store} & 1, r_1 \\
\mathsf{mul} & r_1, r_1, r_1 \\
\mathsf{subi} & r_1, r_1, 7 \\
\mathsf{store} & 2, r_1
\end{array}$$

□

Even for the simple processor architecture of Example 1.7.1, the code generator must take into account that the number of intermediate results to be stored in registers does not exceed the number of currently available registers. These and similar constraints are to be found in realistic target architectures. Furthermore, they typically offer dedicated instructions for frequent special cases. This makes the task of generating efficient code both intricate and challenging. The necessary techniques are presented in the volume: *Compiler Design: Code Generation and Machine-Level Optimization*.

1.8 Specification and Generation of Compiler Components

All the tasks to be solved during the syntactic analysis can be elegantly specified by different types of grammars. Symbols, the lexical units of the languages, can be described by regular expressions. A nondeterministic finite automaton recognizing the language described by a regular expression R can be automatically derived from R expression. This nondeterministic finite automaton then can be automatically converted into a deterministic finite automaton.

A similar correspondence is known between context-free grammars as used for specifying the hierarchical structure of programs, and pushdown automata. A *nondeterministic* pushdown automaton recognizing the language of a context-free grammar can be automatically constructed from the context-free grammar. For practical applications such as compilation, clearly deterministic pushdown

Table 1.1 Compiler subtasks, specification mechanisms, and corresponding implementation mechanisms

Compilation subtask	Specification mechanism	Implementation mechanism
Lexical analysis	Regular expressions	Deterministic finite automata
Syntactic analysis	Context-free grammars	Deterministic pushdown automata
Semantic analysis	Attribute grammars	Attribute evaluators

automata are preferred. Unlike in the case of finite automata, however, nondeterministic pushdown automata are more powerful than deterministic pushdown automata. Most designers of programming languages have succeeded in staying within the class of deterministically analyzable context-free languages, so that syntactic analysis of their languages is relatively simple and efficient. The example of C++, however, shows that a badly designed syntax requires nondeterministic parsers and thus considerably more effort, both in building a parser and in actually parsing programs in the language.

The compiler components for lexical and syntactic analysis need not be programmed by hand, but can be automatically generated from appropriate specifications. These two example suggest to search for further compiler subtasks that can be solved by automatically generated components. As another example for this approach, we meet *attribute grammars* in this volume. Attribute grammars are an extension of context-free grammars in which computations on syntax trees can be specified. These computations typically check the conformance of the program to semantic conditions such as typing rules. Table 1.1 lists compiler subtasks treated in this volume that can be formally specified in such a way that implementations of the corresponding components can be automatically generated. The specification and the implementation mechanisms are listed with the subtask.

Program invariants as they are needed for the semantics-preserving application of optimizing program transformations can be computed using generic approaches based on the theory of *abstract interpretation*. This is the subject of the volume *Compiler Design: Analysis and Transformation*.

There are also methods to automatically produce components of the compiler back-end. For instance, problems of register allocation and instruction scheduling can be conveniently formulated as instances of *Partitioned Boolean Quadratic Programming* (PBQP) or *Integer Linear Programming* (ILP). Various subtasks of code generation are treated in the volume *Compiler Design: Code Generation and Machine-Level Optimization*.

Lexical Analysis 2

We start this chapter by describing the task of lexical analysis. Then we present regular expressions as specifications for this task. Regular expressions can be automatically converted into nondeterministic finite automata, which implement lexical analysis. Nondeterministic finite automata can be made deterministic, which is preferred for implementing lexical analyzers, often called *scanners*. Another transformation on the resulting deterministic finite automata attempts to reduce the sizes of the automata. These three steps together make up an automatic process generating lexical analyzers from specifications. Another module working in close cooperation with such a finite automaton is the *screener*. It filters out keywords, comments, etc., and may do some bookkeeping or conversion.

2.1 The Task of Lexical Analysis

Let us assume that the source program is stored in a file. It consists of a sequence of characters. Lexical analysis, i.e., the scanner, reads this sequence from left to right and decomposes it into a sequence of lexical units, called *symbols*. Scanner, screener, and parser may work in an integrated way. In this case, the parser calls the combination scanner-screener to obtain the next symbol. The scanner starts the analysis with the character that follows the end of the last found symbol. It searches for the longest prefix of the remaining input that is a symbol of the language. It passes a representation of this symbol on to the screener, which checks whether this symbol is relevant for the parser. If not, it is ignored, and the screener reactivates the scanner. Otherwise, it passes a possibly transformed representation of the symbol on to the parser.

The scanner must, in general, be able to recognize infinitely many or at least very many different symbols. The set of symbols is, therefore, divided into finitely many classes. One *symbol class* consists of symbols that have a similar syntactic role. We distinguish:

- The *alphabet* is the set of characters that may occur in program texts. We use the letter Σ to denote alphabets.

R. Wilhelm, H. Seidl, S. Hack, *Compiler Design*, DOI 10.1007/978-3-642-17540-4_2,

- A *symbol* is a word over the alphabet Σ. Examples are *xyz*12, 125, class, "*abc*".
- A *symbol class* is a set of symbols. Examples are the set of identifiers, the set of *int*-constants, and the set of character strings. We denote these by Id, Intconst, and String, respectively.
- The *representation of a symbol* comprises all of the mentioned information about a symbol that may be relevant for later phases of compilation. The scanner may represent the word *xyz*12 as pair (Id, "*xyz*12"), consisting of the name of the class and the found symbol, and pass this representation on to the screener. The screener may replace "*xyz*12" by the internal representation of an identifier, for example, a unique number, and then pass this on to the parser.

2.2 Regular Expressions and Finite Automata

2.2.1 Words and Languages

We introduce some basic terminology. We use Σ to denote the *alphabet*, that is a finite, nonempty set of characters. A *word* x over Σ of length n is a sequence of n characters from Σ. The *empty word* ε is the empty sequence of characters, i.e., the sequence of length 0. We consider individual characters from Σ as words of length 1.

Σ^n denotes the set of words of length n for $n \geq 0$. In particular, $\Sigma^0 = \{\varepsilon\}$ and $\Sigma^1 = \Sigma$. The set of all words is denoted as Σ^*. Correspondingly, Σ^+ is the set of *nonempty* words, i.e.,

$$\Sigma^* = \bigcup_{n \geq 0} \Sigma^n \qquad \text{and} \qquad \Sigma^+ = \bigcup_{n \geq 1} \Sigma^n .$$

Several words can be concatenated to a new word. The *concatenation* of the words x and y puts the sequence of characters of y after the sequence of characters of x, i.e.,

$$x \,.\, y = x_1 \ldots x_m y_1 \ldots y_n ,$$

if $x = x_1 \ldots x_m$, $y = y_1 \ldots y_n$ for $x_i, y_j \in \Sigma$.

Concatenation of x and y produces a word of length $n + m$ if x and y have lengths n and m, respectively. Concatenation is a binary operation on the set Σ^*. In contrast to addition of numbers, concatenation of words is not *commutative*. This means that the word $x \,.\, y$ is , in general, different from the word $y \,.\, x$. Like addition of numbers, concatenation of words is *associative*, i.e.,

$$x \,.\, (y \,.\, z) = (x \,.\, y) \,.\, z \qquad \text{for all } x, y, z \in \Sigma^*$$

The empty word ε is the *neutral* element with respect to concatenation of words, i.e.,

$$x \,.\, \varepsilon = \varepsilon \,.\, x = x \qquad \text{for all } x \in \Sigma^* .$$

In the following, we will write xy for $x \,.\, y$.

For a word $w = xy$ with $x, y \in \Sigma^*$ we call x a *prefix* and y a *suffix* of w. Prefixes and suffixes are special *subwords*. In general, word y is a subword of word w, if $w = xyz$ for words $x, z \in \Sigma^*$. Prefixes, suffixes, and, in general, subwords of w are called *proper*, if they are different from w.

Subsets of Σ^* are called (formal) *languages*. We need some operations on languages. Assume that $L, L_1, L_2 \subseteq \Sigma^*$ are languages. The *union* $L_1 \cup L_2$ consists of all words from L_1 and L_2:

$$L_1 \cup L_2 = \{w \in \Sigma^* \mid w \in L_1 \text{ or } w \in L_2\}.$$

The *concatenation* $L_1.L_2$ (abbreviated L_1L_2) consists of all words resulting from concatenation of a word from L_1 with a word from L_2:

$$L_1 \,.\, L_2 = \{xy \mid x \in L_1, y \in L_2\}.$$

The *complement* $\overline{L}$ of language L consists of all words in Σ^* that are not contained in L:

$$\overline{L} = \Sigma^* - L.$$

For $L \subseteq \Sigma^*$ we denote L^n as the n-times concatenation of L, L^* as the union of arbitrary concatenations, and L^+ as the union of nonempty concatenations of L, i.e.,

$$\begin{array}{lclcl} L^n & = & \{w_1 \dots w_n \mid w_1, \dots, w_n \in L\} & & \\ L^* & = & \{w_1 \dots w_n \mid \exists n \geq 0.\ w_1, \dots, w_n \in L\} & = & \bigcup_{n \geq 0} L^n \\ L^+ & = & \{w_1 \dots w_n \mid \exists n > 0.\ w_1, \dots, w_n \in L\} & = & \bigcup_{n \geq 1} L^n \end{array}$$

The operation $(\_)^*$ is called *Kleene star*.

Regular Languages and Regular Expressions

The languages described by symbol classes as they are recognized by the scanner are nonempty *regular languages*.

Each nonempty regular language can be constructed starting with singleton languages and applying the operations union, concatenation, and Kleene star. Formally, the set of all *regular languages* over an alphabet Σ is inductively defined by:

- The empty set $\emptyset$ and the set $\{\varepsilon\}$, consisting only of the empty word, are regular.
- The sets $\{a\}$ for all $a \in \Sigma$ are regular over Σ.
- If R_1 and R_2 are regular languages over Σ, so are $R_1 \cup R_2$ and R_1R_2.
- If R is regular over Σ, then so is R^*.

According to this definition, each regular language can be specified by a regular expression. *Regular expressions* over Σ and the regular languages described by them are also defined inductively:

- $\emptyset$ is a regular expression over Σ, which specifies the regular language $\emptyset$.
  ε is a regular expression over Σ, and it specifies the regular language $\{\varepsilon\}$.

- For each $a \in \Sigma$, a is a regular expression over Σ that specifies the regular language $\{a\}$.
- If r_1 and r_2 are regular expressions over Σ that specify the regular languages R_1 and R_2, respectively, then $(r_1 \mid r_2)$ and $(r_1 r_2)$ are regular expressions over Σ that specify the regular languages $R_1 \cup R_2$ and $R_1 R_2$, respectively.
- If r is a regular expression over Σ, that specifies the regular language R, then r^* is a regular expression over Σ that specifies the regular language R^*.

In practical applications, $r?$ is often used as abbreviation for $(r \mid \varepsilon)$ and sometimes also r^+ for the expression (rr^*).

In the definition of regular expressions, we assume that the symbols for the empty set and the empty word are not contained in Σ, and the same also holds for parentheses (and)i, and the operators | and $*$, and also ? and $+$. These characters belong to the specification language for regular expressions and not to the languages denoted by the regular expressions. They are called *metacharacters*. The set of representable characters is limited, though, so that some metacharacters may also occur in the specified languages. A programming system for generating scanners which uses regular expressions as specification language, must provide a mechanism to distinguish for a character between when it is used as a metacharacter and when as a character of the specified language. One such mechanism relies on an *escape character*. In many specification formalisms for regular languages, the character \ is used as escape character. If, for example, the metacharacter | is also included in the alphabet, then every occurrence of | as an alphabet character is preceded with a \. So, the set of all sequences of | is represented by \|*.

As in programming languages, we introduce operator precedences to save on parentheses: The ?-operator has the highest precedence, followed by the Kleene star $(\_)^*$, and then possibly the operator $(\_)^+$, then concatenation, and finally the alternative operator |.

Example 2.2.1 The following table lists a number of regular expressions together with the specified languages, and some or even all of their elements.

Regular expression	Described language	Elements of the language
$a \mid b$	$\{a, b\}$	a, b
ab^*a	$\{a\}\{b\}^*\{a\}$	$aa, aba, abba, abbba, \ldots$
$(ab)^*$	$\{ab\}^*$	$\varepsilon, ab, abab, \ldots$
$abba$	$\{abba\}$	$abba$ □

Regular expressions that contain the empty set as symbol can be simplified by repeated application of the following equalities:

$$\begin{array}{lclcl} r \mid \emptyset & = & \emptyset \mid r & = & r \\ r \,.\, \emptyset & = & \emptyset \,.\, r & = & \emptyset \\ \emptyset^* & = & \epsilon & & \end{array}$$

The equality symbol $=$ between two regular expressions means that both specify the same language. We can prove:

Lemma 2.2.1 For every regular expression r over alphabet Σ, a regular expression r' can be constructed which specifies the same language and additionally has the following properties:

1. If r is a specification of the empty language, then r' is the regular expression $\emptyset$;
2. If r is a specification of a nonempty language, then r' does not contain the symbol $\emptyset$. □

Our applications only have regular expressions that specify nonempty languages. A symbol to describe the empty set, therefore, need not be included into the specification language of regular expressions. The empty word, on the other hand, cannot so easily be omitted. For instance, we may want to specify that the sign of a number constant is *optional*, i.e., may be present or absent. Often, however, specification languages used by scanners do not provide a dedicated metacharacter for the empty word: The ?-operator suffices in all practical situations. In order to remove explicit occurrences of ε in a regular expression by means of ?, the following equalities can be applied:

$$\begin{array}{lllll} r \mid \varepsilon & = & \varepsilon \mid r & = & r? \\ r \,.\, \varepsilon & = & \varepsilon \,.\, r & = & r \\ \varepsilon* & = & \varepsilon? & = & \varepsilon \end{array}$$

We obtain:

Lemma 2.2.2 For every regular expression r over alphabet Σ, a regular expression r' (possibly containing ?) can be constructed which specifies the same language as r and additionally has the following properties:

1. If r is a specification of the language $\{\varepsilon\}$, then r' is the regular expression ε;
2. If r is a specification of a language different from $\{\varepsilon\}$, then r' does not contain ε. □

Finite Automata

We have seen that regular expressions are used for the specification of symbol classes. The implementation of recognizers uses finite automata (FA). Finite automata are acceptors for regular languages. They maintain one state variable that can only take on finitely many values, the *states* of the FA. According to Fig. 2.1, an FA has an input tape and an input head, which reads the input on the tape from left to right. The behavior of the FA is specified by means of a *transition relation* Δ.

Formally, a *nondeterministic finite automaton (with ε-transitions)* (or FA for short) is represented as a tuple $M = (Q, \Sigma, \Delta, q_0, F)$, where

- Q is a finite set of *states*,
- Σ is a finite alphabet, the *input alphabet*,
- $q_0 \in Q$ is the *initial state*,

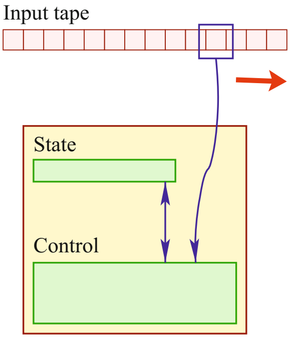


Fig. 2.1 Schematic representation of an FA

- $F \subseteq Q$ is the set of *final states*, and
- $\Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ is the *transition relation*.

A transition $(p, x, q) \in \Delta$ expresses that M can change from its current state p into the state q. If $x \in \Sigma$ then x must be the next character in the input, and after reading x the input head is moved by one character. If $x = \varepsilon$ then no character of the input is read upon this transition. The input head remains at its actual position. Such a transition is called an *ε-transition*.

Of particular interest for implementations are FAs without ε-transitions, which in addition have in each state exactly one transition under each character. Such a finite automaton is called a *deterministic finite automaton* (DFA). For such a DFA the transition relation Δ is a *function* $\Delta : Q \times \Sigma \to Q$.

We describe the behavior of a DFA in comparison with a DFA used as a scanner. The description of the behavior of a scanner is put into boxes. A DFA checks whether given input words are contained in a language or not. It accepts the input word if it arrives in a final state after reading the whole word.

A DFA used as a scanner decomposes the input word into a sequence of subwords corresponding to *symbols* of the language. Each symbol drives the DFA from its initial state into one of its final states.

The DFA starts in its initial state. Its input head is positioned at the beginning of the input head.

A scanner's input head is always positioned at the first not-yet-consumed character.

It then makes a number of steps. Depending on the actual state and the next input symbol, the DFA changes its state and moves its input head to the next character.

The DFA accepts the input word when the input is exhausted, and the actual state is a final state.

Quite analogously, the scanner performs a number of steps. It reports that it has found a symbol or that it has detected an error. If in the actual state no transition under the next input character in the direction of a final state is possible, the scanner rewinds to the last input character that brought it into a final state. The class corresponding to this final state is returned together with the consumed prefix of the input. Then the scanner restarts in the initial state with its input head positioned at the first input character not yet consumed. An error is detected if while rewinding, no final state is found.

Our goal is to derive an implementation of an acceptor of a regular language out of a specification of the language, that is, to construct out of a regular expression r a DFA that accepts the language described by r. In a first step, a *nondeterministic* FA for r is constructed that accepts the language described by r.

An FA $M = (Q, \Sigma, \Delta, q_0, F)$ starts in its initial state q_0 and nondeterministically performs a sequence of steps, a *computation*, under the given input word The input word is accepted if the computation leads to a final state, The future behavior of an FA is fully determined by its actual state $q \in Q$ and the remaining input $w \in \Sigma^*$. This pair (q, w) makes up the *configuration* of the FA. A pair (q_0, w) is an *initial configuration*. Pairs (q, ε) such that $q \in F$ are *final configurations*.

The *step*-relation $\vdash_M$ is a binary relation on configurations. For $q, p \in Q$, $a \in \Sigma \cup \{\varepsilon\}$ and $w \in \Sigma^*$, it holds that $(q, aw) \vdash_M (p, w)$ if and only if $(q, a, p) \in \Delta$ and $a \in \Sigma \cup \{\varepsilon\}$. $\overset{*}{\vdash}_M$ denotes the reflexive, transitive hull of the relation $\vdash_M$. The language accepted by the FA M is defined as

$$L(M) = \{w \in \Sigma^* | (q_0, w) \overset{*}{\vdash}_M (q_f, \varepsilon) \text{ with } q_f \in F\}.$$

Example 2.2.2 Table 2.1 shows the transition relation of an FA M in the form of a two-dimensional matrix T_M. The states of the FA are denoted by the numbers $0, \ldots, 7$. The alphabet is the set $\{0, \ldots, 9, ., E, +, -\}$. Each row of the table describes the transitions for one of the states of the FA. The columns correspond to the characters in $\Sigma \cup \{\varepsilon\}$. The entry $T_M[q, x]$ contains the set of states p such that $(q, x, p) \in \Delta$. The state 0 is the initial state, $\{1, 4, 7\}$ is the set of final states. This FA recognizes unsigned *int*- and *float*-constants. The accepting (final) state 1 can be reached through computations on *int*-constants. Accepting states 4 and 6 can be reached under *float*-constants. □

An FA M can be graphically represented as a finite *transition diagram*. A transition diagram is a finite, directed, edge-labeled graph. The vertices of this graph correspond to the states of M, the edges to the transitions of M. An edge from

Table 2.1 The transition relation of an FA to recognize unsigned *int*- and *float*-constants. The first column represents the identical columns for the digits $i = 0, \ldots, 9$; the fourth the ones for $+$ and $-$

T_M	i	.	E	$+, -$	ε
0	$\{1, 2\}$	$\{3\}$	$\emptyset$	$\emptyset$	$\emptyset$
1	$\{1\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\{4\}$
2	$\{2\}$	$\{4\}$	$\emptyset$	$\emptyset$	$\emptyset$
3	$\{4\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
4	$\{4\}$	$\emptyset$	$\{5\}$	$\emptyset$	$\{7\}$
5	$\emptyset$	$\emptyset$	$\emptyset$	$\{6\}$	$\{6\}$
6	$\{7\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
7	$\{7\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

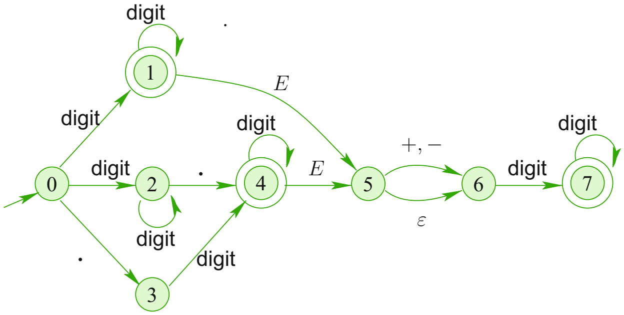


Fig. 2.2 The transition diagram for the FA of Example 2.2.2. The character digit stands for the set $\{0, 1, \ldots, 9\}$, an edge labeled with digit for edges labeled with $0, 1, \ldots 9$ with the same source and target vertices

p to q that is labeled with x corresponds to a transition (p, x, q). The start vertex of the transition diagram, corresponding to the initial state, is marked by an arrow pointing to it. The *end vertices*, corresponding to final states, are represented by doubly encircled vertices. A *w-path* in this graph for a word $w \in \Sigma^*$ is a path from a vertex q to a vertex p, such that w is the concatenation of the edge labels. The language accepted by M consists of all words in $w \in \Sigma^*$, for which there exists a w-path in the transition diagram from q_0 to a vertex $q \in F$.

Example 2.2.3 Figure 2.2 shows the transition diagram corresponding to the FA of example 2.2.2. □

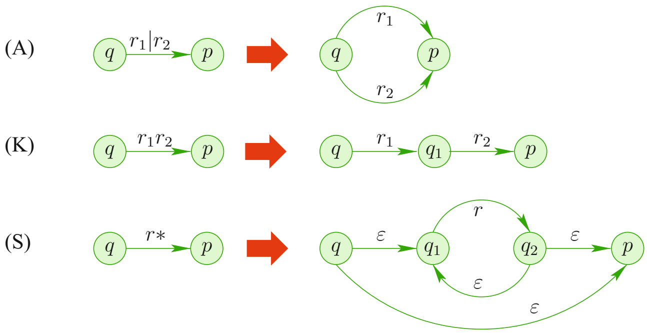


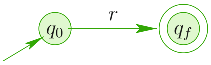
Fig. 2.3 The rules for constructing an FA from a regular expression.

Acceptors

The next theorem guarantees that every regular expression can be compiled into an FA that accepts the language specified by the expression.

Theorem 2.2.1 For each regular expression r over an alphabet Σ, an FA M_r with input alphabet Σ can be constructed such that $L(M_r)$ is the regular language specified by r.

We now present a method that constructs the transition diagram of an FA for a regular expression r over an alphabet Σ. The construction starts with one edge leading from the initial state to a final state. This edge is labeled with r.



While the expression r is decomposed according to its syntactical structure, an implementing transition diagram is built up. The transformation rules are given in Fig. 2.3. The rules are applied until all remaining edges are labeled with $\emptyset$, ε or characters from Σ. Then, the edges labeled with $\emptyset$ are removed.

The application of a rule replaces the edge whose label is matched by the label of the left side with a corresponding copy of the subgraph of the right side. Exactly one rule is applicable for each operator. The application of the rule removes an edge labeled with a regular expression r and inserts new edges that are labeled with the argument expressions of the outermost constructor in r. The rule for the Kleene star inserts additional ε-edges. This method can be implemented by the following program snippet if we take natural numbers as states of the FA:

```
$trans \leftarrow \emptyset$;
$count \leftarrow 1$;
generate$(0, r, 1)$;
return $(count, trans)$;
```

The set *trans* globally collects the transitions of the generated FA, and the global counter *count* keeps track of the largest natural number used as state. A call to a procedure generate for (p, r', q) inserts all transitions of an FA for the regular expression r' with initial state p and final state q into the set *trans*. New states are created by incrementing the counter *count*. This procedure is recursively defined over the structure of the regular expression r':

```
void generate (int $p$, Exp $r'$, int $q$)  {
      switch ($r'$)  {
       case $(r_1 \mid r_2)$ :   generate($p, r_1, q$);
                          generate($p, r_2, q$);  return;
       case $(r_1.r_2)$ :   int $q_1 \leftarrow ++count$;
                          generate($p, r_1, q_1$);
                          generate($q_1, r_2, q$);  return;
       case $r_1^*$ :         int $q_1 \leftarrow ++count$;
                          int $q_2 \leftarrow ++count$;
                          $trans \leftarrow trans \cup \{(p, \varepsilon, q_1), (q_2, \varepsilon, q), (q_2, \varepsilon, q_1)\}$
                          generate($q_1, r_1, q_2$);  return;
       case $\emptyset$ :          return;
       case $x$ :          $trans \leftarrow trans \cup \{(p, x, q)\}$;  return;
      }
}
```

Exp denotes a datatype for regular expressions over the alphabet Σ. We have used a JAVA-like programming language as implementation language. The *switch*-statement was extended by *pattern matching* to elegantly deal with structured data such as regular expressions. This means that patterns are not only used to select between alternatives but also to identify substructures.

A procedure call generate$(0, r, 1)$ terminates after n rule applications, where n is the number of occurrences of operators and symbols in the regular expression r. If l is the value of the counter after the call, the generated FA has $\{0, \ldots, l\}$ as set of states, where 0 is the initial state and 1 the only final state. The transitions are collected in the set *trans*. The FA M_r can be computed in linear time.

Example 2.2.4 The regular expression $a(a|0)^*$ over the alphabet $\{a, 0\}$ describes the set of words $\{a, 0\}^*$ beginning with an a. Figure 2.4 shows the construction of the state diagram of a FA that accepts this language.

□

The Subset Construction

For implementations, *deterministic* finite automata are preferable to nondeterministic finite automata. A deterministic finite automaton M has no transitions under ε and for each pair (q, a) with $q \in Q$ and $a \in \Sigma$, it has exactly one successor state. So, for each state q in M and each word $w \in \Sigma^*$ it has exactly one w-path in the

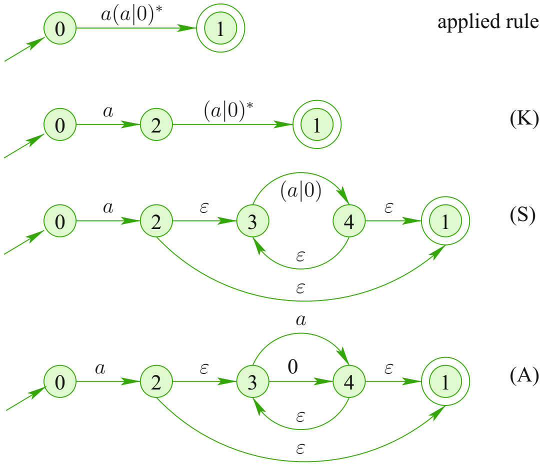


Fig. 2.4 Construction of a transition diagram for the regular expression $a(a \mid 0)^*$

transition diagram of M starting in q. If q is chosen as initial state of M then w is in the language of M if and only if this path leads to a final state of M. Fortunately, we have Theorem 2.2.2.

Theorem 2.2.2 For each FA a DFA can be constructed that accepts the same language. □

Proof The proof provides the second step of the generation procedure for scanners. It uses the *subset construction*. Let $M = (Q, \Sigma, \Delta, q_0, F)$ be an FA. The goal of the subset construction is to construct a DFA $\mathcal{P}(M) = (\mathcal{P}(Q), \Sigma, \mathcal{P}(\Delta), \mathcal{P}(q_0)\mathcal{P}(F))$ that recognizes the same language as M. For a word $w \in \Sigma^*$ let $\mathsf{states}(w) \subseteq Q$ be the set of all states $q \in Q$ for which there exists a w-path leading from the initial state q_0 to q. The DFA $\mathcal{P}(M)$ is given by:

$$\begin{array}{lcll}
\mathcal{P}(Q) & = & \{\mathsf{states}(w) \mid w \in \Sigma^*\} & \\
\mathcal{P}(q_0) & = & \mathsf{states}(\varepsilon) & \\
\mathcal{P}(F) & = & \{\mathsf{states}(w) \mid w \in L(M)\} & \\
\mathcal{P}(\Delta)(S, a) & = & \mathsf{states}(wa) & \text{for } S \in \mathcal{P}(Q) \text{ and } a \in \Sigma \text{ if } S = \mathsf{states}(w)
\end{array}$$

We convince ourselves that our definition of the transition function $\mathcal{P}(\Delta)$ is *reasonable*. For this, we show that for words $w, w' \in \Sigma^*$ with $\mathsf{states}(w) = \mathsf{states}(w')$ it holds that $\mathsf{states}(wa) = \mathsf{states}(w'a)$ for all $a \in \Sigma$. It follows that M and $\mathcal{P}(M)$ accept the same language.

We need a systematic way to construct the states and the transitions of $\mathcal{P}(M)$. The set of final states of $\mathcal{P}(M)$ can be constructed – once the set of states of $\mathcal{P}(M)$ is known, because it holds that:

$$\mathcal{P}(F) = \{A \in \mathcal{P}(M) \mid A \cap F \neq \emptyset\}$$

For a set $A \subseteq Q$ we define the set of ε-successor states A as

$$\mathsf{SS}_\epsilon(S) = \{p \in Q \mid \exists\, q \in S.\, (q, \varepsilon) \vdash_M^* (p, \varepsilon)\}$$

This set consists of all states that can be reached from states in S by ε-paths in the transition diagram of M. This closure can be computed by the following function:

```
set ⟨$state$⟩ closure(set ⟨$state$⟩ $S$) {
    set ⟨$state$⟩ $result \leftarrow \emptyset$;
    list ⟨$state$⟩ $W \leftarrow$ list_of($S$);
    $state\ \ q, q'$;
    while ($W \neq []$) {
        $q \leftarrow$ hd($W$); $W \leftarrow$ tl($W$);
        if ($q \notin result$) {
            $result \leftarrow result \cup \{q\}$;
            forall ($q' : (q, \varepsilon, q') \in \Delta$)
                $W \leftarrow q' :: W$;
        }
    }
    return $result$;
}
```

The states of the resulting DFA that are reachable from A, are collected in the set *result*. The list W contains all elements in *result* whose ε-transitions have not yet been processed. As long as W is not empty, the first state q from W is selected. To do this, functions hd and tl are used that return the first element and the tail of a list, respectively. If q is already contained in *result*, nothing needs to be done. Otherwise, q is inserted into the set *result*. All transitions (q, ε, q') for q in Δ are considered, and the successor states q' are added to W. By applying the closure operator $\mathsf{SS}_\epsilon(\_)$, the initial state $\mathcal{P}(q_0)$ of the subset automaton can be computed:

$$\mathcal{P}(q_0) = S_\varepsilon = \mathsf{SS}_\epsilon(\{q_0\})$$

When constructing the set of all states $\mathcal{P}(M)$ together with the transition function $\mathcal{P}(\Delta)$ of $\mathcal{P}(M)$, bookkeeping is required of the set $Q' \subseteq \mathcal{P}(M)$ of already generated states and of the set $\Delta' \subseteq \mathcal{P}(\Delta)$ of already created transitions.

Initially, $Q' = \{\mathcal{P}(q_0)\}$ and $\Delta' = \emptyset$. For a state $S \in Q'$ and each $a \in \Sigma$, its *successor state* S' under a and Q' and the transition (S, a, S') are added to Δ. The successor state S' for S under a character $a \in \Sigma$ is obtained by collecting the successor states under a of all states $q \in S$ and adding all ε-successor states:

$$S' = \mathsf{SS}_\epsilon(\{p \in Q \mid \exists q \in S : \ (q, a, p) \in \Delta\})$$

The function nextState() serves to compute this set:

```
set ⟨state⟩ nextState(set ⟨state⟩ $S$, symbol $x$) {
    set ⟨state⟩ $S' \leftarrow \emptyset$;
    state $q, q'$;
    forall $(q' : q \in S, (q, x, q') \in \Delta)$ $S' \leftarrow S' \cup \{q'\}$;
    return closure($S'$);
    }
```

Insertions into the sets Q' and Δ' are performed until all successor states of the states in Q' under transitions for characters from Σ are already contained in the set Q'. Technically, this means that the set of all states *states* and the set of all transitions *trans* of the subset automaton can be computed iteratively by the following loop:

```
list ⟨set ⟨state⟩⟩ $W$;
set ⟨state⟩ $S_0 \leftarrow$ closure($\{q_0\}$);
states $\leftarrow \{S_0\}$;  $W \leftarrow [S_0]$;
trans $\leftarrow \emptyset$;
set ⟨state⟩ $S, S'$;
while ($W \neq []$) {
    $S \leftarrow$ hd($W$);  $W \leftarrow$ tl($W$);
    forall ($x \in \Sigma$) {
        $S' \leftarrow$ nextState($S, x$);
        trans $\leftarrow$ trans $\cup \{(S, x, S')\}$;
        if ($S' \notin$ states) {
            states $\leftarrow$ states $\cup \{S'\}$;
            $W \leftarrow W \cup \{S'\}$;
        }
    }
}
```

□

Example 2.2.5 The subset construction applied to the FA of Example 2.2.4 can be executed by the steps described in Fig. 2.5. The states of the DFA to be constructed are denoted by primed natural numbers $0', 1', \ldots$. The initial state $0'$ is the set $\{0\}$. The states in Q' whose successor states are already computed are underlined. The state $3'$ is the empty set of states, i.e., the *error state*. It can never be left. It is the successor state of a state S under a if there is no transition of the FA under a for any FA state in S. □

Minimization

The DFA generated from a regular expression in the given two steps is not necessarily the smallest possible for the specified language. There may be states that have the same *acceptance behavior*. Let $M = (Q, \Sigma, \Delta, q_0, F)$ be a DFA. We say states

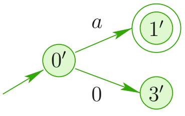
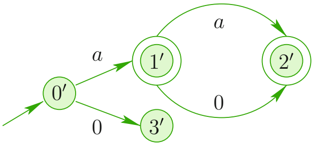
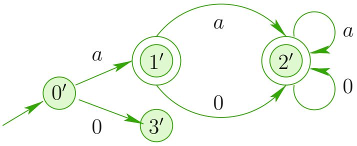
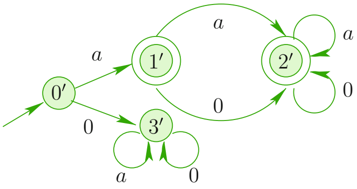
Selected state	New Q'	New (partial) DFA
$0'$	$\{\underline{0'}, 1', 3'\}$ with $1' = \{1, 2, 3\}$	
$1'$	$\{\underline{0'}, \underline{1'}, 2', 3'\}$ with $2' = \{1, 3, 4\}$	
$2'$	$\{\underline{0'}, \underline{1'}, \underline{2'}, 3'\}$	 a a a $0'$ $1'$ $2'$ 0 0 0 $3'$
$3'$	$\{\underline{0'}, \underline{1'}, \underline{2'}, \underline{3'}\}$	 a a a $0'$ $1'$ $2'$ 0 0 0 $3'$ a 0

Fig. 2.5 The subset construction for the FA of Example 2.2.4

p and q of M have the same acceptance behavior if p and q are *indistinguishable* by means of words from Σ^*. In order to make this idea precise, we extend the transition function $\Delta : Q \times \Sigma \to Q$ of the DFA M to a function $\Delta^* : Q \times \Sigma^* \to Q$ that maps each pair $(q, w) \in Q \times \Sigma^*$ to the unique state which is reached by the w-path from q in the transition diagram of M. The function Δ^* is defined by induction over the length of words:

$$\Delta^*(q, \varepsilon) = q \quad \text{and} \quad \Delta^*(q, aw) = \Delta^*(\Delta(q, a), w)$$

for all $q \in Q$, $w \in \Sigma^*$ and $a \in \Sigma$. Then states $p, q \in Q$ have the same acceptance behavior if for all $w \in \Sigma^*$,

$$\Delta^*(p, w) \in F \quad \text{if and only if} \quad \Delta^*(q, w) \in F$$

In this case we write $p \sim_M q$. The relation $\sim_M$ is an equivalence relation on Q. The DFA M is called *minimal* if there is no DFA with fewer states which accepts the same language as M. It turns out that for each DFA, a minimal DFA can be constructed which accepts the same language, and this minimal DFA is unique up to isomorphism. This is the claim of the following theorem.

Theorem 2.2.3 For each DFA M, a minimal DFA M' can be constructed that accepts the same language as M. This minimal DFA is unique up to renaming of states.

Proof For a DFA $M = (Q, \Sigma, \Delta, q_0, F)$ we define a DFA $M' = (Q', \Sigma, \Delta', q_0', F')$ that is minimal. W.l.o.g., we assume that every state in Q is reachable from the initial state q_0 of M. As set of states of the DFA M' we choose the set of *equivalence classes* of states of the DFA M under $\sim_M$. For a state $q \in Q$ let $[q]_M$ denote the equivalence class of states q with respect to the relation $\sim_M$, i.e.,

$$[q]_M = \{p \in Q \mid q \sim_M p\}$$

The set of states of M' is given by:

$$Q' = \{[q]_M \mid q \in Q\}$$

Correspondingly, the initial state and the set of final states of M' are defined by

$$q_0' = [q_0]_M \qquad\qquad F' = \{[q]_M \mid q \in F\},$$

and the transition function of M for $q' \in Q'$ and $a \in \Sigma$ is defined by

$$\Delta'(q', a) = [\Delta(q, a)]_M \quad \text{for a } q \in Q \text{ such that } q' = [q]_M.$$

It can be verified that the new transition function Δ' is well-defined, i.e., that for $[q_1]_M = [q_2]_M$ it holds that $[\Delta(q_1, a)]_M = [\Delta(q_2, a)]_M$ for all $a \in \Sigma$. Furthermore,

$$\Delta^*(q, w) \in F \quad \text{if and only if} \quad (\Delta')^*([q]_M, a) \in F'$$

holds for all $q \in Q$ and $w \in \Sigma^*$. This implies that $L(M) = L(M')$. We now claim that the DFA M' is minimal. For a proof of this claim, consider another DFA $M'' = (Q'', \Sigma, \Delta'', q_0'', F'')$ with $L(M'') = L(M')$ whose states are all reachable from q_0''. Assume for a contradiction that there is a state $q \in Q''$ and words $u_1, u_2 \in \Sigma^*$ such that $(\Delta'')^*(q_0'', u_1) = (\Delta'')^*(q_0'', u_2) = q$, but $(\Delta')^*([q_0]_M, u_1) \neq (\Delta')^*([q_0]_M, u_2)$ holds. For $i = 1, 2$, let $p_i \in Q$ denote a state with $(\Delta')^*([q_0]_M, u_i) = [p_i]_M$. Since $[p_1]_M \neq [p_2]_M$ holds, the states p_1 and p_2 cannot be equivalent. On the other hand, we have for all words $w \in \Sigma^*$,

$$\begin{aligned} \Delta^*(p_1, w) \in F \quad &\text{iff} \quad (\Delta')^*([p_1]_M, w) \in F' \\ &\text{iff} \quad (\Delta'')^*(q, w) \in F'' \\ &\text{iff} \quad (\Delta')^*([p_2]_M, w) \in F' \\ &\text{iff} \quad \Delta^*(p_2, w) \in F \end{aligned}$$

Therefore, the states p_1, p_2 are equivalent – in contradiction to our assumption. Since for every state $[p]_M$ of the DFA M', there is a word u such that $(\Delta')^*([q_0]_M, u) = [p]_M$, we conclude that there is a surjective mapping of the states of M'' onto the states of M'. Therefore, M'' must have at least as many states as M'. Therefore, the DFA M' is minimal. □

The practical construction of M' requires us to compute the equivalence classes $[q]_M$ of the relation $\sim_M$. If *each* state is a final state, i.e., $Q = F$, or none of the states is final, i.e., $F = \emptyset$, then all states are equivalent, and $Q = [q_0]_M$ is the only state of M'.

Let us assume in the following that there is at least one final and one nonfinal state, i.e., $Q \neq F \neq \emptyset$. The algorithm maintains a *partition* Π on the set Q of the states of the DFA M. A partition on the set Q is a set of nonempty subsets of Q, whose union is Q.

A partition Π is called *stable* under the transition relation Δ, if for all $q' \in \Pi$ and all $a \in \Sigma$ there is a $p' \in \Pi$ such that

$$\{\Delta(q, a) \mid q \in q'\} \subseteq p'$$

In a stable partition, all transitions from one set of the partition lead into exactly one set of the partition.

In the partition Π, those sets of states are maintained of which we assume that they have the same acceptance behavior. If it turns out that a set $q' \in \Pi$ contains states with different acceptance behavior, then the set q' is split up. Different acceptance behavior of two states q_1 and q_2 is recognized when the successor states $\Delta(q_1, a)$ and $\Delta(q_2, a)$ for some $a \in \Sigma$ lie in different sets of Π. Then the partition is apparently not stable. Such a split of a set in a partition is called *refinement* of Π. The successive refinement of the partition Π terminates if there is no need for further splitting of any set in the obtained partition. Then the partition is stable under the transition relation Δ.

In detail, the construction of the minimal DFA proceeds as follows. The partition Π is initialized with $\Pi = \{F, Q \backslash F\}$. Let us assume that the actual partition Π of the set Q of states of M' is not yet stable under Δ. Then there exists a set $q' \in \Pi$ and some $a \in \Sigma$ such that the set $\{\Delta(q, a) \mid q \in q'\}$ is not completely contained in any of the sets in $p' \in \Pi$. Such a set q' is then split to obtain a new partition Π' that consists of all nonempty elements of the set

$$\{\{q \in q' \mid \Delta(q, a) \in p'\} \mid p' \in \Pi\}$$

The partition Π' of q' consists of all nonempty subsets of states from q' that lead under a into the same sets in $p' \in \Pi$. The set q' in Π is replaced by the partition Π' of q', i.e., the partition Π is refined to the partition $(\Pi \backslash \{q'\}) \cup \Pi'$.

If a sequence of such refinement steps arrives at a stable partition in Π the set of states of M' has been computed.

$$\Pi = \{[q]_M \mid q \in Q\}$$

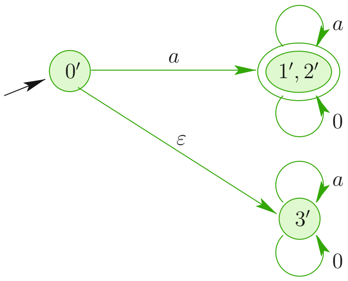


Fig. 2.6 The minimal DFA of Example 2.2.6

Each refinement step increases the number of sets in partition Π. A partition of the set Q may only have as many sets as Q has elements. Therefore, the algorithm terminates after finitely many steps. □

Example 2.2.6 We illustrate the presented method by minimizing the DFA of Example 2.2.5. At the beginning, partition Π is given by

$$\{\{0', 3'\}, \{1', 2'\}\}$$

This partition is not stable. The first set $\{0', 3'\}$ must be split into the partition $\Pi' = \{\{0'\}, \{3'\}\}$. The corresponding refinement of partition Π produces the partition

$$\{\{0'\}, \{3'\}, \{1', 2'\}\}$$

This partition is stable under Δ. It therefore delivers the states of the minimal DFA. The transition diagram of the resulting DFA is shown in Fig. 2.6. □

2.3 A Language for Specifying Lexical Analyzers

We have met regular expressions as specifcation mechanism for symbol classes in lexical analysis. For practical purposes, one often would like to have something more comfortable.

Example 2.3.1 The following regular expression describes the language of unsigned *int*-constants of Examples 2.2.2 and 2.2.3.

$$(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$$

A similar specification of *float*-constants would stretch over three lines. □

In the following, we present several extensions of the specification mechanism of regular expressions that increase the comfort, but not the expressive power of this mechanism. The class of languages that can be described remains the same.

2.3.1 Character Classes

In the specification of a lexical analyzer, one should be able to group sets of characters into *classes* if these characters can be exchanged against each other without changing the symbol class of symbols in which they appear. This is particularly helpful in the case of large alphabets, for instance the alphabet of all *Unicode*-characters. Examples of frequently occurring character classes are:

$$\begin{array}{lcl} \mathsf{alpha} & = & a - zA - Z \\ \mathsf{digit} & = & 0 - 9 \end{array}$$

The first two definitions of character classes define classes by using intervals in the underlying character code, e.g. the ASCII. Note that we need another metacharacter, '-', for the specification of intervals. Using this feature, we can nicely specify the symbol class of identifiers:

$$\mathsf{Id} = \mathsf{alpha}(\mathsf{alpha} \mid \mathsf{digit})^*$$

The specification of character classes uses three metacharacters, namely '=', '−', and the blank. For the *usage* of identifiers for character classes, though, the description formalism must provide another mechanism to distinguish them from ordinary character strings. In our example, we us a dedicated font. In practical systems, the defined names of character classes might be enclosed in dedicated brackets such as {...}.

Example 2.3.2 The regular expression for unsigned *int*- and *float*-constants is simplified through the use of the character classes $\mathsf{digit} = 0 - 9$ to:

$$\begin{array}{l} \mathsf{digit}\ \mathsf{digit}^* \\ \mathsf{digit}\ \mathsf{digit}^* E(+ \mid -)?\mathsf{digit}\ \mathsf{digit}^* \mid \mathsf{digit}^*(.\mathsf{digit} \mid \mathsf{digit}.) \\ \quad \mathsf{digit}^*(E(+ \mid -)?\mathsf{digit}\ \mathsf{digit}^*)? \end{array}$$

□

2.3.2 Nonrecursive Parentheses

Programming languages have lexical units that are characterized by the enclosing parentheses. Examples are string constants and comments. Parentheses limiting comments can be composed of several characters: $(*$ and $*)$ or $/*$ and $*/$ or // and $\backslash n$ (newline). More or less arbitrary texts can be enclosed in the opening and the closing parentheses. This is not easily described. A comfortable abbreviation for this is:

$$r_1 \ \mathsf{until}\ r_2$$

Let L_1, L_2 be the languages specified by the expressions r_1 and r_2, respectively, where L_2 does not contain the empty word. The language specified by the *until*-expression then is given by:

$$L_1 \ \overline{\Sigma^* L_2 \Sigma^*} \ L_2$$

A comment starting with // and ending at the end of line, for example, can be described by:

$$// \ \mathsf{until} \ \backslash n$$

2.4 Scanner Generation

Section 2.2 presented methods for deriving FAs from regular expressions, for compiling FAs into DFAs and finally for minimizing DFAs. In what follows we present the extensions of these methods which are necessary for the implementation of scanners.

2.4.1 Character Classes

Character classes were introduced to simplify regular expressions. They may also lead to smaller automata. The character-class definition

$$\begin{array}{lcl} \mathsf{alpha} & = & a - z \\ \mathsf{digit} & = & 0 - 9 \end{array}$$

can be used to replace the 26 transitions between states under letters by one transition under bu. This may simplify the DFA for the expression

$$\mathsf{Id} = \mathsf{alpha}(\mathsf{alpha} \mid \mathsf{digit})^*$$

considerably. An implementation of character classes uses a map χ that associates each character a with its class. This map can be represented by means of an array that is indexed by characters. The array component indexed with a character, provides the character class of the character. In order for χ to be a function each character must be member of exactly one character class. Character classes are therefore implicitly introduced for characters that do not explicitly occur in a class and those that explicitly occur in the definition of a symbol class. The problem of non-disjoint character classes is resolved by refining the classes to become disjoint. Let us assume that the classes $z_1, \ldots, z_k$ have been specified. Then the generator introduces a new character class for each intersection $\tilde{z}_1 \cap \ldots \cap \tilde{z}_k$ that is nonempty where $\tilde{z}_i$ either denotes z_i or the complement of z_i. Let D be the set of these newly introduced character classes. Each character class z_i corresponds to one of the alternatives $d_i = (d_{i1} \mid \ldots \mid d_{ir_i})$ of character classes d_{ij} in D. Each occurrence of the character class z_i in regular expressions is then replaced by d_i.

Example 2.4.1 Let us assume we have introduced the two classes

$$\begin{array}{lcl} \mathsf{alpha} & = & a - z \\ \mathsf{alphanum} & = & a - z0 - 9 \end{array}$$

to define the symbol class $\mathsf{Id} = \mathsf{alpha}\ \mathsf{alphanum}^*$. The generator would divide one of these character classes into

$$\begin{array}{lclcl} \mathsf{digit}' & = & \mathsf{alphanum}\backslash\mathsf{alpha} & & \\ \mathsf{alpha}' & = & \mathsf{alpha} \cap \mathsf{alphanum} & = & \mathsf{alpha} \end{array}$$

The occurrence of alphanum in the regular expression will be replaced by $(\mathsf{alpha}' \mid \mathsf{digit}')$. □

2.4.2 An Implementation of the *until*-Construct

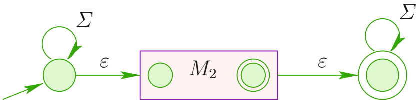
Let us assume the scanner should recognize symbols whose symbol class is specified by the expression $r = r_1$ until r_2. After recognizing a word of the language for r_1 it needs to find a word of the language for r_2 and then halt. This task is a generalization of the *pattern-matching* problem on strings. There exist algorithms that solve this problem for regular patterns in time, linear in the length of the input. These are, for example, used in the UNIX-program EGREP. They construct an FA for this task. Likewise, we now present a single construction of a DFA for the expression r.

Let L_1, L_2 be the languages described by the expressions r_1 and r_2. The language L defined by the expression r_1 until r_2 is:

$$L = L_1\ \overline{\Sigma^* L_2 \Sigma^*}\ L_2$$

The process starts with automata for the languages L_1 and L_2, decomposes the regular expression describing the language, and applies standard constructions for automata. The process has the following seven steps: Fig. 2.7 shows all seven steps for an example.

1. The first step constructs FA M_1 and M_2 for the regular expressions r_1, r_2, where $L(M_1) = L_1$ and $L(M_2) = L_2$. A copy of the FA for M_2 is needed for step 2 and one more in step 6.
2. An FA M_3 is constructed for $\Sigma^* L_2 \Sigma^*$ using the first copy of M_2.



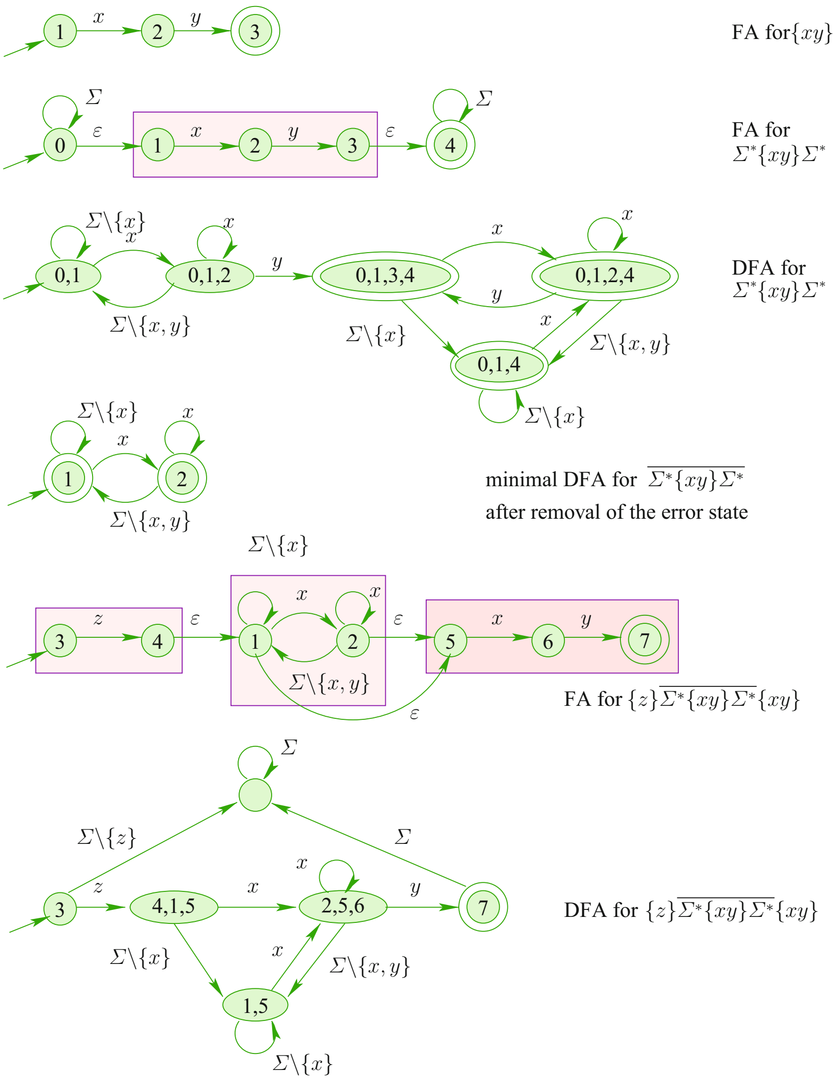
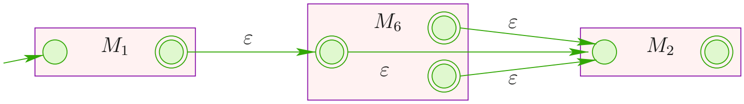

FA for$\{xy\}$
FA for $\Sigma^*\{xy\}\Sigma^*$
DFA for $\Sigma^*\{xy\}\Sigma^*$
minimal DFA for $\overline{\Sigma^*\{xy\}\Sigma^*}$
after removal of the error state
FA for $\{z\}\overline{\Sigma^*\{xy\}\Sigma^*}\{xy\}$
DFA for $\{z\}\overline{\Sigma^*\{xy\}\Sigma^*}\{xy\}$
1
2
3
4
0
0,1
0,1,2
0,1,3,4
0,1,2,4
0,1,4
5
6
7
4,1,5
2,5,6
1,5
x
y
z
ε
Σ
$\Sigma\backslash\{x\}$
$\Sigma\backslash\{x,y\}$
$\Sigma\backslash\{z\}$

Fig. 2.7 The derivation of a DFA for z until xy with $x, y, z \in \Sigma$

The FA M_3 nondeterministically accepts all words over Σ that contain a subword from L_2.

3. The FA M_3 is transformed into a DFA M_4 by the subset construction.
4. A DFA M_5 is constructed that recognizes the language for $\overline{\Sigma^* L_2 \Sigma^*}$. To achieve this, the set of final states of M_4 is exchanged with the set of nonfinal states. In particular, M_5 accepts the empty word since according to our assumption $\varepsilon \notin L_2$. Therefore, the initial state of M_5 also is a final state.

5. The DFA M_5 is transformed into a minimal DFA M_6. All final states of M_4 are equivalent and dead in M_5 since it is not possible to reach a final state of M_5 from any final state of M_4.
6. Using the FA M_1, M_2 for L_1 and L_2 and M_6 an FA M_7 for the language $L_1\ \overline{\Sigma^* L_2 \Sigma^*}\ L_2$ is constructed.



From each final state of M_6 including the initial state of M_6, there is an ε-transition to the initial state of M_2. From there paths under all words $w \in L_2$ lead into the final state of M_2, which is the only final state of M_7.
7. The FA M_7 is converted into a DFA M_8 and possibly minimized.

2.4.3 Sequences of Regular Expressions

Let a sequence

$$r_0, \ldots, r_{n-1}$$

of regular expression be given for the symbol classes to be recognized by the scanner. A scanner recognizing the symbols in these classes can be generated in the following steps:

1. In a first step, FAs $M_i = (Q_i, \Sigma, \Delta_i, q_{0,i}, F_i)$ for the regular expressions r_i are generated, where the Q_i should be pairwise disjoint.
2. The FAs M_i are combined into a single FA $M = (\Sigma, Q, \Delta, q_0, F)$ by adding a new initial state q_0 together with ε-transitions to the initial states $q_{0,i}$ of the FA M_i. The FA M therefore looks as follows:

$$\begin{array}{lll} Q & = & \{q_0\} \cup Q_0 \cup \ldots \cup Q_{n-1} \quad \text{for some} \quad q_0 \notin Q_0 \cup \ldots \cup Q_{n-1} \\ F & = & F_0 \cup \ldots \cup F_{n-1} \\ \Delta & = & \{(q_0, \varepsilon, q_{0,i}) \mid 0 \leq i \leq n-1\} \cup \Delta_0 \cup \ldots \cup \Delta_{n-1}\ . \end{array}$$

The FA M for the sequence accepts the *union* of the languages that are accepted by the FA M_i. The final state reached by a succesful run of the automaton indicates to which class the found symbol belongs.
3. The subset construction is applied to the FA M resulting in a DFA $\mathcal{P}(M)$. A word w is associated with the ith symbol class if it belongs to the language of r_i, but to no language of the other regular expressions r_j, $j < i$. Expressions with smaller indices are here preferred over expressions with larger indices.
To which symbol class a word w belongs can be computed by the DFA $\mathcal{P}(M)$. The word w belongs to the ith symbol class if and only if it drives the DFA

$\mathcal{P}(M)$ into a state $q' \subseteq Q$ such that

$$q' \cap F_i \neq \emptyset \quad \text{and} \quad q' \cap F_j = \emptyset \quad \text{for all } j < i.$$

The set of all these states q' is denoted by F_i'.

4. Hereafter, the DFA $\mathcal{P}(M)$ may be minimized. During minimization, the sets of final states F_i' and F_j' for $i \neq j$ should be kept separate. The minimization algorithm should therefore start with the initial partition

$$\Pi = \{F_0', F_1', \ldots, F_{n-1}', \mathcal{P}(Q) \backslash \bigcup_{i=0}^{n-1} F_i'\}$$

Example 2.4.2 Let the following sequence of character classes be given:

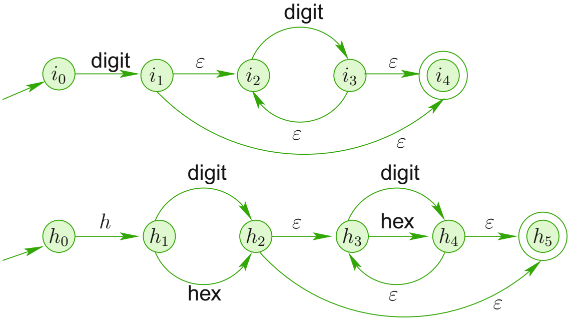
$$\begin{array}{lcl} \textsf{digit} & = & 0-9 \\ \textsf{hex} & = & A-F \end{array}$$

The sequence of regular definitions

$$\textsf{digit digit}^*$$
$$h(\textsf{digit} \mid \textsf{hex})(\textsf{digit} \mid \textsf{hex})^*$$

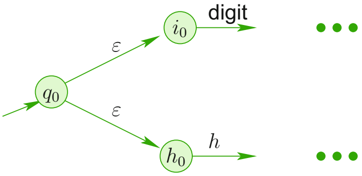
for the symbol classes Intconst and Hexconst are processed in the following steps:

- FA are generated for these regular expressions.

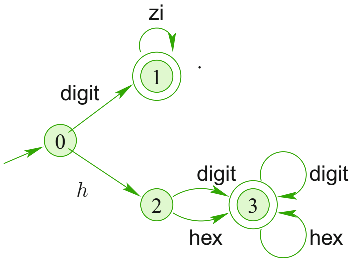


The final state i_4 stands for symbols of the class Intconst, while the final state h_5 stands for symbols of the class Hexconst.

- The two FA are combined with a new initial state q_0:



- The resulting FA is then made deterministic:



An additional state 4 is needed, which is the error state corresponding to the empty set of original states. This state and all transitions into it are left out in the transition diagramm in order to keep readability.

- Minimzation of the DFA does not change it in this example.

The new final state of the generated DFA contains the old final state i_4 and therefore signals the recoginition of symbols of symbol class Intconst. Final state 3 contains h_5 and therefore signals the symbol class Hexconst.

Generated scanners always search for longest prefixes of the remaining input that lead into a final state. The scanner will therefore make a transition out of state 1 if this is possible, that is, if a digit follows. If the next input character is not a digit, the scanner should return to state 1 and reset its reading head. □

2.4.4 The Implementation of a Scanner

We have seen that the core of a scanner is a deterministic finte automaton. The transition function of this automaton can be represented by a two-dimensional array delta. This array is indexed by the actual state and the character class of the next input character. The selected array component contains the new state into which the DFA should go when reading this character in the actual state. While the access to delta$[q, a]$ is usually fast, the size of the array delta can be quite large. We observe, however, that a DFA often contains many transitions into the error state *error*. This state can therfore be chosen as the *default value* for the entries in delta. Representing transitions into only non-error states, may then lead to a sparsely populated array, which can be compressed using well-known methods. These save much space at the cost of slightly increased access times. The now empty entries represent transitions into the error state. Since they are still important for error detection of the scanner, the corresponding information must be preserved.

Let us consider one such compression method. Instead of using the original array delta to represent the transition function an array RowPtr is introduced, which is indexed by states and whose components are addresses of the original rows of delta, see Fig. 2.8.

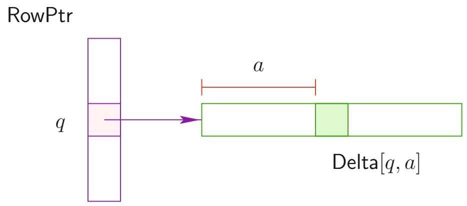


Fig. 2.8 Representation of the transition function of a DFA

We have not gained anything so far, and have even lost a bit of access efficiency. The rows of delta to which entries in RowPtr point are often almost empty. The rows are therefore overlaid into a single 1-dimensional array Delta in such a way that non-empty entries of delta do not collide. To find the starting position for the next row to be inserted into Delta, the *first-fit*-strategy can be used. The row is shifted over the array Delta starting at its beginning, until no non-empty entries of this row collide with non-empty entries already allocated in Delta.

The index in Delta at which the qth row of delta is allocated, is stored ina RowPtr[q] (see Fig. 2.9). One problem is that the represented DFA now has lost its ability to identify errors, that is, undefined transitions. Even if $\Delta(q,a)$ is undefined (representing a transtion into the error state), the component Delta[RowPtr[q] $+ a$] may contain a non-empty entry stemming from a shifted row of a state $p \neq q$. Therefore, another 1-dimensional array Valid is added, which has the same length as Delta. The array Valid contains the information to which states the entries in Delta belong. This means that Valid[RowPtr[q] $+ a$] $= q$ if and only if $\Delta(q,a)$ is defined. The transition function of the DFA can then be implemented by a function

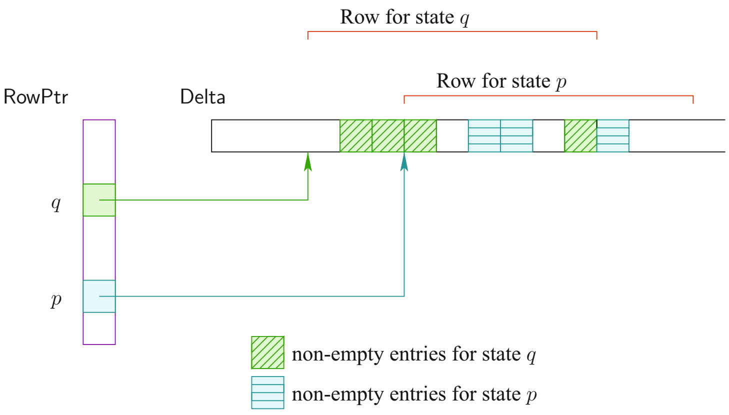

Row for state q
Row for state p
RowPtr
Delta
q
p
non-empty entries for state q
non-empty entries for state p

Fig. 2.9 Compressed representation of the transition function of a DFA

next() as follows:

```
State  next (State q, CharClass a) {
           if  (Valid[RowPtr[q] + a] ≠ q) return error;
           return Delta[RowPtr[q] + a];
}
```

2.5 The Screener

Scanners can be used in many applications, even beyond the pure splitting of a stream of characters according to a specification by means of a sequence of regular expressions. Scanners often provide the possibility of further processing the recognized elements.

To specify this extended functionality, each symbol class is associated with a corresponding semantic action. A screener can therefore be specified as a sequence of pairs of the form

$$\begin{array}{ll} r_0 & \{\mathsf{action}_0\} \\ \ldots & \\ r_{n-1} & \{\mathsf{action}_{n-1}\} \end{array}$$

where r_i is a (possibly extended) regular expression over character classes specifying the ith symbol class, and action_i denotes the semantic action to be executed when a symbol of this class is found. If the screener is to be implemented in a particular programming language, the semanic actions are typically specified as code in this language. Different languages offer different ways to return a representation of the found symbol. An implementation in C would, for instance, return an *int*-value to identify the symbol class, while all other relevant values have to be returned in suitable global values. Somewhat more comfort would be offered for an implementation of the screener in a modern object-oriented language such as JAVA. There a class Token can be introduced whose subclasses C_i correspond to the symbol classes. The last statement in action_i should then be a *return*-statement returning an object of class C_i whose attibutes store all properties of the identified symbol. In a functional language such as OCAML, a data type TOKEN can be supplied whose constructors C_i correspond to the different symbol classes. In this case, the semantic action action_i should be an expression of type token whose value $C_i(\ldots)$ represents the identified symbol of class C_i.

Semantic actions often need to access the text of the actual symbol. Some generated scanners offer access to it by means of a *global* variable *yytext*. Further global variables contain information such as the position of the actual symbol in the input. These are important for the generation of meaningful error messages. Some symbols should be ignored by the screener. Semantic actions therefore should also be able not to return a result but instead ask the scanner for another symbol from the input stream. A comment may, for example, be skipped or a compiler directive be realized without returning a symbol. In a corresponding action in a generator for C or JAVA, the *return*-statement simply would be omitted.

A function yylex() is generated from such a specification. It returns the next symbol every time it is called. Let us assume that a function scan() has been generated for the sequence $r_0, \ldots, r_{n-1}$ of regular expressions which stores the next symbol as a string in the global variable *yytext* and returns the number i of the class of ths symbol. The function yylex() then can be implemented by:

```
Token yylex() {
    while(true)
        switch scan() {
         case 0      :  action_0; break;
                        ...
         case n − 1  :  action_{n−1}; break;
         default     :  return error();
         }
}
```

The function error() is meant to handle the case that an error occurs while the scanner attempts to identify the next symbol. If an action action_i does not have a *return*-statement, the execution is resumed at the beginning of the *switch*-statement and reads the next symbol in the remaining input. If an action action_i terminates by executing a *return*-statement, the *switch*-statement together with the *while*-loop is terminated, and the corresponding value is returned as the return value of the actual call of the function yylex().

2.5.1 Scanner States

Sometimes it is useful to recognize different symbol classes depending on some context. Many scanner generators produce scanners with *scanner states*. The scanner may pass from one state to another one upon reading a symbol.

Example 2.5.1 Skipping comments can be elegantly implemented using scanner states. For this purpose, a distinction is made between a state normal and a state comment.

Symbols from symbol classes that are relevant for the semantics are processed in state normal. An additional symbol class CommentInit contains the start symbol of a comment, e.g., $/*$. The semantic action triggered by recognizing the symbol $/*$ switches to state comment. In state comment, only the end symbol for comments, $*/$, is recognized. All other input characters are skipped. The semantic action triggered upon finding the end-comment symbol switches back to state normal.

The actual scanner state can be kept in a global variable *yystate*. The assignment *yystate* ← state changes the state to the new state state. The specification of a scanner possessing scanner states has the form

$$\begin{array}{ll} A_0 : & \mathit{class\_list}_0 \\ & \ldots \\ A_{r-1} : & \mathit{class\_list}_{r-1} \end{array}$$

where $\mathit{class\_list}_j$ is the sequence of regular expressions and semantic actions for state A_j. For the states normal and comment of Example 2.5.1 we get:

```
normal :
        /*   { $yystate \leftarrow$ comment; }
             ...        //   further symbol classes
comment :
        */   { $yystate \leftarrow$ normal; }
        .    {   }
```

The character . stands for an arbitrary input symbol. Since none of the actions for start, content, or end of comment has a *return*-statement, no symbol is returned for the whole comment. □

Scanner states determine the subsequence of symbol classes of which symbols are recognized. In order to support scanner states, the generation process of the function yylex() can still be applied to the concatenation of the sequence $\mathit{class\_list}_j$. The only function that needs to be modified is the function scan(). To identify the next symbol this function no longer has a single deterministic finite-state automaton but one automaton M_j for each subsequence $\mathit{class\_list}_j$. Depending on the actual scanner state A_j first the corresponding DFA M_j is selected and then applied for the identification of the next symbol.

2.5.2 Recognizing Reserved Words

The duties may be distributed between scanner and screener in many ways. Accordingly, there are also various choices for the functionality of the screener. The advantages and disadvantages are not easily determined. One example for two alternatives is the recognition of keywords. According to the distribution of duties given in the last chapter, the screener is in charge of recognizing reserved symbols (keywords). One possibility to do this is to form an extra symbol class for each reserved word. Figure 2.10 shows a finite-state automaton that recognizes several reserved words in its final states. Reserved keywords in C, Java, and OCaml, on the other hand, have the same form as identifiers. An alternative to recognizing them in the final states of a DFA therefore is to delegate the recognition of keywords to the screener while processing found identifiers.

The function scan() then signals that an identifier has been found. The semantic action associated with the symbol class identifier additionally checks whether, and

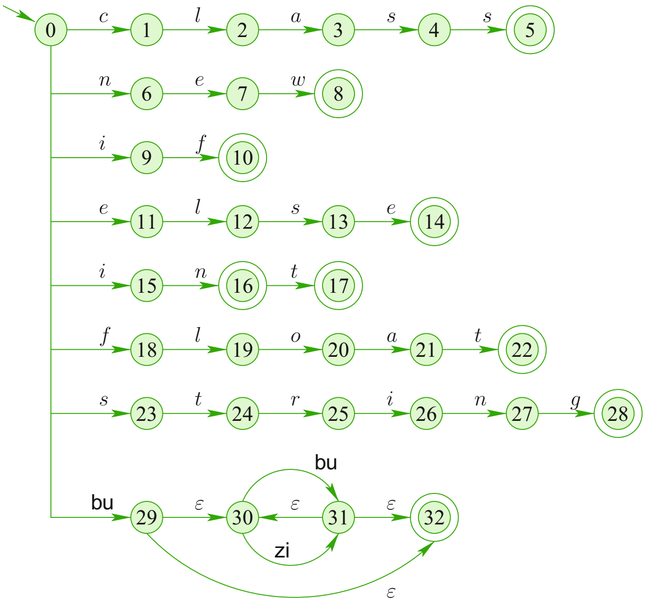


Fig. 2.10 Finite-state automaton for the recognition of identifiers and keywords class, new, if, else, in, int, float, string

if yes, which keyword was found. This distribution of work between scanner and screener keeps the size of the DFA small. A prerequisite, however, is that keywords can be quickly recognized.

Internally, identifiers are often represented by unique *int*-values, where the screener uses a hash table to compute this internal code. A hash table supports the efficient comparison of a newly found identifier with identifiers that have already been entered. If keywords have been entered into the table before lexical analysis starts, the screener thus can then identify their occurrences with approximately the same effort that is necessary for processing other identifiers.

2.6 Exercises

1. **Kleene star**
   Let Σ be an alphabet and $L, M \subseteq \Sigma^*$. Show:
   (a) $L \subseteq L^*$.
   (b) $\varepsilon \in L^*$.
   (c) $u, v \in L^*$ implies $uv \in L^*$.
   (d) L^* is the smallest set with properties (1) - (3), that is, if a set M satisfies: $L \subseteq M,\ \varepsilon \in M$ and $(u, v \in M\ \Rightarrow\ uv \in M)$ it follows $L^* \subseteq M$.

(e) $L \subseteq M$ implies $L^* \subseteq M^*$.
(f) $(L^*)^* = L^*$.

2. **Symbol classes**
FORTRAN provides the implicit declaration of identifiers according to their leading character. Identifiers beginning with one of the letters i, j, k, l, m, n are taken as *int*-variables or *int*-function results. All other identifiers denote *float*-variables.
Give a definition of the symbol classes FloatId and IntId.
3. **Extended regular expressions**
Extend the construction of finite automata for regular expressions from Fig. 2.3 in a way that it processes regular expressions r^+ and $r?$ directly. r^+ stands for rr^* and $r?$ for $(r \mid \varepsilon)$.
4. **Extended regular expressions (cont.)**
Extend the construction of finite automata for regular expressions by a treatment of *counting iteration*, that is, by regular expressions of the form:
$r\{u - o\}$ at least u and at most o consecutive instances of r
$r\{u-\}$ at least u consecutive instances of r
$r\{-o\}$ at most o consecutive instances of r
5. **Deterministic finite automata**
Convert the FA of Fig. 2.10 into a DFA.
6. **Character classes and symbol classes**
Consider the following definitions of character classes:

$$\begin{array}{lcl} \mathsf{bu} & = & a - z \\ \mathsf{zi} & = & 0 - 9 \\ \mathsf{bzi} & = & 0 \mid 1 \\ \mathsf{ozi} & = & 0 - 7 \\ \mathsf{hzi} & = & 0 - 9 \mid A - F \end{array}$$

and the definitions of symbol classes:

$$\begin{array}{l} b\ \mathsf{bzi}^+ \\ o\ \mathsf{ozi}^+ \\ h\ \mathsf{hzi}^+ \\ \mathsf{zi}^+ \\ \mathsf{bu}\ (\mathsf{bu} \mid \mathsf{zi})^* \end{array}$$

(a) Give the partitioning of the character classes that a scanner generator would compute.
(b) Describe the generated finite automaton using these character classes.
(c) Convert this finite automaton into a deterministic one.

7. **Reserved identifiers**
Construct a DFA for the FA of Fig. 2.10.
8. **Table compression**
Compress the table of the deterministic finite automaton using the method of Sect. 2.2.

9. **Processing of Roman numbers**
   (a) Give a regular expression for Roman numbers.
   (b) Generate a deterministic finite automaton from this regular expression.
   (c) Extend this finite automaton such that it computes the decimal value of a Roman number. The finite automaton can perform an assignment to *one* variable w with each state transition. The value is composed of the value of w and of constants. w is initialized with 0. Give an appropriate assignment for each state transition such that w contains the value of the recognized Roman number in each final state.
10. **Generation of a Scanner**
    Generate an OCAML-function yylex from a scanner specification in OCAML. Use functional constructs wherever possible.
    (a) Write a function skip that skips the recognized symbol.
    (b) Extend the generator by scanner states. Write a function next that receives the successor state as argument.

2.7 Bibliographic Notes

The conceptual separation of scanner and screener was proposed by F.L. DeRemer [15]. Many so-called compiler generators support the generation of scanners from regular expressions. Johnson et al. [29] describe such a system. The corresponding routine under UNIX, LEX, was realized by M. Lesk [42]. FLEX was implemented by Vern Paxson. The approach described in this chapter follows the scanner generator JFLEX for JAVA.

Compression methods for sparsely populated matrices as they are generated in scanner and parser generators are described and analyzed in [61] and [11].

Syntactic Analysis 3

3.1 The Task of Syntactic Analysis

The parser realizes the *syntactic analysis* of programs. Its input is a sequence of symbols as produced by the combination of scanner and screener. The parser is meant to identify the syntactic structure in this sequence of symbols, that is how the syntactic units are composed from other units. Syntactic units in imperative languages are, for example, variables, expressions, declarations, statements, and sequences of statements. Functional languages have variables, expressions, patterns, definitions, and declarations, while logic languages such as PROLOG have variables, terms, goals, and clauses.

The parser represents the syntactic structure of the input program in a data structure that allows the subsequent phases of the compiler to access the individual program components. One possible representation is the *syntax tree* or *parse tree*. The syntax tree may later be decorated with more information about the program. Transformations of the program may rely on this data structure, and even code for a target machine can be generated from it.

For some languages, the compilation task is so simple that programs can be translated in one pass over the program text. In this case, the parser can avoid the construction of the intermediate representation. The parser acts as the main function calling routines for semantic analysis and for code generation.

Many programs that are presented to a compiler contain errors. Many of the errors are violations of the rules for forming valid programs. Often such *syntax errors* are simply scambled letters, nonmatching brackets or missing semicolons. The compiler is expected to adequately react to errors. It should at least attempt to locate errors precisely. However, often only the localization of error *symptoms* is possible, not the localization of the errors themselves. The error symptom is the position where no continuation of the syntactic analysis is possible. The compiler is also often expected not to give up after the first error found, but to continue to analyze the rest of the program in order to detect more errors in the same run.

R. Wilhelm, H. Seidl, S. Hack, *Compiler Design*, DOI 10.1007/978-3-642-17540-4_3,

The syntactic structure of the programs written in some programming language can be described by a context-free grammar. There exist methods to automatically generate a parser from such a description. For efficiency and unambiguation, parsing methods are often restricted to deterministically analyzable context-free languages for which parsers can be generated automatically. The parsing methods used in practice fall into two categories, *top-down* and *bottom-up* parsing methods. Both read the input from left to right. The differences in the way they work can best be made clear by regarding how they construct parse trees.

Top-down parsers start the syntactic analysis of a given program and the construction of the parse tree with the start symbol of the grammar and with the root node of the parse tree which is labeled with the start symbol. *Top-down* parsers are called *predictive* parsers since they make predictions about what they expect to find next in the input. They then attempt to verify the prediction by comparing it with the remaining input. The first prediction is the start symbol of the grammar. It says that the parser expects to find a word for the start symbol. Let us assume that a prefix of the prediction is already confirmed. Then there are two cases:

- The nonconfirmed part of the prediction starts with a nonterminal. The *top-down* parser will then refine its prediction by selecting one of the alternatives of this nonteminal.
- The nonconfirmed part of the prediction starts with a terminal symbol. The *top-down* parser will then compare this with the next input symbol. If they agree, another symbol of the prediction is confirmed.

The *top-down* parser terminates successfully when the whole input has been predicted and confirmed.

Bottom-up parsers start the syntactic analysis of a given program and the construction of the parse tree with the input, that is, the given program. They attempt to discover the syntactic structure of longer and longer prefixes of the input program. They attempt to replace occurrences of right sides of productions by their left-side nonterminals. Such a replacement is called a *reduction*. If the parser cannot perform a reduction it performs a *shift*, that is, it reads the next input symbol. Since these are the only two actions, a *bottom-up* parser is also called *shift-reduce* parser. The analysis terminates successfully when the parser has reduced the input program by a sequence of *shift* and *reduce* steps to the start symbol of the grammar.

The Treatment of Syntax Errors

Most programs that are submitted to a compiler are erroneous. Many contain syntax errors. The compiler should, therefore, treat the *normal* case, namely the erroneous program, adequately. Lexical errors are rather local. Syntax errors, for instance in the parenthesis structure of a program, are often more difficult to diagnose. This chapter covers required and possible reactions to syntax errors by the parser. There are essentially four nonexclusive reactions to syntax errors that are desirable:

1. The error is localized and reported;
2. The error is diagnozed;
3. The error is corrected;
4. The parser gets back into a state in which it can possibly detect further errors.

The first reaction is absolutely required. Later stages of the compiler assume that they receive syntactically correct programs in the form of syntax trees. And given that there are errors in the program, the programmer better be informed about them. There are, however, two significant problems: First, further syntax errors can remain undetected in the vicinity of a detected error. Second, since the parser only gets suspicious when it gets stuck, the parser will, in general, only detect error *symptoms*, not errors themselves.

Example 3.1.1 Consider the following erroneous assignment statement:

$$\begin{array}{llcl} & a \;=\; a * (b + c * d & & ; \\ & & \uparrow & \\ \text{error symptom:} & &) & \text{is missing} \end{array}$$

Several errors could lead to the same error symptom: Either there is an extra open parenthesis, or a closing parenthesis is missing after c or is missing after d. Each of the three corrections leads to a program with different meaning. □

At errors of extra or missing parentheses such as {, }, begin, end, if, etc., the position of the error and the position of the error-symptom can be far apart. Practical parsing methods, such as $LL(k)$- and $LR(k)$ parsing, have the *viable-prefix* property:

> Whenever the prefix u of a word has been analyzed without announcing an error, then there exists a word w such that uw is a word of the language.

Parsers possessing this property report error and error symptoms at the earliest possible time. We have explained above that, in general, the parser only discovers error symptoms, not errors themselves. Still, we will speak of *errors* in the following. In this sense, the discussed parsers perform the first two listed actions: they report and try to diagnose errors.

Example 3.1.1 shows that the second action is not as easily realized. The parser can only attempt a diagnosis of the error symptom. It should at least provide the following information:

- the position of the error in the program,
- a description of the parser configuration, i.e., the current state, the expected symbol, and the found symbol.

For the third listed action, the correction of an error, the parser would need to guess the intention of the programmer. This is, in general, difficult. Slightly more realistic is to search for an error correction that is globally optimal. The parser is given the capability to insert or delete symbols in the input word. The *globally optimal* error correction for an erroneous input word w is a word w' that is obtained from w by a minimal number of such insertions and deletions. Such methods have been proposed in the literature, but have not been used in practice due to the tremendous effort that is required.

Instead, most parsers perform only local corrections to have the parser move from the error configuration to a new configuration in which it can at least read the next input symbol. This prevents the parser from going into an endless loop while trying to repair an error.

The Structure of this Chapter

Section 3.2 presents the theoretical foundations of syntax analysis, context-free grammars, their notion of derivation and pushdown automata as their acceptors. A special nondeterministic pushdown automaton for a context-free grammar is introduced that recognizes the language defined by the grammar. Deterministic *top-down* and *bottom-up* parser for the grammar are derived from this pushdown automaton.

Sections 3.3 and 3.4 describe *top-down*- and *bottom-up* syntax analysis, respectively. The corresponding grammar classes are characterized and methods for generating corresponding parsers are presented. Techniques for error handling are described in detail for both *top-down* and *bottom-up* parsers.

3.2 Foundations

In the same way as lexical analysis is specified by regular expressions and implemented by finite automata, so is syntax analysis specified by context-free grammars (CFG) and implemented by pushdown automata (PDA). Regular expressions alone are not sufficient to describe the syntax of programming languages since they cannot express *embedded recursion* as occurs in the nesting of expressions, statements, and blocks.

In Sects. 3.2.1 and 3.2.3, we introduce the necessary notions about context-free grammars and pushdown automata. Readers familiar with these notions can skip them and go directly to Sect. 3.2.4. In Sect. 3.2.4, a pushdown automaton is introduced for a context-free grammar that accepts the language defined by that grammar.

3.2.1 Context-Free Grammars

Context-free grammars can be used to describe the syntactic structure of programs of a programming language. The grammar specifies the elementary components of programs and how fragments of programs can be composed to form bigger fragments.

Example 3.2.1 Productions of a grammar to describe a C-like programming language may look like the following:

$$\begin{array}{lll}
\langle \mathit{stat} \rangle & \rightarrow & \langle \mathit{if\_stat} \rangle \ \mid \\
 & & \langle \mathit{while\_stat} \rangle \ \mid \\
 & & \langle \mathit{do\_while\_stat} \rangle \ \mid \\
 & & \langle \mathit{exp} \rangle \ ; \ \mid \\
 & & ; \ \mid \\
 & & \{ \ \langle \mathit{stats} \rangle \ \} \\
\langle \mathit{if\_stat} \rangle & \rightarrow & \mathsf{if(} \ \langle \mathit{exp} \rangle \ \mathsf{)else} \ \langle \mathit{stat} \rangle \ \mid \\
 & & \mathsf{if(} \ \langle \mathit{exp} \rangle \ \mathsf{)} \ \langle \mathit{stat} \rangle \\
\langle \mathit{while\_stat} \rangle & \rightarrow & \mathsf{while(} \ \langle \mathit{exp} \rangle \ \mathsf{)} \ \langle \mathit{stat} \rangle \\
\langle \mathit{do\_while\_stat} \rangle & \rightarrow & \mathsf{do} \ \langle \mathit{stat} \rangle \ \mathsf{while(} \ \langle \mathit{exp} \rangle \ \mathsf{);} \\
\langle \mathit{exp} \rangle & \rightarrow & \langle \mathit{assign} \rangle \ \mid \\
 & & \langle \mathit{call} \rangle \ \mid \\
 & & \mathsf{Id} \ \mid \\
 & & \ldots \\
\langle \mathit{call} \rangle & \rightarrow & \mathsf{Id} \ \mathsf{(} \ \langle \mathit{exps} \rangle \ \mathsf{)} \ \mid \\
 & & \mathsf{Id()} \\
\langle \mathit{assign} \rangle & \rightarrow & \mathsf{Id} \ {}'{=}' \ \langle \mathit{exp} \rangle \\
\langle \mathit{stats} \rangle & \rightarrow & \langle \mathit{stat} \rangle \ \mid \\
 & & \langle \mathit{stats} \rangle \ \langle \mathit{stat} \rangle \\
\langle \mathit{exps} \rangle & \rightarrow & \langle \mathit{exp} \rangle \ \mid \\
 & & \langle \mathit{exps} \rangle \ , \ \langle \mathit{exp} \rangle
\end{array}$$

The nonterminal symbol $\langle \mathit{stat} \rangle$ generates statements. As for regular expressions, the metacharacter | is used to combine several alternatives, here for possible right sides for one nonterminal. According to these productions, a statement is either an *if*-statement, a *while*-statement, a *do-while*-statement, an expression followed by a semicolon, an empty statement, or a sequence of statements in parentheses. The nonterminal symbol $\langle \mathit{if\_stat} \rangle$ generates *if*-statements in which the *else*-part may be present or absent. These statements always start with the keyword if, followed by an expression in parentheses, and a statement. This statement may be followed by the keyword else and another statement. Further productions describe how *while*- and *do-while*-statements and expressions are constructed. For expressions, only a few alternatives are explicitly given. Further alternatives are indicated by .... □

Formally, a *context-free grammar* (CFG for short) is a quadruple $G = (V_N, V_T, P, S)$, where V_N, V_T are disjoint alphabets, V_N is the set of *nonterminals*, V_T is the set of *terminals*, $P \subseteq V_N \times (V_N \cup V_T)^*$ is the finite set of *production rules*, and $S \in V_N$ is the *start symbol*.

Terminal symbols (in short: terminals) are the symbols from which the programs of the programming language are built. While we spoke of alphabets of *characters* in the section on lexical analysis, typically ASCII or Unicode characters, we now speak of alphabets of *symbols* as they are provided by the scanner or the screener. Examples for such symbols are reserved keywords of the language, or symbol classes such as identifiers which comprise a set of symbols.

The nonterminals of the grammar denote sets of words that can be produced from them by means of the production rules of the grammar. In the example grammar 3.2.1, nonterminals are enclosed in angle brackets. A production rule (in short: production) (A, α) in the relation P describes a possible replacement: an occurrence of the left side A in a word $\beta = \gamma_1 A \gamma_2$ can be replaced by the right side $\alpha \in (V_T \cup V_N)^*$. In the view of a *top-down* parser, a new word $\beta' = \gamma_1 \alpha \gamma_2$ is *produced* or *derived* from the word β.

A *bottom-up* parser, on the other hand, interprets the production (A, α) as a replacement of the right side α by the left side A. Applying the production to a word $\beta' = \gamma_1 \alpha \gamma_2$ *reduces* this to the word $\beta = \gamma_1 A \gamma_2$.

We introduce some useful conventions concerning a CFG $G = (V_N, V_T, P, S)$. Capital latin letters from the beginning of the alphabet, e.g., A, B, C, are used to denote nonterminals from V_N; capital latin letters from the end of the alphabet, e.g., X, Y, Z, denote terminals or nonterminals. Small latin letters from the beginning of the alphabet, e.g., $a, b, c, \ldots$, stand for terminals from V_T; small latin letters from the end of the alphabet, like u, v, w, x, y, z, stand for terminal words, that is, elements from V_T^*; small greek letters such as $\alpha, \beta, \gamma, \varphi, \psi$ stand for words from $(V_T \cup V_N)^*$.

The relation P is seen as a set of production rules. Each element (A, α) of this relation is, more intuitively, written as $A \to \alpha$. All productions $A \to \alpha_1, A \to \alpha_2, \ldots, A \to \alpha_n$ for a nonterminal A are combined to

$$A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$$

The $\alpha_1, \alpha_2, \ldots, \alpha_n$ are called the *alternatives* of A.

Example 3.2.2 The two grammars G_0 and G_1 describe the same language:

$$\begin{array}{lcll}
G_0 & = & \{E, T, F\}, \{+, *, (,), \mathsf{Id}\}, P_0, E) & \text{where } P_0 \text{ is given by:} \\
 & & \quad E \;\to\; E + T \mid T & \\
 & & \quad T \;\to\; T * F \mid F & \\
 & & \quad F \;\to\; (E) \mid \mathsf{Id} & \\
G_1 & = & (\{E\}, \{+, *, (,), \mathsf{Id}\}, P_1, E) & \text{where } P_1 \text{ is given by:} \\
 & & E \to E + E \mid E * E \mid (E) \mid \mathsf{Id} &
\end{array}$$

□

We say, a word φ *directly produces* a word ψ according to G, written as $\varphi \underset{G}{\Longrightarrow} \psi$ if $\varphi = \sigma A \tau, \psi = \sigma \alpha \tau$ holds for some words σ, τ and a production $A \to \alpha \in P$. A word φ *produces* a word ψ according to G, or ψ is *derivable* from φ according to G, written as $\varphi \overset{*}{\underset{G}{\Longrightarrow}} \psi$, if there is a finite sequence $\varphi_0, \varphi_1, \ldots \varphi_n$, $(n \geq 0)$ of words such that

$$\varphi = \varphi_0, \psi = \varphi_n \quad \text{and} \quad \varphi_i \underset{G}{\Longrightarrow} \varphi_{i+1} \quad \text{for all} \quad 0 \leq i < n.$$

The sequence $\varphi_0, \varphi_1, \ldots, \varphi_n$ is called a *derivation* of ψ from φ according to G. The existence of a derivation of length n is written as $\varphi \underset{G}{\overset{n}{\Longrightarrow}} \psi$. The relation $\underset{G}{\overset{*}{\Longrightarrow}}$ denotes the reflexive and transitive closure of $\underset{G}{\Longrightarrow}$.

Example 3.2.3 The grammars of Example 3.2.2 have, among others, the derivations

$$\begin{aligned}
&E \underset{G_0}{\Longrightarrow} E + T \underset{G_0}{\Longrightarrow} T + T \underset{G_0}{\Longrightarrow} T * F + T \underset{G_0}{\Longrightarrow} T * \mathsf{Id} + T \underset{G_0}{\Longrightarrow} \\
&\qquad F * \mathsf{Id} + T \underset{G_0}{\Longrightarrow} F * \mathsf{Id} + F \underset{G_0}{\Longrightarrow} \mathsf{Id} * \mathsf{Id} + F \underset{G_0}{\Longrightarrow} \mathsf{Id} * \mathsf{Id} + \mathsf{Id}, \\
&E \underset{G_1}{\Longrightarrow} E + E \underset{G_1}{\Longrightarrow} E * E + E \underset{G_1}{\Longrightarrow} \mathsf{Id} * E + E \underset{G_1}{\Longrightarrow} \mathsf{Id} * E + \mathsf{Id} \underset{G_1}{\Longrightarrow} \\
&\qquad \mathsf{Id} * \mathsf{Id} + \mathsf{Id}\,.
\end{aligned}$$

We conclude from these derivations that $E \underset{G_1}{\overset{*}{\Longrightarrow}} \mathsf{Id} * \mathsf{Id} + \mathsf{Id}$ holds as well as $E \underset{G_0}{\overset{*}{\Longrightarrow}} \mathsf{Id} * \mathsf{Id} + \mathsf{Id}$. □

The *language defined by* a CFG $G = (V_N, V_T, P, S)$ is the set

$$L(G) = \{u \in V_T^* \mid S \underset{G}{\overset{*}{\Longrightarrow}} u\}.$$

A word $x \in L(G)$ is called a *word* of G. A word $\alpha \in (V_T \cup V_N)^*$ where $S \underset{G}{\overset{*}{\Longrightarrow}} \alpha$ is called a *sentential form* of G.

Example 3.2.4

Let us consider again the grammars of Example 3.2.3. The word $\mathsf{Id} * \mathsf{Id} + \mathsf{Id}$ is a word of both G_0 and G_1, since $E \underset{G_0}{\overset{*}{\Longrightarrow}} \mathsf{Id} * \mathsf{Id} + \mathsf{Id}$ as well as $E \underset{G_1}{\overset{*}{\Longrightarrow}} \mathsf{Id} * \mathsf{Id} + \mathsf{Id}$ hold. □

We omit the index G in $\underset{G}{\Longrightarrow}$ when the grammar to which the derivation refers is clear from the context.

The syntactic structure of a program, as it results from syntactic analysis, is the *syntax tree* or *parse tree* (we will use these two notions synonymously). The parse tree provides a canonical representation of derivations. Within a compiler, the parse tree serves as the *interface* to the subsequent compiler phases. Most approaches to the evaluation of semantic attributes, as they are described in Chap. 4, about semantic analysis, work on this tree structure.

Let $G = (V_N, V_T, P, S)$ be a CFG. Let t be an ordered tree whose inner nodes are labeled with symbols from V_N and whose leaves are labeled with symbols from $V_T \cup \{\varepsilon\}$. A *parse* or *syntax* tree t is an ordered tree where the inner nodes and leaf

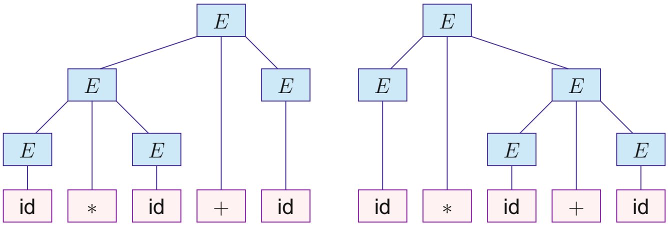


Fig. 3.1 Two syntax trees according to grammar G_1 of Example 3.2.2 for the word Id $*$ Id $+$ Id

nodes are labeled with symbols from V_N and elements from $V_T \cup \{\epsilon\}$, respectively. Moreover, the label B of each inner node n of t together with the sequence of labels $X_1, \ldots, X_k$ of the children of n in t has the following properties:

1. $B \to X_1 \ldots X_k$ is a production from P.
2. If $X_1 \ldots X_k = \varepsilon$, then node n has exactly one child and this child is labeled with ε.
3. If $X_1 \ldots X_k \neq \varepsilon$ then $X_i \in V_N \cup C_T$ for each i.

If the root of t is labeled with nonterminal symbol A, and if the concatenation of the leaf labels yields the terminal word w we call t a parse tree for nonterminal A and word w according to grammar G. If the root is labeled with S, the start symbol of the grammar, we just call t a parse tree for w.

Example 3.2.5 Figure 3.1 shows two syntax trees according to grammar G_1 of Example 3.2.2 for the word Id $*$ Id $+$ Id. □

A syntax tree can be viewed as a representation of derivations where one abstracts from the order and the direction, *derivation* or *reduction*, in which productions are applied. A word of the language is called *ambiguous* if there exists more than one parse tree for it. Correspondingly, the grammar G is called *ambiguous* if $L(G)$ contains at least one ambiguous word. A CFG that is not ambiguous is called *unambiguous*.

Example 3.2.6 The grammar G_1 is ambiguous because the word Id $*$ Id $+$ Id has more than one parse tree. The grammar G_0, on the other hand, is unambiguous. □

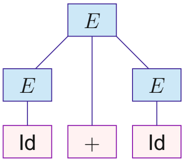
The definition implies that each word $x \in L(G)$ has at least one derivation from S. To each derivation for a word x corresponds a parse tree for x. Thus, each word $x \in L(G)$ has at least one parse tree. On the other hand, to each parse tree for a word x corresponds at least one derivation for x. Any such derivation can be easily read off the parse tree.

Example 3.2.7 The word Id $+$ Id has the one parse tree of Fig. 3.2 according to grammar G_1. Two different derivations result depending on the order in which the nonterminals are replaced:

$$\begin{array}{l} E \Rightarrow E + E \Rightarrow \mathsf{Id} + E \Rightarrow \mathsf{Id} + \mathsf{Id} \\ E \Rightarrow E + E \Rightarrow E + \mathsf{Id} \Rightarrow \mathsf{Id} + \mathsf{Id} \end{array}$$

□

Fig. 3.2 The uniquely determined parse tree for the word Id + Id.



In Example 3.2.7 we saw that – even with unambiguous words – several derivations may correspond to one parse tree. This results from the different possibilities to choose a nonterminal in a sentential form for the next application of a production. There are two different canonical replacement strategies: one is to replace in each step the leftmost nonterminal, while the other one replaces in each step the rightmost nonterminal. The corresponding uniquely determined derivations are called *leftmost* and *rightmost* derivations, respectively.

Formally, a derivation $\varphi_1 \Longrightarrow \ldots \Longrightarrow \varphi_n$ of $\varphi = \varphi_n$ from $S = \varphi_1$ is is a *leftmost derivation* of φ, denoted as $S \underset{lm}{\overset{*}{\Longrightarrow}} \varphi$, if in the derivation step from φ_i to φ_{i+1} the leftmost nonterminal of φ_i is replaced, i.e., $\varphi_i = uA\tau$, $\varphi_{i+1} = u\alpha\tau$ for a word $u \in V_T^*$ and a production $A \to \alpha \in P$. Similarly, a derivation $\varphi_1 \Longrightarrow \ldots \Longrightarrow \varphi_n$ is a *rightmost derivation* of φ, denoted by $S \underset{rm}{\overset{*}{\Longrightarrow}} \varphi$, if the rightmost nonterminal in φ_i is replaced, i.e., $\varphi_i = \sigma A u$, $\varphi_{i+1} = \sigma\alpha u$ with $u \in V_T^*$ and $A \to \alpha \in P$. A sentential form that occurs in a leftmost derivation (rightmost derivation) is called *left sentential form (right sentential form).*

To each parse tree for S there exists exactly one leftmost derivation and exactly one rightmost derivation. Thus, there is exactly one leftmost and one rightmost derivation for each unambiguous word in a language.

Example 3.2.8

The word $\mathsf{Id} * \mathsf{Id} + \mathsf{Id}$ has, according to grammar G_1, the leftmost derivations

$$E \underset{lm}{\Longrightarrow} E + E \underset{lm}{\Longrightarrow} E * E + E \underset{lm}{\Longrightarrow} \mathsf{Id} * E + E \underset{lm}{\Longrightarrow} \mathsf{Id} * \mathsf{Id} + E \underset{lm}{\Longrightarrow}$$
$$\mathsf{Id} * \mathsf{Id} + \mathsf{Id} \qquad \text{and}$$
$$E \underset{lm}{\Longrightarrow} E * E \underset{lm}{\Longrightarrow} \mathsf{Id} * E \underset{lm}{\Longrightarrow} \mathsf{Id} * E + E \underset{lm}{\Longrightarrow} \mathsf{Id} * \mathsf{Id} + E \underset{lm}{\Longrightarrow} \mathsf{Id} * \mathsf{Id} + \mathsf{Id}.$$

It has the rightmost derivations

$$E \underset{rm}{\Longrightarrow} E+E \underset{rm}{\Longrightarrow} E+\mathsf{Id} \underset{rm}{\Longrightarrow} E*E+\mathsf{Id} \underset{rm}{\Longrightarrow} E*\mathsf{Id}+\mathsf{Id} \underset{rm}{\Longrightarrow} \mathsf{Id}*\mathsf{Id}+\mathsf{Id}$$

and

$$E \underset{rm}{\Longrightarrow} E*E \underset{rm}{\Longrightarrow} E*E+E \underset{rm}{\Longrightarrow} E*E+\mathsf{Id} \underset{rm}{\Longrightarrow} E*\mathsf{Id}+\mathsf{Id} \underset{rm}{\Longrightarrow} \mathsf{Id}*\mathsf{Id}+\mathsf{Id}.$$

The word $\mathsf{Id}+\mathsf{Id}$ has, according to G_1, only one leftmost derivation, namely

$$E \underset{lm}{\Longrightarrow} E+E \underset{lm}{\Longrightarrow} \mathsf{Id}+E \underset{lm}{\Longrightarrow} \mathsf{Id}+\mathsf{Id}$$

and one rightmost derivation, namely

$$E \underset{rm}{\Longrightarrow} E+E \underset{rm}{\Longrightarrow} E+\mathsf{Id} \underset{rm}{\Longrightarrow} \mathsf{Id}+\mathsf{Id}.$$

□

In an unambiguous grammar, the leftmost and the rightmost derivation of a word consist of the same productions. The only difference is the order of their applications. The questions is, can one find sentential forms in both derivations that correspond to each other in the following way: in both derivations will, in the next step, the same occurrence of a nonterminal be replaced? The following lemma establishes such a relation.

Lemma 3.2.1

1. If $S \overset{*}{\underset{lm}{\Longrightarrow}} uA\varphi$ holds, then there exists ψ, with $\psi \overset{*}{\Longrightarrow} u$, such that for all v with $\varphi \overset{*}{\Longrightarrow} v$ holds $S \underset{rm}{\Longrightarrow} \psi Av$.
2. If $S \overset{*}{\underset{rm}{\Longrightarrow}} \psi Av$ holds, then there exists a φ with $\varphi \overset{*}{\Longrightarrow} v$, such that for all u with $\psi \overset{*}{\Longrightarrow} u$ holds $S \overset{*}{\underset{lm}{\Longrightarrow}} uA\varphi$. □

Figure 3.3 clarifies the relation between φ and v on one side and ψ and u on the other side.

Context-free grammars that describe programming languages should be unambiguous. If this is the case, then there exist exactly one parse tree, one leftmost and one rightmost derivation for each syntactically correct program.

3.2.2 Productivity and Reachability of Nonterminals

A CFG can have *superfluous* nonterminals and productions. Eliminating them reduces the size of the grammar, but does not change the language. We will now introduce two properties that characterize nonterminals as useful and present methods

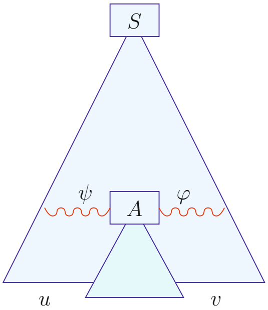


Fig. 3.3 Correspondence between leftmost and rightmost derivations

to compute the subsets of nonterminals that have these properties. All nonterminals *not* having these properties, together with all productions using such nonterminals can be removed. The resulting grammars are called then *reduced*.

The first required property of useful nonterminals is *productivity*. A nonterminal X of a CFG $G = (V_N, V_T, P, S)$ is called *productive*, if there exists a derivation $X \overset{*}{\underset{G}{\Longrightarrow}} w$ for a word $w \in V_T^*$, or equivalently, if there exists a parse tree whose root is labeled with X.

Example 3.2.9 Consider the grammar $G = (\{S', S, X, Y, Z\}, \{a, b\}, P, S')$, where P consists of the productions:

$$\begin{array}{lcl} S' & \rightarrow & S \\ S & \rightarrow & aXZ \mid Y \\ X & \rightarrow & bS \mid aYbY \\ Y & \rightarrow & ba \mid aZ \\ Z & \rightarrow & aZX \end{array}$$

Then Y is productive and therefore also X, S and S'. The nonterminal Z, on the other hand, is not productive since the only production for Z contains an occurrence of Z on its right side. □

The following two-level characterization of productivity gives rise to an algorithm for computing the set of productive nonterminals.

(1) X is *productive through production* p if and only if X is the left side of p, and if all nonterminals on the right side of p are productive.

(2) X is *productive* if X is productive through at least one of its alternatives.

In particular, X is thereby productive if there exists a production $X \rightarrow u \in P$ whose right side u has no nonterminal occurrences, that is, $u \in V_T^*$. Property (1) describes the dependence of the information for X on the information about symbols on the right side of the production for X; property (2) indicates how to combine the information obtained from the different alternatives for X.

We describe a method that computes for a CFG G the set of all productive nonterminals. The idea is to introduce for each production p a *counter* $\mathsf{count}[p]$, which counts the number of occurrences of nonterminals whose productivity is not yet known. When the counter of a production p is decreased to 0 all nonterminals on the right side must be productive. Therefore, also the left side of p is productive through p. To maintain the productions whose counter has been decreased to 0 the algorithm uses a *worklist* W.

Further, for each nonterminal X a list $\mathsf{occ}[X]$ is maintained which consists of all productions where X occurs in the right sides.

```
set ⟨nonterminal⟩  productive ← ∅;          //  result-set
int  count[production];                      //  counter for each production
list ⟨nonterminal⟩  W ← [];
list ⟨production⟩  occ[nonterminal];         //  occurrences in right sides

forall (nonterminal X)  occ[X] ← [];         //  Initialization
forall (production p)  { count[p] ← 0;
                         init(p);
                       }
...
```

The call $\mathsf{init}(p)$ of the routine $\mathsf{init}()$ for a production p, whose code we have not given, iterates over the sequence of symbols on the right side of p. At each occurrence of a nonterminal X the counter $\mathsf{count}[p]$ is incremented, and p is added to the list $\mathsf{occ}[X]$. If at the end, $\mathsf{count}[p] = 0$ still holds then $\mathsf{init}(p)$ enters production p into the list W. This concludes the initialization.

The main iteration processes the productions in W one by one. For each production p in W, the left side is productive through p and therefore productive. When, on the other hand, a nonterminal X is newly discovered as productive, the algorithm iterates through the list $\mathsf{occ}[X]$ of those productions in which X occurs. The counter $\mathsf{count}[r]$ is decremented for each production r in this list. The described method is realized by the following algorithm:

```
...
while (W ≠ [])  {
        X ← hd(W);  W ← tl(W);
        if (X ∉ productive)  {
                productive ← productive ∪ {X};
                forall ((r : A → α) ∈ occ[X])  {
                        count[r]−−;
                        if (count[r] = 0)  W ← A :: W;
                        }                               //  end of  forall
                }                                       //  end of  if
        }                                               //  end of  while
```

Let us derive the run time of this algorithm. The initialization phase essentially runs once over the grammar and does a constant amount of work for each symbol. The main iteration through the worklist enters the left side of each production at most once into the list W and so removes it also at most once from the list. At the removal of a nonterminal X from W more than a constant amount of work has to be done only when X has not yet been marked as productive. The effort for such an X is proportional to the length of the list $\mathsf{occ}[X]$. The *sum* of these lengths is bounded by the overall size of the grammar G. This means that the total effort is linear in the size of the grammar.

To show the correctness of the procedure, we ascertain that it possesses the following properties:

- If X is entered into the set productive in the jth iteration of the *while*-loop, there exists a parse tree for X of height at most $j - 1$.
- For each parse tree, the root is evenually entered into W.

The efficient algorithm just presented has relevance beyond its application in compiler construction. It can be used with small modifications to compute *least* solutions of *Boolean* systems of equations, that is, of systems of equations in which the right sides are disjunctions of arbitrary conjunctions of unknowns. In our example, the conjunctions stem from the right sides while a disjunction represents the existence of different alternatives for a nonterminal.

The second property of a useful nonterminal is its *reachability*. We call a nonterminal X *reachable* in a CFG $G = (V_N, V_T, P, S)$, if there exists a derivation $S \overset{*}{\underset{G}{\Longrightarrow}} \alpha X \beta$.

Example 3.2.10 Consider the grammar $G = (\{S, U, V, X, Y, Z\}, \{a, b, c, d\}, P, S)$, where P consists of the following productions:

$$\begin{array}{lllcllll} S & \rightarrow & Y & \qquad & X & \rightarrow & c \\ Y & \rightarrow & YZ \mid Ya \mid b & & V & \rightarrow & Vd \mid d \\ U & \rightarrow & V & & Z & \rightarrow & ZX \end{array}$$

The nonterminals S, Y, Z, and X are reachable, while U and V are not. □

Reachability can also be characterized by a two-level definition:

(1) If a nonterminal X is reachable and $X \rightarrow \alpha \in P$, then each nonterminal occurring in the right side α is reachable through this occurrence.

(2) A nonterminal is reachable if it is reachable through at least one of its occurrences.

(3) The start symbol S is always reachable.

Let $\mathsf{rhs}[X]$ for a nonterminal X be the set of all nonterminals that occur in the right side of productions with left side X. These sets can be computed in linear time.

The set *reachable* of reachable nonterminals of a grammar can be computed by:

```
set ⟨nonterminal⟩  reachable ← ∅;
list ⟨nonterminal⟩  W ← S :: [];
nonterminal Y;
while (W ≠ []) {
    X ← hd(W);  W ← tl(W);
    if (X ∉ reachable) {
        reachable ← reachable ∪ {X};
        forall (Y ∈ rhs[X]) W ← W ∪ {Y};
        }
```

To reduce a grammar G, first all nonproductive nonterminals are removed from the grammar together with all productions in which they occur. Only in a second step are the unreachable nonterminals eliminated, again together with the productions in which they occur. This second step is, therefore, based on the assumption that all remaining nonterminals are productive.

Example 3.2.11 Let us consider again the grammar of Example 3.2.9 with the productions

$$\begin{array}{lcl} S' & \to & S \\ S & \to & aXZ \mid Y \\ X & \to & bS \mid aYbY \\ Y & \to & ba \mid aZ \\ Z & \to & aZX \end{array}$$

The set of productive nonterminals is $\{S', S, X, Y\}$, while Z is not productive. To reduce the grammar, a first step removes all productions in which Z occurs. The resulting set is P_1:

$$\begin{array}{lcl} S' & \to & S \\ S & \to & Y \\ X & \to & bS \mid aYbY \\ Y & \to & ba \end{array}$$

Although X was reachable according to the original set of productions X is no longer reachable after the first step. The set of reachable nonterminals is $V'_N = \{S', S, Y\}$. By removing all productions whose left side is no longer reachable the following set is obtained:

$$\begin{array}{lcl} S' & \to & S \\ S & \to & Y \\ Y & \to & ba \end{array}$$

□

We assume in the following that grammars are always reduced in this way.

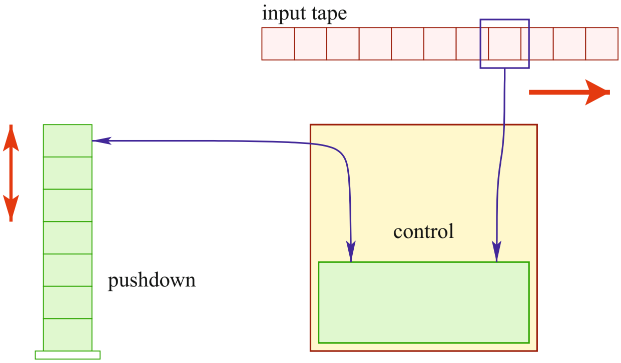


Fig. 3.4 Schematic representation of a PDA

3.2.3 Pushdown Automata

This section introduces an automata model corresponding to CFGs, namely, pushdown automata (PDA for short). Our goal is to realize a compiler component that performs syntax analysis according to a given CFG. Section 3.2.4 describes such a method. The PDA constructed for a CFG, however, has one problem: it is nondeterministic for most grammars. In Sects. 3.3 and 3.4 we describe how for appropriate subclasses of CFGs the given PDA can be modified to obtain deterministic parsers.

In contrast to the finite automata of the preceding chapter, a PDA has an unlimited storage capacity. It has a (conceptually) unbounded data structure, the *pushdown*, which works according to a *last-in, first-out* principle. Figure 3.4 shows a schematic picture of a PDA. The reading head is only allowed to move from left to right, as was the case with finite automata. In contrast to finite automata, transitions of the PDA not only depend on the actual state and the next input symbol, but also on some topmost section of the pushdown. A transition may change this upper section of the pushdown, and it may consume the next input symbol by moving the reading head one place to the right.

Formally, a *pushdown automaton* (PDA for short) is a tuple $P = (Q, V_T, \Delta, q_0, F)$, where

- Q is a finite set of *states*,
- V_T is the *input alphabet*,
- $q_0 \in Q$ is the *initial state* and
- $F \subseteq Q$ is the set of *final states*, and
- Δ is a finite subset of $Q^+ \times V_T \times Q^*$, the *transition relation*. The transition relation Δ can be seen as a finite partial function Δ from $Q^+ \times V_T$ into the finites subsets of Q^*.

Our definition of a PDA is somewhat unusual as it does not distinguish between the states of the automaton and its pushdown symbols. It uses the same alphabet for both. In this way, the topmost pushdown symbol is interpreted as the *actual* *state*. The transition relation describes the possible computation steps of the PDA. It lists finitely many transitions. Executing the transition (γ, x, γ') replaces the upper section $\gamma \in Q^+$ of the pushdown by the new sequence $\gamma' \in Q^*$ of states and reads $x \in V_T \cup \{\varepsilon\}$ in the input. The replaced section of the pushdown has at least the length 1. A transition that does not inspect the next input symbol is called an *ε-transition*.

Similarly to the case for finite automata, we introduce the notion of a *configuration* for PDAs. A configuration encompasses all components that may influence the future behavior of the automaton. With our kind of PDA these are the contents of the pushdown and the remaining input. Formally, a *configuration* of the PDA P is a pair $(\gamma, w) \in Q^+ \times V_T^*$. In the linear representation the topmost position of the pushdown is always at the right end of γ, while the next input symbol is situated at the left end of w. A *transition* of P is represented through the binary relation $\underset{P}{\vdash}$ between configurations. This relation is defined by:

$$(\gamma, w) \underset{P}{\vdash} (\gamma', w'), \text{if} \quad \gamma = \alpha\beta,\ \gamma' = \alpha\beta', \quad w = xw' \quad \text{und} \quad (\beta, x, \beta') \in \Delta$$

for a suitable $\alpha \in Q^*$. As was the case with finite automata, a *computation* is a sequence of configurations, where a transition exists between each two consecutive members. We denote them by $C \overset{n}{\underset{P}{\vdash}} C'$ if there exist configurations $C_1, \ldots, C_{n+1}$ such that $C_1 = C$, $C_{n+1} = C'$ and $C_i \underset{P}{\vdash} C_{i+1}$ for $1 \leq i \leq n$ holds. The relations $\overset{+}{\underset{P}{\vdash}}$ and $\overset{*}{\underset{P}{\vdash}}$ are the transitive and the reflexive and transitive closure of $\underset{P}{\vdash}$, resp. We have:

$$\overset{+}{\underset{P}{\vdash}} = \bigcup_{n \geq 1} \overset{n}{\underset{P}{\vdash}} \quad \text{and} \quad \overset{*}{\underset{P}{\vdash}} = \bigcup_{n \geq 0} \overset{n}{\underset{P}{\vdash}}$$

A configuration (q_0, w) for an input word $w \in V_T^*$ is called an *initial configuration*, (q, ε), for $q \in F$, a *final configuration* of the PDA P. A word $w \in V_T^*$ is *accepted* by a PDA P if $(q_0, w) \overset{*}{\underset{P}{\vdash}} (q, \varepsilon)$ holds for some $q \in F$. The *language* $L(P)$ of the PDA P is the set of words accepted by P:

$$L(P) = \{w \in V_T^* \mid \exists f \in F : (q_0, w) \overset{*}{\underset{P}{\vdash}} (f, \varepsilon)\}$$

This means that a word w is accepted by a PDA if there exists at least one computation that goes from the initial configuration (q_0, w) to a final configuration. Such computations are called *accepting*. Several accepting computations may exist for the same word, as well as several computations that can only read a prefix of a word w or that can read w, but do not reach a final configuration.

In practice, accepting computations should not be found by trial and error. Therefore, *deterministic* PDAs are of particular importance.

A PDA P is called *deterministic* if the transition relation Δ has the following property:

(D) If $(\gamma_1, x, \gamma_2), (\gamma_1', x', \gamma_2')$ are two different transitions in Δ and γ_1' is a suffix of γ_1 then x and x' are in Σ and are different from each other, that is, $x \neq \varepsilon \neq x'$ and $x \neq x'$.

If the transition relation has the property (D) there exists at most one transition out of each configuration.

3.2.4 The Item-Pushdown Automaton to a Context-Free Grammar

In this section, we meet a method that constructs for each CFG a PDA that accepts the language defined by the grammar. This automaton is nondeterministic and therefore not in itself useful for a practical application. However, we can derive the LL-parsers of Sect. 3.3, as well as the LR-parsers of Sect. 3.4 by appropriate design decisions.

The notion of context-free *item* is crucial for the construction. Let $G = (V_N, V_T, P, S)$ be a CFG. A *context-free item* of G is a triple (A, α, β) with $A \to \alpha\beta \in P$. This triple is, more intuitively, written as $[A \to \alpha.\beta]$. The item $[A \to \alpha.\beta]$ describes the situation that in an attempt to derive a word w from A a prefix of w has already been derived from α. α is therefore called the *history* of the item.

An item $[A \to \alpha.\beta]$ with $\beta = \varepsilon$ is called *complete*. The set of all context-free items of G is denoted by It_G. If ρ is the sequence of items:

$$\rho = [A_1 \to \alpha_1.\beta_1][A_2 \to \alpha_2.\beta_2]\dots[A_n \to \alpha_n.\beta_n]$$

then $\mathsf{hist}(\rho)$ denotes the concatenation of the histories of the items of ρ, i.e.,

$$\mathsf{hist}(\rho) = \alpha_1\alpha_2\dots\alpha_n.$$

We now describe how to construct the *item-pushdown automaton* (IPDA) to a CFG $G = (V_N, V_T, P, S)$. The items of the grammar act as its states and, therefore, also as pushdown symbols. The actual state is the item whose right side the automaton is just processing. Below this state in the pushdown are the items, where processing of their right sides has been begun, but not yet been finished.

Before we construct the IPDA P_G to a grammar G, we extend the grammar G in such a way that termination of the IPDA can be recognized by looking at the actual state alone. Is S the start symbol of the grammar, candidates for final states of the IPDA are all complete items $[S \to \alpha.]$ of the grammar. If S also occurs on the right side of a production such complete items can occur on the pushdown. Then the computation still may continue after such an item has been encountered. We, therefore, extend the grammar G by a new start symbol S', which does not occur in any right side. For S' we add the productions $S' \to S$ to the set of productions of G. As initial state of the IPDA for the extended grammar we choose the item $[S' \to .S]$, and as single final state the complete item $[S' \to S.]$. The *item-pushdown automaton* to the grammar G (IPDA for short) is the PDA

$$P_G = (\mathsf{It}_G, V_T, \Delta, [S' \to .S], \{[S' \to S.]\})$$

where the transition relation Δ has three types of transitions:

$$\begin{array}{lll}
(E) & \Delta([X \to \beta.Y\gamma], \varepsilon) & = \{[X \to \beta.Y\gamma][Y \to .\alpha] \mid Y \to \alpha \in P\} \\
(S) & \Delta([X \to \beta.a\gamma], a) & = \{[X \to \beta a.\gamma]\} \\
(R) & \Delta([X \to \beta.Y\gamma][Y \to \alpha.], \varepsilon) & = \{[X \to \beta Y.\gamma]\}.
\end{array}$$

Transitions according to (E) are called *expanding transitions*, those according to (S) *shifting transitions*, and those according to (R) *reducing transitions*. Each sequence of items that occurs as pushdown in the computation of an IPDA satisfies the following invariant (I):

(I) If $([S' \to .S], uv) \vdash_{P_G}^{*} (\rho, v)$ then $\mathsf{hist}(\rho) \overset{*}{\underset{G}{\Longrightarrow}} u$.

This invariant is an essential part of the proof that the IPDA P_G only accepts words of G, that is, that $L(P_G) \subseteq L(G)$ holds. We now explain the way the automaton P_G works and at the same time give a proof by induction over the length of computations that the invariant (I) holds for each configuration reachable from an initial configuration. Let us first consider the initial configuration for the input w. The initial configuration is $([S' \to .S], w)$. The word $u = \varepsilon$ has already been read, $\mathsf{hist}([S' \to .S]) = \varepsilon$, and $\varepsilon \overset{*}{\Longrightarrow} \varepsilon$ holds. Therefore, the invariant holds in this configuration.

Let us now consider derivations that consist of at least one transition. Let us first assume that the last transition was an expanding transition. Before this transition, a configuration $(\rho[X \to \beta.Y\gamma], v)$ was reached from the initial configuration $([S' \to .S], uv)$. This configuration satisfies the invariant (I) by the induction hypothesis, i.e., $\mathsf{hist}(\rho)\beta \overset{*}{\Longrightarrow} u$ holds. The item $[X \to \beta.Y\gamma]$ as actual state suggests to derive a prefix v from Y. To do this, the automaton should nondeterministically select one of the alternatives for Y. This is described by the transitions according to (E). All the successor configurations $(\rho[X \to \beta.Y\gamma][Y \to .\alpha], v)$ for $Y \to \alpha \in P$ also satisfy the invariant (I) because

$$\mathsf{hist}(\rho[X \to \beta.Y\gamma][Y \to .\alpha]) = \mathsf{hist}(\rho)\beta \overset{*}{\Longrightarrow} u\,.$$

As our next case, we assume that the last transition was a shifting transition. Before this transition, a configuration $(\rho[X \to \beta.a\gamma], av)$ has been reached from the initial configuration $([S' \to .S], uav)$. This configuration again satisfies the invariant (I) by the induction hypothesis, that is, $\mathsf{hist}(\rho)\beta \overset{*}{\Longrightarrow} u$ holds. The successor configuration $(\rho[X \to \beta a.\gamma], v)$ also satisfies the invariant (I) because

$$\mathsf{hist}(\rho[X \to \beta a.\gamma]) = \mathsf{hist}(\rho)\beta a \overset{*}{\Longrightarrow} ua$$

For the final case, let us assume that the last transition was a reducing transition. Before this transition, a configuration $(\rho[X \to \beta.Y\gamma][Y \to \alpha.], v)$ has been reached from the initial configuration $([S' \to .S], uv)$. This configuration satisfies the invariant (I) according to the induction hypothesis, that is, $\mathsf{hist}(\rho)\beta\alpha \overset{*}{\underset{G}{\Longrightarrow}} u$ holds. The actual state is the complete item $[Y \to \alpha.]$. It is the result of a computation that started with the item $[Y \to .\alpha]$, when $[X \to \beta.Y\gamma]$ was the actual state and the alternative $Y \to \alpha$ for Y was selected. This alternative has been successfully processed. The successor configuration $(\rho[X \to \beta Y.\gamma], v)$ also satisfies the invariant (I) because $\mathsf{hist}(\rho)\beta\alpha \underset{G}{\overset{*}{\Longrightarrow}} u$ implies that also $\mathsf{hist}(\rho)\beta Y \underset{G}{\overset{*}{\Longrightarrow}} u$. □

In summary, the following theorem holds:

Theorem 3.2.1 For each CFG G, $L(P_G) = L(G)$.

Proof Let us assume $w \in L(P_G)$. We then have

$$([S' \to .S], w) \vdash^*_{P_G} ([S' \to S.], \varepsilon)\ .$$

Because of the invariant (I), which we have already proved, it follows that

$$S = \mathsf{hist}([S' \to S.]) \underset{G}{\overset{*}{\Longrightarrow}} w$$

Therefore $w \in L(G)$. For the other direction, we assume $w \in L(G)$. We then have $S \underset{G}{\overset{*}{\Longrightarrow}} w$. To prove

$$([S' \to .S], w) \vdash^*_{P_G} ([S' \to S.], \varepsilon)$$

we show a more general statement, namely that for each derivation $A \underset{G}{\Longrightarrow} \alpha \underset{G}{\overset{*}{\Longrightarrow}} w$ with $A \in V_N$,

$$(\rho[A \to .\alpha], wv) \vdash^*_{P_G} (\rho[A \to \alpha.], v)$$

for arbitrary $\rho \in \mathsf{It}^*_G$ and arbitrary $v \in V_T^*$. This general claim can be proved by induction over the length of the derivation $A \underset{G}{\Longrightarrow} \alpha \underset{G}{\overset{*}{\Longrightarrow}} w$. □

Example 3.2.12 Let $G' = (\{S, E, T, F\}, \{+, *, (,), \mathsf{Id}\}, P', S)$ be the extension of grammar G_0 by the new start symbol S. The set of productions P' is given by

$$\begin{array}{lcl} S & \to & E \\ E & \to & E + T \mid T \\ T & \to & T * F \mid F \\ F & \to & (E) \mid \mathsf{Id} \end{array}$$

The transition relation Δ of P_{G_0} is presented in Table 3.1. Table 3.2 shows an accepting computation of P_{G_0} for the word $\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$. □

Table 3.1 Tabular representation of the transition relation of Example 3.2.12. The *middle column* shows the consumed input

Top of the pushdown	Input	New top of the pushdown
$[S \to .E]$	ε	$[S \to .E][E \to .E + T]$
$[S \to .E]$	ε	$[S \to .E][E \to .T]$
$[E \to .E + T]$	ε	$[E \to .E + T][E \to .E + T]$
$[E \to .E + T]$	ε	$[E \to .E + T][E \to .T]$
$[F \to (.E)]$	ε	$[F \to (.E)][E \to .E + T]$
$[F \to (.E)]$	ε	$[F \to (.E)][E \to .T]$
$[E \to .T]$	ε	$[E \to .T][T \to .T * F]$
$[E \to .T]$	ε	$[E \to .T][T \to .F]$
$[T \to .T * F]$	ε	$[T \to .T * F][T \to .T * F]$
$[T \to .T * F]$	ε	$[T \to .T * F][T \to .F]$
$[E \to E + .T]$	ε	$[E \to E + .T][T \to .T * F]$
$[E \to E + .T]$	ε	$[E \to E + .T][T \to .F]$
$[T \to .F]$	ε	$[T \to .F][F \to .(E)]$
$[T \to .F]$	ε	$[T \to .F][F \to .\mathsf{Id}]$
$[T \to T * .F]$	ε	$[T \to T * .F][F \to .(E)]$
$[T \to T * .F]$	ε	$[T \to T * .F][F \to .\mathsf{Id}]$
$[F \to .(E)]$	(	$[F \to (.E)]$
$[F \to .\mathsf{Id}]$	Id	$[F \to \mathsf{Id}.]$
$[F \to (E.)]$	)	$[E \to (E).]$
$[E \to E. + T]$	$+$	$[E \to E + .T]$
$[T \to T. * F]$	$*$	$[T \to T * .F]$
$[T \to .F][F \to \mathsf{Id}.]$	ε	$[T \to F.]$
$[T \to T * .F][F \to \mathsf{Id}.]$	ε	$[T \to T * F.]$
$[T \to .F][F \to (E).]$	ε	$[T \to F.]$
$[T \to T * .F][F \to (E).]$	ε	$[T \to T * F.]$
$[T \to .T * F][T \to F.]$	ε	$[T \to T. * F]$
$[E \to .T][T \to F.]$	ε	$[E \to T.]$
$[E \to E + .T][T \to F.]$	ε	$[E \to E + T.]$
$[E \to E + .T][T \to T * F.]$	ε	$[E \to E + T.]$
$[T \to .T * F][T \to T * F.]$	ε	$[T \to T. * F]$
$[E \to .T][T \to T * F.]$	ε	$[E \to T.]$
$[F \to (.E)][E \to T.]$	ε	$[F \to (E.)]$
$[F \to (.E)][E \to E + T.]$	ε	$[F \to (E.)]$
$[E \to .E + T][E \to T.]$	ε	$[E \to E. + T]$
$[E \to .E + T][E \to E + T.]$	ε	$[E \to E. + T]$
$[S \to .E][E \to T.]$	ε	$[S \to E.]$
$[S \to .E][E \to E + T.]$	ε	$[S \to E.]$

Table 3.2 The accepting computation of P_G for the word Id + Id ∗ Id

Pushdown	Remaining input
$[S \to .E]$	Id + Id ∗ Id
$[S \to .E][E \to .E + T]$	Id + Id ∗ Id
$[S \to .E][E \to .E + T][E \to .T]$	Id + Id ∗ Id
$[S \to .E][E \to .E + T][E \to .T][T \to .F]$	Id + Id ∗ Id
$[S \to .E][E \to .E + T][E \to .T][T \to .F][F \to .\mathsf{Id}]$	Id + Id ∗ Id
$[S \to .E][E \to .E + T][E \to .T][T \to .F][F \to \mathsf{Id}.]$	+Id ∗ Id
$[S \to .E][E \to .E + T][E \to .T][T \to F.]$	+Id ∗ Id
$[S \to .E][E \to .E + T][E \to T.]$	+Id ∗ Id
$[S \to .E][E \to E. + T]$	+Id ∗ Id
$[S \to .E][E \to E + .T]$	Id ∗ Id
$[S \to .E][E \to E + .T][T \to .T * F]$	Id ∗ Id
$[S \to .E][E \to E + .T][T \to .T * F][T \to .F]$	Id ∗ Id
$[S \to .E][E \to E + .T][T \to .T * F][T \to .F][F \to .\mathsf{Id}]$	Id ∗ Id
$[S \to .E][E \to E + .T][T \to .T * F][T \to .F][F \to \mathsf{Id}.]$	∗Id
$[S \to .E][E \to E + .T][T \to .T * F][T \to F.]$	∗Id
$[S \to .E][E \to E + .T][T \to T. * F]$	∗Id
$[S \to .E][E \to E + .T][T \to T * .F]$	Id
$[S \to .E][E \to E + .T][T \to T * .F][F \to .\mathsf{Id}]$	Id
$[S \to .E][E \to E + .T][T \to T * .F][F \to \mathsf{Id}.]$	
$[S \to .E][E \to E + .T][T \to T * F.]$	
$[S \to .E][E \to E + T.]$	
$[S \to E.]$	

Pushdown Automata with Output

PDAs as such are only acceptors; that is, they decide whether or not an input string is a word of the language. To use a PDA for syntactic analysis in a compiler needs more than a yes/no answer. The automaton should output the syntactic structure of every accepted input word. This structure can be returned as the parse tree directly or, equivalently, as the sequence of productions as they were applied in a leftmost or rightmost derivation. PDAs therefore are extended to produce output.

A PDA *with output* is a tuple $P = (Q, V_T, O, \Delta, q_0, F)$, where Q, V_T, q_0, F are the same as with an ordinary PDA. the set O is a finite output alphabet, and Δ is now a finite relation between $Q^+ \times (V_T \cup \{\varepsilon\})$ and $Q^* \times (O \cup \{\varepsilon\})$. A *configuration* consists of the actual pushdown, the remaining input, and the already produced output. It is an element of $Q^+ \times V_T^* \times O^*$.

At each transition, the automaton can output one symbol from O (or ϵ). If a PDA with output is used as a parser, its output alphabet consists of the productions of the CFG or their numbers.

The IPDA can be extended to return the syntactic structure in essentially two different ways. It can output the applied production whenever it performs an expansion. In this case, the overall output of an accepting computation is a leftmost derivation. A PDA with this output discipline is called a *left-parser*.

Instead at expansion, the IPDA can output the applied production also at each reduction. In this case, it delivers a rightmost derivation, but in reversed order. A PDA using such an output discipline is called a *right-parser*.

Deterministic Parsers

By Theorem 3.2.1, the IPDA P_G to a CFG G accepts the grammar's language $L(G)$. The nondeterministic way of working of the IPDA, though, is not suited for the practical application in a compiler. The source of nondeterminism lies in the transitions of type (E): the IPDA must choose between several alternatives for a nonterminal at expanding transitions. With a unambiguous grammar at most one is the correct choice to derive a prefix of the remaining input, while the other alternatives lead sooner or later to dead ends. The IPDA still can only *guess* the right alternative.

In Sects. 3.3 and 3.4, we describe two different ways to replace guessing. The LL-parsers of Sect. 3.3 deterministically choose one alternative for the actual nonterminal using a bounded look-ahead into the remaining input. For grammars of class $LL(k)$ a corresponding parser can deterministically select one (E)-transition based on the already consumed input, the nonterminal to be expanded and the next k input symbols. LL-parsers are left-parsers.

LR-parsers work differently. They *delay* the decision, which LL-parsers take at expansion, until reduction. During the analysis, they try to pursue all possibilities in parallel that may lead to a reverse rightmost derivation for the input word. A decision has to be taken only when one of these possibilities signals a reduction. This decision concerns whether to continue shifting or to reduce, and in the latter case, to reduce by which production. The basis for this decision is again the actual pushdown and a bounded look-ahead into the remaining input. LR-parsers signal reductions, and therefore are right-parsers. There does not exist an LR-parser for each CFG, but only for grammars of the class $LR(k)$, where k again is the number of necessary look-ahead symbols.

3.2.5 first- and follow-Sets

Let us consider the IPDA P_G to a CFG G when it performs an expansion, that is, at an (E)-transition. Just before such a transition, P_G is in a state of the form $[X \to \alpha.Y\beta]$. In this state, the IPDA P_G must select nondeterministically one of the alternatives $Y \to \alpha_1 \mid \ldots \mid \alpha_n$ for the nonterminal Y. This selection can be guided by knowledge about the sets of words that can be produced from the different alternatives. If the beginning of the remaining input only matches words in the set of words derivable from one alternative $Y \to \alpha_i$, this alternative is to be selected. If some of the alternatives may produce short words or even ε, the set of words that may follow Y must be taken into account.

It is wise only to consider *prefixes* of words up to a given length k since the set of words that can be derived from an alternative α_i is in general infinite. Sets of prefixes up to a given length, in contrast, are always finite. A generated parser bases its decisions on a comparison of prefixes of the remaining input of length k with the elements in these precomputed sets. For this purpose, we introduce the two functions first_k and follow_k, which associate these sets with words over $(V_N \cup V_T)^*$ and V_N, respectively. For an alphabet V_T, we write $V_T^{\leq k}$ for $\bigcup\limits_{i=0}^{k} V_T^i$ and $V_{T,\#}^{\leq k}$ for $V_T^{\leq k} \cup (V_T^{\leq k-1}\ \{\#\})$, where # is a symbol that is not contained in V_T. Like the EOF symbol, eof, it marks the end of a word. Let $w = a_1 \ldots a_n$ be a word $a_i \in V_T$ for $(1 \leq i \leq n)$, $n \geq 0$. For $k \geq 0$, we define the k-*prefix* of w by

$$w|_k = \begin{cases} a_1 \ldots a_n & \text{if} \quad n \leq k \\ a_1 \ldots a_k & \text{otherwise} \end{cases}$$

Further, we introduce the operator $\odot_k : V_T \times V_T \to V_T^{\leq k}$ defined by

$$u \odot_k v = (uv)|_k$$

This operator is called k-*concatenation*. We extend both operators to sets of words. For sets $L \subseteq V_T^*$ and $L_1, L_2 \subseteq V^{\leq k}$ we define

$$L|_k = \{w|_k \mid w \in L\} \quad \text{and} \quad L_1 \odot_k L_2 = \{x \odot_k y \mid x \in L_1, y \in L_2\}\,.$$

Let $G = (V_N, V_T, P, S)$ be a CFG. For $k \geq 1$, we define the function $\mathsf{first}_k : (V_N \cup V_T)^* \to 2^{V_T^{\leq k}}$ that returns for each word α the set of all prefixes of length k of terminal words that can be derived from α.

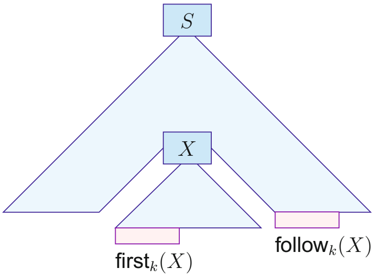
$$\mathsf{first}_k(\alpha) = \{u|_k \mid \alpha \overset{*}{\Longrightarrow} u\}$$

Correspondingly, the function $\mathsf{follow}_k : V_N \to 2^{V_{T,\#}^{\leq k}}$ returns for a nonterminal X the set of terminal words of length at most k that can directly follow a nonterminal X in a sentential form:

$$\mathsf{follow}_k(X) = \{w \in V_T^* \mid S \overset{*}{\Longrightarrow} \beta X \gamma \quad \text{and} \quad w \in \mathsf{first}_k(\gamma\#)\}$$

The set $\mathsf{first}_k(X)$ consists of the k-prefixes of leaf words of all trees for X, $\mathsf{follow}_k(X)$ of the k-prefixes of the second part of leaf words of all upper tree fragments for X (see Fig. 3.5). The following lemma describes some properties of k-concatenation and the function first_k.

Fig. 3.5 first_k and follow_k in a parse tree



Lemma 3.2.2 Let $k \geq 1$, and let $L_1, L_2, L_3 \subseteq V^{\leq k}$ be given. We then have:

$$\begin{array}{ll}
(a) & L_1 \odot_k (L_2 \odot_k L_3) = (L_1 \odot_k L_2) \odot_k L_3 \\
(b) & L_1 \odot_k \{\varepsilon\} = \{\varepsilon\} \odot_k L_1 = L_1|_k \\
(c) & L_1 \odot_k L_2 = \emptyset \quad \text{iff} \quad L_1 = \emptyset \vee L_2 = \emptyset \\
(d) & \varepsilon \in L_1 \odot_k L_2 \quad \text{iff} \quad \varepsilon \in L_1 \wedge \varepsilon \in L_2 \\
(e) & (L_1 L_2)|_k = L_1|_k \odot_k L_2|_k \\
(f) & \mathsf{first}_k(X_1 \ldots X_n) = \mathsf{first}_k(X_1) \odot_k \ldots \odot_k \mathsf{first}_k(X_n) \\
 & \qquad\qquad \text{for } X_1, \ldots, X_n \in (V_T \cup V_N)
\end{array}$$

The proofs for (b), (c), (d), and (e) are trivial. (a) is obtained by case distinctions over the length of words $x \in L_1$, $y \in L_2$, $z \in L_3$. The proof for (f) uses (e) and the observation that $X_1 \ldots X_n \overset{*}{\Longrightarrow} u$ holds if and only if $u = u_1 \ldots u_n$ for suitable words u_i with $X_i \overset{*}{\Longrightarrow} u_i$.

Because of property (f), the computation of the set $\mathsf{first}_k(\alpha)$ can be reduced to the computation of the set $\mathsf{first}_k(X)$ for single symbols $X \in V_T \cup V_N$. Since $\mathsf{first}_k(a) = \{a\}$ holds for $a \in V_T$ it suffices to determine the sets $\mathsf{first}_k(X)$ for nonterminals X. A word $w \in V_T^{\leq k}$ is in $\mathsf{first}_k(X)$ if and ony if w is contained in the set $\mathsf{first}_k(\alpha)$ for one of the productions $X \to \alpha \in P$.

Due to property (f) of Lemma 3.2.2, the first_k-sets satisfy the equation system (fi):

$$\mathsf{first}_k(X) = \bigcup \{\mathsf{first}_k(X_1) \odot_k \ldots \odot_k \mathsf{first}_k(X_n) \mid X \to X_1 \ldots X_n \in P\}, \atop X_i \in V_N \tag{fi}$$

Example 3.2.13 Let G_2 be the CFG with the productions:

0 :	S	$\to$	E	3 :	E'	$\to$	$+E$	6 :	T'	$\to$	$*T$
1 :	E	$\to$	TE'	4 :	T	$\to$	FT'	7 :	F	$\to$	(E)
2 :	E'	$\to$	ε	5 :	T'	$\to$	ε	8 :	F	$\to$	Id

G_2 generates the same language of arithmetic expressions as G_0 and G_1. We obtain as system of equations for the computation of the first_1-sets:

$$\begin{array}{lcl}
\mathsf{first}_1(S) & = & \mathsf{first}_1(E) \\
\mathsf{first}_1(E) & = & \mathsf{first}_1(T) \odot_1 \mathsf{first}_1(E') \\
\mathsf{first}_1(E') & = & \{\varepsilon\} \cup \{+\} \odot_1 \mathsf{first}_1(E) \\
\mathsf{first}_1(T) & = & \mathsf{first}_1(F) \odot_1 \mathsf{first}_1(T') \\
\mathsf{first}_1(T') & = & \{\varepsilon\} \cup \{*\} \odot_1 \mathsf{first}_1(T) \\
\mathsf{first}_1(F) & = & \{\mathsf{Id}\} \cup \{(\} \odot_1 \mathsf{first}_1(E) \odot_1 \{)\}
\end{array}$$

□

The right sides of the system of equations of the first_k-sets can be represented as expressions consisting of unknowns $\mathsf{first}_k(Y), Y \in V_N$ and the set constants $\{x\}, x \in V_T \cup \{\varepsilon\}$ and built using the operators $\odot_k$ and $\cup$. Immediately the following questions arise:

- Does this system of equations always have solutions?
- If yes, which is the solution corresponding to the first_k-sets?
- How does one compute this solution?

To answer these questions we first consider general systems of equations like (fi) and look for an algorithmic approach to solve such systems: Let $\mathbf{x}_1, \ldots, \mathbf{x}_n$ be a set of unknowns,

$$\begin{array}{rcl}
\mathbf{x}_1 & = & f_1(\mathbf{x}_1, \ldots, \mathbf{x}_n) \\
\mathbf{x}_2 & = & f_2(\mathbf{x}_1, \ldots, \mathbf{x}_n) \\
 & \vdots & \\
\mathbf{x}_n & = & f_n(\mathbf{x}_1, \ldots, \mathbf{x}_n)
\end{array}$$

a system of equations to be solved over a domain $\mathbb{D}$. Each f_i on the right side denotes a function $f_i : \mathbb{D}^n \to \mathbb{D}$. A solution I^* of this system of equations associates a value $I^*(\mathbf{x}_i)$ with each unknown $\mathbf{x}_i$ such that all equations are satisfied, that is

$$I^*(\mathbf{x}_i) = f_i(I^*(\mathbf{x}_1), \ldots, I^*(\mathbf{x}_n))$$

holds for all $i = 1, \ldots, n$.

Let us assume that $\mathbb{D}$ contains a distinctive element d_0 that offers itself as start value for the calculation of a solution. A simple idea to determine a solution then consists in setting all the unknowns $\mathbf{x}_1, \ldots, \mathbf{x}_n$ to this start value d_0. Let $I^{(0)}$ be this variable binding. All right sides f_i are evaluated in this variable binding. This associates each variable $\mathbf{x}_i$ with a new value. All these new values form a new variable binding $I^{(1)}$, in which the right sides are again evaluated, and so on. Let us assume that an actual variable binding $I^{(j)}$ has been computed. The new variable binding $I^{(j+1)}$ is then determined through:

$$I^{(j+1)}(\mathbf{x}_i) = f_i(I^{(j)}(\mathbf{x}_1), \ldots, I^{(j)}(\mathbf{x}_n))$$

A sequence of variable bindings $I^{(0)}, I^{(1)}, \ldots$, results. If for a $j \geq 0$ holds that $I^{(j+1)} = I^{(j)}$, then

$$I^{(j)}(\mathbf{x}_i) = f_i(I^{(j)}(\mathbf{x}_1), \ldots, I^{(j)}(\mathbf{x}_n)) \qquad (i = 1, \ldots, n).$$

Therefore $I^{(j)} = I^*$ is a solution.

Without further assumptions it is unclear whether a j with $I^{(j+1)} = I^{(j)}$ is ever reached. In the special cases considered in this volume, we can guarantee that this procedure converges not only against some solution, but against the desired solution. This is based on the specific properties of the domains $\mathbb{D}$ that occur in our application.

- There exists a *partial order* on the domain $\mathbb{D}$ represented by the symbol $\sqsubseteq$. In the case of the first_k-sets the set $\mathbb{D}$ consists of all subsets of the finite base set $V_T^{\leq k}$ of terminal words of length at most k. The partial order over this domain is the *subset relation*.
- $\mathbb{D}$ contains a uniquely determined least element with which the iteration can start. This element is denoted as $\bot$ (bottom). In the case of the first_k-sets, this least element is the empty set.
- For each subset $Y \subseteq \mathbb{D}$, there exists a *least upper bound* $\bigsqcup Y$ wrt. to the relation $\sqsubseteq$. In the case of the first_k-sets, the least upper bound of a set of sets is the union of its sets. Partial orders with this property are called *complete lattices*.

Furthermore, all functions f_i are *monotonic*, that is, they respect the order $\sqsubseteq$ of their arguments. In the case of the first_k-sets this holds because the right sides of the equations are built from the operators union and k-concatenation, which are both monotonic and because the composition of monotonic functions is again monotonic.

If the algorithm is started with $d_0 = \bot$, it holds that $I^{(0)} \sqsubseteq I^{(1)}$. Hereby, a variable binding is less than or equal to another variable binding, if this holds for the value of each variable. The monotonicity of the functions f_i implies by induction that the algorithm produces an *ascending* sequence

$$I^{(0)} \sqsubseteq I^{(1)} \sqsubseteq I^{(2)} \sqsubseteq \ldots I^{(k)} \sqsubseteq \ldots$$

of variable bindings. If the domain $\mathbb{D}$ is finite, there exists a number j, such that $I^{(j)} = I^{(j+1)}$ holds. This means that the algorithm finds a solution. In fact, this solution is the *least* solution of the system. Such a least solution even exists if the complete lattice is not finite, and if the simple iteration does not terminate. This follows from the fixed-point theorem of Knaster-Tarski, which we treat in detail in the third volume, *Compiler Design: Analysis and Transformation*.

Example 3.2.14 Let us apply this algorithm to determine a solution of the system of equations of Example 3.2.13. Initially, all nonterminals are associated with the empty set. The following table shows the words added to the first_1-sets in the ith iteration.

	1	2	3	4	5	6	7	8
S				Id			(	
E			Id			(		
E'	ϵ		+					
T		Id			(			
T'	ϵ		$*$					
F	Id			(				

The following result is obtained:

$$\begin{array}{lcllcl} \mathsf{first}_1(S) & = & \{\mathsf{Id}, (\} & \mathsf{first}_1(T) & = & \{\mathsf{Id}, (\} \\ \mathsf{first}_1(E) & = & \{\mathsf{Id}, (\} & \mathsf{first}_1(T') & = & \{\varepsilon, *\} \\ \mathsf{first}_1(E') & = & \{\varepsilon, +\} & \mathsf{first}_1(F) & = & \{\mathsf{Id}, (\} \end{array}$$

□

It suffices to show that all right sides are monotonic and that the domain is finite to guarantee the applicability of the iterative algorithm for a given system of equations over a complete lattice. The following theorem makes sure that the *least* solution of the system of equations (fi) indeed characterizes the first_k-sets.

Theorem 3.2.2 (Correctness of the first_k-sets) Let $G = (V_N, V_T, P, S)$ be a CFG, $\mathbb{D}$ the complete lattice of the subsets of $V_T^{\leq k}$, and $I : V_N \rightarrow \mathbb{D}$ be the least solution of the system of equations (fi). We then have:

$$I(X) = \mathsf{first}_k(X) \quad \text{for all } X \in V_N$$

Proof For $i \geq 0$, let $I^{(i)}$ be the variable binding after the ith iteration of the algorithm to find solutions for (fi). By induction over i, we show that for all $i \geq 0$ $I^{(i)}(X) \subseteq \mathsf{first}_k(X)$ holds for all $X \in V_N$. Therefore, it also holds $I(X) = \bigcup_{i \geq 0}(X) \subseteq \mathsf{first}_k(X)$ for all $X \in V_N$. For the other direction it suffices to show that for each derivation $X \underset{lm}{\overset{*}{\Longrightarrow}} w$, there exists an $i \geq 0$ with $w|_k \in I^{(i)}(X)$. This claim is again shown by induction, this time by induction over the length $n \geq 1$ of the leftmost derivation. For $n = 1$ the grammar has a production $X \rightarrow w$. We then have

$$I^{(1)}(X) \supseteq \mathsf{first}_k(w) = \{w|_k\}$$

and the claim follows with $i = 1$. For $n > 1$, there exists a production $X \rightarrow u_0 X_1 u_1 \ldots X_m u_m$ with $u_0, \ldots, u_m \in V_T^*$ and $X_1, \ldots, X_m \in V_N$ and leftmost derivations $X_i \underset{lm}{\overset{*}{\Longrightarrow}} w_j$, $j = 1, \ldots, k$ that all have a length less than n, with $w = u_0 w_1 u_1 \ldots w_m u_m$. According to the induction hypothesis, for each $j \in \{1, \ldots, m\}$

there exists a i_j, such that $(w_i|_k) \in I^{(i_j)}(X_i)$ holds. Let i' be the maximum of these i_j. For $i = i' + 1$ it holds that

$$\begin{array}{lll} I^{(i)}(X) & \supseteq & \{u_0\} \odot_k I^{(i')}(X_1) \odot_k \{u_1\} \ldots \odot_k I^{(i')}(X_m) \odot_k \{u_m\} \\ & \supseteq & \{u_0\} \odot_k \{w_1|_k\} \odot_k \{u_1\} \ldots \odot_k \{w_m|_k\} \odot_k \{u_m\} \\ & \supseteq & \{w|_k\} \end{array}$$

and the claim follows. □

To compute least solutions of systems of equations, or similarly for systems of inequalities over complete lattices, is a problem that also appears in the computation of program invariants. Such program invariants are required, e.g., to guarantee the applicability of program transformations, that are to increase the efficiency of programs. Such analyses and transformations are presented in the volume *Compiler Design: Analysis and Transformation*. The global iterative approach which we have sketched, is not necessarily the best method to solve systems of equations. In the volume *Compiler Design: Analysis and Transformation* more efficient methods are presented.

Let us now consider how to compute follow_k-sets for an extended CFG G. Again, we start with an adequate recursive property. For a word $w \in V_T^k \cup V_T^{\leq k-1}\{\#\}$, $w \in \mathsf{follow}_k(X)$ holds if

(1) $X = S'$ is the start symbol of the grammar and $w = \#$ holds,
(2) or there exists a production $Y \to \alpha X \beta$ in G such that $w \in \mathsf{first}_k(\beta) \odot_k \mathsf{follow}_k(Y)$ holds.

The sets $\mathsf{follow}_k(X)$ therefore satisfy the following system of equations:

$$\begin{aligned} \mathsf{follow}_k(S') &= \{\#\} \\ \mathsf{follow}_k(X) &= \bigcup \{\mathsf{first}_k(\beta) \odot_k \mathsf{follow}_k(Y) \mid Y \to \alpha X \beta \in P\}, \\ & S' \neq X \in V_N \end{aligned} \qquad (\mathsf{fo})$$

Example 3.2.15 Let us again consider the CFG G_2 of Example 3.2.13. To calculate the follow_1-sets for the grammar G_2 we use the system of equations:

$$\begin{array}{lcl} \mathsf{follow}_1(S) & = & \{\#\} \\ \mathsf{follow}_1(E) & = & \mathsf{follow}_1(S) \;\cup\; \mathsf{follow}_1(E') \;\cup\; \{)\} \odot_1 \mathsf{follow}_1(F) \\ \mathsf{follow}_1(E') & = & \mathsf{follow}_1(E) \\ \mathsf{follow}_1(T) & = & \{\varepsilon, +\} \odot_1 \mathsf{follow}_1(E) \;\cup\; \mathsf{follow}_1(T') \\ \mathsf{follow}_1(T') & = & \mathsf{follow}_1(T) \\ \mathsf{follow}_1(F) & = & \{\varepsilon, *\} \odot_1 \mathsf{follow}_1(T) \end{array}$$

□

The system of equations (fo) must again be solved over a subset lattice. The right sides of the equations are built from constant sets and unknowns by monotonic operators. Therefore, (fo) has a solution which can be computed by global iteration. The next theorem ascertains that this algorithm indeed computes the right sets.

Theorem 3.2.3 (Correctness of the follow$_k$-sets) Let $G = (V_N, V_T, P, S')$ be an extended CFG, $\mathbb{D}$ be the complete lattice of subsets of $V_T^k \cup V_T^{\leq k-1}\{\#\}$, and $I : V_N \to \mathbb{D}$ be the least solution of the system of equations (fo). Then we have:

$$I(X) = \mathsf{follow}_k(X) \quad \text{for all } X \in V_N$$

□

The proof is simiilar to the proof of Theorem 3.2.2 and is left to the reader (Exercise 6).

Example 3.2.16 We consider the system of equations of Example 3.2.15. To compute the solution the iteration again starts with the value ∅ for each nonterminal. The words added in the subsequent iterations are shown in the following table:

	1	2	3	4	5	6	7
S	#						
E		#			)		
E'			#			)	
T			+,#,)				
T'				+, #,)			
F				$*$, +, #,)			

Altogether we obtain the following sets:

$$\begin{array}{lcllcl}
\mathsf{follow}_1(S) & = & \{\#\} & \mathsf{follow}_1(T) & = & \{+, \#,)\} \\
\mathsf{follow}_1(E) & = & \{\#,)\} & \mathsf{follow}_1(T') & = & \{+, \#,)\} \\
\mathsf{follow}_1(E') & = & \{\#,)\} & \mathsf{follow}_1(F) & = & \{*, +, \#,)\}
\end{array}$$

□

3.2.6 The Special Case first$_1$ and follow$_1$

The iterative method for the computation of least solutions of systems of equations for the first$_1$- and follow$_1$-sets is not very efficient. But even with more efficient methods, the computation of first$_k$- and follow$_1$-sets needs a large effort when k gets larger. Therefore, practical parsers mostly use look-ahead of length $k = 1$. In this case, the computation of the first- and follow-sets can be performed particularly efficiently. The following lemma is the foundation for our further treatment.

Lemma 3.2.3 Let $L_1, L_2 \subseteq V_T^{\leq 1}$ be nonempty languages. We then have:

$$L_1 \odot_1 L_2 = \begin{cases} L_1 & \text{if } L_2 \neq \emptyset \text{ and } \varepsilon \notin L_1 \\ (L_1 \backslash \{\varepsilon\}) \cup L_2 & \text{if } L_2 \neq \emptyset \text{ and } \varepsilon \in L_1 \end{cases}$$

According to our assumption, the considered grammars are always reduced. They therefore contain neither nonproductive nor unreachable nonterminals. Accordingly, for all $X \in V_N$ the sets $\mathsf{first}_1(X)$ as well as the sets $\mathsf{follow}_1(X)$ are nonempty. Together with Lemma 3.2.3, this observation allows us to simplify the equations for first_1 and follow_1 in such a way that the 1-concatenation can be (essentially) replaced with *union*. In order to eliminate the case distinction of whether ε is contained in a first_1-set or not, we proceed in two steps. In the first step, the set of nonterminals X is determined that satisfy $\varepsilon \in \mathsf{first}_1(X)$. In the second step, the *ε-free* first_1-set is determined for each nonterminal X instead of the first_1-set. For a symbol $X \in V_N \cup V_T$, The *ε-free* first_1-set is defined by

$$\begin{aligned} \mathsf{eff}(X) &= \mathsf{first}_1(X) \backslash \{\varepsilon\} \\ &= \{(w|_k) \mid X \overset{*}{\underset{G}{\Longrightarrow}} w, w \neq \varepsilon\} \end{aligned}$$

To implement the first step, it helps to exploit that for each nonterminal X

$$\varepsilon \in \mathsf{first}_1(X) \quad \text{if and only if} \quad X \overset{*}{\Longrightarrow} \varepsilon$$

In a derivation of the word ε no production can be used that contains a terminal symbol $a \in V_T$. Let G_ε be the grammar that is obtained from G by eliminating all these productions. Then it holds that $X \overset{*}{\underset{G}{\Longrightarrow}} \varepsilon$ if and only if X is productive with respect to the grammar G_ε. For the latter problem, the efficient solver for productivity of Sect. 3.2.2 can be applied.

Example 3.2.17

Consider the grammar G_2 of Example 3.2.13. The set of productions in which no terminal symbol occurs is

$$\begin{array}{lllllllll} 0: & S & \to & E & & & & & \\ 1: & E & \to & TE' & \qquad & 4: & T & \to & FT' \\ 2: & E' & \to & \varepsilon & & 5: & T' & \to & \varepsilon \end{array}$$

With respect to this set of productions only the nonterminals E' and T' are productive. These two nonterminals are thus the only ε-productive nonterminals of grammar G_2. □

Let us now turn to the second step, the computation of the ε-free first_1-sets. Consider a production of the form $X \to X_1 \dots X_m$. Its contribution to $\mathsf{eff}(X)$ can be written as

$$\bigcup \{\mathsf{eff}(X_j) \mid X_1 \dots X_{j-1} \overset{*}{\underset{G}{\Longrightarrow}} \varepsilon\}$$

Altogether, we obtain the system of equations:

$$\mathsf{eff}(X) = \bigcup \{\mathsf{eff}(Y) \mid X \to \alpha Y \beta \in P, \alpha \overset{*}{\underset{G}{\Longrightarrow}} \varepsilon\}, \quad X \in V_N \qquad (\mathsf{eff})$$

Example 3.2.18

Consider again the CFG G_2 of Example 3.2.13. The following system of equations serves to compute the ε-free first_1-sets.

$$\begin{aligned} \mathsf{eff}(S) &= \mathsf{eff}(E) & \mathsf{eff}(T) &= \mathsf{eff}(F) \\ \mathsf{eff}(E) &= \mathsf{eff}(T) & \mathsf{eff}(T') &= \emptyset \ \cup \ \{*\} \\ \mathsf{eff}(E') &= \emptyset \ \cup \ \{+\} & \mathsf{eff}(F) &= \{\mathsf{Id}\} \ \cup \ \{(\} \end{aligned}$$

All occurrences of the $\odot_1$-operator have disappeared. Instead, only constant sets, unions, and variables $\mathsf{eff}(X)$ appear on right sides. The least solution is

$$\begin{aligned} \mathsf{eff}(S) &= \{\mathsf{Id}, (\} & \mathsf{eff}(T) &= \{\mathsf{Id}, (\} \\ \mathsf{eff}(E) &= \{\mathsf{Id}, (\} & \mathsf{eff}(T') &= \{*\} \\ \mathsf{eff}(E') &= \{+\} & \mathsf{eff}(F) &= \{\mathsf{Id}, (\} \end{aligned}$$

□

Nonterminals that occur to the right of terminals do not contribute to the ε-free first_1-sets. Therefore, it is important for the correctness of the construction that all nonterminals of the grammar are productive.

The ε-free first_1-sets $\mathsf{eff}(X)$ can also be used to simplify the system of equations for the computation of the follow_1-sets. Consider a production of the form $Y \to \alpha X X_1 \dots X_m$. The contribution of the occurrence of X in the right side of Y to the set $\mathsf{follow}_1(X)$ is

$$\bigcup \{\mathsf{eff}(X_j) \mid X_1 \dots X_{j-1} \overset{*}{\underset{G}{\Longrightarrow}} \varepsilon\} \cup \{\mathsf{follow}_1(Y) \mid X_1 \dots X_m \overset{*}{\underset{G}{\Longrightarrow}} \varepsilon\}$$

If all nonterminals are not only productive, but also reachable, the equation system for the computation of the follow_1-sets can be simplified to

$$\begin{aligned} \mathsf{follow}_1(S') &= \{\#\} \\ \mathsf{follow}_1(X) &= \bigcup \{\mathsf{eff}(Y) \mid A \to \alpha X \beta Y \gamma \in P, \beta \overset{*}{\underset{G}{\Longrightarrow}} \varepsilon\} \\ &\qquad \cup \bigcup \{\mathsf{follow}_1(A) \mid A \to \alpha X \beta, \beta \overset{*}{\underset{G}{\Longrightarrow}} \varepsilon\}, \quad X \in V_N \backslash \{S'\} \end{aligned}$$

Example 3.2.19 The simplified system of equations for the computation of the follow_1-sets of the CFG G_2 of Example 3.2.13 is given by:

$$\begin{aligned}
\mathsf{follow}_1(S) &= \{\#\}\\
\mathsf{follow}_1(E) &= \mathsf{follow}_1(S) \ \cup\ \mathsf{follow}_1(E') \ \cup\ \{)\}\\
\mathsf{follow}_1(E') &= \mathsf{follow}_1(E)\\
\mathsf{follow}_1(T) &= \{+\} \cup \mathsf{follow}_1(E) \cup \mathsf{follow}_1(T')\\
\mathsf{follow}_1(T') &= \mathsf{follow}_1(T)\\
\mathsf{follow}_1(F) &= \{*\} \cup \mathsf{follow}_1(T)
\end{aligned}$$

Again we observe that all occurrences of the operators $\odot_1$ have disappeared. Only constant sets and variables $\mathsf{follow}_1(X)$ occur on the right sides of equations together with the union operator. □

The next section presents a method that solves systems of equations efficiently that are similar to the simplified systems of equations for the sets $\mathsf{eff}(X)$ and $\mathsf{follow}_1(X)$. We first describe the general method and then apply it to the computations of the first_1- and follow_1-sets.

3.2.7 Pure Union Problems

Let us assume we are given a system of equations

$$\mathbf{x}_i = e_i, \qquad i = 1, \ldots, n$$

over an arbitrary complete lattice $\mathbb{D}$ such that the right sides of the equations are expressions e_i that are built from constants in $\mathbb{D}$ and variables $\mathbf{x}_j$ by means of applications of $\sqcup$, the least upper bound operator of the complete lattice $\mathbb{D}$. The problem of computing the least solution of this system of equations is called a *pure union problem*.

The computation of the set of reachable nonterminals of a CFG is a pure union problem over the Boolean lattice $\mathbb{B} = \{\mathit{false}, \mathit{true}\}$. Also the problems to compute ε-free first_1-sets and follow_1-sets for a reduced CFG are pure union problems. In these cases, the complete lattices are 2^{V_T} and $2^{V_T \cup \{\#\}}$, respectively, both ordered by the subset relation.

Example 3.2.20 As running example we consider the subset lattice $\mathbb{D} = 2^{\{a,b,c\}}$ together with the system of equations

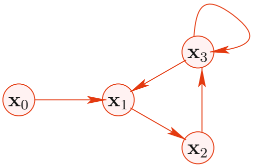


Fig. 3.6 The variable-dependence graph for the system of equations of Example 3.2.20

$$\begin{aligned}
\mathbf{x}_0 &= \{a\}\\
\mathbf{x}_1 &= \{b\} \cup \mathbf{x}_0 \cup \mathbf{x}_3\\
\mathbf{x}_2 &= \{c\} \cup \mathbf{x}_1\\
\mathbf{x}_3 &= \{c\} \cup \mathbf{x}_2 \cup \mathbf{x}_3
\end{aligned}$$

□

In order to come up with an efficient algorithm for pure union problems, we consider the variable-dependence graph of the system of equations. The nodes of this graph are the variables $\mathbf{x}_i$ of the system. An edge $(\mathbf{x}_i, \mathbf{x}_j)$ exists if and only if the variable $\mathbf{x}_i$ occurs in the right side of the variable $\mathbf{x}_j$. Figure 3.6 shows the variable-dependence graph for the system of equations of Example 3.2.20.

Let I be the least solution of the system of equations. We observe that always $I(\mathbf{x}_i) \sqsubseteq I(\mathbf{x}_j)$ must hold if there exists a path from $\mathbf{x}_i$ to $\mathbf{x}_j$ in the variable-dependence graph. In consequence, all values of variables within the same *strongly connected component* of the variable-dependence graph are equal.

We determine each variable $\mathbf{x}_i$ with the least upper bound of all constants that occur on the right side of the equation for variable $\mathbf{x}_i$. We call this value $I_0(\mathbf{x}_i)$. Then it holds for all j that

$$I(\mathbf{x}_j) = \sqcup\{I_0(\mathbf{x}_i) \mid \mathbf{x}_j \text{ is reachable from } \mathbf{x}_i\}$$

Example 3.2.21 (Continuation of Example 3.2.20) For the system of equations of Example 3.2.20 we find:

$$\begin{aligned}
I_0(\mathbf{x}_0) &= \{a\}\\
I_0(\mathbf{x}_1) &= \{b\}\\
I_0(\mathbf{x}_2) &= \{c\}\\
I_0(\mathbf{x}_3) &= \{c\}
\end{aligned}$$

It follows:

$$\begin{aligned}
I(\mathbf{x}_0) &= I_0(\mathbf{x}_0) && = \{a\}\\
I_0(\mathbf{x}_1) &= I_0(\mathbf{x}_0) \cup I_0(\mathbf{x}_1) \cup I_0(\mathbf{x}_2) \cup I_0(\mathbf{x}_3) &&= \{a, b, c\}\\
I_0(\mathbf{x}_2) &= I_0(\mathbf{x}_0) \cup I_0(\mathbf{x}_1) \cup I_0(\mathbf{x}_2) \cup I_0(\mathbf{x}_3) &&= \{a, b, c\}\\
I_0(\mathbf{x}_3) &= I_0(\mathbf{x}_0) \cup I_0(\mathbf{x}_1) \cup I_0(\mathbf{x}_2) \cup I_0(\mathbf{x}_3) &&= \{a, b, c\}
\end{aligned}$$

□

This observation suggests the following method for computing the least solution I of the system of equations. First, the strongly connected components of the variable-dependence graph are computed. This needs a linear number of steps. Then an iteration over the list of strongly connected components is performed.

One starts with a strongly connected component Q that has no edges entering from other strongly connected components. The values of all variables $\mathbf{x}_j \in Q$ are:

$$I(\mathbf{x}_j) = \bigsqcup\{I_0(\mathbf{x}_i) \mid \mathbf{x}_i \in Q\}$$

The values $I(\mathbf{x}_j)$ can be computed by the two loops:

```
$\mathbb{D}\ t \leftarrow \bot;$
**forall** $(\mathbf{x}_i \in Q)$
        $t \leftarrow t \sqcup I_0(\mathbf{x}_i);$
**forall** $(\mathbf{x}_i \in Q)$
        $I(\mathbf{x}_i) \leftarrow t;$
```

The run time of both loops is proportional to the number of elements in the strongly connected component Q. The values of the variables in Q are now propagated along the outgoing edges. Let E_Q be the set of edges $(\mathbf{x}_i, \mathbf{x}_j)$ of the variable-dependence graph with $\mathbf{x}_i \in Q$ and $\mathbf{x}_j \notin Q$, that is, the edges leaving Q. For E_Q it is set:

```
**forall** $((\mathbf{x}_i, \mathbf{x}_j) \in E_Q)$
        $I_0(\mathbf{x}_j) \leftarrow I_0(\mathbf{x}_j) \sqcup I(\mathbf{x}_i);$
```

The number of steps for the propagation is proportional to the number of edges in E_Q. Then the strongly connected component Q together with the set E_Q of outgoing edges is removed from the graph and the algorithm continues with the next strongly connected component without ingoing edges. This is repeated until no further strongly connected component is left. Altogether, the algorithm performs a linear number of operations $\sqcup$ of the complete lattice $\mathbb{D}$.

Example 3.2.22 (Continuation of Example 3.2.20) The dependence graph of the system of equations of Example 3.2.20 has the strongly connected components:

$$Q_0 = \{\mathbf{x}_0\} \quad \text{and} \quad Q_1 = \{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3\}\,.$$

For Q_0, the value $I_0(\mathbf{x}_0) = \{a\}$ is obtained. After removal of Q_0 and the edge $(\mathbf{x}_0, \mathbf{x}_1)$, the new assignment is:

$$\begin{array}{lcl} I_0(\mathbf{x}_1) & = & \{a, b\} \\ I_0(\mathbf{x}_2) & = & \{c\} \\ I_0(\mathbf{x}_3) & = & \{c\} \end{array}$$

The value of all variables in the strongly connected component Q_1 then are determined as $I_0(\mathbf{x}_1) \cup I_0(\mathbf{x}_2) \cup I_0(\mathbf{x}_3) = \{a, b, c\}$. □

3.3 *Top-Down* Syntax Analysis

3.3.1 Introduction

The way different parsers work can be understood best by observing how they construct the parse tree to an input word. *Top-down* parsers start the construction of the parse tree at the root. In the initial situation, the constructed fragment of the parse tree consists of the root, which is labeled by the start symbol of the CFG; nothing of the input word w is consumed. In this situation, one alternative for the start symbol is selected for expansion. The symbols of the right side of this alternative are attached under the root extending the upper fragment of the parse tree. The next nonterminal to be considered is the one on the leftmost position. The selection of one alternative for this nonterminal and the attachment of the right side below the node labeled with the left side is repeated until the parse tree is complete. Among the symbols of the right side of a production that are attached to the growing tree fragment, there can also be terminal symbols. If there is no nonterminal to the left of such a terminal symbol, the *top-down* parser compares it with the next symbol in the input. If they agree, the parser consumes the corresponding symbol in the input. Otherwise, the parser reports a syntax error. Thus, a *top-down* analysis performs the following two types of actions:

- Selection of an alternative for the actual leftmost nonterminal and attachment of the right side of the production to the actual tree fragment.
- Comparison of terminal symbols to the left of the leftmost nonterminal with the remaining input.

Figures 3.7, 3.8, 3.9, and 3.10 show some parse tree fragments for the arithmetic expression $\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$ according to grammar G_2. The selection of alternatives for the nonterminals to be expanded was cleverly done in such a way as to lead to a successful termination of the analysis.

$$\begin{array}{lll} S \to E & E' \to +E \mid \varepsilon & T' \to *T \mid \varepsilon \\ E \to T\,E' & T \;\to F\,T' & F \;\to (E) \mid \mathsf{Id} \end{array}$$

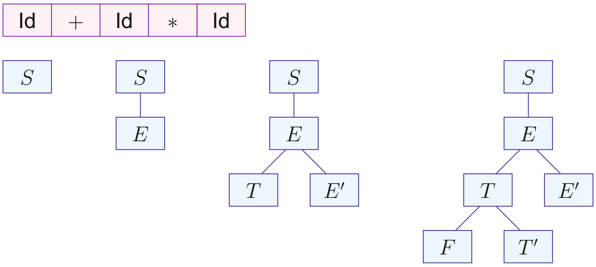


Fig. 3.7 The first parse-tree fragments of a *top-down* analysis of the word $\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$ according to grammar G_2. They are constructed without reading any symbol from the input

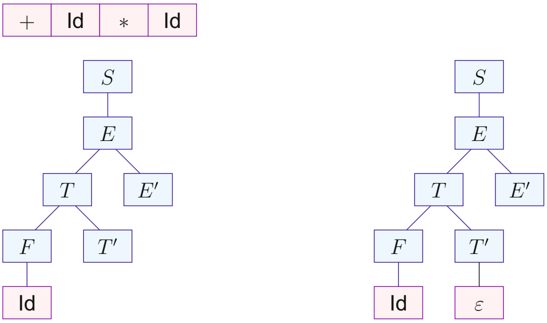


Fig. 3.8 The parse tree fragments after reading of the symbol Id and before the terminal symbol $+$ is attached to the fragment

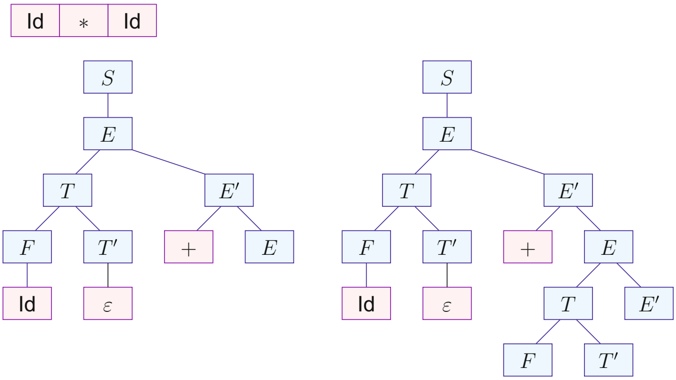


Fig. 3.9 The first and the last parse tree after reading of the symbols $+$ and before the second symbol Id appears in the parse tree

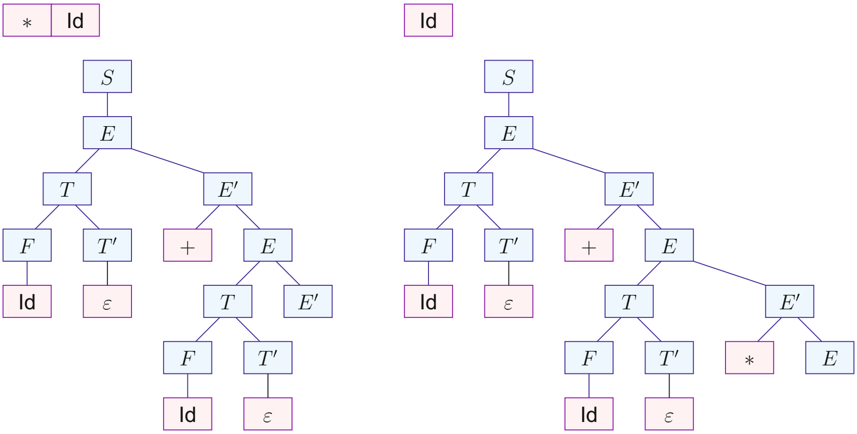


Fig. 3.10 The parse tree after the reduction for the second occurrence of T' and the parse tree after reading the symbol $*$, together with the remaining input

3.3.2 $LL(k)$: Definition, Examples, and Properties

The IPDA P_G to a CFG G works in principle like a *top-down* parser; its (E)-transitions predict which alternative to select for the actual nonterminal to derive the input word. The trouble is that the IPDA P_G takes this decision in a nondeterministic way. The nondeterminism stems from the (E) transitions. If $[X \to \beta.Y\gamma]$ is the actual state and if Y has the alternatives $Y \to \alpha_1 \mid \ldots \mid \alpha_n$, there are n transitions:

$$\Delta([X \to \beta.Y\gamma], \varepsilon) = \{[X \to \beta.Y\gamma][Y \to .\alpha_i] \mid 1 \leq i \leq n\}$$

To derive a deterministic automaton from the IPDA P_G we equip the automaton with a *bounded look-ahead* into the remaining input. We fix a natural number $k \geq 1$ and allow the IPDA to inspect the k first symbols of the remaining input at each (E) transition to aid in its decision. If this look-ahead of depth k always suffices to select the right alternative we call the grammar (strong) $LL(k)$-grammar.

Let us regard a configuration that the IPDA P_G has reached from an initial configuration:

$$([S' \to .S], uv) \vdash_{P_G}^{*} (\rho[X \to \beta.Y\gamma], v)$$

Because of invariant (I) of Sect. 3.2.4 it holds that $\mathsf{hist}(\rho)\beta \overset{*}{\Longrightarrow} u$.

Let $\rho = [X_1 \longrightarrow \beta_1.X_2\gamma_1]\ldots[X_n \longrightarrow \beta_n.X_{n+1}\gamma_n]$ be a sequence of items. We call the sequence

$$\mathsf{fut}(\rho) = \gamma_n \ldots \gamma_1$$

the *future* of ρ. Let $\delta = \mathsf{fut}(\rho)$. So far, the leftmost derivation $S' \underset{lm}{\overset{*}{\Longrightarrow}} uY\gamma\delta$ has been found. If this derivation can be extended to a derivation of the terminal word uv, that is, $S' \underset{lm}{\overset{*}{\Longrightarrow}} uY\gamma\delta \underset{lm}{\overset{*}{\Longrightarrow}} uv$, then in a $LL(k)$-grammar the alternative to be selected for Y only depends on u, Y and $v|_k$.

Let $k \geq 1$ be a natural number. The reduced CFG G is a $LL(k)$-*grammar* if for every two leftmost derivations:

$$S \underset{lm}{\overset{*}{\Longrightarrow}} uY\alpha \underset{lm}{\Longrightarrow} u\beta\alpha \underset{lm}{\overset{*}{\Longrightarrow}} ux \quad \text{and} \quad S \underset{lm}{\overset{*}{\Longrightarrow}} uY\alpha \underset{lm}{\Longrightarrow} u\gamma\alpha \underset{lm}{\overset{*}{\Longrightarrow}} uy$$

and $x|_k = y|_k$ implies $\beta = \gamma$.

For an $LL(k)$ grammar, the selection of the alternative for the next nonterminal Y in general depends not only on Y and the next k symbols, but also on the already consumed prefix u of the input. If this selection does, however, not depend on the already consumed left context u, we call the grammar *strong*-$LL(k)$.

Example 3.3.1 Let G_1 the CFG with the productions:

$$\begin{array}{lcll} \langle stat \rangle & \rightarrow & \mathsf{if}\ (\mathsf{Id})\ \langle stat \rangle\ \mathsf{else}\ \langle stat \rangle & | \\ & & \mathsf{while}\ (\mathsf{Id})\ \langle stat \rangle & | \\ & & \{\ \langle stats \rangle\ \} & | \\ & & \mathsf{Id}\ '='\ \mathsf{Id}; & \\ \langle stats \rangle & \rightarrow & \langle stat \rangle\ \langle stats \rangle & | \\ & & \varepsilon & \end{array}$$

The grammar G_1 is an $LL(1)$-grammar. If $\langle stat \rangle$ occurs as leftmost nonterminal in a sentential form then the next input symbol determines which alternative must be applied. More precisely, it means that for two derivations of the form

$$\begin{array}{lllllll} \langle stat \rangle & \overset{*}{\underset{lm}{\Longrightarrow}} & w\ \langle stat \rangle\ \alpha & \underset{lm}{\Longrightarrow} & w\,\beta\,\alpha & \overset{*}{\underset{lm}{\Longrightarrow}} & w\,x \\ \langle stat \rangle & \overset{*}{\underset{lm}{\Longrightarrow}} & w\ \langle stat \rangle\ \alpha & \underset{lm}{\Longrightarrow} & w\,\gamma\,\alpha & \overset{*}{\underset{lm}{\Longrightarrow}} & w\,y \end{array}$$

it follows from $x|_1 = y|_1$ that $\beta = \gamma$. If for instance $x|_1 = y|_1 = \mathsf{if}$, then $\beta = \gamma = \mathsf{if}\ (\mathsf{Id})\ \langle stat \rangle\ \mathsf{else}\ \langle stat \rangle$. □

Example 3.3.2 We now add the following production to the grammar G_1 of Example 3.3.1:

$$\begin{array}{lcllll} \langle stat \rangle & \rightarrow & \mathsf{Id} :\ \langle stat \rangle & | & // & \text{labeled statement} \\ & & \mathsf{Id}\,(\mathsf{Id}); & & // & \text{procedure call} \end{array}$$

The resulting grammar G_2 is no longer an $LL(1)$-grammar because it holds that

$$\begin{array}{lllllll} \langle stat \rangle & \overset{*}{\underset{lm}{\Longrightarrow}} & w\ \langle stat \rangle\ \alpha & \underset{lm}{\Longrightarrow} & w\ \overbrace{\mathsf{Id}\,'='\,\mathsf{Id};}^{\beta}\ \alpha & \overset{*}{\underset{lm}{\Longrightarrow}} & w\,x \\ \langle stat \rangle & \overset{*}{\underset{lm}{\Longrightarrow}} & w\ \langle stat \rangle\ \alpha & \underset{lm}{\Longrightarrow} & w\ \overbrace{\mathsf{Id} : \langle stat \rangle}^{\gamma}\ \alpha & \overset{*}{\underset{lm}{\Longrightarrow}} & w\,y \\ \langle stat \rangle & \overset{*}{\underset{lm}{\Longrightarrow}} & w\ \langle stat \rangle\ \alpha & \underset{lm}{\Longrightarrow} & w\ \overbrace{\mathsf{Id}(\mathsf{Id});}^{\delta}\ \alpha & \overset{*}{\underset{lm}{\Longrightarrow}} & w\,z \end{array}$$

with $x|_1 = y|_1 = z|_1 = \mathsf{Id}$, but β, γ, δ are pairwise different.

On the other hand, G_2 is a $LL(2)$-grammar. For the given three leftmost derivations we find that

$$x|_2 = \mathsf{Id}\,'='\qquad y|_2 = \mathsf{Id}\ :\qquad z|_2 = \mathsf{Id}\ ($$

are pairwise different. And these are indeed the only critical cases. □

Example 3.3.3 Consider a CFG G_3 which has the productions

$$\begin{array}{lcll}
\langle stat\rangle & \to & \textsf{if}\ (\langle var\rangle)\ \langle stat\rangle\ \textsf{else}\ \langle stat\rangle & | \\
 & & \textsf{while}\ (\langle var\rangle)\ \langle stat\rangle & | \\
 & & \{\ \langle stats\rangle\ \} & | \\
 & & \langle var\rangle\ '='\ \langle var\rangle\ ; & | \\
 & & \langle var\rangle\ ; & \\
\langle stats\rangle & \to & \langle stat\rangle\ \langle stats\rangle & | \\
 & & \varepsilon & \\
\langle var\rangle & \to & \textsf{Id} & | \\
 & & \textsf{Id}() & | \\
 & & \textsf{Id}(\langle vars\rangle) & \\
\langle vars\rangle & \to & \langle var\rangle\ ,\ \langle vars\rangle & | \\
 & & \langle var\rangle &
\end{array}$$

For no $k \geq 1$, the grammar G_3 has the $LL(k)$-property. For a contradiction, assume G_3 is an $LL(k)$ grammar for some $k \geq 1$.

$$\text{Let}\quad \langle stat\rangle \Rightarrow \beta \underset{lm}{\overset{*}{\Longrightarrow}} x \quad\text{and}\quad \langle stat\rangle \Rightarrow \gamma \underset{lm}{\overset{*}{\Longrightarrow}} y \quad\text{with}$$

$$x = \textsf{Id}\ (\underbrace{\textsf{Id}, \textsf{Id}, \ldots, \textsf{Id}}_{k})\ '='\ \textsf{Id}; \quad\text{and}\quad y = \textsf{Id}\ (\underbrace{\textsf{Id}, \textsf{Id}, \ldots, \textsf{Id}}_{k});$$

Then $x|_k = y|_k$ holds, but

$$\beta = \langle var\rangle\ '='\ \langle var\rangle \qquad \gamma \ = \ \langle var\rangle\ ;$$

and therefore $\beta \neq \gamma$. □

There exists, however, an $LL(2)$-grammar for the language $L(G_3)$ of grammar G_3, which can be obtained from G_3 by *factorization*. Critical in G_3 are the productions for assignment and procedure call. Factorization introduces sharing of common prefixes of those productions. A new nonterminal symbol follows this common prefix from which the different continuations can be derived. In the example, the productions

$$\langle stat\rangle \to \langle var\rangle\ '='\ \langle var\rangle\ ;\ \mid\ \langle var\rangle\ ;$$

are replaced with

$$\begin{array}{lcl}
\langle stat\rangle & \to & \langle var\rangle\ Z \\
Z & \to & '='\ \langle var\rangle\ ;\ \mid ;
\end{array}$$

Now, an $LL(1)$-parser can decide between the critical alternatives using the next symbols Id and “;”.

Example 3.3.4 Let $G_4 = (\{S, A, B\}, \{0, 1, a, b\}, P_4, S)$, where the set P_4 of productions is given by

$$\begin{array}{lcl} S & \rightarrow & A \mid B \\ A & \rightarrow & aAb \mid 0 \\ B & \rightarrow & aBbb \mid 1 \end{array}$$

Then

$$L(G_4) = \{a^n 0 b^n \mid n \geq 0\} \cup \{a^n 1 b^{2n} \mid n \geq 0\}$$

and G_4 is not an $LL(k)$-grammar for any $k \geq 1$. To see this we consider the two leftmost derivations:

$$\begin{array}{lclcl} S & \underset{lm}{\Longrightarrow} & A & \overset{*}{\underset{lm}{\Longrightarrow}} & a^k 0 b^k \\ S & \underset{lm}{\Longrightarrow} & B & \overset{*}{\underset{lm}{\Longrightarrow}} & a^k 1 b^{2k} \end{array}$$

For no $k \geq 1$, G_4 is an $LL(k)$ grammar. For each $k \geq 1$ it holds $(a^k 0 b^k)|_k = (a^k 1 b^{2k})|_k$, but the right sides A and B for S are different. In this case it can be shown that for every $k \geq 1$, the language $L(G_4)$ cannot be generated by an $LL(k)$-grammar. □

Theorem 3.3.1 The reduced CFG $G = (V_N, V_T, P, S)$ is an $LL(k)$-grammar if and only if for each two different productions $A \rightarrow \beta$ and $A \rightarrow \gamma$ of G, it holds that

$$\mathsf{first}_k(\beta\alpha) \cap \mathsf{first}_k(\gamma\alpha) = \emptyset \quad \text{for all } \alpha \quad \text{with } S \overset{*}{\underset{lm}{\Longrightarrow}} wA\alpha$$

Proof To prove the direction, "$\Rightarrow$", we assume that G is an $LL(k)$-grammar, but there exists an $x \in \mathsf{first}_k(\beta\alpha) \cap \mathsf{first}_k(\gamma\alpha)$. According to the definition of first_k and because G is reduced, there exist derivations

$$\begin{array}{l} S \overset{*}{\underset{lm}{\Longrightarrow}} uA\alpha \underset{lm}{\Longrightarrow} u\beta\alpha \overset{*}{\underset{lm}{\Longrightarrow}} uxy \\ S \overset{*}{\underset{lm}{\Longrightarrow}} uA\alpha \underset{lm}{\Longrightarrow} u\gamma\alpha \overset{*}{\underset{lm}{\Longrightarrow}} uxz, \end{array}$$

where in the case $|x| < k$, we have $y = z = \varepsilon$. But then $\beta \neq \gamma$ implies that G cannot be an $LL(k)$-grammar – a contradiction to our assumption.

To prove the other direction, "$\Leftarrow$", we assume, G is not an $LL(k)$-grammar. Then there exist two leftmost derivations

$$\begin{array}{l} S \overset{*}{\underset{lm}{\Longrightarrow}} uA\alpha \underset{lm}{\Longrightarrow} u\beta\alpha \overset{*}{\underset{lm}{\Longrightarrow}} ux \\ S \overset{*}{\underset{lm}{\Longrightarrow}} uA\alpha \underset{lm}{\Longrightarrow} u\gamma\alpha \overset{*}{\underset{lm}{\Longrightarrow}} uy \end{array}$$

with $x|_k = y|_k$, where $A \to \beta$, $A \to \gamma$ are different productions. Then the word $x|_k = y|_k$ is contained in $\mathsf{first}_k(\beta\alpha) \cap \mathsf{first}_k(\gamma\alpha)$ – a contradiction to the claim of the theorem. □

Theorem 3.3.1 states that in an $LL(k)$-grammar, two different productions applied to the same left-sentential form always lead to two different k-prefixes of the remaining input. Theorem 3.3.1 allows us to derive useful criteria for membership in certain subclasses of $LL(k)$-grammars. The first concerns the case $k = 1$.

The set $\mathsf{first}_1(\beta\alpha) \cap \mathsf{first}_1(\gamma\alpha)$ for all left-sentential forms $wA\alpha$ and any two different alternatives $A \to \beta$ and $A \to \gamma$ can be simplified to $\mathsf{first}_1(\beta) \cap \mathsf{first}_1(\gamma)$, if neither β nor γ produce the empty word ε. This is the case if no nonterminal of G is ε-productive. In practice, however, it would be too restrictive to rule out ε-productions. Consider the case that the empty word is produced by at least one of the two right sides β or γ. If ϵ is produced both by β and γ, G cannot be an $LL(1)$-grammar. Let us, therefore, assume that $\beta \overset{*}{\Longrightarrow} \varepsilon$, but that ε cannot be derived from γ. Then the following holds for all left-sentential forms $uA\alpha, u'A\alpha'$:

$$\begin{aligned}
\mathsf{first}_1(\beta\alpha) \cap \mathsf{first}_1(\gamma\alpha') &= \mathsf{first}_1(\beta\alpha) \;\cap\; \mathsf{first}_1(\gamma) \odot_1 \mathsf{first}_1(\alpha') \\
&= \mathsf{first}_1(\beta\alpha) \;\cap\; \mathsf{first}_1(\gamma) \\
&= \mathsf{first}_1(\beta\alpha) \;\cap\; \mathsf{first}_1(\gamma\alpha) \\
&= \emptyset
\end{aligned}$$

This implies that

$$\begin{aligned}
&\mathsf{first}_1(\beta) \odot_1 \mathsf{follow}_1(A) \;\cap\; \mathsf{first}_1(\gamma) \odot_1 \mathsf{follow}_1(A) \\
&\quad= \bigcup\{\mathsf{first}_1(\beta\alpha) \mid S \underset{lm}{\overset{*}{\Longrightarrow}} uA\alpha\} \;\cap\; \bigcup\{\mathsf{first}_1(\gamma\alpha') \mid S \underset{lm}{\overset{*}{\Longrightarrow}} u'A\alpha'\} \\
&\quad= \emptyset
\end{aligned}$$

Hereby, we obtain the following theorem:

Theorem 3.3.2 A reduced CFG G is an $LL(1)$-grammar if and only if for each two different productions $A \to \beta$ and $A \to \gamma$, it holds that

$$\mathsf{first}_1(\beta) \odot_1 \mathsf{follow}_1(A) \;\cap\; \mathsf{first}_1(\gamma) \odot_1 \mathsf{follow}_1(A) = \emptyset \,.$$

□

In contrast to the characterization of Theorem 3.3.1, the characterization of Theorem 3.3.2 can be easily verified for a given grammar. A perhaps handier formulation is obtained, if the properties of 1-concatenation are taken into account.

Corollary 3.3.2.1 A reduced CFG G is an $LL(1)$-grammar if and only if for all alternatives $A \to \alpha_1 \mid \ldots \mid \alpha_n$, it holds that

1. $\mathsf{first}_1(\alpha_1), \ldots, \mathsf{first}_1(\alpha_n)$ are pairwise disjoint; in particular, at most one of these sets contains ε;
2. $\varepsilon \in \mathsf{first}_1(\alpha_i)$ for some i implies that $\mathsf{first}_1(\alpha_j) \cap \mathsf{follow}_1(A) = \emptyset$ for all $1 \leq j \leq n, j \neq i$. □

The property of Theorem 3.3.2 can be generalized to arbitrary lengths $k \geq 1$ of look-aheads.

A reduced CFG $G = (V_N, V_T, P, S)$ is called *strong* $LL(k)$-grammar, if for each two different productions $A \to \beta$ and $A \to \gamma$ of a nonterminal A, it always holds that

$$\mathsf{first}_k(\beta) \odot_k \mathsf{follow}_k(A) \ \cap \ \mathsf{first}_k(\gamma) \odot_k \mathsf{follow}_k(A) = \emptyset.$$

According to this definition and Theorem 3.3.2 every $LL(1)$-grammar is a strong $LL(1)$-grammar. However for $k > 1$, an $LL(k)$-grammar is not necessarily also a strong $LL(k)$-grammar. The reason is that the set $\mathsf{follow}_k(A)$ contains the following words of *all* left sentential forms with occurrences of A. In contrast, the $LL(k)$ condition only refers to following words of *one* left sentential form.

Example 3.3.5 Let G be the CFG with the productions

$$S \to aAaa \mid bAba \qquad\qquad A \to b \mid \varepsilon$$

We check:
Case 1: The derivation starts with $S \Rightarrow aAaa$. Then $\mathsf{first}_2(baa) \cap \mathsf{first}_2(aa) = \emptyset$.
Case 2: The derivation starts with $S \Rightarrow bAba$. Then $\mathsf{first}_2(bba) \cap \mathsf{first}_2(ba) = \emptyset$.
Hence G is an $LL(2)$-grammar according to Theorem 3.3.1. However, the grammar G is not a strong $LL(2)$-grammar, because

$$\begin{aligned}
\mathsf{first}_2(b) \odot_2 \mathsf{follow}_2(A) \ &\cap \ \mathsf{first}_2(\varepsilon) \odot_2 \mathsf{follow}_2(A) \\
&= \{b\} \odot_2 \{aa, ba\} \ \cap \ \{\varepsilon\} \odot_2 \{aa, ba\} \\
&= \{ba, bb\} \ \cap \ \{aa, ba\} \\
&= \{ba\}
\end{aligned}$$

In the example, $\mathsf{follow}_1(A)$ is too undifferentiated because it collects terminal following words that may occur in *different* sentential forms. □

3.3.3 Left Recursion

Deterministic parsers that construct the parse tree for the input *top-down* cannot deal with *left recursive* nonterminals. A nonterminal A of a CFG G is called *left recursive* if there exists a derivation $A \overset{+}{\Longrightarrow} A\beta$.

Theorem 3.3.3 Let G be a reduced CFG. G is not an $LL(k)$-grammar for any $k \geq 1$ if at least one nonterminal of the grammar G is left-recursive.

Proof Let X be a left recursive nonterminal of grammar G. For simplicity we assume that G has a production $X \to X\beta$. Since G is reduced, there must exist another production $X \to \alpha$. If X occurs in a left sentential form, that is, $S \underset{lm}{\overset{*}{\Longrightarrow}} uX\gamma$, the production $X \to X\beta$ can be applied arbitrarily often. This means that for each $n \geq 1$ there exists a leftmost derivation

$$S \underset{lm}{\overset{*}{\Longrightarrow}} wX\gamma \underset{lm}{\overset{n}{\Longrightarrow}} wX\beta^n\gamma \;.$$

Let us assume that grammar G is an $LL(k)$-grammar. Then by Theorem 3.3.1, it holds that

$$\mathsf{first}_k(X\beta^{n+1}\gamma) \cap \mathsf{first}_k(\alpha\beta^n\gamma) = \emptyset.$$

Due to $X \to \alpha$, we have

$$\mathsf{first}_k(\alpha\beta^{n+1}\gamma) \subseteq \mathsf{first}_k(X\beta^{n+1}\gamma),$$

and hence also

$$\mathsf{first}_k(\alpha\beta^{n+1}\gamma) \cap \mathsf{first}_k(\alpha\beta^n\gamma) = \emptyset.$$

If $\beta \overset{*}{\Longrightarrow} \varepsilon$ holds, we immediately obtain a contradiction. Otherwise, we choose $n \geq k$ and again obtain a contradiction. Hence, G cannot be an $LL(k)$-grammar. □

We conclude that no generator of $LL(k)$-parsers can cope with left recursive grammars. However, each grammar with left recursion can be transformed into a grammar without left recursion that defines the same language. Let $G = (V_N, V_T, P, S)$ and assume for simplicity that the grammar G has no ε-productions (see Exercise 3) and no recursive chain productions, that is, there is no nonterminal A with $A \underset{G}{\overset{+}{\Longrightarrow}} A$. For G, we construct a CFG $G' = (V'_N, V_T, P', S)$ with the same set V_T of terminal symbols, the same start symbol S, a set V'_N of nonterminal symbols

$$V'_N = V_N \cup \{\langle A, B\rangle \mid A, B \in V_N\},$$

and a set of productions P'.

- If $B \to a\beta \ \in \ P$ for a terminal symbol $a \in V_T$, then $A \to a\beta \ \langle A, B\rangle \ \in \ P'$ for each $A \in V_N$;
- If $C \to B\beta \ \in \ P$ then $\langle A, B\rangle \to \beta \, \langle A, C\rangle \ \in \ P'$;
- Finally, $\langle A, A\rangle \to \varepsilon \ \in \ P'$ for all $A \in V_N$.

Example 3.3.6 For the grammar G_0 with the productions

$$\begin{array}{l} E \to E + T \mid T \\ T \to T * F \mid F \\ F \to (E) \mid \mathsf{Id} \end{array}$$

we obtain after removal of nonproductive nonterminals

$$\begin{array}{lcl}
E & \to & (E)\ \langle E, F\rangle \mid \mathsf{Id}\ \langle E, F\rangle \\
\langle E, F\rangle & \to & \langle E, T\rangle \\
\langle E, T\rangle & \to & *\,F\ \langle E, T\rangle \mid \langle E, E\rangle \\
\langle E, E\rangle & \to & +T\ \langle E, E\rangle \mid \varepsilon \\
T & \to & (E)\ \langle T, F\rangle \mid \mathsf{Id}\ \langle E, F\rangle \\
\multicolumn{3}{l}{\langle T, F\rangle \to \langle T, T\rangle} \\
\multicolumn{3}{l}{\langle T, T\rangle \to *F\ \langle T, T\rangle \mid \varepsilon} \\
F & \to & (E)\ \langle F, F\rangle \mid \mathsf{Id}\ \langle F, F\rangle \\
\langle F, F\rangle & \to & \varepsilon
\end{array}$$

While the grammar G_0 has only three nonterminals and six productions, the grammar G_1 needs nine nonterminals and 15 productions.

The parse tree for $\mathsf{Id} + \mathsf{Id}$ according to grammar G_0 is shown in Fig. 3.11a, the one according to grammar G_1 in Fig. 3.11b. The latter one has quite a different structure. Intuitively, the grammar produces for a nonterminal the first possible terminal symbol *directly* and afterwards collects in a backward fashion the remainders of the right sides, which follow the nonterminal symbol on the left. The nonterminal $\langle A, B\rangle$ thus represents the task of performing this collection backward from B to A. □

We convince ourselves that the grammar G' constructed from grammar G has the following properties:

- Grammar G' has no left recursive nonterminals.
- There exists a leftmost derivation

  $$A \underset{G}{\overset{*}{\Longrightarrow}} B\gamma \underset{G}{\Longrightarrow} a\beta\gamma$$

  if and only there exists a rightmost derivation

  $$A \underset{G'}{\Longrightarrow} a\beta\ \langle A, B\rangle \underset{G'}{\overset{*}{\Longrightarrow}} a\beta\gamma\ \langle A, A\rangle$$

  in which after the first step only nonterminals of the form $\langle X, Y\rangle$ are replaced.

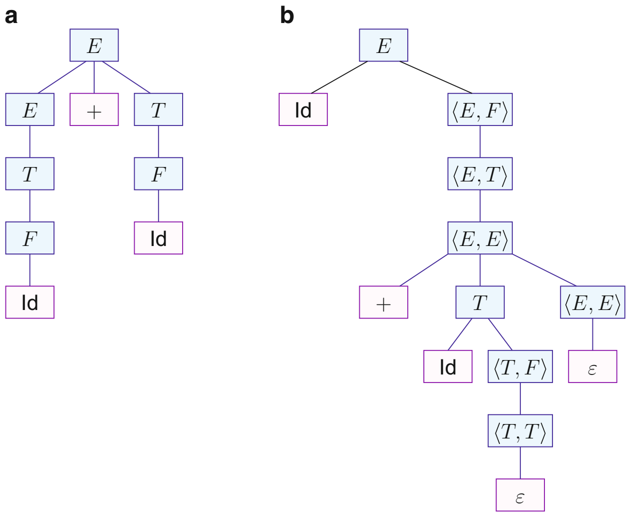


Fig. 3.11 Parse trees for Id + Id according to grammar G_0 of Example 3.3.6 and according to the grammar after removal of left recursion

The last property implies, in particular, that grammars G and G' are equivalent, i.e., that $L(G) = L(G')$ holds.

In some cases, the grammar obtained by removing left recursion is an $LL(k)$-grammar. This is the case for grammar G_0 of Example 3.3.6. We have already seen that the transformation to remove left recursion also has disadvantages. Let n be the number of nonterminals. The number of nonterminals as well as the number of productions can increase by a factor of $n + 1$. In large grammars, it therefore may not be advisable to perform this transformation *manually*. A parser generator, however, can do the transformation automatically and also can generate a program that automatically converts parse trees of the transformed grammar back into parse trees of the original grammar (see Exercise 7 of the next chapter). The user then would not even notice the grammar transformation.

Example 3.3.6 illustrates how much the parse tree of a word according to the transformed grammar can be different from the one according to the original grammar. The operator sits somewhat isolated between its remotely located operands. An alternative to the elimination of left recursion are grammars with *regular* right sides, which we will treat in Sect. 3.3.5.

3.3.4 Strong $LL(k)$-Parsers

Figure 3.12 shows the structure of a parser for strong $LL(k)$-grammars. The prefix w of the input is already read. The remaining input starts with a prefix u of length k. The pushdown contains a sequence of items of the CFG. The topmost item, the

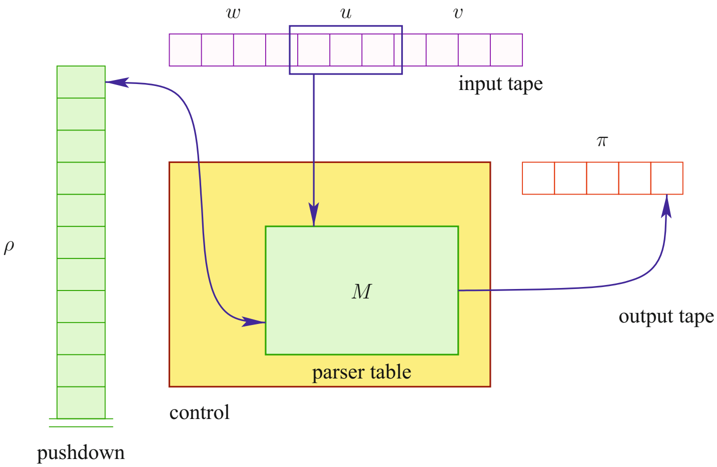


Fig. 3.12 Schematic representation of a strong $LL(k)$-parser.

actual state, Z, determines whether

- to read the next input symbol,
- to test for the successful end of the analysis, or
- to expand the actual nonterminal.

Upon expansion, the parser uses the parser table to select the correct alternative for the nonterminal. The parser table M is a two-dimensional array whose rows are indexed by nonterminals and whose columns are indexed by words of length at most k. It represents a selection function

$$V_N \times V_{T\#}^{\leq k} \rightarrow (V_T \cup V_N)^* \cup \{\mathsf{error}\}$$

which associates each nonterminal with the alternative that should be applied based on the given look-ahead, and signals an error if no alternative is available for the given combination of actual state and look-ahead. Let $[X \rightarrow \beta.Y\gamma]$ be the topmost item on the pushdown and u be the prefix of length k of the remaining input. If $M[Y,u] = (Y \rightarrow \alpha)$ then $[Y \rightarrow .\alpha]$ will be the new topmost pushdown symbol and the production $Y \rightarrow \alpha$ is written to the output tape.

The table entries in M for a nonterminal Y are determined in the following way. Let $Y \rightarrow \alpha_1 \mid \ldots \mid \alpha_r$ be the alternatives for Y. For a strong $LL(k)$-grammar, the sets $\mathsf{first}_k(\alpha_i) \odot_k \mathsf{follow}_k(Y)$ are disjoint. For each of the $u \in \mathsf{first}_k(\alpha_1) \odot_k \mathsf{follow}_k(Y) \cup \ldots \cup, \mathsf{first}_k(\alpha_r) \odot_k \mathsf{follow}_k(Y)$, we therefore set

$$M[Y,u] \leftarrow \alpha_i \qquad \text{if and only if} \qquad u \in \mathsf{first}_k(\alpha_i) \odot_k \mathsf{follow}_k(Y)$$

Otherwise, $M[Y,u]$ is set to error. The entry $M[Y,u] = \mathsf{error}$ means that the actual nonterminal and the prefix of the remaining input do not go together. This means that a syntax error has been found. An error-diagnosis and error-handling routine is

Table 3.3 $LL(1)$-parser table for the grammar of Example 3.2.13

	(	)	+	*	Id	#
S	E	error	error	error	E	error
E	$(E)\ \langle E,F\rangle$	error	error	error	Id $\langle E,F\rangle$	error
T	$(E)\ \langle T,F\rangle$	error	error	error	Id $\langle T,F\rangle$	error
F	$(E)\ \langle F,F\rangle$	error	error	error	Id $\langle F,F\rangle$	error
$\langle E,F\rangle$	error	$\langle E,T\rangle$	$\langle E,T\rangle$	$\langle E,T\rangle$	error	$\langle E,T\rangle$
$\langle E,T\rangle$	error	$\langle E,E\rangle$	$\langle E,E\rangle$	$*F\ \langle E,T\rangle$	error	$\langle E,E\rangle$
$\langle E,E\rangle$	error	ε	$+T\ \langle E,E\rangle$	error	error	ε
$\langle T,F\rangle$	error	$\langle T,T\rangle$	$\langle T,T\rangle$	$\langle T,T\rangle$	error	$\langle T,T\rangle$
$\langle T,T\rangle$	error	ε	ε	$*F\ \langle T,T\rangle$	error	ε
$\langle F,F\rangle$	error	ε	ε	ε	error	ε

started, which attempts to continue the analysis. Such approaches are described in Sect. 3.3.8.

For $k = 1$, the construction of the parser table is particularly simple. Because of Corollary 3.3.2.1, it does not require k-concatenation. Instead, it suffices to test u for membership in one of the sets $\mathsf{first}_1(\alpha_i)$ and maybe in $\mathsf{follow}_1(Y)$.

Example 3.3.7 Table 3.3 is the $LL(1)$-parser table for the grammar of Example 3.2.13. Table 3.4 describes the run of the associated parser for input Id $*$ Id#. □

Our construction of $LL(k)$-parsers are only applicable to *strong* $LL(k)$-grammars. This restriction, however, is not really severe.

- The case occurring most often in practice is the case $k = 1$, and each $LL(1)$-grammar is a strong $LL(1)$-grammar.
- If a look-ahead $k > 1$ is needed, and is the grammar $LL(k)$, but not strong $LL(k)$, a general transformation can be applied to convert the grammar into a strong $LL(k)$-grammar that accepts the same language. (see Exercise 7).

We therefore do not present a parsing method for general $LL(k)$-grammars. In the following Sect. 3.3.5, we instead provide for the case $k = 1$ a second, much more popular implememtation of $LL(k)$-parsers that is based on mutually recursive procedures.

3.3.5 Right-Regular Context-Free Grammars

Left recursive nonterminals destroy the $LL(k)$-property of CFGs. Left recursion is often used in order to specify lists, for instance, of parameters, of statements and declarations, or sequences of operands which are to be combined by some associative operator. As a substitute to left recursion, we will instead allow regular expressions in right sides of grammar productions.

A *right-regular* CFG is a tuple $G = (V_N, V_T, p, S)$, where V_N, V_T, S denote as usual the alphabets of nonterminals and terminals, respectively, together with the

Table 3.4 Parser run for input $Id * Id\#$

Stack	Input
$[S \to .E]$	$\mathsf{Id} * \mathsf{Id}\#$
$[S \to .E][E \to .\mathsf{Id}\ \langle E, F\rangle]$	$\mathsf{Id} * \mathsf{Id}\#$
$[S \to .E][E \to \mathsf{Id}.\ \langle E, F\rangle]$	$*\mathsf{Id}\#$
$[S \to .E][E \to \mathsf{Id}.\ \langle E, F\rangle][\langle E, F\rangle \to .\ \langle E, T\rangle]$	$*\mathsf{Id}\#$
$[S \to .E][E \to \mathsf{Id}.\ \langle E, F\rangle][\langle E, F\rangle \to .\ \langle E, T\rangle][\langle E, T\rangle \to .*F\ \langle E, T\rangle]$	$*\mathsf{Id}\#$
$[S \to .E][E \to \mathsf{Id}.\ \langle E, F\rangle][\langle E, F\rangle \to .\ \langle E, T\rangle][\langle E, T\rangle \to *.F\ \langle E, T\rangle]$	$\mathsf{Id}\#$
$[S \to .E][E \to \mathsf{Id}.\ \langle E, F\rangle][\langle E, F\rangle \to .\ \langle E, T\rangle]$ $[\langle E, T\rangle \to *.F\ \langle E, T\rangle][F \to .\mathsf{Id}\ \langle F, F\rangle]$	$\mathsf{Id}\#$
$[S \to .E][E \to \mathsf{Id}.\ \langle E, F\rangle][\langle E, F\rangle \to .\ \langle E, T\rangle]$ $[\langle E, T\rangle \to *.F\ \langle E, T\rangle][F \to \mathsf{Id}.\ \langle F, F\rangle]$	$\#$
$[S \to .E][E \to \mathsf{Id}.\ \langle E, F\rangle][\langle E, F\rangle \to .\ \langle E, T\rangle]$ $[\langle E, T\rangle \to *.F\ \langle E, T\rangle][F \to \mathsf{Id}.\ \langle F, F\rangle][\langle F, F\rangle \to .]$	$\#$
$[S \to .E][E \to \mathsf{Id}.\ \langle E, F\rangle][\langle E, F\rangle \to .\ \langle E, T\rangle]$ $[\langle E, T\rangle \to *.F\ \langle E, T\rangle][F \to \mathsf{Id}\ \langle F, F\rangle\ .]$	$\#$
$[S \to .E][E \to \mathsf{Id}.\ \langle E, F\rangle][\langle E, F\rangle \to .\ \langle E, T\rangle][\langle E, T\rangle \to *F.\ \langle E, T\rangle]$	$\#$
$[S \to .E][E \to \mathsf{Id}.\ \langle E, F\rangle][\langle E, F\rangle \to .\ \langle E, T\rangle]$ $[\langle E, T\rangle \to *F.\ \langle E, T\rangle][\langle E, T\rangle \to .\ \langle E, E\rangle]$	$\#$
$[S \to .E][E \to \mathsf{Id}.\ \langle E, F\rangle][\langle E, F\rangle \to .\ \langle E, T\rangle]$ $[\langle E, T\rangle \to *F.\ \langle E, T\rangle][\langle E, T\rangle \to .\ \langle E, E\rangle][\langle E, E\rangle \to .]$	$\#$
$[S \to .E][E \to \mathsf{Id}.\ \langle E, F\rangle][\langle E, F\rangle \to .\ \langle E, T\rangle]$ $[\langle E, T\rangle \to *F.\ \langle E, T\rangle][\langle E, T\rangle \to \langle E, E\rangle\ .]$	$\#$
$[S \to .E][E \to \mathsf{Id}.\ \langle E, F\rangle][\langle E, F\rangle \to .\ \langle E, T\rangle][\langle E, T\rangle \to *F\ \langle E, T\rangle\ .]$	$\#$
$[S \to .E][E \to \mathsf{Id}.\ \langle E, F\rangle][\langle E, F\rangle \to \langle E, T\rangle\ .]$	$\#$
$[S \to .E][E \to \mathsf{Id}\ \langle E, F\rangle\ .]$	$\#$
$[S \to E.]$	$\#$

Output:

$(S \to E)\ (E \to \mathsf{Id}\ \langle E, F\rangle)\ (\langle E, F\rangle \to \langle E, T\rangle)\ (\langle E, T\rangle \to *\ F\ \langle E, T\rangle)$
$(F \to \mathsf{Id}\ \langle F, F\rangle)\ (\langle F, F\rangle \to \varepsilon)\ (\langle E, T\rangle \to \langle E, E\rangle)\ (\langle E, E\rangle \to \varepsilon)$

start symbol, and $p : V_N \to RA$ is a mapping that assigns to each nonterminal symbol a regular expression over $V_N \cup V_T$.

Example 3.3.8

A right-regular CFG for arithmetic expressions is given by

$$G_e = (\{S, E, T, F\}, \{\mathsf{Id}, \boxed{(}, \boxed{)}, \boxed{+}, \boxed{-}, \boxed{*}, \boxed{/}\}, p, S),$$

The terminal symbols $+, -, *, /$ as well as the two bracket symbols have been set into boxes in order to distinguish them from the metacharacters to denote regular expressions. The mapping p is given by:

$$\begin{array}{lcl}
S & \to & E \\
E & \to & T\,((\boxed{+} \mid \boxed{-})\; T)^{*} \\
T & \to & F\,\Big(\Big(\boxed{*} \mid \boxed{/}\Big)\; F\Big)^{*} \\
F & \to & \Big(\boxed{(}\,E\,\boxed{)} \mid \mathsf{Id}\Big)
\end{array}$$

□

Assume that $G = (V_N, V_T, p, S)$ denotes a right-regular CFG, and $\mathcal{R}$ the set of regular subexpressions r which occur in right sides $r(A)$, $A \in V_N$. The grammar G can be considered as the specification of an ordinary CFG $\langle G\rangle$ that additionally has the auxiliary nonterminal symbols $\langle r\rangle$, $r \in \mathcal{R}$, together with a set of productions that allow to derive from $\langle r\rangle$ all words in the language $L(r) \subseteq (V_N \cup V_T)^*$. Formally, the grammar $\langle G\rangle$ is defined as the CFG (V'_N, V_T, P', S) with $V'_N = V_N \cup \{\langle r\rangle \mid r \in \mathcal{R}\}$, where the set of productions P' is given by:

$$\begin{array}{lcll}
A & \longrightarrow & \langle r\rangle & \text{if } A \in V_N,\, p(A) = r \\
\langle X\rangle & \longrightarrow & X & \text{if } X \in V_N \cup V_T \\
\langle \varepsilon\rangle & \longrightarrow & \varepsilon & \text{if } X \in V_N \cup V_T \\
\langle r^*\rangle & \longrightarrow & \varepsilon \mid \langle r\rangle\,\langle r^*\rangle & \text{if } r^* \in \mathcal{R} \\
\langle (r_1 \mid \ldots \mid r_n)\rangle & \longrightarrow & \langle r_1\rangle \mid \ldots \mid \langle r_n\rangle & \text{if } (r_1 \mid \ldots \mid r_n) \in \mathcal{R} \\
\langle (r_1 \ldots r_n)\rangle & \longrightarrow & \langle r_1\rangle \ldots \langle r_n\rangle & \text{if } (r_1 \ldots r_n) \in \mathcal{R}
\end{array}$$

The language $L(G)$ of the right-regular CFG G then is defined as the language $L(\langle G\rangle)$ of the ordinary CFG $\langle G\rangle$.

Example 3.3.9 The transformed CFG $\langle G_e\rangle$ corresponding to the right-regular CFG G_e of Example 3.3.8 has the following productions:

$$\begin{array}{ll}
S & \to E \\
E & \to \Big\langle T\,((\boxed{+} \mid \boxed{-})\; T)^{*}\Big\rangle \\
T & \to \Big\langle F\,\Big(\Big(\boxed{*} \mid \boxed{/}\Big)\; F\Big)^{*}\Big\rangle \\
F & \to \Big\langle \boxed{(}\,E\,\boxed{)} \mid \mathsf{Id}\Big\rangle \\
\Big\langle T\,((\boxed{+} \mid \boxed{-})\; T)^{*}\Big\rangle & \to T\,\Big\langle ((\boxed{+} \mid \boxed{-})\; T)^{*}\Big\rangle \\
\Big\langle ((\boxed{+} \mid \boxed{-})\; T)^{*}\Big\rangle & \to \varepsilon \mid \langle (\boxed{+} \mid \boxed{-})\; T\rangle \Big\langle ((\boxed{+} \mid \boxed{-})\; T)^{*}\Big\rangle \\
\langle (\boxed{+} \mid \boxed{-})\; T\rangle & \to \langle \boxed{+} \mid \boxed{-}\rangle\, T \\
\langle \boxed{+} \mid \boxed{-}\rangle & \to \boxed{+} \mid \boxed{-} \\
\Big\langle F\,\Big(\Big(\boxed{*} \mid \boxed{/}\Big)\; F\Big)^{*}\Big\rangle & \to F\,\Big\langle \Big(\Big(\boxed{*} \mid \boxed{/}\Big)\; F\Big)^{*}\Big\rangle \\
\Big\langle \Big(\Big(\boxed{*} \mid \boxed{/}\Big)\; F\Big)^{*}\Big\rangle & \to \varepsilon \mid \Big\langle \boxed{*} \mid \boxed{/}\Big\rangle \Big\langle \Big(\Big(\boxed{*} \mid \boxed{/}\Big)\; F\Big)^{*}\Big\rangle \\
\Big\langle \boxed{*} \mid \boxed{/}\Big\rangle & \to \boxed{*} \mid \boxed{/} \\
\Big\langle \boxed{(}\,E\,\boxed{)} \mid \mathsf{Id}\Big\rangle & \to \boxed{(}\,E\,\boxed{)} \mid \mathsf{Id}
\end{array}$$

In order to increase readability of the occurring regular expressions, outermost brackets are omitted as well as inner brackets if the brackets can be recovered from operator precedences. Moreover, occurrences of nonterminals $\langle X\rangle$ have been directly replaced with X both for terminal and nonterminal symbols X. □

The right-regular CFG G is called $RLL(1)$-grammar if the corresponding CFG $\langle G\rangle$ is a $LL(1)$-grammar. For an $RLL(1)$-grammar G, the $LL(1)$-parser of the transformed grammar $\langle G\rangle$ can directly be used also as a parser for G. This parser, however, has the disadvantage that it performs pushdown operations – just in order to realize the evaluation of regular expressions in right sides. As a better alternative, we therefore propose a method that operates on the structure of the regular expressions directly and thus omits the detour through the transformed grammar.

$RLL(1)$-Parsers can be generated by using just the first_1-sets for the regular subexpressions r of right sides. These are defined as the first_1-sets for the corresponding nonterminals $\langle r\rangle$ of the transformed CFG $\langle G\rangle$:

$$\mathsf{first}_1(r) = \mathsf{first}_1(\langle r\rangle)$$

Example 3.3.10

The eff- as well as the first_1-sets for the nonterminals of the grammar G_e of Example 3.3.8 are:

$$\mathsf{first}_1(S) = \mathsf{first}_1(E) = \mathsf{first}_1(T) = \mathsf{first}_1(F) = \{\boxed{(}, \mathsf{Id}\}$$

□

3.3.6 *Recursive-Descent* Parsers for $RLL(1)$-Grammars

Analogously to the table-driven parser for $LL(1)$-grammars, a table-driven parser for $RLL(1)$-grammars could also be constructed. It is much more common, though, to implement the $RLL(1)$-parser by means of the technique of *recursice descent*. Compared to table-driven parsers, *recursive-descent* parsers have several practical advantages:

- They are typically faster since, due to the sizes of the tables, table-driven parsers often cause cache misses when accessing the tables.
- *Recursive-descent* parsers are so simple that they can be directly written in the programming language of the compiler and need not be generated. This simplifies error detection during implementation of the parser itself: potential flaws need not be hunted down within generated code that looks unreadable and strange, if the author is not acquainted with the technical detail of the generator.
- Moreover, the parser together with all semantic actions can be written as a single program in a single programming language. Error detection within partly generated, partly hand-written code is particularly cumbersome.

- Generators provide only limited support for the realization of decent error handling routines which, however, are mandatory for any reasonably practical parser. Error handling, on the other hand, can be elegantly integrated into a *recursive-descent* parser.

Assume we are given a right-regular CFG $G = (V_N, V_T, p, S)$ which is $RLL(1)$. For each nonterminal A, a procedure with name A is introduced. Calls to the function generate then translate the regular expressions in right sides into corresponding imperative program fragments. The function generate is a metaprogram that, for a given regular expression r, generates code which is able to analyze the input derived from $\langle r \rangle$: a $*$ expression becomes a *while*-loop; a concatenation becomes sequential composition; and a choice between alternatives results in a *switch*-statement. Terminals are turned into tests and nonterminals into recursive calls of the corresponding procedures. The first_1-sets of regular subexpressions are used to decide whether a loop should be exited and which alternative to choose. Such a parser is called *recursive-descent* parser, since it replaces the explicit manipulation of a pushdown store by means of the run-time stack and recursion.

The parser relies on a procedure expect in order to test whether the next input symbol is contained in the first_1-set of the next regular expression. In the following, we assume that the next input symbol is stored in the global variable *next*.

```
void expect(set⟨string ⟨terminal⟩⟩ $E$) {
    if ($\{\varepsilon, next\} \cap E = \emptyset$){
        output("Expected one of " + $E$ + ". Found " + next); exit();
    }
}
```

A call to procedure expect receives as actual parameter a set E of terminal words of lengths at most 1. This set thus may also contain the element ε. If neither the next input symbol nor ε is contained in the set E of expected symbols, then expect outputs an error message and exits the syntactic analysis by calling the procedure exit. This treatment of errors is clearly unsatisfactory. Instead, we expect the parser to continue despite the error, to try to continue at a suitable point and to also process the remaining part of the input. In this way, a single run of the parser may possibly detect several errors. In the following section we therefore present an extension of the procedure expect (and the function generate) that enables this behavior.

The parser is started by means of the procedure parse that initially establishes an invariant that is maintained throughout the run of the parser:

- the global variable *next* always contains the next input symbol (1).
- Whenever the code for analyzing a regular expression r is going to be executed, a call to expect has verified that the next input symbol is contained in the first_1-set of r (2).

```
void parse() {
    $next \leftarrow$ scan();            // (1)
    expect($\mathsf{first}_1(S)$);       // (2)
    $S$();
    expect({#});
}
```

For a nonterminal symbol $A \in V_N$, the procedure $A()$ is defined by:

```
void  $A$() {
          generate $r$
}
```

given that $p(A) = r$ is the right side for A in p. For the regular subexpressions of right sides, we define:

```
generate $(r_1 \cdots r_k)$        $\equiv$  generate $r_1$
                                   expect($\mathsf{first}_1(r_2)$)
                                   generate $r_2$
                                   ⋮
                                   expect($\mathsf{first}_1(r_k)$)
                                   generate $r_k$
generate $(r_1 \mid \ldots \mid r_k)$  $\equiv$  switch ($next$) {
                                   labels($\mathsf{first}_1(r_1)$)   generate $r_1$   break;
                                   ⋮
                                   labels($\mathsf{first}_1(r_k)$)   generate $r_k$   break;
                                   }
labels $\{a_1, \ldots, a_m\}$      $\equiv$  label $a_1$ : ... label $a_m$ :
label $a$                          $\equiv$  case $a$
label $\varepsilon$                $\equiv$  default
generate $r^*$                     $\equiv$  while (next $\in$ eff($r$)) {
                                       generate $r$
                                   }
generate $\varepsilon$             $\equiv$  ;
generate $a$                       $\equiv$  consume();
generate $A$                       $\equiv$  $A$();
```

for $a \in V_T$ and $A \in V_N$. We use the function consume() as an abbreviation for *next* $\leftarrow$ scan();. The procedure $A()$ corresponding to the nonterminal A is meant to identify words that are derived from A. When $A()$ is called, the first input symbol has already been provided by the function scan of the joint scanner/screener. If the procedure $A()$ has found a word for the nonterminal A and returns, the input symbol following this word has already been read and stored in the global *next*.

Example 3.3.11 For the extended expression grammar G_e of Example 3.3.8, the following parser is obtained (the procedures expect and parse have been omitted).

```
void   $S$() { $E$(); }
void   $E$() {
       $T$();
       expect({$\varepsilon$, $\boxed{+}$, $\boxed{-}$});
       while ($next \in \{\boxed{+}, \boxed{-}\}$)   {
             switch ($next$)   {
               case $\boxed{+}$ :   consume(); break;
               case $\boxed{-}$ :   consume(); break;
             }
           expect({Id, $\boxed{(}$});
           $T$();
       }
}

void   $T$() {
       $F$();
       expect({$\varepsilon$, $\boxed{*}$, $\boxed{/}$});
       while ($next \in \{\boxed{*}, \boxed{/}\}$)   {
             switch ($next$)   {
               case $\boxed{*}$ :   consume(); break;
               case $\boxed{/}$ :   consume(); break;
             }
           expect({Id, $\boxed{(}$});
           $F$();
       }
}

    void   $F$() {
           switch ($next$)   {
             case $\boxed{\text{Id}}$ :   consume(); break;
             case ( :      consume();
                           expect({$\boxed{\text{Id}}$, $\boxed{(}$});
                           $E$();
                           expect({$\boxed{)}$});
                           consume();
                           break;
           }
    }
```

□

3.3.7 Tree Generation by Means of *Recursive-Descent* Parsers

The *recursiver-descent* parser for a right-regular CFG G, as described so far, succeeds if the given input is syntactically correct and gives an error message otherwise. The parsing procedures can, however, be modified in such a way that every call $A()$ for a correct input returns a representation of the syntax tree for the input w.r.t. to the grammar $\langle G \rangle$. Instead, they can also be instrumented so that they return tree representations that are better suited to later applications.

Example 3.3.12 Consider again the grammar G_e from Example 3.3.8, and for that grammar the parsing function for the nonterminal E. Let us assume that every node of the syntax tree is goint to be represented by an object of the class Tree. For the nonterminal E, we define:

```
Tree   $E()$ {
          Tree $l \leftarrow T()$;
          expect($\{\varepsilon, \boxed{+}, \boxed{-}\}$);
          while ($next \in \{\boxed{+}, \boxed{-}\}$)   {
                switch ($next$)   {
                  case $\boxed{+}$ :  consume();
                                   $op \leftarrow$ PLUS;
                                   break;
                  case $\boxed{-}$ :  consume();
                                   $op \leftarrow$ MINUS;
                                   break;
                }
                expect($\{$Id, $\boxed{(}\}$);
                Tree $r \leftarrow T()$;
                $l \leftarrow$ new Tree($op, l, r$);
          }
          return $l$;
}
```

The function E builds up tree representations for sequences of additions and subtractions. The labels of the case distinction, which did not play any particular role in the last example, are now used for selecting the right operator for the tree node. In principle, any shape of the abstract syntax tree can be chosen when processing the sequence corresponding to a regular right side. Alternatively, all tree nodes of the sequence could be collected into a plain list (cf. Exercise 15). Or a sequence is returned that is nested to the right, since the processed operators associate to the right. □

3.3.8 Error Handling within $RLL(1)$-Parsers

$RLL(1)$-parsers have the property of the *extendible prefix*: every prefix of the input accepted by an $RLL(1)$-parser can be extended in at least one way to a sentence of the language. Although parsers generally may only detect symptoms of errors and not their causes, this property suggests not to perform corrections within the part of the input which has already been parsed. Instead the parser may modify or ignore some input symbols until a configuration is reached from where the remaining input can be parsed. The method that we propose now tries by skipping a prefix of the remaining input to reach a contents of the pushdown such that the analysis can be continued.

An obvious idea to this end is to search for a closing bracket or a separator for the current nonterminal and to ignore all input symbols inbetween. If no such symbol is found but instead a meaningful end symbol for another nonterminal, then the entries in the pushdown are popped until the corresponding nonterminal occurs on top of the pushdown. This means in C or similar languages, for instance:

- during the analysis of an assignment to search for a semicolon;
- during the analysis of a declaration for commas or semicolons;
- during the analysis of conditionals for the keyword else;
- and when analyzing a block starting with an opening bracket { for a closing bracket }.

Such a *panic mode*, however, has several disadvantages.

Even if the expected symbol occurs in the program, a longer sequence of words may be skipped until the symbol is found. If the symbol does not occur or does not belong to the current incarnation of the current nonterminal, the parser generally is doomed.

Our error handling therefore refines the basic idea. During the syntactic analysis, the *recursive-descent* parser maintains a set T of *anchor terminals*. This anchor set consists of terminals where the parsing mode may be recovered. This means that they occur in the right context of one of the subexpressions or productions that are currently processed. The anchor set is not static, but dynamically adapted to the current parser configuration. Consider, for instance, the parser for an *if*-statement in C:

```
void ⟨if_stat⟩ () {
                                        consume();
        expect($\{\boxed{(}\}$);                  consume();
        expect($first_1(E)$);             $E()$;
        expect($\{\boxed{)}\}$);                  consume();
        expect($first_1(\langle stat\rangle)$);       $\langle stat\rangle$ ();
        expect($\{else, \varepsilon\}$)            switch ($next$) {
```

```
                    case else :   consume();
                                  expect(first$_1$($\langle stat \rangle$));   $\langle stat \rangle$ ();
                                  break;
                    default :
                  }
        }
```

If an error occurs during the analysis of the condition, then parsing may continue at the closing bracket following the condition. In the same way, however, also a first terminal of a statement or an else could be used for recovery. After the closing bracket has been accepted, it is no longer available for future recoveries. Therefore, it must be removed from the anchor set. Accordingly, the *if*-parser passes a set of terminals where parsing can be recovered after an error, to the nonterminal parsers of the right side of ⟨if_stat⟩.

Before revealing how the anchor sets are determined, we first discuss how they are used. This provides us with an intuition how they should be constructed. The main task of error handling is realized by a modified version of procedure expect:

```
void expect(set ⟨string⟩ ⟨terminal⟩ $E$, set ⟨terminal⟩ $anc$) {
     bool $skipped\_input \leftarrow$ discard_until($E \setminus \{\varepsilon\} \cup anc$);
     if $(\neg in\_error\_mode \wedge (skipped\_input \vee \{next, \varepsilon\} \cap E = \emptyset))$ {
          $in\_error\_mode \leftarrow$ true;
          $output(\ldots)$;
     }
}
```

The new version of expect receives the actual anchor set in the extra parameter *anc*. First, expect skips input symbols until the symbol is either expected or an anchor terminal. The skipped input can neither be consumed (since it is not expected), nor used for recovery. If input has been skipped or the current input symbol is an anchor terminal which is *not* expected (the intersection of E and *anc* need not be empty!), then an error message is issued and the parser switches to the error mode. The error mode is necessary in order to suppress the consumption of unexpected input symbols.

Example 3.3.13

Consider the example of a syntacticly incorrect C program, where the programmer has omitted the right operand of the comparison together with the closing bracket of the if:

if (Id < while ...

During the syntactic analysis of the *if* condition, the first symbols of the nonterminal S which analyzes statements are assumed to be contained in the anchor set. This means that the symbol while is well suited for recovery and allows it to continue with the analysis of the *if* statement. The call to expect by which the analysis of the right side of the expression is started expects a terminal symbol from the set $\mathsf{first}_1(F)$ (see Example 3.3.11), but finds while. Since while is not expected, but is contained in the anchor set, the parser switches into the error mode without skipping any input. After return from the analysis of E, the procedure $\langle if\_stat\rangle$ expects a closing bracket) which, however, is missing, since the next input symbol is still while. Having switched into the error mode allows it to suppress the follow-up error message by this second call to expect. The error mode is active until expect identifies an expected symbol without having to skip input. □

The parser may leave the error mode, when it may consume an expected symbol. This is implemented by means of the procedure consume:

```
void consume($terminal\ a$) {
    if ($next = a$) {
        $in\_error\_mode \leftarrow$ false;
        $next \leftarrow scan()$;
    }
}
```

The input is only advanced if it is the expected terminal which is consumed. Otherwise, consume does nothing. In case of an error where the next input symbol is not expected but is contained in the anchor set, this behavior enables the parser to continue until it reaches a position where the symbol is indeed expected and consumable.

In the following, the generator schemes of the parsers are extended by a computation of the anchor sets. Of particular interest is the code generated for a sequence $(r_1 \cdots r_k)$ of regular expressions. The parser for r_i receives the union of the first symbols of $r_{i+1}, \ldots, r_k$. This enables the parser to recover at every symbol to the right of r_i. At alternatives, the anchor set is not modified. Each anchor symbol of the whole alternative is also an anchor symbol of each individual case. At a regular expression r^*, recovery may proceed to the right of r^* as well as before r^* itself.

```
generate $(r_1 \cdots r_k)\ t$          $\equiv$  generate $r_1\ t_1$
                                         expect($\mathsf{first}_1(r_2), t_2$);
                                         generate $r_2\ t_2$
                                         ⋮
                                         expect($\mathsf{first}_1(r_k), t_k$);
                                         generate $r_k\ t_k$
                                         where $t_i = \mathsf{eff}(r_{i+1}) \cup \ldots \cup \mathsf{eff}(r_k) \cup t$
generate $(r_1 \mid \ldots \mid r_k)\ t$    $\equiv$  switch ($next$) {
                                         labels($\mathsf{first}_1(r_1)$)   generate $r_1\ t$   break;
                                         ⋮
                                         labels($\mathsf{first}_1(r_k)$)   generate $r_k\ t$   break;
                                         }
```

```
labels $\{a_1, \ldots, a_m\}$   $\equiv$  label $a_1$ : ... label $a_m$ :
label $a$                       $\equiv$  **case** $a$
label $\varepsilon$             $\equiv$  **default**
generate $r^*$ $t$              $\equiv$  **while** ($next \in$ eff($r$)) {
                                               generate $r$ ($t \cup$ eff($r$))
                                               expect(eff($r$), $t \cup$ eff($r$));
                                          }
generate $\varepsilon$ $t$      $\equiv$  ;
generate $a$ $t$                $\equiv$  consume($a$);
generate $A$ $t$                $\equiv$  $A(t)$;
```

The procedures for nonterminals A now also receive a parameter *anc* in which the actual anchor set is passed. Accordingly, the code for the procedure of a nonterminal A with right side r is changed to:

```
**void** $A$(**set** $\langle terminal \rangle$ $anc$) {
     generate $r$ $anc$
}
```

where the procedure parse passes the anchor set $\{\#\}$ to the start symbol of the grammar.

Example 3.3.14 The parser for an *if* statement in C including error handling now looks as follows:

```
**void** $\langle if\_stat \rangle$ (**set** $\langle terminal \rangle$ $anc$) {
                                                  consume(if);
    expect($\{\boxed{(}\}, anc$
      $\cup$ first$_1(E) \cup \{\boxed{)}\}$
      $\cup$ first$_1(\langle$stat$\rangle) \cup \{$else$\}$);          consume($\boxed{(}$);
    expect(first$_1(E), anc \cup \{\boxed{)}\}$
      $\cup$ first$_1(\langle$stat$\rangle) \cup \{$else$\}$);          $E(anc \cup \{\boxed{)}\} \cup$ first$_1(\langle$stat$\rangle) \cup \{$else$\}$);
    expect($\{\boxed{)}\}, anc$
      $\cup$ first$_1(\langle$stat$\rangle) \cup \{$else$\}$);          consume($\boxed{)}$);
    expect(first$_1(\langle$stat$\rangle), anc \cup \{$else$\}$);    $\langle$stat$\rangle$ ($anc \cup \{$else$\}$);
    expect($\{$else$, \varepsilon\}, anc$);                      **switch** ($next$) {
                                                   **case** else :  consume(else);
                                                                   expect(first$_1(\langle$stat$\rangle), anc$);
                                                                   $\langle$stat$\rangle$ ($anc$);
                                                                   **break**;
                                                   **default** :
                                                   }
}
```

□

3.4 ***Bottom-up*** Syntax Analysis

3.4.1 Introduction

Bottom-up parsers read their input like *top-down* parsers from left to right. They are pushdown automata that can essentially perform two kinds of operations:

- Read the next input symbol (*shift*), and
- Reduce the right side of a production $X \to \alpha$ at the top of the pushdown by the left side X of the production (*reduce*).

Because of these operations they are called *shift-reduce* parsers. *Shift-reduce* parsers are right parsers; they output the application of a production when they do a reduction. Since *shift-reduce* parsers always reduce at the top of the pushdown, the result of the successful analysis of an input word is a rightmost derivation in reverse order.

A *shift-reduce* parser must never miss a *required* reduction, that is, cover it in the pushdown by newly read input symbols. A reduction is *required*, if no rightmost derivation from the start symbol is possible without it. A right side covered by an input symbol will never reappear at the top of the pushdown and can therefore never be reduced. A right side at the top of the pushdown that must be reduced to obtain a derivation is called a *handle*.

Not all occurrences of right sides that appear at the top of the pushdown are handles. Some reductions when performed at the top of the pushdown lead into dead ends, that is, they cannot be continued to a reverse rightmost derivation although the input is correct.

Example 3.4.1 Let G_0 be again the grammar for arithmetic expressions with the productions:

$$\begin{array}{lcl} S & \to & E \\ E & \to & E + T \mid T \\ T & \to & T * F \mid F \\ F & \to & (E) \mid \mathsf{Id} \end{array}$$

Table 3.5 shows a successful *bottom-up* analysis of the word $\mathsf{Id} * \mathsf{Id}$ of G_0. The third column lists actions that were also possible, but would lead into dead ends. In the third step, the parser would miss a required reduction. In the other two steps, the alternative reductions would lead into dead ends, that is, not to right sentential-forms. □

Bottom-up parsers construct the parse tree *bottom up*. They start with the leaf word of the parse tree, the input word, and construct for ever larger parts of the read input subtrees of the parse tree: upon reduction by a production $X \to \alpha$, the subtrees for the right side α are attached below a newly created node for X. The analysis is successful if a parse tree has been constructed for the whole input word whose root is labeled with the start symbol of the grammar.

Figure 3.13 shows some snapshots during the construction of the parse tree according to the derivation shown in Table 3.5. The tree on the left contains all nodes

Table 3.5 A successful analysis of the word Id $*$ Id together with potential dead ends

Pushdown	Input	Erroneous alternative actions
	Id $*$ Id	
Id	$*$ Id	
F	$*$ Id	Reading of $*$ misses a required reduction
T	$*$ Id	Reduction of T to E leads into a dead end
$T\ *$	Id	
$T *$ Id		
$T * F$		Reduction of F to T leads into a dead end
T		
E		
S		

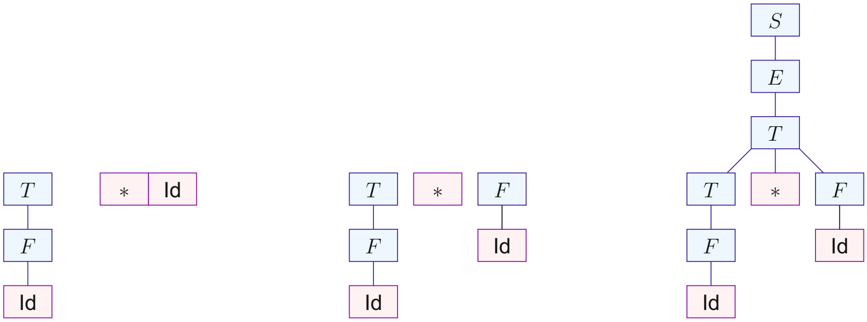


Fig. 3.13 Construction of the parse tree after reading the first symbol, Id, together with the remaining input, before the reduction of the handle $T\ *\ F$, and the complete parse tree

that can be created when the input Id has been read. The sequence of three trees in the middle represents the state before the handle $T\ *\ F$ is being reduced, while the tree on the right shows the complete parse tree.

3.4.2 $LR(k)$-Parsers

This section presents the most powerful deterministic method that works *bottom-up*, namely, $LR(k)$ analysis. The letter L says that the parsers of this class read their input from left to right, The R characterizes them as Right parsers, while k is the length of the considered look-ahead.

We start again with the IPDA P_G for a CFG G and transform it into a *shift-reduce* parser. Let us recall what we do in the case of *top-down* analysis. Sets of look-ahead words are computed from the grammar, which are used to select the right alternative for a nonterminal at *expansion transitions* of P_G. So, the $LL(k)$-parser decides about the alternative for a nonterminal at the earliest possible time, when the nonterminal has to be expanded. $LR(k)$-parsers follow a different strategy; they pursue *all* possibilities to expand and to read in *parallel*.

A decision has to be taken when one of the possibilities to continue asks for a reduction. What is there to decide? There can be several productions by which to reduce, and a shift can be possible in addition to a reduction. The parser uses the next k symbols to take its decision.

In this section, first an $LR(0)$-parser is developed, which does not yet take any look-ahead into account. Section 3.4.3 presents the *canonical* $LR(k)$-parser. In Sect. 3.4.3, less powerful variants of $LR(k)$ are described, which are often powerful enough for practice. Finally, Sect. 3.4.4 describes a error recovery method for $LR(k)$. Note that all CFGs are assumed to be reduced of nonproductive and unreachable nonterminals and extended by a new start symbol.

The Characteristic Finite Automaton to a Context-free Grammar

Our goal is to realize a PDA that pursues all potential expansion and read transitions of the IPDA in parallel and only at reduction decides which production is the one to select. We define the *characteristic* finite automaton $\mathsf{char}(G)$ to a reduced CFG G. The states of the characteristic finite automaton $\mathsf{char}(G)$ are the items $[A \to \alpha.\beta]$ of the grammar G, that is, the states of the IPDA P_G. The automaton $\mathsf{char}(G)$ should reach a state $[X \to \alpha.\beta]$ with a word γ if and only if the same item can be reached by the IPDA with a pushdown store whose *history* equals γ. Accordingly, the characteristic automaton reads the concatenation of the prefixes of productions which have already been processed by the IPDA. Since these prefixes may contain both terminal and nonterminal symbols, the set of input symbols of the characteristic finite automaton $\mathsf{char}(G)$ is given by $V_T \cup V_N$. Its initial state is the start item $[S' \to .S]$ of the IPDA P_G. The final states of the characteristic finite automaton are the complete items $[X \to \alpha.]$. Such a final state signals that the word just read represents the history of a pushdown store of the IPDA in which a reduction with production $A \to \alpha$ can be performed. The transition relation Δ of the characteristic finite automaton consists of the transitions:

$$\begin{array}{lll} ([X \to \alpha.Y\beta], \varepsilon, [Y \to .\gamma]) & \text{for} & X \to \alpha Y\beta \in P, \quad Y \to \gamma \in P \\ ([X \to \alpha.Y\beta], Y, [X \to \alpha Y.\beta]) & \text{for} & X \to \alpha Y\beta \in P, \quad Y \in V_N \cup V_T \end{array}$$

Reading a terminal symbols a in $\mathsf{char}(G)$ corresponds to a *shift* transition of the IPDA under a. ε transitions of $\mathsf{char}(G)$ correspond to the expansion transitions of the IPDA, while reading a nonterminal symbol X corresponds to shifting the dot after a reduction for a production with left side X.

Example 3.4.2 Let G_0 again be the grammar for arithmetic expressions with the productions

$$\begin{array}{lcl} S & \to & E \\ E & \to & E + T \mid T \\ T & \to & T * F \mid F \\ F & \to & (E) \mid \mathsf{Id} \end{array}$$

Figure 3.14 shows the characteristic finite automaton to grammar G_0. □

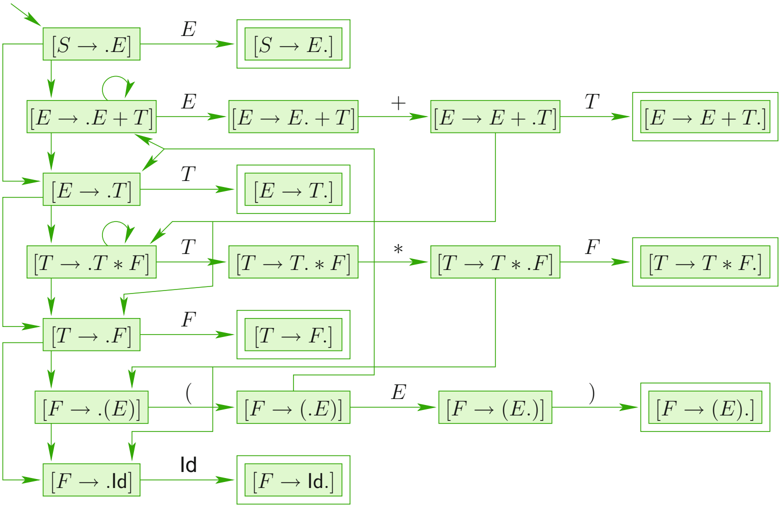


Fig. 3.14 The characteristic finite automaton $\mathsf{char}(G_0)$ for the grammar G_0

The following theorem clarifies the exact relation between the characteristic finite automaton and the IPDA:

Theorem 3.4.1 Let G be a CFG and $\gamma \in (V_T \cup V_N)^*$. The following three statements are equivalent:

1. There exists a computation $([S' \to .S], \gamma) \vdash^*_{\mathsf{char}(G)} ([A \to \alpha.\beta], \varepsilon)$ of the characteristic finite automaton $\mathsf{char}(G)$.
2. There exists a computation $(\rho\,[A \to \alpha.\beta], w) \vdash^*_{P_G} ([S' \to S.], \varepsilon)$ of the IPDA P_G such that $\gamma = \mathsf{hist}(\rho)\,\alpha$ holds.
3. There exists a rightmost derivation $S' \overset{*}{\underset{rm}{\Longrightarrow}} \gamma' A w \underset{rm}{\Longrightarrow} \gamma' \alpha\beta w$ with $\gamma = \gamma'\alpha$.

□

The equivalence of statements (1) and (2) means that words that lead to an item of the characteristic finite automaton $\mathsf{char}(G)$ are exactly the histories of pushdown contents of the IPDA P_G whose topmost symbol is this item and from which P_G can reach one of its final states assuming appropriate input w. The equivalence of statements (2) and (3) means that an accepting computation of the IPDA for an input word w that starts with a pushdown contents ρ corresponds to a rightmost derivation that leads to a sentential form αw where α is the history of the pushdown contents ρ.

Before proving Theorem 3.4.1, we introduce some terminology. For a rightmost derivation

$$S' \overset{*}{\underset{rm}{\Longrightarrow}} \gamma' A v \underset{rm}{\Longrightarrow} \gamma \alpha v$$

and a production $A \rightarrow \alpha$ we call α the *handle* of the right sentential-form $\gamma\alpha v$. If the right side $\alpha = \alpha'\beta$, the prefix $\gamma = \gamma'\alpha'$ is called a *reliable prefix* of G for the item $[A \rightarrow \alpha'.\beta]$. The item $[A \rightarrow \alpha.\beta]$ is *valid* for γ. Theorem 3.4.1, thus, means, that the set of words under which the characteristic finite automaton reaches an item $[A \rightarrow \alpha'.\beta]$ is exactly the set of reliable prefixes for this item.

Example 3.4.3 For the grammar G_0 we have:

Right sentential-form	Handle	Reliable prefixess	Reason
$E + F$	F	$E,\ E+,\ E+F$	$S \underset{rm}{\Longrightarrow} E \underset{rm}{\Longrightarrow} E+T \underset{rm}{\Longrightarrow} E+F$
$T * \mathsf{Id}$	Id	$T,\ T*,\ T*\mathsf{Id}$	$S \overset{3}{\underset{rm}{\Longrightarrow}} T*F \underset{rm}{\Longrightarrow} T*\mathsf{Id}$

□

In an unambiguous grammar, the handle of a right sentential-form is the uniquely determined word that the *bottom-up* parser should replace by a nonterminal in the next reduction step to arrive at a rightmost derivation. A reliable prefix is a prefix of a right sentential-form that does not properly extend beyond the handle.

Example 3.4.4 We give two reliable prefixes of G_0 and some items that are valid for them.

Reliable prefix	Valid item	Reason
$E+$	$[E \rightarrow E + .T]$	$S \underset{rm}{\Longrightarrow} E \underset{rm}{\Longrightarrow} E+T$
	$[T \rightarrow .F]$	$S \overset{*}{\underset{rm}{\Longrightarrow}} E+T \underset{rm}{\Longrightarrow} E+F$
	$[F \rightarrow .\mathsf{Id}]$	$S \overset{*}{\underset{rm}{\Longrightarrow}} E+F \underset{rm}{\Longrightarrow} E+\mathsf{Id}$
$(E+($	$[F \rightarrow (.E)]$	$S \overset{*}{\underset{rm}{\Longrightarrow}} (E+F) \underset{rm}{\Longrightarrow} (E+(E))$
	$[T \rightarrow .F]$	$S \overset{*}{\underset{rm}{\Longrightarrow}} (E+(.T) \underset{rm}{\Longrightarrow} (E+(F))$
	$[F \rightarrow .\mathsf{Id}]$	$S \overset{*}{\underset{rm}{\Longrightarrow}} (E+(F) \underset{rm}{\Longrightarrow} (E+(\mathsf{Id}))$

□

If during the construction of a rightmost derivation for a word, the prefix u of the word is reduced to a reliable prefix γ, then each item $[X \rightarrow \alpha.\beta]$, valid for γ, describes one possible interpretation of the analysis situation. Thus, there is a rightmost derivation in which γ is prefix of a right sentential-form and $X \rightarrow \alpha\beta$ is one of the possibly just processed productions. All such productions are candidates for later reductions.

Consider the rightmost derivation

$$S' \overset{*}{\underset{rm}{\Longrightarrow}} \gamma A w \underset{rm}{\Longrightarrow} \gamma\alpha\beta w$$

It should be extended to a rightmost derivation of a terminal word. This requires that

1. β is derived to a terminal word v, and after that,
2. α is derived to a terminal word u.

Altogether,

$$S' \overset{*}{\underset{rm}{\Longrightarrow}} \gamma A w \underset{rm}{\Longrightarrow} \gamma\alpha\beta w \overset{*}{\underset{rm}{\Longrightarrow}} \gamma\alpha v w \overset{*}{\underset{rm}{\Longrightarrow}} \gamma u v w \overset{*}{\underset{rm}{\Longrightarrow}} x u v w$$

We now consider this rightmost derivation in the direction of reduction, that is, in the direction in which it is constructed by a *bottom-up* parser. First, x is reduced to γ in a number of steps, then u to α, then v to β. The valid item $[A \to \alpha.\beta]$ for the reliable prefix $\gamma\alpha$ describes the analysis situation in which the reduction of u to α has already been done, while the reduction of v to β has not yet started. A possible long-range goal in this situation is the application of the production $X \to \alpha\beta$.

We come back to the question of which language is accepted by the characteristic finite automaton of P_G. Theorem 3.4.1 says that by reading a reliable prefix $\mathsf{char}(G)$, will enter a state that is a valid item for this prefix. Final states, i.e., complete items, are only valid for reliable prefixes where a reduction is possible at their ends.

Proof of Theorem 3.4.1 We do a circular proof (1) ⇒ (2) ⇒ (3) ⇒ (1). Let us first assume that $([S' \to .S], \gamma) \vdash^*_{\mathsf{char}(G)} ([A \to \alpha.\beta], \varepsilon)$. By induction over the number n of ε transitions we construct a rightmost derivation $S' \overset{rm}{\underset{*}{\Longrightarrow}} \gamma A w \overset{rm}{\Longrightarrow} \gamma\alpha\beta w$.

If $n = 0$ holds then $\gamma = \varepsilon$ and $[A \to \alpha.\beta] = [S' \to .S]$ also hold. Since $S' \overset{rm}{\underset{*}{\Longrightarrow}} S'$ holds, the claim holds in this case. If $n > 0$ holds we consider the last ε transition. The computation of the characteristic finite automaton can be decomposed into

$$\begin{aligned}([S' \to .S], \gamma) &\vdash^*_{\mathsf{char}(G)} ([X \to \alpha'.A\beta'], \varepsilon) \vdash_{\mathsf{char}(G)} ([A \to .\alpha\beta], \alpha)\\ &\vdash^*_{\mathsf{char}(G)} ([A \to \alpha.\beta], \varepsilon)\end{aligned}$$

where $\gamma = \gamma'\alpha$. According to the induction hypothesis there exists a rightmost derivation

$$S' \overset{rm}{\underset{*}{\Longrightarrow}} \gamma'' X w' \overset{rm}{\Longrightarrow} \gamma''\alpha' A \beta' w'$$

with $\gamma' = \gamma''\alpha'$. Since the grammar G is reduced, there also exists a rightmost derivation $\beta' \overset{rm}{\underset{*}{\Longrightarrow}} v$. We therefore have

$$S' \overset{rm}{\underset{*}{\Longrightarrow}} \gamma' A v w' \overset{rm}{\Longrightarrow} \gamma'\alpha\beta w$$

with $w = vw'$. This proves the direction (1) ⇒ (2).

Let us now assume that we have a rightmost derivation $S' \overset{rm}{\underset{*}{\Longrightarrow}} \gamma' A w \overset{rm}{\Longrightarrow} \gamma'\alpha\beta w$. This derivation can be decomposed into

$$\begin{aligned} S' &\overset{rm}{\Longrightarrow} \alpha_1 X_1 \beta_1 \overset{rm}{\underset{*}{\Longrightarrow}} \alpha_1 X_1 v_1 \overset{rm}{\underset{*}{\Longrightarrow}} \dots \overset{rm}{\underset{*}{\Longrightarrow}} (\alpha_1 \dots \alpha_n) X_n (v_n \dots v_1) \\ &\overset{rm}{\Longrightarrow} (\alpha_1 \dots \alpha_n)\alpha\beta(v_n \dots v_1) \end{aligned}$$

for $X_n = A$. We can prove by induction over n that $(\rho, vw) \vdash_{K_G}^{*} ([S' \to S.], \varepsilon)$ holds for

$$\begin{array}{rcl} \rho & = & [S' \to \alpha_1.X_1\beta_1] \dots [X_{n-1} \to \alpha_n.X_n\beta_n] \\ w & = & v v_n \dots v_1 \end{array}$$

as long as $\beta \underset{rm}{\overset{*}{\Longrightarrow}} v$, $\alpha_1 = \beta_1 = \varepsilon$ and $X_1 = S$. This proves the direction (2) $\Rightarrow$ (3).

For the last implication we consider a pushdown store $\rho = \rho'\ [A \to \alpha.\beta]$ with $(\rho, w) \vdash_{K_G}^{*} ([S' \to S.], \varepsilon)$. We first convince ourselves by induction over the number of transitions in such a computation that ρ' must be of the form:

$$\rho' = [S' \to \alpha_1.X_1\beta_1] \dots [X_{n-1} \to \alpha_n.X_n\beta_n]$$

for some $n \geq 0$ and $X_n = A$. By induction over n follows that $([S' \to .S], \gamma) \vdash_{\mathsf{char}(G)}^{*} ([A \to \alpha.\beta], \varepsilon)$ holds for $\gamma = \alpha_1 \dots \alpha_n \alpha$. From $\gamma = \mathsf{hist}(\rho)$ follows claim 1. This completes the proof. □

The Canonical $LR(0)$-Automaton

In Chapt. 2, an algorithm has been presented which takes an nondeterministic finite automaton and constructs an equivalent DFA. This DFA pursues all paths in parallel which the nondeterministic automaton could take for a given input. Its states are sets of states of the nondeterministic automaton. This *subset construction* is now applied to the characteristic finite automaton $\mathsf{char}(G)$ of a CFG G. The resulting DFA is called the *canonical* $LR(0)$-automaton for G and denoted by $LR_0(G)$.

Example 3.4.5 The canonical $LR(0)$-automaton for the CFG G_0 of Example 3.2.2 is obtained by applying the subset construction to the characteristic FA $\mathsf{char}(G_0)$ of

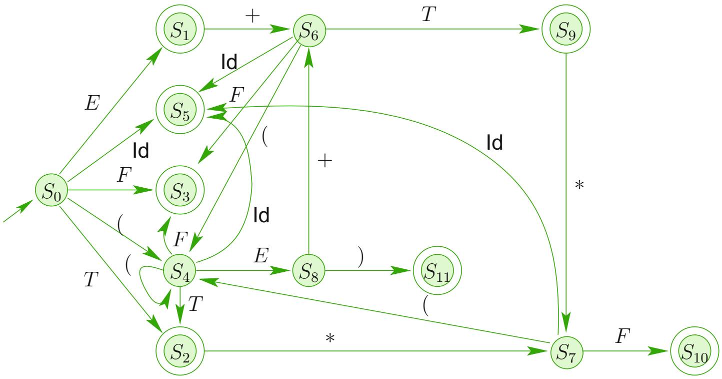


Fig. 3.15 The transition diagram of the $LR(0)$-automaton for the grammar G_0 obtained from the characteristic finite automaton $\mathsf{char}(G_0)$ in Fig. 3.14. The error state $S_{12} = \emptyset$ and all transitions into it are omitted

Fig. 3.14. It is shown in Fig. 3.15. It states are:

$$\begin{aligned}
S_0 = \{\, & [S \to .E], \\
& [E \to .E+T], \\
& [E \to .T], \\
& [T \to .T*F], \\
& [T \to .F], \\
& [F \to .(E)], \\
& [F \to .\mathsf{Id}]\,\} \\
S_1 = \{\, & [S \to E.], \\
& [E \to E.+T]\,\} \\
S_2 = \{\, & [E \to T.], \\
& [T \to T.*F]\,\} \\
S_3 = \{\, & [T \to F.]\,\}
\end{aligned}$$

$$\begin{aligned}
S_4 = \{\, & [F \to (.E)], \\
& [E \to .E+T], \\
& [E \to .T], \\
& [T \to .T*F] \\
& [T \to .F] \\
& [F \to .(E)] \\
& [F \to .\mathsf{Id}]\,\} \\
S_5 = \{\, & [F \to \mathsf{Id}.]\,\} \\
S_6 = \{\, & [E \to E+.T], \\
& [T \to .T*F], \\
& [T \to .F], \\
& [F \to .(E)], \\
& [F \to .\mathsf{Id}]\,\}
\end{aligned}$$

$$\begin{aligned}
S_7 &= \{\, [T \to T*.F], \\
& \quad\ \ [F \to .(E)], \\
& \quad\ \ [F \to .\mathsf{Id}]\,\} \\
S_8 &= \{\, [F \to (E.)], \\
& \quad\ \ [E \to E.+T]\,\} \\
S_9 &= \{\, [E \to E+T.], \\
& \quad\ \ [T \to T.*F]\,\} \\
S_{10} &= \{\, [T \to T*F.]\} \\
S_{11} &= \{\, [F \to (E).]\,\} \\
S_{12} &= \emptyset
\end{aligned}$$

□

The canonical $LR(0)$-automaton $LR_0(G)$ to a CFG G has interesting properties. Let $LR_0(G) = (Q_G, V_T \cup V_N, \Delta_G, q_{G,0}, F_G)$, and let $\Delta_G^* : Q_G \times (V_T \cup V_N)^* \to Q_G$ be the lifting of the transition function Δ_G from symbols to words. We then have:

1. $\Delta_G^*(q_{G,0}, \gamma)$ is the set of all items in $\mathcal{I}_G$ for which γ is a reliable prefix.
2. $L(LR_0(G))$ is the set of all reliable prefixes for complete items $[A \to \alpha.] \in \mathcal{I}_G$.

Reliable prefixes are prefixes of right sentential-forms, as they occur during the reduction of an input word. A reduction that will again lead to a right sentential-form can only happen at the right end of this sentential form. An item valid for a reliable prefix describes one possible interpretation of the actual analysis situation.

Example 3.4.6 $E + F$ is a reliable prefix for the grammar G_0. The state $\Delta^*_{G_0}(S_0, E + F) = S_3$ is also reached by the following reliable prefixes:

$$\begin{array}{llll} F\,, \quad (F\,, & ((F\,, & (((F\,, & \ldots \\ T * (F\,, & T * ((F\,, & T * (((F\,, & \ldots \\ E + F\,, & E + (F\,, & E + ((F\,, & \ldots \end{array}$$

The state S_6 in the canonical $LR(0)$-automaton to G_0 contains all valid items for the reliable prefix $E+$, namely the items

$$[E \to E + .T], [T \to .T * F], [T \to .F], [F \to .\mathsf{Id}], [F \to .(E)].$$

For $E+$ a prefix of the right sentential-form $E + T$:

$$\begin{array}{lccccccc} S \underset{rm}{\Longrightarrow} E \underset{rm}{\Longrightarrow} & E + T & \underset{rm}{\Longrightarrow} & E + F & \underset{rm}{\Longrightarrow} & E + \mathsf{Id} \\ & \uparrow & & \uparrow & & \uparrow \\ \text{Valid for instance} & [E \to E + .T] & & [T \to .F] & & [F \to .\mathsf{Id}] \end{array}$$

□

The canonical $LR(0)$-automaton $LR_0(G)$ to a CFG G is a DFA that accepts the set of reliable prefixes to complete items. In this way, it identifies positions for reduction, and therefore offers itself for the construction of a right parser. Instead of items (as the IPDA), this parser stores on its pushdown states of the canonical $LR(0)$-automaton, that is *sets* of items. The underlying PDA P_0 is defined as the tuple $K_0 = (Q_G \cup \{f\}, V_T, \Delta_0, q_{G,0}, \{f\})$. The set of states is the set Q_G of states of the canonical $LR(0)$-automaton $LR_0(G)$, extended by a new state f, the final state. The initial state of P_0 is identical to the initial state $q_{G,0}$ of $LR_0(G)$. The transition relation Δ_0 consists of the following kinds of transitions:

Read: $(q, a, q\; \Delta_G(q, a)) \in \Delta_0$, if $\Delta_G(q, a) \neq \emptyset$. This transition reads the next input symbol a and pushes the successor state q under a onto the pushdown. It can only be taken if at least one item of the form $[X \to \alpha.a\beta]$ is contained in q.

Reduce: $(qq_1 \ldots q_n, \varepsilon, q\; \Delta_G(q, X)) \in \Delta_0$ if $[X \to \alpha.] \in q_n$ holds with $|\alpha| = n$. The complete item $[X \to \alpha.]$ in the topmost pushdown entry signals a potential reduction. As many entries are removed from the top of the pushdown as the length of the right side indicates. After that, the X successor of the new topmost pushdown entry is pushed onto the pushdown. Figure 3.16 shows a part of the transition diagram of a $LR(0)$-automaton $LR_0(G)$ that demonstrates this situation. The α path in the transition diagram corresponds to $|\alpha|$ entries on top of the pushdown. These entries are removed at reduction. The new actual state, previously below these removed entries, has a transition under X, which is now taken.

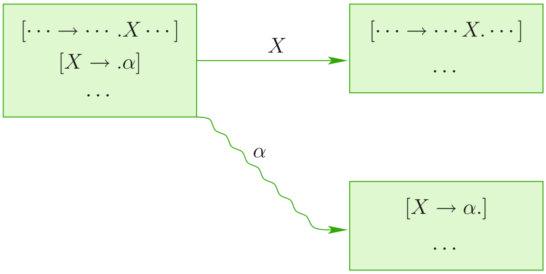


Fig. 3.16 Part of the transition diagram of a canonical $LR(0)$-automaton

Finish: $(q_{G,0}\,q, \varepsilon, f) \in \Delta_0$ if $[S' \to S.] \in q$. This transition is the reduction transition to the production $S' \to S$. The property $[S' \to S.] \in q$ signals that a word was successfully reduced to the start symbol. This transition empties the pushdown and inserts the final state f.

The special case $[X \to .]$ merits special consideration. According to our description, $|\varepsilon| = 0$ topmost pushdown entries need to be removed from the pushdown upon this reduction, and a transition from the new, and old, actual state q under X should be taken, and the state $\Delta_G(q, X)$ is pushed onto the pushdown. This latter reduction transition therefore *extends* the length of the pushdown.

The construction of $LR_0(G)$ guarantees that for each noninitial and nonfinal state q there exists exactly one entry symbol under which the automaton can make a transition into q. The pushdown contents $q_0, \ldots, q_n$ with $q_0 = q_{G,0}$ corresponds therefore to a uniquely determined word $\alpha = X_1 \ldots X_n \in (V_T \cup V_N)^*$ for which $\Delta_G(q_i, X_{i+1}) = q_{i+1}$ holds. This word α is a reliable prefix, and q_n is the set of all items valid for α.

The PDA P_0 just constructed is not necessarily deterministic. There are two kinds of conflicts that cause nondeterminism:

shift-reduce conflict: a state q allows a read transition under a symbol $a \in V_T$ as well as a reduce transition, and

reduce-reduce conflict: a state q permits reduction transitions according to two different productions.

Here, we consider a finish transition as a specific reduce transition. In case of a shift-reduce conflict, the actual state contains at least one item $[X \to \alpha.a\beta]$ and at least one complete item $[Y \to \gamma.]$. In case of a reduce-reduce conflict on the other hand, q contains two different complete items $[Y \to \alpha.]$, $[Z \to \beta.]$. A state q of the $LR(0)$-automaton with one of these properties is called $LR(0)$-*inadequate*. Otherwise, we call q $LR(0)$-*adequate*. We have:

Lemma 3.4.4 For an $LR(0)$-*adequate* state q there are three possibilities:

1. The state q contains no complete item.
2. The state q consists of exactly one complete item $[A \to \alpha.]$;
3. The state q contains exactly one complete item $[A \to .]$, and all noncomplete items in q are of the form $[X \to \alpha.Y\beta]$, where all rightmost derivations for Y

that lead to a terminal word are of the form:

$$Y \underset{rm}{\overset{*}{\Longrightarrow}} Aw \underset{rm}{\Longrightarrow} w$$

for a $w \in V_T^*$. □

Inadequate states of the canonical $LR(0)$-automaton make the PDA P_0 nondeterministic. We obtain a deterministic parser by permitting the parser to look ahead into the remaining input in order to select the correct action in inadequate states.

Example 3.4.7 The states S_1, S_2 and S_9 of the canonical $LR(0)$-automaton in Fig. 3.15 are $LR(0)$-inadequate. In state S_1, the parser can reduce the right side E to the left side S (complete item $[S \rightarrow E.]$) and it can read the terminal symbol $+$ in the input (item $[E \rightarrow E. + T]$). In state S_2 the parser can reduce the right side T to E (complete item $[E \rightarrow T.]$) and it can read the terminal symbol $*$ (item $[T \rightarrow T. * F]$). In state S_9 finally, the parser can reduce the right side $E + T$ to E (complete item $[E \rightarrow E + T.]$), and it can read the terminal symbol $*$ (item $[T \rightarrow T. * F]$). □

Direct Construction of the Canonical $LR(0)$-Automaton

The canonical $LR(0)$-automaton $LR_0(G)$ to a CFG G need not be derived from the characteristic finite automaton char(G) by means of the subset construction. It can also be constructed directly from G. The direct construction requires a function $\Delta_{G,\varepsilon}$ that adds to each set q of items all items that are reachable by ε transitions of the characteristic finite automaton. The set $\Delta_{G,\varepsilon}(q)$ is the least solution of the following equation

$$I = q \cup \{[A \rightarrow .\gamma] \mid \exists X \rightarrow \alpha A\beta \in P : [X \rightarrow \alpha.A\beta] \in I\}$$

Similar to the function closure() of the subset construction, it can be computed by

```
set ⟨item⟩ closure(set ⟨item⟩ q) {
    set ⟨item⟩ result ← q;
    list ⟨item⟩ W ← list_of(q);
    symbol X; string ⟨symbol⟩ α;
    while (W ≠ []) {
        item i ← hd(W); W ← tl(W);
        switch (i) {
         case [_ → _ .X _] : forall (α : (X → α) ∈ P)
                                  if ([X → .α] ∉ result) {
                                      result ← result ∪ {[X → .α]};
                                      W ← [X → .α] :: W;
                                  }
```

```
                default :   break;
              }
          }
          return  result;
      }
```

where V is the set of symbols $V = V_T \cup V_N$. The set Q_G of states and the transition relation Δ_G are computed by first constructing the initial state $q_{G,0} = \Delta_{G,\varepsilon}(\{[S' \to .S]\})$ and then adding successor states and transitions until all successor states are already in the set of constructed states. For an implementation, we specialize the function nextState() of the subset construction:

```
set ⟨item⟩ nextState(set ⟨item⟩ q, symbol X) {
    set ⟨item⟩ $q' \leftarrow \emptyset$;
    nonterminal A; string ⟨symbol⟩ $\alpha, \beta$;
    forall $(A, \alpha, \beta : ([A \to \alpha.X\beta] \in q))$
        $q' \leftarrow q' \cup \{[A \to \alpha X.\beta]\}$;
    return closure($q'$);
    }
```

As in the subset construction, the set of states *states* and the set of transitions *trans* can be computed iteratively:

```
list ⟨set ⟨item⟩⟩  W;
set ⟨item⟩  $q_0 \leftarrow$ closure($\{[S' \to .S]\}$);
states $\leftarrow \{q_0\}$;  $W \leftarrow [q_0]$;
trans $\leftarrow \emptyset$;
set ⟨item⟩  $q, q'$;
while  $(W \neq [])$  {
    $q \leftarrow$ hd($W$);  $W \leftarrow$ tl($W$);
    forall (symbol X)  {
        $q' \leftarrow$ nextState($q, X$);
        trans $\leftarrow$ trans $\cup \{(q, X, q')\}$;
        if  ($q' \notin$ states)  {
            states $\leftarrow$ states $\cup \{q'\}$;
            $W \leftarrow q' :: W$;
        }
    }
}
```

3.4.3 $LR(k)$: Definition, Properties, and Examples

We call a CFG G an $LR(k)$-grammar, if in each of its rightmost derivations $S' = \alpha_0 \underset{rm}{\Longrightarrow} \alpha_1 \underset{rm}{\Longrightarrow} \alpha_2 \cdots \underset{rm}{\Longrightarrow} \alpha_m = v$ and each right sentential-forms α_i occurring in the derivation

- the handle can be localized, and
- the production to be applied can be determined

by considering α_i from the left to at most k symbols following the handle. In an $LR(k)$-grammar, the decomposition of α_i into $\gamma\beta w$ and the determination of $X \to \beta$, such that $\alpha_{i-1} = \gamma X w$ holds is uniquely determined by $\gamma\beta$ and $w|_k$. Formally, we call G an $LR(k)$-grammar if

$$\begin{array}{lll} S' \underset{rm}{\overset{*}{\Longrightarrow}} \alpha X w \underset{rm}{\Longrightarrow} \alpha\beta w & \text{and} & \\ S' \underset{rm}{\overset{*}{\Longrightarrow}} \gamma Y x \underset{rm}{\Longrightarrow} \alpha\beta y & \text{and} & \\ w|_k = y|_k & \text{implies} & \alpha = \gamma \wedge X = Y \wedge x = y. \end{array}$$

Example 3.4.8 Let G be the grammar with the productions

$$S \to A \mid B \qquad A \to aAb \mid 0 \qquad B \to aBbb \mid 1$$

Then $L(G) = \{a^n 0 b^n \mid n \geq 0\} \cup \{a^n 1 b^{2n} \mid n \geq 0\}$. As we know, G is not an $LL(k)$-grammar for any $k \geq 1$. Grammar G is an $LR(0)$-grammar, though.

The right sentential-forms of G have the form

$$S, \quad \underline{A}, \quad \underline{B}, \quad a^n\underline{aAb}b^n, \quad a^n\underline{aBbb}b^{2n}, \quad a^n a\underline{0}bb^n, \quad a^n a\underline{1}bbb^{2n}$$

for $n \geq 0$. The handles are always underlined. Two different possibilities to reduce exist only in the case of right sentential-forms $a^n aAbb^n$ and $a^n aBbbb^{2n}$. The sentential-form $a^n aAbb^n$ can be reduced to $a^n Ab^n$ as well as to $a^n aSbb^n$. The first choice belongs to the rightmost derivation

$$S \underset{rm}{\overset{*}{\Longrightarrow}} a^n Ab^n \underset{rm}{\Longrightarrow} a^n aAbb^n$$

while the second one does not occur in any rightmost derivation. The prefix a^n of $a^n Ab^n$ uniquely determines whether A is the handle, namely in the case $n = 0$, or whether aAb is the handle, namely in the case $n > 0$. The right sentential-forms $a^n Bb^{2n}$ are handled analogously. □

Example 3.4.9 The grammar G_1 with the productions

$$S \to aAc \qquad A \to Abb \mid b$$

and the language $L(G_1) = \{ab^{2n+1}c \mid n \geq 0\}$ is an $LR(0)$-grammar. In a right sentential-form $aAbbb^{2n}c$ only the reduction to $aAb^{2n}c$ is possible as part of a rightmost derivation, which is uniquely determined by the prefix $aAbb$. For the right sentential-form $abb^{2n}c$, b is the handle, which is uniquely determined by the prefix ab. □

Example 3.4.10 The grammar G_2 with the productions

$$S \to aAc \qquad A \to bbA \mid b$$

and the language $L(G_2) = L(G_1)$ is an $LR(1)$-grammar. The critical right sentential-forms have the form $ab^n w$. If $w|_1 = b$, the handle is contained in w; if $w|_1 = c$, the last b in b^n forms the handle. □

Example 3.4.11 The grammar G_3 with the productions

$$S \to aAc \qquad A \to bAb \mid b$$

and the language $L(G_3) = L(G_1)$ is not an $LR(k)$-grammar for any $k \geq 0$. In order to see this, let k be arbitrary, but fixed. Consider the two rightmost derivations

$$\begin{aligned} S &\overset{*}{\underset{rm}{\Longrightarrow}} ab^n Ab^n c \underset{rm}{\Longrightarrow} ab^n bb^n c \\ S &\overset{*}{\underset{rm}{\Longrightarrow}} ab^{n+1} Ab^{n+1} c \underset{rm}{\Longrightarrow} ab^{n+1} bb^{n+1} c \end{aligned}$$

with $n \geq k$. With the names introduced in the definition of $LR(k)$-grammar, we have $\alpha = ab^n, \beta = b, \gamma = ab^{n+1}, w = b^n c, y = b^{n+2} c$. Here $w|_k = y|_k = b^k$. Therefore, $\alpha \neq \gamma$ implies that G_3 cannot be an $LR(k)$-grammar. □

The following theorem clarifies the relation between the definition of a $LR(0)$-grammar and the properties of the canonic $LR(0)$-automaton.

Theorem 3.4.2 A CFG G is an $LR(0)$-grammar if and only if the canonical $LR(0)$-automaton for G has no $LR(0)$-inadequate states.

Proof "$\Rightarrow$" Assume that G is an $LR(0)$-grammar, and that the canonical $LR(0)$-automaton $LR_0(G)$ has a $LR(0)$-inadequate state p.

Case 1: The state p has a *reduce-reduce*-conflict, i.e., p contains two distinct items $[X \to \beta.]$, $[Y \to \delta.]$. Associated to state p is a nonempty set of words that are all reliable prefixes for each of the two items. Let $\gamma = \gamma'\beta$ be one such reliable prefix. Since both items are valid for γ, there are rightmost derivations

$$\begin{aligned} S' &\overset{*}{\underset{rm}{\Longrightarrow}} \gamma' X w \underset{rm}{\Longrightarrow} \gamma'\beta w && \text{and} \\ S' &\overset{*}{\underset{rm}{\Longrightarrow}} \nu Y y \underset{rm}{\Longrightarrow} \nu\delta y && \text{with} \quad \nu\delta = \gamma'\beta = \gamma \end{aligned}$$

This, however, is a contradiction the the $LR(0)$-property of G.

Case 2: state p has a *shift-reduce*-conflict, i.e., p contains items $[X \rightarrow \beta.]$ and $[Y \rightarrow \delta.a\alpha]$. Let γ be a reliable prefix for both items. Since both items are valid for γ, there are rightmost derivations

$$\begin{array}{ll} S' \overset{*}{\underset{rm}{\Longrightarrow}} \gamma' X w \underset{rm}{\Longrightarrow} \gamma'\beta w & \text{and} \\ S' \overset{*}{\underset{rm}{\Longrightarrow}} \nu Y y \underset{rm}{\Longrightarrow} \nu\delta a\alpha y & \text{with} \quad \nu\delta = \gamma'\beta = \gamma \end{array}$$

If $\beta' \in V_T^*$, we immediately obtain a contradiction. Otherwise, there is a rightmost derivation

$$\alpha \overset{*}{\underset{rm}{\Longrightarrow}} v_1 X v_3 \underset{rm}{\Longrightarrow} v_1 v_2 v_3$$

Since $y \neq a v_1 v_2 v_3 y$ holds, the $LR(0)$-property is violated.

"$\Leftarrow$" For a proof of this direction, assume that the canonical $LR(0)$-automaton $LR_0(G)$ has no $LR(0)$-inadequate states, and consider zwei rightmost derivations:

$$\begin{array}{l} S' \overset{*}{\underset{rm}{\Longrightarrow}} \alpha X w \underset{rm}{\Longrightarrow} \alpha\beta w \\ S' \overset{*}{\underset{rm}{\Longrightarrow}} \gamma Y x \underset{rm}{\Longrightarrow} \alpha\beta y \end{array}$$

We have to show that $\alpha = \gamma$, $X = Y$, and $x = y$ hold. Let p the state of the canonical $LR(0)$-automaton after reading $\alpha\beta$. Then p consists of all valid items for $\alpha\beta$ where by assumption, p is $LR(0)$-adequate. We again distinguish two cases.

Case 1: $\beta \neq \varepsilon$. By Lemma 3.4.4, $p = \{[X \rightarrow \beta.]\}$, i.e., $[X \rightarrow \beta.]$ is the only valid item for $\alpha\beta$. But then $\alpha = \gamma$, $X = Y$, and $x = y$ must hold.
Case 2: $\beta = \varepsilon$. Assume that the second rightmost derivation violates the $LR(0)$-condition. Then a further item $[Y \rightarrow \delta.Y'\eta]$ must be contained in p such that $\alpha = \alpha'\delta$. The last application of a production in the second rightmost derivation is the last application of a production in a terminal rightmost derivation for Y'. By Lemma 3.4.4, the second derivation therefore has the form:

$$S' \overset{*}{\underset{rm}{\Longrightarrow}} \alpha'\delta Y' w \overset{*}{\underset{rm}{\Longrightarrow}} \alpha'\delta X v w \underset{rm}{\Longrightarrow} \alpha'\delta v w$$

where $y = vw$. But then $\alpha = \alpha'\delta = \gamma$, $Y = X$ and $x = vw = y$, in contradiction to our assumption. □

Let us summarize. We have seen how to construct the $LR(0)$-automaton $LR_0(G)$ from a given CFG G. This can be done either directly or through the characteristic finite automaton $\mathsf{char}(G)$. From the DFA $LR_0(G)$ a PDA P_0 can be constructed. This PDA P_0 is deterministic if $LR_0(G)$ does not contain $LR(0)$-inadequate states. Theorem 3.4.2 states this is the case if and only if the grammar G is an $LR(0)$-grammar. Thereby, we have obtained a method to generate parsers for $LR(0)$-grammars.

In real life, though, $LR(0)$-grammars are rare. Often look-ahead of length $k > 0$ is required in order to select from the different choices in a parsing situation. In an $LR(0)$-parser, the actual state determines the next action, independently of the next input symbols. $LR(k)$-parsers for $k > 0$ have states consisting of sets of items as well. A different kind of items are used, though, namely $LR(k)$-items. $LR(k)$-items are context-free items, extended by look-ahead words. An $LR(k)$-item is of the form $i = [A \to \alpha.\beta, x]$ for a production $A \to \alpha\beta$ of G and a word $x \in (V_T^k \cup V_T^{<k}\#)$. The context-free item $[A \to \alpha.\beta]$ is called the *core*, the word x the *look-ahead* of the $LR(k)$-item i. The set of $LR(k)$-items of grammar G is written as $\mathcal{I}_{G,k}$. The $LR(k)$-item $[A \to \alpha.\beta, x]$ is *valid* for a reliable prefix γ, if there exists a rightmost derivation

$$S'\# \underset{rm}{\overset{*}{\Longrightarrow}} \gamma' X w\# \underset{rm}{\Longrightarrow} \gamma'\alpha\beta w\#$$

with $x = (w\#)|_k$. A context-free item $[A \to \alpha.\beta]$ can be understood as an $LR(0)$-item that is extended by look-ahead ε.

Example 3.4.12 Consider again grammar G_0. We have:
(1) $[E \to E + .T,)]$ and $[E \to E + .T, +]$
are valid $LR(1)$-items for the prefix $(E+$
(2) $[E \to T., *]$
is not a valid $LR(1)$-item for any reliable prefix.
To see observation (1), consider the two rightmost derivations:

$$S' \underset{rm}{\overset{*}{\Longrightarrow}} (E) \underset{rm}{\Longrightarrow} (E + T)$$
$$S' \underset{rm}{\overset{*}{\Longrightarrow}} (E + \mathsf{Id}) \underset{rm}{\Longrightarrow} (E + T + \mathsf{Id})$$

Observation (2) follows since the subword $E*$ cannot occur in a right sentential-form. □

The folllowing theorem provides a characterization of the $LR(k)$-property based on valid $LR(k)$-items.

Theorem 3.4.3 Let G be a CFG. For a reliable prefix γ let $It(\gamma)$ be the set of $LR(k)$-items of G that are valid for γ.

The grammar G is an $LR(k)$-grammar if and only if for all reliable prefixes γ and all $LR(k)$-items $[A \to \alpha., x] \in It(\gamma)$ the following holds.

1. If there is another $LR(k)$-item $[X \to \delta., y] \in It(\gamma)$, then $x \neq y$.
2. If there is tanother $LR(k)$-item $[X \to \delta.a\beta, y] \in It(\gamma)$, then $x \notin \mathsf{first}_k(a\beta) \odot_k \{y\}$. □

Theorem 3.4.3 suggests to define $LR(k)$-adequate and $LR(k)$-inadequate sets of items also for $k > 0$. Let I be a set of $LR(k)$-items. I has a *reduce-reduce* conflict if there are $LR(k)$-items $[X \to \alpha., x], [Y \to \beta., y] \in I$ with $x = y$. I has a *shift-reduce*-conflict if there are $LR(k)$-items $[X \to \alpha.a\beta, x], [Y \to \gamma., y] \in I$ with

$$y \in \{a\} \odot_k \mathsf{first}_k(\beta) \odot_k \{x\}$$

For $k = 1$ this condition is simplified to $y = a$.

The set I is called $LR(k)$-inadequate, if it has a *reduce-reduce*- or a *shift-reduce*-conflict. Otherwise, we call it $LR(k)$-adequate.

The $LR(k)$-property means that when reading a right sentential-form, a candidate for a reduction together with production to be applied can be uniquely determined by the help of the associated reliable prefixes and the k next symbols of the input. However, if we were to tabulate all combinations of reliable prefixes with words of length k this would be infeasible since, in general, there are infinitely many reliable prefixes. In analogy to our way of dealing with $LR(0)$-grammars one could construct a canonical $LR(k)$-automaton. The canonical $LR(k)$-automaton $LR_k(G)$ is a DFA. Its states are sets of $LR(k)$-items. For each reliable prefix γ the DFA $LR_k(G)$ determines the set of $LR(k)$-items that are valid for γ. Theorem 3.4.3 helps us in our derivation. It says that for an $LR(k)$-grammar, the set of $LR(k)$-items valid for γ together with the look-ahead determines uniquely whether to reduce in the next step, and if so, by which production.

In much the same way as the $LR(0)$-parser stores states of the canonical $LR(0)$-automaton on its pushdown, the $LR(k)$-parser stores states of the canonical $LR(k)$-automaton on its pushdown. The selection of the correct choice of several possible actions of the $LR(k)$-parser is controlled by the *action*-table. This table contains for each combination of state and look-ahead one of the following entries:

shift:	read the next input symbol;
reduce$(X \to \alpha)$:	reduce by production $X \to \alpha$;
error:	report error
accept:	announce successful end of the parser run

A second table, the *goto*-table, contains the representation of the transition function of the canonic $LR(k)$-automaton $LR_k(G)$. It is consulted after a *shift*-action or a *reduce*-action to determine the new state on top of the pushdown. Upon a *shift*, it computes the transition under the read symbol out of the actual state. Upon a reduction by $X \to \alpha$, it gives the transition under X out of the state underneath those pushdown symbols that belong to α.

The $LR(k)$-parser for a grammar G requires a *driver* that interprets the *action*- and *goto*-table. Again, we consider the case $k = 1$. This is, in principle, sufficient because for each language that has an $LR(k)$-grammar and therefore also an $LR(k)$-parser one can construct an $LR(1)$-grammar and consequently also an $LR(1)$-parser. Let us assume that the set of states of the $LR(1)$-parser were Q. One such driver program then is:

```
list ⟨state⟩  pd ← [$q_0$];
terminal  buffer ← scan();
state  q;  nonterminal  X;  string ⟨symbol⟩  $\alpha$;
while  (true)  {
    $q \leftarrow$ hd($pd$);
    switch  ($action[q, buffer]$)  {
     case  shift :               $pd \leftarrow goto[q, buffer] :: pd$;
                                 $buffer \leftarrow$ scan();
                                 break;
     case  reduce($X \to \alpha$) :  output($X \to \alpha$);
                                 $pd \leftarrow$ tl($|\alpha|, pd$);  $q \leftarrow$ hd($pd$);
                                 $pd \leftarrow goto[q, X] :: pd$;
                                 break;
     case  accept :              $pd \leftarrow f ::$ tl($2, pd$);
                                 return accept;
     case  error :               output(″...″);  goto  err;
    }
```

The function **list** ⟨*state*⟩ tl(**int** n, **list** ⟨*state*⟩ s) returns in its second argument the list s with the topmost n elements removed. As with the driver program for $LL(1)$-parsers, in the case of an error, it jumps to a label *err* at which the code for error handling is to be found.

We present three approaches to construct an $LR(1)$-parser for a CFG G. The most general method is the canonical $LR(1)$-method. For each $LR(1)$-grammar G there exists a canonical $LR(1)$-parser. The number of states of this parser can be large. Therefore, other methods were proposed that have state sets of the size of the $LR(0)$-automaton. Of these we consider the $SLR(1)$- and the $LALR(1)$-method.

The given driver program for $LR(1)$-parsers works for all three parsing methods; the driver interprets the *action*- and a *goto*-table, but their contents are computed in different ways. In consequence, the actions for some combinations of state and look-ahead may be different.

Construction of an $LR(1)$-Parser

The $LR(1)$-parser is based on the canonical $LR(1)$-automaton $LR_1(G)$. Its states therefore are sets of $LR(1)$-items. We construct the canonical $LR(1)$-automaton much in the same way as we constructed the canonical $LR(0)$-automaton. The only difference is that $LR(1)$-items are used instead of $LR(0)$-items. This means that the look-ahead symbols need to be computed when the closure of a set q of $LR(1)$-items under ε-transitions is formed. This set is the least solution of the following equation

$$\begin{aligned} I = & q \cup \{[A \to .\gamma, y] \mid \exists X \to \alpha A \beta \in P : [X \to \alpha.A\beta, x] \in I, \\ & y \in \mathsf{first}_1(\beta) \odot_1 \{x\}\} \end{aligned}$$

It is computed by the following function

```
set ⟨$item_1$⟩ closure(set ⟨$item_1$⟩ $q$) {
    set ⟨$item_1$⟩ $result \leftarrow q$;
    list ⟨$item_1$⟩ $W \leftarrow$ list_of($q$);
    nonterminal $X$; string ⟨symbol⟩ $\alpha, \beta$; terminal $x, y$;
    while ($W \neq []$) {
        $item_1$ $i \leftarrow$ hd($W$); $W \leftarrow$ tl($W$);
        switch ($i$) {
        case [_ $\to$ _ $.X\beta, x$] :
                    forall ($\alpha$ : ($X \to \alpha$) $\in P$)
                        forall ($y \in$ first$_1(\beta) \odot_1 \{x\}$)
                            if ([$X \to .\alpha, y$] $\notin$ result) {
                                result $\leftarrow$ result $\cup \{[X \to .\alpha, y]\}$;
                                $W \leftarrow [X \to .\alpha, y] :: W$;
                            }
         default : break;
        }
    }
    return result;
}
```

where V is the set of all symbols, $V = V_T \cup V_N$. The initial state q_0 of $LR_1(G)$ is given by

$$q_0 = \text{closure}(\{[S' \to .S, \#]\})$$

Moreover, a function nextState() is required that computes the successor state to a given set q of $LR(1)$-items and a symbol $X \in V = V_N \cup V_T$. For that, the corresponding function of the construction of $LR_0(G)$ is extended by a calculation on look-ahead symbols:

```
set ⟨$item_1$⟩ nextState(set ⟨$item_1$⟩ $q$, symbol $X$) {
    set ⟨$item_1$⟩ $q' \leftarrow \emptyset$;
    nonterminal $A$; string ⟨symbol⟩ $\alpha, \beta$; terminal $x$;
    forall ($A, \alpha, \beta, x$ : ([$A \to \alpha.X\beta, x$] $\in q$))
        $q' \leftarrow q' \cup \{[A \to \alpha X.\beta, x]\}$;
    return closure($q'$);
    }
```

The set of states and the transition relation of the canonical $LR(1)$-automaton is computed in analogy to the canonical $LR(0)$-automaton. The generator starts with the initial state and an empty set of transitions and adds successors states until all successor states are already contained in the set of computed states. The transition function of the canonical $LR(1)$-automaton gives the *goto*-table of the $LR(1)$-parser.

Let us turn to the construction of the *action*-table of the $LR(1)$-parser. No *reduce-reduce*-conflict occurs in a state q of the canonical $LR(1)$-automaton with complete $LR(1)$-items $[X \to \alpha., x], [Y \to \beta., y]$ if $x \neq y$. If the $LR(1)$-parser is in state q it selects to reduce with the production whose look-ahead symbol is given by the next input symbol. If state q contains one complete $LR(1)$-item $[X \to \alpha., x]$ together with an $LR(1)$-item $[Y \to \beta.a\gamma, y]$, still no *shift-reduce*-conflict occurs if $a \neq x$. In state q the generated parser reduces if the next input symbol equals x and shifts if it equals a. Therefore, the *action*-table can be computed by the following iteration:

```
forall (state q) {
    forall (terminal x) $action[q, x] \leftarrow error$;
    forall ($[X \to \alpha.\beta, x] \in q$)
        if ($\beta = \varepsilon$)
            if ($X = S' \wedge \alpha = S \wedge x = \#$) $action[q, \#] \leftarrow accept$;
            else $action[q, x] \leftarrow reduce(X \to \alpha)$;
        else if ($\beta = a\beta'$) $action[q, a] \leftarrow shift$;
}
```

Example 3.4.13 We consider some states of the canonical $LR(1)$-automaton for the CFG G_0. The numbering of states is the same as in Fig. 3.15. To make the representation of sets S of $LR(1)$-items more readable all look-ahead symbols in $LR(1)$-items from S with the same kernel $[A \to \alpha.\beta]$ are collected in one look-ahead set

$$L = \{x \mid [A \to \alpha.\beta, x] \in q\}$$

in order to represent the subset $\{[A \to \alpha.\beta, x] \mid x \in L\}$ as $[A \to \alpha.\beta, L]$. Then we obtain:

$$\begin{array}{lll} S'_0 & = & \text{closure}(\{[S \to .E, \{\#\}]\}) \\ & = \{ & [S \to .E, \{\#\}] \\ & & [E \to .E + T, \{\#, +\}], \\ & & [E \to .T, \{\#, +\}], \\ & & [T \to .T * F, \{\#, +, *\}], \\ & & [T \to .F, \{\#, +, *\}], \\ & & [F \to .(E), \{\#, +, *\}], \\ & & [F \to .\text{Id}, \{\#, +, *\}] \ \} \end{array}$$

$$\begin{array}{lll} S'_1 & = & \text{nextState}(S'_0, E) \\ & = \{ & [S \to E., \{\#\}], \\ & & [E \to E. + T, \{\#, +\}] \ \} \end{array}$$

$$\begin{array}{lll} S'_2 & = & \text{nextState}(S'_1, T) \\ & = \{ & [E \to T., \{\#, +\}], \\ & & [T \to T. * F, \{\#, +, *\}] \ \} \end{array}$$

$$\begin{array}{lll} S'_6 & = & \text{nextState}(S'_1, +) \\ & = \{ & [E \to E + .T, \{\#, +\}], \\ & & [T \to .T * F, \{\#, +, *\}], \\ & & [T \to .F, \{\#, +, *\}], \\ & & [F \to .(E), \{\#, +, *\}], \\ & & [F \to .\text{Id}, \{\#, +, *\}] \ \} \end{array}$$

$$\begin{array}{lll} S'_9 & = & \text{nextState}(S'_6, T)) \\ & = \{ & [E \to E + T., \{\#, +\}], \\ & & [T \to T. * F, \{\#, +, *\}] \ \} \end{array}$$

Table 3.6 Some rows of the *action*-table of the canonical $LR(1)$-parser for G_0. s stands for *shift*, $r(i)$ for *reduce* by production i, *acc* for *accept*. All empty entries represent *error*

	Id	( )	$*$	$+$	#
S_0'	s	s			
S_1'				s	acc
S_2'			s	$r(3)$	$r(3)$
S_6'	s	s			
S_9'			s	$r(2)$	$r(2)$

Numbering of the productions used:

$$1 : S \to E$$
$$2 : E \to E + T$$
$$3 : E \to T$$
$$4 : T \to T * F$$
$$5 : T \to F$$
$$6 : F \to (E)$$
$$7 : F \to \mathsf{Id}$$

After the extension by look-ahead symbols, the states S_1, S_2, and S_9, which were $LR(0)$-inadequate, no longer have conflicts. In state S_1' the next input symbol $+$ indicates to shift, and the next input symbol # indicates to reduce. In state S_2' look-ahead symbol $*$ indicates to shift, and # and $+$ to reduce; similarly in state S_9'.

Table 3.6 shows the rows of the *action*-table of the canonical $LR(1)$-parser for the grammar G_0, which belong to the states S_0', S_1', S_2', S_6', and S_9'. □

$SLR(1)$- and $LALR(1)$-Parsers

The set of states of $LR(1)$-parsers can become quite large. Therefore, often LR-analysis methods are employed that are not as powerful as canonical LR-parsers, but have fewer states. Two such LR-analysis methods are the $SLR(1)$- (*simple* LR-) and $LALR(1)$- (*look-ahead* LR-)methods. The $SLR(1)$-parser is a special $LALR(1)$-parser, and each grammar that has an $LALR(1)$-parser is an $LR(1)$-grammar.

The starting point of the construction of $SLR(1)$- and $LALR(1)$-parsers is the canonical $LR(0)$-automaton $LR_0(G)$. The set Q of states and the *goto*-table for these parsers are the set of states and the *goto*-table of the corresponding $LR(0)$-parser. Lookahead is used to resolve conflicts in the states in Q. Let $q \in Q$ be a state of the canonical $LR(0)$-automaton and $[X \to \alpha.\beta]$ an item in q. Let $\lambda(q, [X \to \alpha.\beta])$ denote the look-ahead set that is attached to the item $[X \to \alpha.\beta]$ in q. The $SLR(1)$-method differs from the $LALR(1)$-method in the definition of the function λ. Relative to such a function λ, the state q of $LR_0(G)$ has a *reduce-reduce*-conflict if it has different complete items $[X \to \alpha.], [Y \to \beta.] \in q$ with

$$\lambda(q, [X \to \alpha.]) \cap \lambda(q, [Y \to \beta.]) \neq \emptyset$$

Relative to λ , q has a *shift-reduce*-conflict if it has items $[X \to \alpha.a\beta], [Y \to \gamma.] \in q$ with $a \in \lambda(q, [Y \to \gamma.])$. If no state of the canonic $LR(0)$-automaton has a conflict, the look-ahead sets $\lambda(q, [X \to \alpha.])$ suffice to construct an *action*-table.

In $SLR(1)$-parsers, the look-ahead sets for items are independent of the states in which they occur. The look-ahead only depends on the left side of the production in the item:

$$\lambda_S(q, [X \to \alpha.\beta]) = \{a \in V_T \cup \{\#\} \mid S'\# \overset{*}{\Longrightarrow} \gamma X a w\} = \mathsf{follow}_1(X)$$

for all states q with $[X \to \alpha.] \in q$. A state q of the canonical $LR(0)$-automaton is called $SLR(1)$-*inadequate* if it contains conflicts with respect to the function λ_S. The grammar G is an $SLR(1)$-*grammar* if there are no $SLR(1)$-inadequate states.

Example 3.4.14 We consider again grammar G_0 of Example 3.4.1, whose canonical $LR(0)$-automaton $LR_0(G_0)$ has the inadequate states S_1, S_2 and S_9. In order to represent the function λ_S in a readable way, the complete items in the states are extended by the follow_1-sets of their left sides. Since $\mathsf{follow}_1(S) = \{\#\}$ and $\mathsf{follow}_1(E) = \{\#, +,)\}$ we obtain:

$$\begin{array}{lll} S_1'' = \{ & [S \to E., \{\#\}], & \text{conflict eliminated,} \\ & [E \to E. + T]\} & \text{since} \quad + \notin \{\#\} \\[1em] S_2'' = \{ & [E \to T., \{\#, +,)\}], & \text{conflict eliminated,} \\ & [T \to T. * F]\ \} & \text{since} \quad * \notin \{\#, +,)\} \\[1em] S_9'' = \{ & [E \to E + T., \{\#, +,)\}], & \text{conflict eliminated,} \\ & [T \to T. * F]\ \} & \text{since} \quad * \notin \{\#, +,)\} \end{array}$$

Accordingly, G_0 is an $SLR(1)$-grammar and therefore has an $SLR(1)$-parser. □

The set $\mathsf{follow}_1(X)$ collects all symbols that can follow the nonterminal X in a sentential form of the grammar. Only the follow_1-sets are used to resolve conflicts in the construction of an $SLR(1)$-parser. In many cases this is not sufficient. More conflicts can be resolved if the state is taken into consideration in which the complete item $[X \to \alpha.]$ occurs. The *most precise* look-ahead set that considers the state is defined by:

$$\lambda_L(q, [X \to \alpha.\beta]) = \{a \in V_T \cup \{\#\} \mid S'\# \underset{rm}{\overset{*}{\Longrightarrow}} \gamma X a w \wedge \Delta_G^*(q_0, \gamma\alpha) = q\}$$

Here, q_0 is the initial state, and Δ_G is the transition function of the canonical $LR(0)$-automaton $LR_0(G)$. In $\lambda_L(q, [X \to \alpha.])$ only terminal symbols are contained that can follow X in a right sentential-form $\beta X a w$ such that $\beta\alpha$ drives the canonical $LR(0)$-automaton into the state q. We call state q of the canonical $LR(0)$-automaton $LALR(1)$-*inadequate* if it contains conflicts with respect to the function λ_L. The grammar G is an $LALR(1)$-grammar if the canonical $LR(0)$-automaton has no $LALR(1)$-inadequate states.

There always exists an $LALR(1)$-parser to an $LALR(1)$-grammar. The definition of the function λ_L, however, is not constructive since the occurring sets of right sentential-forms can be infinite. Albeit in an inefficient way, an $LALR(1)$-parser can be constructed from a canonical $LR(1)$-automaton for the CFG G. The look-ahead set for an item of a state q of the canonical $LR(0)$-automaton of G is obtained by collecting all look-aheads of the same item in all $LR(1)$-states of with core q. Formally, let S_q be the subset of states all with core q. The function λ_L then is obtained by

$$\lambda_L(q, [A \to \alpha.\beta]) = \{x \in V_T \cup \{\#\} \mid \exists\, q' \in S_q :\; [A \to \alpha.\beta, x] \in q'\}$$

Without referring to the canonical $LR(1)$-automaton, the sets $\lambda_L(q, [A \to \alpha.\beta])$ can also be characterized as the least solution of the following system of equations:

$$\begin{array}{ll}
\lambda_L(q_0, [S' \to .S]) & = \{\#\} \\
\lambda_L(q, [A \to \alpha X.\beta]) & = \bigcup\{\lambda_L(p, [A \to \alpha.X\beta]) \mid \Delta_G(p, X) = q\}, \\
 & \qquad X \in (V_T \cup V_N) \\
\lambda_L(q, [A \to .\alpha]) & = \bigcup\{\mathsf{first}_1(\beta) \odot_1 \lambda_L(q, [X \to \gamma.A\beta]) \mid [X \to \gamma.A\beta] \in q'\}
\end{array}$$

This system of equations describes how sets of look-ahead symbols for items in states are generated. The first equation says that only # can follow the start symbol S'. The second type of equations describes that the look-ahead symbols of an item $[A \to \alpha X.\beta]$ in a state q result from the look-ahead symbols of the item $[A \to \alpha.X\beta]$ in states p from which one can reach q by reading X. The third class of equations formalizes that the follow symbols of an item $[A \to .\alpha]$ in a state q result from the follow symbols of *occurrences* of A in items in q, that is, from the sets $\mathsf{first}_1(\beta) \odot_1 \lambda_L(q, [X \to \gamma.A\beta])$ for items $[X \to \gamma.A\beta]$ in q.

The system of equations for the sets $\lambda_L(q, [A \to \alpha.\beta])$ over the finite subset lattice $2^{V_T \cup \{\#\}}$ can be solved by the iterative method for computing least solutions. By taking into account which nonterminal may produce ε. the occurrences of 1-concatenation in the system of equations can be replaced with unions. In this way, we obtain a reformulation as a pure union problem, that can be solved by the efficient method of Sect. 3.2.7.

Example 3.4.15 The following grammar taken from [2] describes a simplified version of the C assignment statement:

$$\begin{array}{lcl}
S' & \to & S \\
S & \to & L = R \mid R \\
L & \to & *R \mid \mathsf{Id} \\
R & \to & L
\end{array}$$

This grammar is not an $SLR(1)$-grammar, but a $LALR(1)$-grammar. The states of the canonical $LR(0)$-automaton are given by:

$$\begin{aligned} S_0 = \{\, & [S' \to .S], \\ & [S \to .L = R], \\ & [S \to .R], \\ & [L \to . * R], \\ & [L \to .\mathsf{Id}], \\ & [R \to .L] \,\} \end{aligned}$$

$$S_1 = \{\, [S' \to S.] \,\}$$

$$\begin{aligned} S_2 = \{\, & [S \to L. = R], \\ & [R \to L.] \,\} \end{aligned}$$

$$S_3 = \{\, [S \to R.] \,\}$$

$$\begin{aligned} S_4 = \{\, & [L \to *.R], \\ & [R \to .L], \\ & [L \to . * R], \\ & [L \to .\mathsf{Id}] \,\} \end{aligned}$$

$$S_5 = \{\, [L \to \mathsf{Id}.] \,\}$$

$$\begin{aligned} S_6 = \{\, & [S \to L = .R], \\ & [R \to .L], \\ & [L \to . * R], \\ & [L \to .\mathsf{Id}] \;\} \end{aligned}$$

$$S_7 = \{\, [L \to *R.] \,\}$$

$$S_8 = \{\, [R \to L.] \,\}$$

$$S_9 = \{\, [S \to L = R.] \,\}$$

State S_2 is the only $LR(0)$-inadequate state. We have $\mathsf{follow}_1(R) = \{\#, =\}$. This look-ahead set for the item $[R \to L.]$ is not sufficient to resolve the *shift-reduce*-conflict in S_2 since the next input symbol $=$ is in the look-ahead set. Therefore, the grammar is not an $SLR(1)$-grammar.

The grammar is, however, an $LALR(1)$-grammar. The transition diagram of its $LALR(1)$-parser is shown in Fig. 3.17. To increase readability, the look-ahead sets $\lambda_L(q, [A \to \alpha.\beta])$ are directly associated with the item $[A \to \alpha.\beta]$ of state q. In state S_2, the item $[R \to L.]$ receives the look-ahead set $\{\#\}$. The conflict is resolved since this set does not contain the next input symbol $=$. □

3.4.4 Error Handling in LR-Parsers

LR-parsers like LL-parsers have the viable-prefix property. This means that each prefix of the input that could be analyzed by an LR-parser without finding an error can be completed to a correct input word, a word of the language. The earliest situation in which an error can be detected is when an LR-parser reaches a state q where for the actual input symbol a the *action*-table only provides the entry error. We call such a configuration an *error configuration* and q the *error state* of this configuration. There are at least two approaches to error handling in LR-parsers:

Forward error recovery: Modifications are made in the remaining input, not in the pushdown.

Backward error recovery: Modifications are also made in the pushdown.

Let us assume that the actual state is q and the next symbol in the input is a. Potential corrections are the following actions: A generalized *shift*(βa) for an item $[A \to \alpha.\beta a \gamma]$ in q, a *reduce* for incomplete items in q, and *skip*.

- The correction *shift*(βa) assumes that the subword for β is missing. It therefore pushes the states that the IPDA would run through when reading the word β starting in state q. After that the symbol a is read and the associated *shift*-transition of the parser is performed.

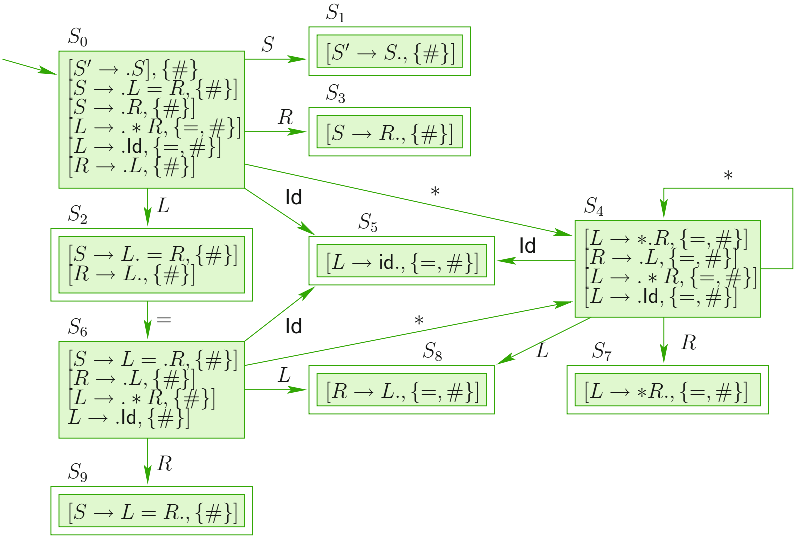


Fig. 3.17 Transition diagram of the $LALR(1)$-parser for the grammar of Example 3.4.15

- The correction *reduce*$(A \to \alpha.\beta)$ also assumes that the subword that belongs to β is missing. Therefore $|\alpha|$ many states are removed from the pushdown. Let p be the newly appearing state on top of the pushdown. Then the successor state of p and the left side A according to the *goto*-table is pushed onto the pushdown.
- The correction *skip* continues with the next symbol a' in the input.

A straight forward method for error recovery based on these actions may proceed as follows. Let us assume that the *action*-table only contains an *error*-entry for the actual input symbal a. If the actual state q contains an item $[A \to \alpha.\beta a \gamma]$, the parser may restart by reading a. As a correction therefore *shift*(βa) is performed. If the symbol a does not occur in any right side of an items in q, but as a look-ahead of an incomplete item $[A \to \alpha.\beta]$ in q, then *reduce*$(A \to \alpha.\beta)$ may be performed as correction. If several corrections are possible in q, a *plausible* correction is chosen. It may be plausible, for example, to choose the operation *shift*(βa) or *reduce*$(A \to \alpha.\beta)$ in which the missing subword β is the shortest. If neither a *shift*- nor a *reduce*-correction is possible the correction *skip* is applied.

Example 3.4.16 Consider the grammar G_0 with the productions

$$\begin{array}{lllllllll} E & \to & E + T & \quad T & \to & T * F & \quad F & \to & (E) \\ E & \to & T & \quad T & \to & F & \quad F & \to & \mathsf{Id} \end{array}$$

for which the canonical $LR(0)$-automaton has been constructed in Example 3.4.5. As input we choose

$$(\ \mathsf{Id} + \ )$$

After reading the prefix (Id + the pushdown of an $SLR(1)$-parser contains the sequence of states $S_0S_4S_8S_6$, corresponding to the reliable prefix (E +. The actual state S_6 consists of the items:

$$\begin{array}{rcl} S_6 & = & \{\ [E \to E + .T], \\ & & \quad [T \to .F], \\ & & \quad [F \to .\mathsf{Id}], \\ & & \quad [F \to .(E)] \quad \} \end{array}$$

Since we consider an $SLR(1)$-parser, the look-ahead sets of the items in S_6 are given by the follow_1-sets of the left sides, i.e.,

S_6	λ_S
$[E \to E + .T]$	$+,)$
$[T \to .F]$	$*, +,)$
$[F \to .\mathsf{Id}]$	$*, +,)$
$[F \to .(E)]$	$*, +,)$

While reading) in state S_6 leads to error, there are incomplete items in S_6 with look-ahead). One of these items may be used for reduction; let us choose, e.g., the item $[E \to E + .T]$. The corresponding reduction produces the new pushdown content $S_0S_4S_8$, since S_8 is the successor state of S_4 under the left side E. In state S_8 a *shift*-transition is possible that reads). The corresponding action pushes the new state S_{11} on top of the pushdown. Now, a sequence of reductions will reach the final state f. □

The presented error recovery method is a pure forward recovery. It is similar to the one offered, e.g., by the parser generator CUP for JAVA.

The One-Error Hypothesis

We now present a refined method of error recovery. It assumes that the program is *essentially correct* and therefore requires only minor corrections. The error correction again moves over the input in forward direction. When an error has been detected, the parser attempts to change the input only at a single position. This assumption is called the *one-error hypothesis*.

The proposed method is quite sophisticated in the way it uses precomputed information to efficiently decide how to correct the error in the input. A configuration of the LR-parser is written as $(\varphi q, a_i \ldots a_n)$, where φq is the pushdown contents with actual state q, and the remaining input is $a_i \ldots a_n$. The refined error recovery attempts to identify for each error configuration $(\varphi q, a_i \ldots a_n)$ a *suitable* configuration in which a continuation of the analysis is possible that reads at least one input symbol. A configuration is *suitable* for an error configuration if it is obtained by as few modifications as possible from the error configuration. The *one-error hypothesis* restricts the possible modification drastically. The one-error hypothesis means that the detected error is caused by either *one* missing, *one* extra, or *one* wrong symbol at the position of the error. Accordingly, the error-recovery algorithm offers operations for the insertion, for the deletion, and for the replacement of that single symbol.

Let $(\varphi q, a_i \dots a_n)$ be an error configuration. The goal of error correction by one of the three operations can be described as follows:
Deletion: Find a pushdown contents $\varphi' p$ with

$$(\varphi q, a_{i+1} \dots a_n) \vdash^* (\varphi' p, a_{i+1} \dots a_n) \quad \text{and} \quad \mathit{action}[p, a_{i+1}] = \mathit{shift}$$

Replacement: Find a symbol a and pushdown contents $\varphi' p$ with

$$(\varphi q, a a_{i+1} \dots a_n) \vdash^* (\varphi' p, a_{i+1} \dots a_n) \quad \text{and} \quad \mathit{action}[p, a_{i+1}] = \mathit{shift}$$

Insertion: Find a symbol a and pushdown contents $\varphi' p$ with

$$(\varphi q, a a_i \dots a_n) \vdash^* (\varphi' p, a_i \dots a_n) \quad \text{and} \quad \mathit{action}[p, a_i] = \mathit{shift}$$

The required pushdown contents $\varphi' p$ are determined by the property that reductions are possible under the new next input symbol that were impossible in the error configuration. An important property of all three operations is that they guarantee termination of the error recovery process: each of the three operations advances the input pointer by at least one symbol.

Error recovery methods *with backtracking* additionally allow us to do the last applied production $X \to \alpha Y$ and to consider $Y a_i \dots a_n$ as input, when all other correction attempts have failed.

An immediate realization of the method searches through the different possible error corrections dynamically, that is, at parsing time until a suitable correction is found. Checking one of the possibilities may involve several reductions, followed by a test whether a symbol can be read. Upon failure of the test, the error configuration must be restored, and the next possibility be tested. Finding a *right* modification of the program by one symbol therefore can be expensive. Therefore we are interested in *precomputations* that can be performed at the time when the parser is generated. The result of the precomputations should allow us to recognize many dead ends in the error recovery quickly. Let $(\varphi q, a_i \dots a_n)$ be an error configuration. Let us consider the *insertion* of a symbol $a \in V_T$. The error recovery consists of the following sequence of steps (see Fig. 3.18a):
(1) a sequence of reductions under look-ahead symbol a, followed by
(2) reading of a, followed by
(3) a sequence of reductions under look-ahead symbol a_i.
A precomputation makes it possible to exclude many symbols a from consideration for which there are no subsequences for subtasks (1) or (3). Therefore for each state q and each $a \in V_T$, the set $\mathit{Succ}(q, a)$ of potential *reduction successors* of q under a is computed. The set $\mathit{Succ}(q, a)$ contains the state q together with all states in which the parser may come out of q by reductions under look-ahead a. The set $\mathit{Succ}(q, a)$ is the smallest set Q' with the following properties:

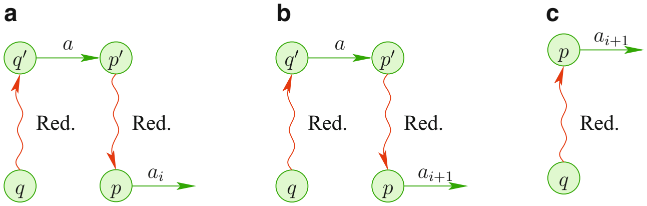


Fig. 3.18 Closing the gap for error correction, **a** by insertion, **b** by replacement, or **c** by deletion of a symbol

- $q \in Q'$.
- Let $q' \in Q'$, and let q' contain a complete item to a production $A \to X_1 \dots X_k$. Then $goto[p, A] \in Q'$ for each state p with

$$goto[\dots goto[p, X_1] \dots, X_k] = q$$

Using the set $Succ(q, a)$ we define the set $Sh(q, a)$ of all states that can be reached from reduction successors of q under a by a *shift* transition for a:

$$Sh(q, a) = \{goto[q', a] \mid q' \in Succ(q, a)\}$$

Using the sets $Sh(q', a')$ for all states q' and terminal symbols a, we define the set of all states that are reached from the states in $Sh(q, a)$ by reductions with lookahead a_i, followed by reading a_i:

$$Sh(q, a, a_i) = \bigcup \{Sh(q', a_i) \mid q' \in Sh(q, a)\}$$

A correction by inserting a symbol a is promising if the set $Sh(q, a, a_i)$ is nonempty. The terminal symbol a is a candidate to bridge the gap in the error configuration. One could even drive preprocessing further by computing for state q and terminal symbol a_i the set of all candidates that can be tried for error correction by insertion:

$$Bridge(q, a_i) = \{a \in V_T \mid \quad Sh(q, a, a_i) \neq \emptyset\}$$

Example 3.4.17 We consider the grammar of Example 3.4.15 with the $LALR(1)$-parser of Fig. 3.17. The sets of reduction successors $Succ(q, a)$ of q under a, the sets $Sh(q, a)$ and $Bridge(q, a)$ for $a \in \{=, *, \mathsf{Id}\}$ given by:

$\underline{Succ(q, a)}$:

q	$=$	$*$	Id
S_0	S_0	S_0	S_0
S_1	S_1	S_1	S_1
S_2	S_2	S_2	S_2
S_3	S_3	S_3	S_3
S_4	S_4	S_4	S_4
S_5	S_2, S_5	S_5	S_5
S_6	S_6	S_6	S_6
S_7	S_2, S_7	S_7	S_7
S_8	S_2, S_8	S_8	S_8
S_9	S_9	S_9	S_9

$\underline{Sh(q, a)}$:

q	$=$	$*$	Id
S_0		S_4	S_5
S_1			
S_2	S_6		
S_3			
S_4		S_4	S_5
S_5	S_6		
S_6		S_4	S_5
S_7	S_6		
S_8	S_6		
S_9			

$\underline{Bridge(q, a)}$:

q	$=$	$*$	Id
S_0	Id	$*$	$*$
S_1			
S_2		$=$	$=$
S_3			
S_4	Id	$*$	$*$
S_5		$=$	$=$
S_6	Id	$*$	$*$
S_7		$=$	$=$
S_8			
S_9			

The following error corrections are proposed by the precomputation:

Input	Error configuration	Bridge	Correction
$* = \mathsf{Id}\ \#$	$(S_0 S_4, = \mathsf{Id}\ \#)$	$bridge(S_4, =) = \{\mathsf{Id}\}$	Insertion of Id
$\mathsf{Id} == \mathsf{Id}\ \#$	$(S_0 S_2 S_6, = \mathsf{Id}\ \#)$	$Bridge(S_6, \mathsf{Id}) = \{*\}$	Replacement of $=$ by $*$

An example for a correction by deletion is:

Input	Error configuration	Bridge	Correction
$\mathsf{Id}\ \mathsf{Id} = \mathsf{Id}\ \#$	$(S_0 S_5, \mathsf{Id} = \mathsf{Id}\ \#)$	$Sh(S_5, =) \neq \emptyset$	Deletion of Id

□

On the basis of the precomputed sets a correction by one-symbol *replacement* can be efficiently tried out. The only difference is that symbols in $Bridge(q, a_{i+1})$ need to be considered (see Fig. 3.18b).

Analogous considerations lead to a test, whether a one-symbol-*deletion* is promising. The deletion of a symbol a is considered as a potential error correction when the set $Sh(q, a_{i+1})$ is nonempty (see Fig. 3.18c). For each combination of a state q and a symbol a we can precompute whether there exists such a state p. This predicate can be tabulated such that the test can be implemented by a simple table lookup.

We have so far ignored the special case of corrections at exhausted input. Since the end symbol # is never read, neither correction by deletion nor by replacement are possible. Only corrections by insertion can be applied.

The insertion of a terminal symbol a makes sense when, after some reductions out of q under look-ahead a, a state p can be reached from which another state p' can be reached by reading a, from which again by reductions under # *accept* configurations are reachable. To support this, the set $Acc(q)$ for each state q can be precomputed that contains all candidate terminal symbols.

Let $(\varphi q, a_i \dots a_n)$ be the error configuration. An optimized error correction can be described as follows:

Attempt to delete: If $Sh(q, a_{i+1}) \neq \emptyset$, then $test(\varphi q, a_{i+1} \dots a_n)$;
Attempt to replace: If there exists an $a \in Bridge(q, a_{i+1})$,
then $test(\varphi q, aa_{i+1} \dots a_n)$;
Attempt to insert: If there exists an $a \in Bridge(q, a_i)$,
then $test(\varphi q, aa_i a_{i+1} \dots a_n)$.

In procedure *test*, parsing is continued after the attempt of an error correction. If the remaining input can be successfully analyzed, the correction attempt is considered successful. If the attempt to process the remaining input fails, the parser assumes the existence of another error and starts a correction attempt for this error.

A more ambitious implementation may be more reluctant to conjecture a second error at a failed correction attempt. Instead it may return to the error location and attempt another correction. Only if all correction attempts fail, is a further error assumed. The parser then selects a *best* attempt and restarts error correction in the reached configuration. One measure for the success of an attempt of error-correction could be the length of the input possibly consumed after the correction attempt.

The Forward Move

The precomputations that we have presented are able to considerably reduce the number of correction candidates that need to be tried. Still, several error corrections can be jointly possible. For speeding up this situation, partial computations that are required by all correction attempts should be *factored out*. Such a partial computation is a *forward move* that is common to all attempts. To identify such moves, the algorithm does not start with a particular pushdown contents, but with all states in which the symbol a_{i+1} can be read. It then attempts to reduce a prefix, which is as long as possible, of $a_{i+1} \dots a_n$. The configurations consist of sequences of *sets* of states Q in an error pushdown and the corresponding remaining input. If the parser is in the set of states Q with next input symbol a, it performs nonerror transitions according to $action[q, a]$ for all $q \in Q$ if either all states q command *shift*, or all command $reduce(X \to \alpha)$ by the same production $X \to \alpha$ under the assumption that the error pushdown is not shorter than $|\alpha|$. The forward move stops,

- when for all $q \in Q$ $action[q, a] = \text{error}$ it holds that: a *second* error has been detected;
- when the *action* table for the set Q and the symbol a contains more than one action;
- when the only action is *accept*: parsing has terminated; or
- when a reduction is required, where the length of the right side is greater than the depth of the error pushdown: this would lead to a *reduction beyond the error location*.

As a result, the forward move returns the word γ to which it has reduced the prefix $a_{i+1} \dots a_k$ read so far, followed by the remaining input $a_{k+1} \dots a_n$. Note that the word γ will often be much shorter than the subword $a_{i+1} \dots a_k$. In the input for the calls of procedure *test* the subword $a_{i+1} \dots a_n$ can be replaced by $\gamma\ a_{k+1} \dots a_n$, where the parser treats a nonterminal A in the input always like a *shift* of the symbol A.

Wrong Reductions in $SLR(1)$- and $LALR(1)$-parsers

Canonical $LR(1)$-parsers detect errors at the earliest location in the input; they neither read a symbol beyond this error location nor reduce under a wrong look-ahead symbol. $SLR(1)$- and $LALR(1)$-parsers also never read a symbol beyond the error location. They may, however, perform extra reductions from a *shift*-state before detecting the error. The reason is that their look-ahead sets are less differentiated. An extra pushdown is used to undo these extra reductions. All reductions performed since the last read are stored on this pushdown. This pushdown is emptied at each read. When an error is detected the stored reductions are undone in reverse order.

3.5 Exercises

1. **Reduced Grammar**
   Check the productivity and the reachability of the nonterminals of the CFG

$$G = \left(\{S, A, B, C, D, E\},\ \{a, b, c\}, \left\{ \begin{array}{lcl} S & \to & aAa \mid bS \\ A & \to & BB \mid C \\ B & \to & bC \\ C & \to & B \mid c \\ D & \to & aAE \\ E & \to & Db \end{array} \right\}, S \right)$$

2. **Items**
   Give a definition of the *future* of a sequence of items, $\mathsf{fut}(\gamma)$, such that you can prove the following invariant (I'):
   (I') For all words $uv \in L(G)$ exists a sequence $\gamma \in \mathsf{It}_G^*$ such that:
   $(q_0, uv) \vdash_{K_G}^{*} (\gamma, v)$ implies $\mathsf{fut}(\gamma) \overset{*}{\Longrightarrow} v$.
3. **ε-Productions**
   Assume that $G = (V_N, V_T, P, S)$ is a reduced CFG, and that the start symbol does not occur in the right side of any production. G is called ε-free if $A \to \varepsilon \in P$ implies that A is the start symbol S. Show that for each grammar G, an ε-free grammar can be constructed that describes the same language.
4. **Item-pushdown Automaton**

   (a) Construct the IPDA to the grammar

$$G = \left(\{S\},\ \{\textbf{if}\ ,\ \textbf{then}\ ,\ \textbf{else}\ ,\ a,\ b\}, \left\{ \begin{array}{l} S \to a \\ S \to \textbf{if}\ b\ \textbf{then}\ S \\ S \to \textbf{if}\ b\ \textbf{then}\ S\ \textbf{else}\ S \end{array} \right\}, S \right)$$

(b) Give an accepting sequence of configurations for

$$\textbf{if}\ \ b\ \textbf{then}\ \ \textbf{if}\ \ b\ \textbf{then}\ \ a\ \textbf{else}\ \ a$$

(c) Show that G is ambiguous.

(d) Give an unambiguous grammar G' with $L(G') = L(G)$.

5. **Item-pushdown Automaton (*Cont.*)**

(a) Construct the IPDA to

$$G = \left(\{S, A, B, C\},\ \{a, b\}, \left\{ \begin{array}{lcl} S & \to & AB \mid BC \\ A & \to & BA \mid a \\ B & \to & CC \mid b \\ C & \to & AB \mid a \end{array} \right\}, S \right)$$

(b) How many accepting sequences of configurations exist for the word $babaab$?

6. ***follow*-Sets**

Prove Theorem 3.2.3.

7. **Strong $LL(k)$-Grammars**

Develop a construction that constructs to an arbitrary $LL(k)$-grammar a strong $LL(k)$-grammar that specifies the same language.

[Hint: Use pairs $\langle A, \mathsf{first}_k(\beta) \rangle$ for nonteminals A and words β with $S'\# \overset{*}{\underset{L}{\Longrightarrow}} wA\beta$ as nonterminals.]

How are the parse trees of the original grammar related to the parse trees of the transformed grammar?

8. ***k*-Concatenation**

Prove that the operation $\odot_k$ is associative.

9. **first_1- and follow_1-Sets**

Assume that the following grammar is given:

$$G = \left(\{S', S, B, E, J, L\}, \{;, :=, (,), ,\}, \left\{ \begin{array}{lcl} S' & \to & S \\ S & \to & LB \\ B & \to & ;S;L \mid := L \\ E & \to & a \mid L \\ J & \to & ,EJ \mid) \\ L & \to & (EJ \end{array} \right\}, S' \right)$$

Compute the first_1- and follow_1-sets using the iterative procedure.

10. **ε-free first_1-sets**

Consider the grammar:

$$G = \left(\{S, A, B\},\ \{a, b\}, \left\{ \begin{array}{lcl} S & \to & aAaB \mid bAbB \\ A & \to & a \mid ab \\ B & \to & aB \mid a \end{array} \right\}, S \right)$$

(a) Set up the system of equations for the computation of the ε-free first_1-sets and the follow_1-sets.
(b) Determine the variable dependency-graphs of the two systems of equations and their strongly connected components.
(c) Solve the two systems of equations.

11. ***LL*(1)-Grammars**
Test the $LL(1)$-property for
(a) the grammar of Exercise 5;
(b) the grammar of Exercise 9;
(c) the grammar of Exercise 10;
(d) the grammar

$$G = \left(\{E, E', D, D', F\},\ \{a, (,), +, *\}, \left\{ \begin{array}{lcl} E & \to & DE' \\ E' & \to & +DE' \mid \varepsilon \\ D & \to & FD' \\ D' & \to & *FD' \mid \varepsilon \\ F & \to & (E) \mid a \end{array} \right\}, E \right)$$

12. ***LL*(1)-Parsers**
(a) Construct the $LL(1)$-parser table for G of Exercise 11(d).
(b) Give a run of the corresponding parser for the input $(a + a) * a + a$.

13. ***LL*(1)-Parsers (*Cont.*)**
Construct the $LL(1)$-table for the grammar with the following set of productions:

$$\begin{array}{lcl} E & \to & -E \mid (E) \mid VE' \\ E' & \to & -E \mid \varepsilon \\ V & \to & \mathsf{Id}\ V' \\ V' & \to & (E) \mid \varepsilon \end{array}$$

Sketch a run of the parser on input $-\ \mathsf{Id}\ (-\ \mathsf{Id})\ -\ \mathsf{Id}$.

14. **Extension of Right-regular Grammars**
Extend right-regular grammars by additionally allowing the operators ? and $(\_)^+$ to occur in right sides of productions.
(a) Provide the productions of the transformed grammar for the nonterminals $\langle r? \rangle$ and $\langle r^+ \rangle$.
(b) Extend the generator scheme $\mathsf{generate}$ of the *recursive-descent* parser to expressions $r?$ and r^+.

15. **Syntax Trees for *RLL*(1)-Grammars**
Instrument the parsing procedures of the *recursive-descent* parser for an $RLL(1)$-grammar G in such a way that they return syntax trees.
(a) Instrument the procedure of a nonterminal A in such a way that syntax trees of the transformed CFG $\langle G \rangle$ are produced.
(b) Instrument the procedure of a nonterminal A in such a way that the subtrees of all symbols of the current word from the language of the regular expression $p(A)$ are collected in a vector.

Adapt the generator schemes for the regular subexpressions of right sides accordingly.

16. **Operator Precedences**

Consider a CFG with a nonterminal symbol A and the productions:

$$\begin{array}{lcl} A & \to & \mathsf{lop}\, A \mid A\, \mathsf{rop} \mid \\ & & A\, \mathsf{bop}\, A \mid \\ & & (\, A\,) \mid \mathsf{var} \mid \mathsf{const} \end{array}$$

for various unary prefix operators lop, unary postfix operators rop, and binary infix operators bop where the sets of postfix and infix operators are disjoint. Assume further that every unary operator has a precedence and every infix operator has both a left precedence and a right precedence by which the strength of the association is determined. If negation has precedence 1 and the operator $+$ has left and right precedences 2 and 3, respectively, the expression

$$-1 + 2 + 3$$

corresponds to the bracketed expression:

$$((-1) + 2) + 3$$

Traditionally, smaller precedences bind stronger. In case of equal precedences, association to the right is favored. The bracketing when all precedences are one, therefore is given by:

$$-(1 + (2 + 3))$$

(a) Transform the grammar in such a way that every expression obtains exactly one parse tree, which corresponds to a correct bracketing according to the operator precedences.
Is your grammar an $LR(1)$-grammar? Justiffy your answer.

(b) Construct *directly* from the grammar a *shift-reduce*-parser which identifies reductions only based on the look-ahead 1 and the precedences of the operators.

(c) Realize your *shift-reduce*-parser by means of recursive functions.

17. ***LR*(*k*)-Grammars**

Which of the folowing grammars are not $LR(0)$-grammars? Justify your answer.

$$\begin{array}{llll} S \to L & S \to L & S \to L & S \to L \\ L \to L; A \mid A & L \to A; L \mid A & L \to L; L \mid A & L \to aT \\ A \to a & A \to a & A \to a & T \to \varepsilon \mid ; L \\ \text{(a)} & \text{(b)} & \text{(c)} & \text{(d)} \end{array}$$

18. ***SLR*(1)-Grammars**
Show that the following grammar is an $SLR(1)$-grammar, and construct an *action*-table for it:

$$\begin{array}{lcl} S & \rightarrow & E \\ E & \rightarrow & T \mid E + T \\ T & \rightarrow & P \mid T * P \\ P & \rightarrow & F \mid F \uparrow P \\ F & \rightarrow & \mathsf{Id} \mid (E) \end{array}$$

19. ***SLR*(1)-Grammars (*Cont.*)**
Show that the following grammar is an $LL(1)$-grammar, but not an $SLR(1)$-grammar:

$$\begin{array}{lcl} S & \rightarrow & AaAb \mid BbBa \\ A & \rightarrow & \varepsilon \\ B & \rightarrow & \varepsilon \end{array}$$

20. ***LALR*(1)-Grammars**
Show that the following grammar is an $LALR(1)$-grammar, but not an $SLR(1)$-grammar:

$$\begin{array}{lcl} S & \rightarrow & Aa \mid bAc \mid dc \mid bda \\ A & \rightarrow & d \end{array}$$

21. ***LALR*(1)-Grammars (*Cont.*)**
Show that the following grammar is an $LR(1)$-grammar, but not an $LALR(1)$-grammar:

$$\begin{array}{lcl} S & \rightarrow & Aa \mid bAc \mid Bc \mid bBa \\ A & \rightarrow & d \\ B & \rightarrow & d \end{array}$$

22. ***LR*(1)-Automata**
Consider the grammar with the following productions:

$$\begin{array}{lcl} S & \rightarrow & A \\ A & \rightarrow & bB \\ B & \rightarrow & cC \\ B & \rightarrow & cCe \\ C & \rightarrow & dA \\ A & \rightarrow & a \end{array}$$

(a) Compute the set of $LR(1)$-items.
(b) Is the grammar an $SLR(1)$-grammar?
(c) Is the grammar an $LALR(1)$-grammar?
(d) Is the grammar an $LR(1)$-grammar?

23. ***LR*(1)-Automata (*Cont.*)**
Consider the grammar with the following productions:

$$\begin{array}{lcl} S & \rightarrow & A \\ B & \rightarrow & \varepsilon \\ C & \rightarrow & \varepsilon \\ A & \rightarrow & BCA \\ A & \rightarrow & a \end{array}$$

(a) Compute the set of $LR(1)$-items. Is the grammar an $LR(1)$-grammar?
(b) Construct the canonical $LR(0)$-automaton for the grammar. Compute $LALR(1)$-look-ahead sets for the inadequate states.

3.6 Bibliographic Notes

Detailed presentations of the theory of formal languages and automata are given in the books by Hopcroft and Ullman [25] and Harrison [22]. Exclusively dedicated to syntactic analysis are the books [46], [59] and [60]. Möncke and Wilhelm developed *grammar-flow analysis* as a generic method to solve some of the problems treated in this chapter, namely the determination of the productivity and the reachability of nonterminals, the computation of first_k and follow_k sets, and later the computation of global attribute dependences. It was first described in [49] and is further developed in [48] and [50]. Knuth presents in [39] a related approach. However, he used totally ordered sets. A similar approach is presented by Courcelle in [9].

$LL(k)$-grammars were introduced by Lewis and Stearns [26], [27]. Heckmann [23] develops an efficient $RLL(1)$ parser generator, which computes first_1- and follow_1 sets noniteratively as solution of a pure-union problem. The presented error-recovery method for $RLL(1)$ parsers is a refinement of the method realized by Ammann in the Zürich Pascal-P4 compiler [4], [69]. Related techniques are also presented in [43]. The transformation for the removal of left recursion essentially follows the method in [5].

As early as in the 1950s, parsing techniques were sought for expressions built up by means of prefix, postfix, and infix operators of different precedences. In [16], the *shunting yard* algorithm was suggested by Dijkstra. This algorithm memorizes operators on a stack and maintains the operands on another stack. A *shift-reduce*-parser for the same problem goes back to Floyd [18]. An alternative approach is presented by Pratt [57]. The *Pratt*-parser extends the *recursive-descent* parser and includes the treatment of different operators and operator precedences directly into the recursive parse functions. A formal elaboration of this idea together with an implementation in LISP is provided by Van De Vanter's Master's thesis [64].

$LR(k)$-grammars were introduced by Knuth [36]. That the effort of $LR(k)$-parsing does not necessarily grow exponentially in k, was shown 2010 by Norbert Blum [6]. The subclasses of $SLR(k)$- and $LALR(k)$-grammars, as supported by most LR-parser generators, were proposed by DeRemer [13], [14]. Besides parsing techniques that are based on these subclasses of $LR(1)$, there has been extensive discussion on how general $LR(1)$-parsers can efficiently be implemented. Useful optimizations for that purpose were already proposed in the 1970s by Pager [53, 54]. An interesting newer approach is by Kannapinn in his PhD dissertation [33], where also extensions are discussed for grammars with regular right sides. The generation of verified $LR(1)$-parsers is described in [32]. The elaborated error-recovery method for $LR(1)$-parsers follows [55].

Tomita presents *generalized* LR-parsing as a syntax-analysis method, which is able to recognize languages of *all* context-free grammars [62, 63]. Whenever a conflict is encountered, all possibilities are tracked in parallel by using several stacks. Still, it attempts to analyze as much as possible deterministically and to work with only one stack in order to increase its efficiency. Nonetheless, its worst-case complexity is $O(n^3)$. This method, although originally developed for natural-language parsing, is also interesting for the analysis of languages like C++, which do not have deterministic context-free grammars.

Semantic Analysis

4

4.1 The Task of Semantic Analysis

Some required properties of programs cannot be described by context-free grammars. These properties are described by *context-conditions*. Fundamental for these requirements are the rules of the programming language for *validity*, and *visibility* of identifiers.

The rules concerning *declaredness* determine whether an explicit declaration has to be given for an identifier, where it has to be placed, and whether multiple declarations of an identifier are allowed. The rules for *validity* determine for identifiers declared in the program what the *scope* of a declaration is, that is, in which part of the program the declaration may have an effect. The rules for *visibility* determine where in its scope an identifier is *visible* or *hidden*.

The goal of such restrictions is to preclude frequent programming errors. This is also the intention of rules to enforce *type consistency* and the initialization of certain variables or attributes. Initialization is enforced in order to preclude read accesses to *uninitialized* variables whose result is undefined. *Type consistency*, on the other hand, is supposed to guarantee that at run-time no operation will access operands that are not compatible with its argument types.

Some Notions

We use the following notions to describe the task of semantic analysis.

An *identifier* is a symbol (in the sense of lexical analysis) which can be used in a program to name a program element. Program elements in imperative languages that may be named are modules, functions or procedures, statement labels, constants, variables and parameters, and their types. In object-oriented languages such as JAVA, classes and interfaces, together with their attributes and their methods, can also be named. In functional languages such as OCAML, variables and functions differ semantically slightly from the corresponding concepts in imperative languages, but can be named by identifiers as well. An important class of data structures can be built using constructors whose identifiers are introduced together

R. Wilhelm, H. Seidl, S. Hack, *Compiler Design*, DOI 10.1007/978-3-642-17540-4_4,

with the type declaration. In logic languages such as PROLOG, identifiers may refer to predicates, atoms, data constructors, and variables.

Some identifiers are introduced in explicit declarations. The occurrence of an identifier in its declaration is the *defining occurrence* of the identifier; all other occurrences are *applied occurrences*. In imperative programming languages such as C and in object-oriented languages such as JAVA, all identifiers need to be explicitly introduced. Essentially, this also holds for functional languages such as OCAML. In PROLOG, however, neither constructors and atoms, nor local variables in clauses are explicitly introduced. Instead, they are introduced by their syntactically first occurrence in the program or the clause. In order to distinguish between variables and atoms, their respective identifiers are taken from distinct name spaces. Variables start with a capital letter or an underscore, while constructors and atoms are identified by leading lower-case letters. Thus, the term $f(X, a)$ represents an application of the binary constructor $f/2$ to the variable X and the atom a.

Each programming language has *scoping constructs*, which introduce boundaries within which identifiers can be used. Imperative languages offer packages, modules, function and procedure declarations as well as blocks that may summarize several statements (and declarations). Object-oriented languages such as JAVA additionally provide classes and interfaces, which may be organized in hierarchies. Functional languages such as OCAML also offer modules to collect sets of declarations. Explicit *let*- and *let-rec*-constructs allow us to restrict declarations of variables and functions to particular parts of the program. Beyond clauses, modern dialects of PROLOG also provide module systems.

Types are simple forms of specifications. If the programming element is a module, the type specifies which operations, data structures, and further programing elements are exported. For a function or method, it specifies the types of the arguments as well as the type of the result. If the programming element is a program variable of an imperative or object-oriented language, the type restricts which values may be stored in the variable. In purely functional languages, values cannot be explicitly assigned to variables, i.e., stored in the storage area corresponding to this variable. A variable here does not identify a storage area, but a value itself. The type of the variable therefore must also match the types of all possibly denoted values. If the programming element is a value then the type also determines how much space must be allocated by the run-time system in order to store its internal representation. A value of type **int** of the programming language JAVA, for example, currently requires 32 bits or 4 bytes, while a value of type **double** requires 64 bits or 8 bytes. The type also determines which internal representation to use and which operations to be applied to the value as well as their semantics. An *int*-value in JAVA, for example, must be represented in two's complement and can be combined with other values of type **int** by means of arithmetic operations to compute new values of type **int**. Overflow is explicitly allowed. C on the other hand, does not specify the size and internal representation of base types with equal precision. For signed *int*-values, for example, may be represented by the ones' as well as by the two's complement – depending on what is provided by the target architecture. Accordingly, the effect of an overflow is left unspecified. Therefore, the program fragment:

$$\textbf{if}\ (\text{MAX\_INT} + 1 = \text{MIN\_INT})\quad \mathsf{printf}(''\texttt{Hello\textbackslash n}'');$$

need not necessarily output `Hello`. Instead, the compiler is allowed to optimize it away into an empty statement. For **unsigned int**, however, C is guaranteed to wrap-around at overflows. Therefore, the statement:

$$\textbf{if}\ (\text{MAX\_UINT} + 1 = 0)\quad \mathsf{printf}(''\texttt{Hello\textbackslash n}'');$$

always will output `Hello`.

Concrete and Abstract Syntax

Conceptually, the input to semantic analysis is some representation of the structure of the program as it is produced by syntactic analysis. The most popular representation of this structure is the parse tree according to the context-free for the language. This tree is called the *concrete syntax* of the program. The context-free grammar for a programming language contains information that is important for syntactic analysis, but is irrelevant for the subsequent compiler phases. Among these are terminal symbols that are required for the syntactic analysis and the parsing, but that have no semantic value of their own such as the key words if, else or while. In many grammars, operator precedences introduce deep nestings of nonterminal nodes in parse trees, typically one nonterminal per precedence depth. These nonterminals together with the corresponding productions are often of no further significance once the syntactic structure has been identified. Therefore compilers often use a simplified representation of the syntactic structure, which therefore is called *abstract syntax*. It only represents the constructs occurring in the program and their nesting.

Example 4.1.1 The concrete syntax of the program fragment

$$\begin{aligned}&\mathsf{int}\ a, b;\\ &\quad a \leftarrow 42;\\ &\quad b \leftarrow a * a - 7;\end{aligned}$$

from the introduction is shown in Fig. 4.1. We assumed that the context-free grammar differentiates between the precedence levels for assignments, comparison, addition, and multiplication operators. Notable are the long chains of chain productions, that is, replacements of one nonterminal by another one, which have been introduced to bridge the precedence differences. A more abstract representation is obtained if in a first step, the applications of chain productions are removed from the syntax tree and then superfluous terminal symbols are omitted. The result of these two simplifications is shown in Fig. 4.2. □

The first step of the transoformation into an abstract syntax that we executed by hand in Example 4.1.1 need not be executed for each syntax tree separately. Instead,

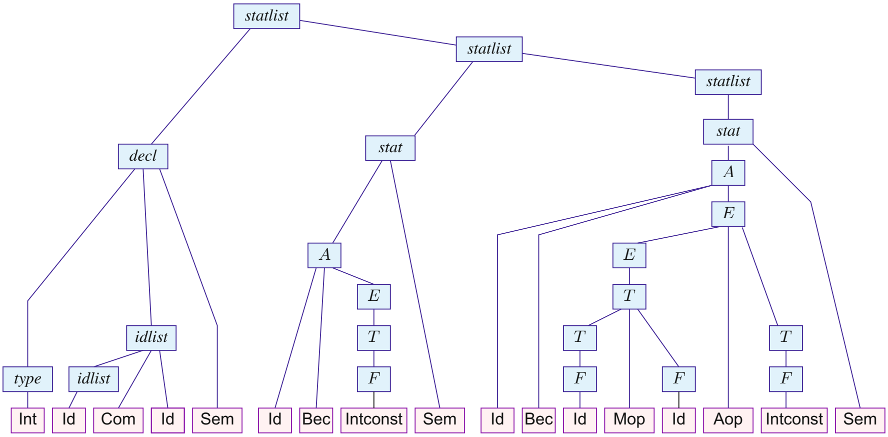


Fig. 4.1 Representation of the concrete syntax

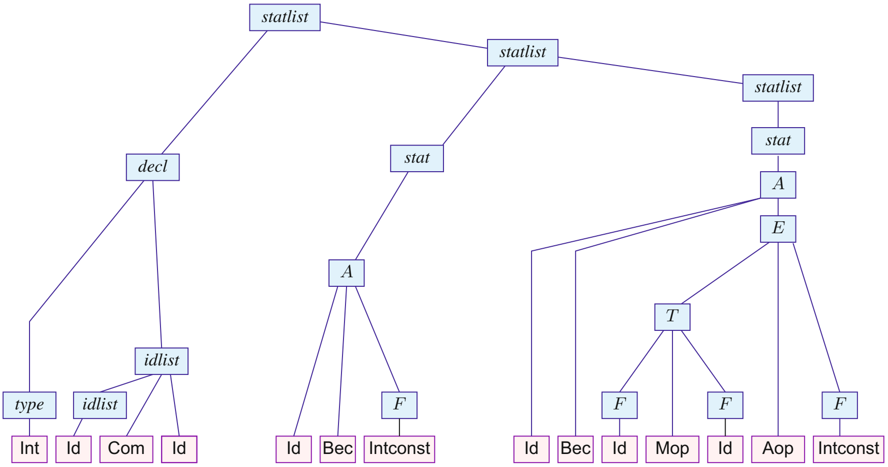


Fig. 4.2 Representation of the abstract syntax

the grammar G can be systematically rewritten such that all chain productions are eliminated. We have:

Theorem 4.1.1 For every CFG G, a CFG G' without chain productions can be constructed that has the following properties:

1. $L(G) = L(G')$;
2. If G is an $LL(k)$-grammar, then also G'.
3. If G is an $LR(k)$-grammar, then also G'.

Proof Assume that the CFG G is given by $G = (V_N, V_T, P, S)$. The CFG G' then is obtained as the tuple $G' = (V_N, V_T, P', S)$ with the same sets V_N and V_T

and the same start symbol S where the set P' of productions of G' consists of all productions $A \to \beta$ for which

$$A_0 \underset{G}{\Longrightarrow} A_1 \underset{G}{\Longrightarrow} \ldots \underset{G}{\Longrightarrow} A_n \underset{G}{\Longrightarrow} \beta \quad \text{whereby} \quad A = A_0, \beta \notin V_N$$

for some $n \geq 0$. The number $\#P'$ of productions in P' thus can be significantly larger than the number $\#P$ of productions of the original grammar G, but is still bounded by the product $\#N \cdot \#P$.

Let $R \subseteq V_N \times V_N$ denote the relation with $(A, B) \in R$ iff $A \to B \in P$. Then it suffices to compute the *reflexive and transitive closure* R^* of this relation in order to determine the set of all pairs (A, B) for which B can be derived from A by an arbitrary sequence of chain transitions. The reflexive and transitive closure of a binary relation can, e.g., be computed by the Floyd-Warshall algorithm [1].

The proof of properties (1), (2) and (3) is left to the reader. □

Example 4.1.2 Consider the example grammar G_1 with the productions:

$$\begin{array}{lcl} E & \to & E + T \mid T \\ T & \to & T * F \mid F \\ F & \to & (E) \mid \mathsf{Id} \end{array}$$

The relations R and R^* are given by:

R	E	T	F
E	0	1	0
T	0	0	1
F	0	0	0

R^*	E	T	F
E	1	1	1
T	0	1	1
F	0	0	1

By eliminating the chain productions, we obtain a grammar with the following productions:

$$\begin{array}{lcl} E & \to & E + T \mid T * F \mid (E) \mid \mathsf{Id} \\ T & \to & T * F \mid (E) \mid \mathsf{Id} \\ F & \to & (E) \mid \mathsf{Id} \end{array}$$

Note that the resulting grammar is no longer a $SLR(1)$-grammar, but is still an $LR(1)$-grammar. □

We conclude that chain rules are, at least when expressiveness is concerned, superfluous. The duplication of right sides, which is introduced for their elimination, is not appreciated during the *specification* of grammars. For the *application* of parse trees, on the other hand, chain rules should be avoided. One meaningful compromise, therefore, could be to allow chain rules in grammars, but delegate their elimination to the parser generator.

In the following, we will sometimes use the concrete syntax, sometimes the abstract syntax, whichever is more intuitive.

4.1.1 Rules for Validity and Visibility

Programming languages often allow us to use the same identifier for several program elements and thus to have several declarations. Thus, a general strategy is needed to determine for an applied occurrence of an identifier the defining occurrence it refers to. This strategy is specified by means of the rules for *validity visibility*.

The *scope* (range of validity) of the defining occurrences of an identifier x is that part of a program in which an applied occurrence of x can refer to this defining occurrence. In programming languages with nested blocks, the validity of a defining occurrence of an identifier stretches over the block containing the declaration. Such languages often require that there is only one declaration of an identifier in a block. All applied occurrences of the identifier refer to this single defining occurrence. Blocks, however, may be nested, and nested blocks may contain new declarations of identifiers that have already been declared in the outer block. This is the case for C and C++. While a declaration in the outer block is also valid in the nested block, it may no longer be *visible*. It may *hidden* by a declaration of the same identifier in the inner block. The range of *visibility* of a defining occurrences of an identifier is the program text in which the identifier is valid and visible, i.e., not hidden.

Java, for example, does not allow us to hide local identifiers since this is a source of unpleasant programming errors. The nested loop

```
for (int $i \leftarrow 0; i < n; i{+}{+}$)   {
    for (int $i \leftarrow 0; i < m; i{+}{+}$)   {
        ...
    }
}
```

is not possible in Java, since the inner declaration of i would overwrite the outer one. With class fields Java is not as restrictive:

```
class Test {
    int $x$;
    void foo (int $x$) {
        $x \leftarrow 5$;
    }
}
```

Method foo does not change the value of field x of its receiver object, but modifies instead the value of parameter x.

Some languages, for example, JAVA and C++, permit that the declaration of a variable is placed *somewhere* in a block before its first applied occurrence:

```
{
    int y;
    ...
    int x ← 2;
    ...
    y ← x + 1;
}
```

Variable x is valid in the whole block, but is visible only after its declaration. The first property prevents further declarations of x within the same block.

The process of *identification of identifiers* identifies for each applied occurrence of an identifier the defining occurrence that belongs to this applied occurrence according to the rules of validity and visibility of the language. The validity and the visibility rules of a programming language are strongly related to the types of nesting of scopes that the language allows.

COBOL has no nesting of scopes at all; all identifiers are valid and visible everywhere. FORTRAN77 only allows one nesting level, that is, no nested scopes. Procedures and functions are all defined in the main program. Identifiers that are defined in a block are only visible within that block. An identifier declared in the main program is visible starting with its declaration, but is hidden within procedure declarations that contain a new declaration of the identifier.

Modern imperative and object-oriented languages such as PASCAL, ADA, C, C++, C#, and JAVA as well as functional programming languages allow arbitrarily deep nesting of blocks. The ranges of validity and visibility of defining occurrences of identifiers are fixed by additional rules. In a *let* construct

$$\textbf{let}\ x = e_1\ \textbf{in}\ e_0$$

in OCAML the identifier x is only valid in the *body* e_0 of the *let*-construct. Applied occurrences of x in the expression e_1 refer to a defining occurrence of x in enclosing blocks. The scope of the identifiers $x_1, \ldots, x_n$ of a *let-rec* construct

$$\textbf{let rec}\ x_1 = e_1\ \textbf{and}\ \ldots\ \textbf{and}\ x_n = e_n\ \textbf{in}\ e_0$$

consists of all expressions $e_0, e_1, \ldots, e_n$ simultaneously. This rule makes it difficult to translate programs in one pass; when translating a declaration the compiler may need information about an identifier whose declaration has not yet been processed. PASCAL, ADA, and C therefore provide*forward*-declarations to avoid this problem.

PROLOG has several classes of identifiers, which are characterized by their syntactic positions. Since there is no hiding, validity and visibility agree. The identifiers of the different classes have the following scopes:

- Predicates, atoms, and constructors have global visibility; they are valid in the whole PROLOG program and in associated queries.
- Identifiers of clause variables are valid only in the clause in which they occur.

Explicit declarations exist only for predicates: These are defined by the list of their alternatives.

An important concept to make identifiers visible in a given context is *qualification*. Consider the expression $x.a$ in the programming language C. The type declaration which the component a refers to depends on the variable x. The variable x, more precisely the type of x, serves as a qualification of component a. Qualification is also used in programming languages with a module concept such as MODULA and OCAML to make (public) identifiers from modules visible outside of the defining module. Let A be an OCAML module. $A.f$ calls a function f declared in A. Similarly, the *use*-directive in ADA lists identifiers of surrounding scopes, making their declaration thereby visible. The visibility of these identifiers stretches from the end of the *use*-directive until the end of the enclosing program unit.

Similar concepts for qualification exist in object-oriented programming languages such as JAVA. Consider a name x in JAVA defined in a class C as public. Within a class A that is different from C and neither an inner class nor a subclass of C, the identifier x of class C is still valid.

First, consider the case that x is declared to be static. Then x exists only once for the whole class C. If class C belongs to a different package than class A, then identfiication of x needs not only class C, but in addition the name of this package. A call of the static method *newInstance*(), for instance, of class *DocumentBuilderFactory* of package *javax.xml.parsers* has the form:

$$javax.xml.parsers.DocumentBuilderFactory.newInstance()$$

Such lengthy qualifications take too much writing effort. JAVA therefore offers an *import* directive. The directive:

$$\textbf{import}\ javax.xml.parsers.*$$

at the beginning of a file makes all public classes of package *javax.xml.parsers* visible in all classes of the current file without further qualification. The call of the static method *newInstance*() of class the *DocumentBuilderFactory* then can be abbreviated to:

$$DocumentBuilderFactory.newInstance()$$

The directive:

$$\textbf{static import}\ javax.xml.parsers.DocumentBuilderFactory.*$$

at the beginning of a file, however, makes not only the class *DocumentBuilderFactory* visible, but also all static public attributes and methods of the class *DocumentBuilderFactory*.

Similar directives that make valid but not directly visible identifiers visible in the actual context exist in many programming languages. In OCAML an **open** A in a module B makes all those variables and types of module A visible in B that are public.

Things are different if a field or a method x in a JAVA program is not **static**. The class to which an instance of the identifier x belongs is determined considering the static type of the expression whose run-time value refers to the object selected by x.

Example 4.1.3 Consider the class declarations:

```
class A {               class B extends A {
    int a ← 1;              int b ← 2;
}                           int foo() {
                                A o ← new B();
                                return o.a;
                            }
                        }
```

The static type of the attribute o is A. At run time the attribute o has as value an object of subclass B of A. Visible of the objects to which o may be evaluated are only the visible attributes, methods, and inner classes of the superclass A. □

Conclusion

Not everywhere in the scope of a defining occurrence of x does an applied occurrence of x referxs to this defining occurrence. If the defining occurrence is *global* to the actual block, it may be hidden by a local (re)declaration of x. Then it is not *directly visible*. There are several possibilities within its scope, though, to make a defining occurrence of an identifier x that is not directly visible, visible. For that, many programming languages provide explicit qualification at a particular use or general directives to omit explicit qualifications within a given context.

4.1.2 Checking the Context-Conditions

We will now sketch how compilers check the context-conditions. We consider the simple case of a programming language with nested scopes, but without overloading.

The task is decomposed in two subtasks. The first subtask consists in checking whether all identifiers are declared and in relating their defining to applied occurrences. We call this task *declaration analysis*. This analysis is determined by the rules for validity and visibility of the programming language. The second subtask, *type checking*, examines whether the types of program objects conform to the rules of the type system. It may also infer types for objects for which no types were given.

Identification of Identifiers

In our simple case, the rules for validity and visibility determine that in a correct program, exactly one defining occurrence of an identifier belongs to each applied occurrence of the identifier. The identification of identifiers consists in linking each applied occurrence to a defining occurrence or to find that no such linking is possible or that more than one exists. The result of this identification is later used for type checking and possibly for code generation. It must therefore be passed on to subsequent compiler phases. There exist several possibilities for the representation of the link between applied and defining occurrence. Traditionally, the compiler constructs a *symbol table*, in which the declarative information for each defining occurrence of an identifier is stored. This symbol table is frequently organized similarly to the block structure of the program. This helps to quickly reach the corresponding defining occurrence starting from an applied occurrence.

The symbol table is not the result of the identification of identifiers, but it supports their identification. The result of the identification is to establish for every node for an applied occurrence of an identifier x a link to the node of that defining occurrence of x to which it refers.

Which operations must be supported by the symbol table? For each declaration of an identifier, the identifier must be entered into the symbol table together with a reference to the declaration's node in the syntax tree. Another operation must register the opening, yet another the closing of a block. The latter operation can delete the entries for the declarations of the closed block from the symbol table. In this way, the symbol table contains at any point in time exactly the entries for declarations of all actually opened blocks that have not yet been closed. When the declaration analyzer arrives at an applied occurrence of an identifier it searches the symbol table according to the rules for validity and visibility for the entry of the corresponding defining occurrence. When it has found this entry it copies the reference to the declaration node to the node for the applied occurrence.

Thus, the following operations on the symbol table are required:

create_table() creates an empty symbol table.

enter_block() registers the opening of a new block.

exit_block() resets the symbol table to the state before the last enter_block.

enter_id(*id*, *ref*) enters an entry for identifier *id* into the symbol table. This contains the reference *ref* to its declaration node.

search_id(*id*) searches the defining occurrence to *id* and returns the reference to the declaration node if it exists and **null** otherwise.

search_block_id(*id*) returns for *id* the reference to the corresponding declaration node within the current block or **null**, if such a declaration does not exist.

Example 4.1.4 We want to apply a symbol table for annotating the parse tree for a simple fragment of a C-like imperative language without functions or procedures.

$$\begin{array}{lcl}
\langle decl \rangle & \longrightarrow & \langle type \rangle \;\; \mathsf{var}; \\
\langle type \rangle & \longrightarrow & \mathbf{int} \\
\langle stat \rangle & \longrightarrow & \mathsf{var} = E; \\
\langle stat \rangle & \longrightarrow & \{ \; \langle block \rangle \; \} \\
E & \longrightarrow & \mathsf{const} \;\mid\; \mathsf{var} \\
\langle block \rangle & \longrightarrow & \langle decl \rangle \;\; \langle block \rangle \\
\langle block \rangle & \longrightarrow & \langle stat \rangle \;\; \langle block \rangle \\
\langle block \rangle & \longrightarrow & \epsilon
\end{array}$$

In order to keep the example grammar small, only a minimalistic set of types and productions for the nonterminals have been included.

In order to assign declarations to uses of identifiers, we consider a simple internal representation that closely resemples the parse tree. Every node of the parse tree is represented by an object whose class name equals the corresponding terminal or nonterminal symbol. Every such object has an array *succs* of references to the successor nodes. Internal nodes corresponding to nonterminals additionally have an attribute *rhs* containing the right side of the corresponding production.

Assume that the tokens of the class var serve as identifiers of variables and are equipped with an attribute *id* containing their concrete name. Additionally, each such token obtains an attribute *ref* that is meant to receive a reference to the declaration to which the identifier refers. When computing these references, the algortihm should take into account that new declarations of an identifier within the same block are ruled out, while they are admitted within subblocks. The attributes *ref* can be computed by means of a *depth-first left-right* traversal over the parse tree. Thereby, a method process() is called for every visited node, which may behave differently depending on the class of the node.
For the class $\langle decl \rangle$, we define:

```
void process()   {
        ⟨decl⟩ ref;
        switch (rhs)   {
          case ′⟨type⟩ var;′ :   ref ← table.search_block_id(succs[1].id);
                                 if (ref ≠ null)   error();
                                 else   table.enter_id(succs[1].id, this);
                                 return;
        }
}
```

For the class $\langle block \rangle$, we define:

```
void process()  {
    switch (rhs){
     case ′ ⟨decl⟩ ⟨block⟩′ :  succs[0].process();
                               succs[1].process();
                               return;
     case ′ ⟨stat⟩ ⟨block⟩′ :  succs[0].process();
                               succs[1].process();
                               return;
     case ′ϵ′ :                return;
    }
}
```

For the class $\langle stat \rangle$ we define:

```
void process()  {
    ⟨decl⟩ ref;
    switch (rhs)  {
     case ′var = E;′ :   ref ← table.search_id(succs[0].id);
                         if (ref = null)  error();
                         else  succs[0].ref ← ref;
                         return;
     case ′{⟨block⟩}′ :  table.enter_block();
                         succs[1].process();
                         table.exit_block();
                         return;

    }
}
```

The class E has:

```
void process()  {
    switch (rhs)  {
     case ′const′ :  return;
     case ′var′ :    ref ← table.search_id(succs[0].id);
                     if (ref = null)  error();
                     else  succs[0].ref ← ref;
                     return;
    }
}
```

During the traversal of the parse tree, the visibility rules for identifiers must be taken into account. If, for example, a new declaration of a variable x is forbidden within the same block, then enter_id must only be executed if x has not yet been declared within the actual block. □

For more complicated programming languages such as the one from Example 4.1.4, a *depth-first left-right* traversal over the parse tree often is not sufficient. In JAVA, for example, members such as methods or attributes may already be used, before they syntactically occur within the declaration of the class. A first pass over the class must therefore collect all declarations in order to map uses of identifiers to their declarations in the second pass. A similar procedure is required for functional languages in definitions of mutually recursive functions.

A specific characteristics of PASCAL-like programming languages as well as C is to stick with the *depth-first left-right* traversal, but to insert a *forward* declaration before the first use of a function or procedure. Such a forward declaration consists of the name together with the return type and the list of parameters.

Example 4.1.5 As an illustration, we extend the grammar from Example 4.1.4 by means of parameterless procedures, which may also occur in inner blocks, and forward declarations. For that, we add the following productions:

$$\begin{array}{lcl}\langle decl\rangle & \longrightarrow & \mathsf{void\ var\ ();} \\ \langle decl\rangle & \longrightarrow & \mathsf{void\ var\ ()}\ \{\ \langle block\rangle\ \} \\ \langle stat\rangle & \longrightarrow & \mathsf{var();}\end{array}$$

The method process() for declarations now are extended to deal with the cases of forward declarations and declarations of procedures:

```
...
case 'void var ();' :          $ref \leftarrow table$.search_block_id();
                               if ($ref$ = null)   $table$.enter_id($succs[1].id$, this);
                               else   return;
case 'void var () {⟨block⟩};' : $ref \leftarrow table$.search_block_id();
                               if ($ref$ = null)   $table$.enter_id($succs[1].id$, this);
                               else   {
                                      if ($ref.impl \neq$ null)   error();
                                      else   $ref.impl \leftarrow$ this;
                               }
                               $succs[5]$.process();
                               return;
...
```

For the declaration of a procedure that provides an implementation must be checked that so far only forward declarations of the given procedure have been seen. For that, ⟨*decl*⟩-objects are equipped with an attribute *impl*, which is initialized with **null** at forward declarations. All other declarations should set it to a value different from null, e.g., to the current object **this**. The attribute *impl* of a forward declaration in the symbol table is thus set to the declaration within the actual block which provides the body for the procedure. Now it only remains to extend the method process()

for ⟨*stat*⟩ for the case of procedure calls:

```
...
case 'var();' :  $ref \leftarrow$ table.search_id($succs[0].id$);
                 if ($ref$ = null)   error();
                 else   $succs[0].ref \leftarrow ref$;
                 return;
...
```

□

The treatment of forward declarations in Example 4.1.5 is not completely general. In fact, for multiple declarations of a procedure within the same block it should be checked that the various declarations also coincide in their *types*. Already the example of forward declarations thus indicates that in more complicated situations, the analysis of declaredness cannot be performed independently of the analysis of types. A similar entanglement of the analyses of declaredness and types is required in programming languages such as Java and Ada, where the static types of parameters (or even the return types) determine which implementation is to be selected.

Implementation of the Symbol Table

The implementation of a symbol table must guarantee that the method search_id() selects the correct entry for an identifier among all available entries according to the given validity rules.

One solution is to maintain all entered blocks in a stack. For every entered block a mapping is provided that maps each encountered identifier to its corresponding declaration. The method enter_block() then pushes a fresh empty mapping onto the stack, while the method exit_block pops the topmost mapping from the stack. The method *search_block_id*(*id*) looks up the identifier *id* within the topmost mapping in the stack, which corresponds to the actual block. The method *search_id*(*id*) looks up the identifier *id* within the topmost mapping in the stack. If no corresponding declaration is found, searching continues in the mapping underneath, which belongs to the enclosing block. This is repeated until the search has arrived in the outermost block. Only if no declaration was found there, the search is unsuccessful.

In this approach, entering and leaving a block requires only marginal effort. The efficiency of the methods search_block_id() and search_id() heavily depends on the choice of the data structures for the identifier mappings. If these data structures are simple *lists*, the runtime within each block grows linearly with the number of identifiers declared in this block. Logarithmic runtime is achieved if, e.g., *binary search trees* are used. Virtually constant runtimes, however, are obtained by means of suitable *hash maps*.

Even when a highly efficient implementation of identifier mappings is chosen, the runtime of the method search_id() may grow linearly with the depth up to which blocks are nested. For a programming language where deep nesting of blocks is common, it therefore can be advantageous to choose an implementation of the symbol table where the runtime of search_id() is *independent* of the nesting depth. The alternative implementation no longer organizes the symbol table as a stack of identifier mappings, but instead as a single mapping that assigns to each identifier a dedicated stack of declarations. At each moment, the declaration stack of an identifier x contains the references to all currently valid defining occurrences of x. A new declaration for x will be pushed onto this stack. Looking up the latest declaration of x therefore is a constant-time operation. The realization of the method exit_block() that implements the exit from the actual block b is slightly more complicated. For this operation, all declarations of the block b must be removed from the table.

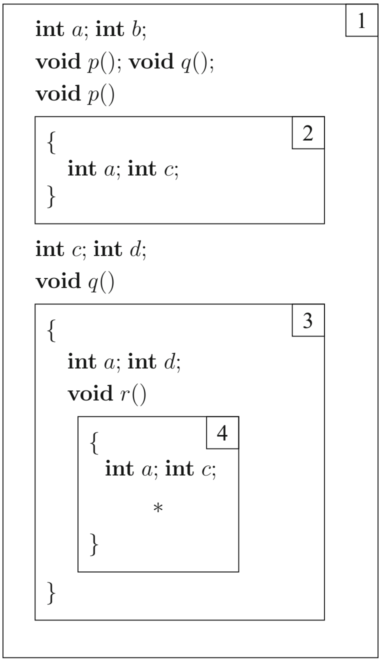
The method exit_block() can be easily realized, if for each block, all identifiers declared in this block are additionally collected in a list. These lists are maintained in a separate stack of blocks. A method call enter_block() allocates a new empty list on top of that stack. The method call enter_id(x, ref) records the identifier x in the list for the actual block and pushes the reference to the declaration of the identifier x onto the declaration stack of x. The method call exit_block() takes the list of all declared identifiers of the actual block and removes all its declarations from the corresponding declaration stacks. Afterwards, the list is popped from the stack of blocks. The extra costs for maintaining the stack of blocks can be distributed over the individual calls to the method enter_id() and thus increase their runtimes by only a (small) constant factor.

Example 4.1.6 For the program in Fig. 4.3 and the program point marked by $*$, the symbol table from Fig. 4.4 is obtained. The light entries represent references to declarations. The modified implementation of the symbol table that maintains a separate declaration stack for each identifier is displayed in Fig. 4.5. Additionally, the stack of blocks is shown whose entries list the identifiers that have been allocated in each entered block. □

Checking Type Consistency

A single *bottom-up* run over a expression tree allows us to determine the type of the corresponding expression and at the same time to check whether the occurring constants, variables, and operators are used consistently with their types. For the terminal operands, the types are already given; for identifiers of variables, the type can be retrieved from the corresponding defining occurrence. For every operator application, it is checked whether the actual parameters are type-consistent with types of the parameters and if so, the return type of the operator provides the type for the result of the operator application.

Sometimes, the programming language supports automatic *type casts*, for example, from the type **int** to **float** or **double**, or from a subclass to a superclass. These must be taken into account when checking for type consistency. In JAVA, the required type casts are identified and inserted during the *bottom-up* run of type checking. In ADA, on the other hand, no automatic type casts are provided.



```
int a; int b;                          1
void p(); void q();
void p()
  {                                    2
    int a; int c;
  }
int c; int d;
void q()
  {                                    3
    int a; int d;
    void r()
      {                        4
        int a; int c;
              *
      }
  }
```

Fig. 4.3 Nested scopes of validity

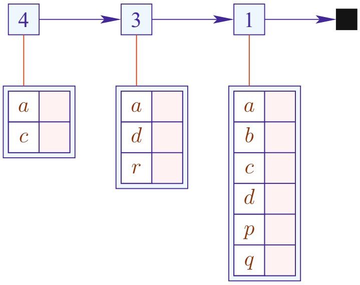


Fig. 4.4 Symbol table for the program of Fig. 4.3

As in JAVA, though, ADA supports multiple operators having the same name. Such operators are called *overloaded*. For overloaded operaots, the programming language must provide rules that determine at every applied occurrence of the operator which operation should be selected.

4.1.3 Overloading of Identifiers

A symbol is called *overloaded* if it may have several meanings at some point in the program. Already mathematics knows overloaded symbols, for example, the arithmetic operators, which, depending on the context, may denote operations on integral, real, or complex numbers, or even operations in rings and fields. The early

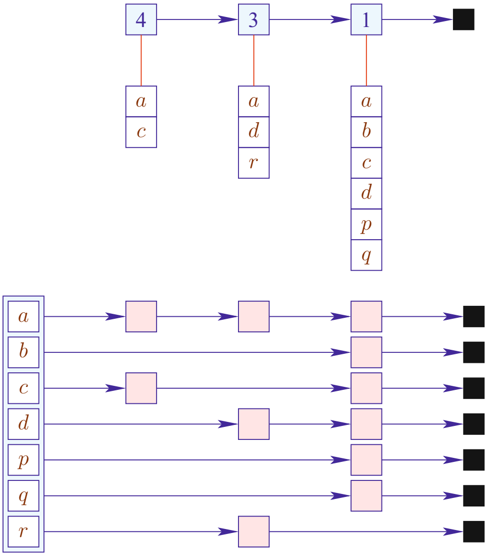


Fig. 4.5 Modified symbol table for the program of Fig. 4.3

programming languages FORTRAN and ALGOL60 have followed the mathematical tradition and admitted overloaded operators.

Programming languages such as ADA or JAVA also allow overloading of user-defined symbols such as procedure, function, or method names. In correct programs one applied occurrence of an identifier x then may correspond to several defining occurrences of x. A redeclaration of an identifier x only hides an enclosing declaration of x if both cannot be distinguished by means of their types. In JAVA, this distinction is based on the types of the parameter lists only. Therefore, the resolution of overloading can be integrated into the *bottom-up* run of type-checking for expressions. In ADA, on the other hand, also return types are taken into account. A program is correct only if the *environment* of each applied occurrence allows the compiler to select exactly one defining occurrence. Thereby, the *environment* for procedure and function calls consists of the combination of the actual parameters and the requested return type.

Example 4.1.7 (ADA program (Istvan Bach))

```
procedure BACH is
    procedure put (x: boolean) is begin null; end;
    procedure put (x: float)   is begin null; end;
    procedure put (x: integer) is begin null; end;
    package x is
       type boolean is (false, true);
       function f return boolean;                          -- (D1)
    end x;
    package body x is
```

```
       function f return boolean is begin null; end;
    end x;
    function f return float is begin null; end;           -- (D2)
    use x;
begin
    put (f);                                              -- (A1)
    A: declare
       f: integer;                                        -- (D3)
    begin
       put (f);                                           -- (A2)
       B: declare
          function f return integer is begin null; end;   -- (D4)
          begin
             put (f);                                     -- (A3)
           end B;
    end A;
end BACH;
```

The package `x` declares in its public part two new identifiers, namely the type identifier `boolean` and the function identifier `f`. These two identifiers are made potentially visible after the semicolon by the use directive `use x;` (see after (D2)). Function identifiers in ADA can be overloaded. The two declarations of `f`, at (D1) and (D2), have different parameter profiles, in this case different result types. Both are therefore (potentially) visible at program point (A1).

The declaration `f: integer` in the program unit `A` (see (D3)) hides the outer declaration (D2) of `f`, since variable identifiers in ADA cannot be overloaded. For this reason the declaration (D1) is not visible. Declaration (D4) of `f` in program unit `B` hides declaration (D3), and since this one hides declaration (D2), transitively also D2. Declaration (D1) potentially made visible through the use directive is not hidden, but still potentially visible. In the context `put(f)` (see (A3)) `f` can only refer to declaration (D4) since the first declaration of `put` uses a type, `boolean`, that is different from the result type of `f` in (D1). □

Consider an assignment $x \leftarrow e$ in the programming language ADA. The overloading of operators in the expression e can be resolved in two passed over the (abstract) syntax tree of e. We assume that for the left side x of the assignment as well as for every variable and every constant in e the type has already been determined. Moreover, we require for each occurrence of an operator f in e the set of all potential implementations. A first *bottom-up* run over e determines for each subexpression e' of e the set $\tau[e']$ of *potential* types of e'.

If e' is a constant or a variable of type t, $\tau[e']$ equals the set $\{t\}$. If e' is an operator application $f(e_1, \ldots, e_k)$, then the set $\tau[e']$ is given by the set of all result types of potential implementations of f whose tuple $(t_1, \ldots, t_k)$ of parameter types is contained in $\tau[e_1] \times \ldots \times \tau[e_k]$.

The selection of the right operator implementations then is realized in a second run select. This second run tries to select implementations of operators in such way that the result type for e equals the type of the left side x of the assignment. Since $\tau[e]$ contains all potential types of e, this type must be contained in $\tau[e]$. The select-run propagates the result type, which is required by the context, *top-down* to each subexpression e' of e.

Assume that the required result type for an operator application $f(e_1, \ldots, e_k)$ is t'. Then an implementation of f must be selected whose result type equals t' and which additionally has a tuple $(t_1, \ldots, t_k)$ of parameter types such that $t_i \in \tau[e_i]$ for all $i = 1, \ldots, k$. If there is exactly one such implementation, this implementation of f is selected. Then select proceeds with the subexpressions e_i where t_i is the required result type for e_i. If there is more than one such implementation of f, the program is rejected.

4.2 Type Inference

Imperative languages typically require us to supply types for identifiers. These are used to derive types of expressions. In modern *functional* programming languages, however, not only the types of expressions, but also the types of identifiers are automatically inferred. Therefore new identifiers are introduced in programs (mostly) without associating them with types.

Example 4.2.1 Consider the following OCAML function:

$$\begin{array}{lll} \textbf{let rec}\ \mathsf{fac} & = \textbf{fun}\ x \rightarrow & \textbf{if}\ x \leq 0\ \textbf{then}\ 1 \\ & & \textbf{else}\ \ x \cdot \mathsf{fac}\ (x-1) \end{array}$$

An arithmetic operation for integral numbers is applied to the argument x of the function fac. Therefore the argument type of fac must be **int**. The return value is either 1 or it is computed using the operator $\cdot$ for integral numbers. Accordingly, the type of the return value must again be **int**. The OCAML compiler infers that the function fac has the type **int** → **int**, meaning that fac is a function that takes *int*-values as arguments and returns *int*-values as results. □

The idea to automatically infer types goes back to J.R. Hindley and R. Milner. We follow them and characterize the set of potential types of an expression by introducing axioms and inference rules, which relate the type of an expression to the types of its subexpressions. For simplicity we only consider a functional *core language*, derived from OCAML. A similar functional core language is also considered in the volume *Compiler Design: Virtual Machines*. A program in this programming language is an expression without free variables, where expressions e are built according to the following grammar:

$$
\begin{array}{rcl}
e & ::= & b \mid x \mid (\Box_1\, e) \mid (e_1\, \Box_2\, e_2) \\
 & \mid & (\mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2) \\
 & \mid & (e_1, \ldots, e_k) \mid [\,] \mid (e_1 :: e_2) \\
 & \mid & (\mathbf{match}\ e_0\ \mathbf{with}\ [\,] \rightarrow e_1 \mid h :: t \rightarrow e_2) \\
 & \mid & (\mathbf{match}\ e_0\ \mathbf{with}\ (x_1, \ldots, x_k) \rightarrow e_1) \\
 & \mid & (e_1\, e_2) \mid (\mathbf{fun}\ x \rightarrow e) \\
 & \mid & (\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_0) \\
 & \mid & (\mathbf{let\ rec}\ x_1 = e_1\ \mathbf{and}\ \ldots\ \mathbf{and}\ x_n = e_n\ \mathbf{in}\ e_0)
\end{array}
$$

Here b are basic values, x are variables, and $\Box_i$ $(i = 1, 2)$ are i-place operators on basic values. For simplicity we consider as structured data types only tuples and lists. Pattern matching can be used to decompose structured data. As patterns for the decomposition we admit only patterns with exactly one constructor. We use the usual precedence rules and associativities to spare parentheses.

Example 4.2.2 A first example program in the functional core language is:

$$
\begin{array}{lcllll}
\mathbf{let}\ \mathsf{rev} & = & \mathbf{fun}\ x \rightarrow & & & \\
 & & \mathbf{let\ rec}\ \mathsf{r} & = & \mathbf{fun}\ x \rightarrow \mathbf{fun}\ y \rightarrow \mathbf{match}\ x\ \mathbf{with} & \\
 & & & & \qquad [\,] \rightarrow y & \\
 & & & \mid & \qquad h :: t \rightarrow \mathsf{r}\ t\ (h :: y) & \\
 & & \mathbf{in} & & \mathsf{r}\ x\ [\,]; & \\
\mathbf{in} & & \mathsf{rev}\ [1; 2; 3] & & &
\end{array}
$$

Here, the abbreviating notation

$$[a_1; \ldots; a_n]$$

stands for lists

$$a_1 :: (a_2 :: (\ldots (a_n :: [\,]) \ldots)$$

□

We use a syntax for types that is also similar to that of OCAML, that is, the unary type constructor list for lists is written to the right of its arguments. Types t are built according to the following grammar:

$$t \quad ::= \quad \mathbf{int} \mid \mathbf{bool} \mid (t_1 * \ldots * t_m) \mid t\ \mathsf{list} \mid (t_1 \rightarrow t_2)$$

The only *basic types* we consider are the type **int** for integral numbers and the type **bool** for boolean values. Expressions may contain *free variables*. The type of an expression depends on the types of free variables that occur in it. The assumptions about the types of free variables are collected in a *type environment*. A type environment Γ is a function mapping finite sets of variables to the set of types. A type environment Γ for an expression e maps each free variable x of e to a type t, but may also associate types with further variables. The *type!judgment* that the expression e has the type t under the assumption Γ is written in short as:

$$\Gamma \vdash e \ : \ t$$

A *type system* consists of a set of axioms and of rules by which *valid* type judgments can be inferred. Axioms are judgments that are valid without further assumptions. Rules permit us to derive new valid type judgments from valid preconditions. We now list the axioms and rules for our functional core language. As axioms we need:

$$\begin{array}{lll}
\textsc{Const:} & \Gamma \vdash b \ : \ t_b & (t_b \quad \text{type of basic value } b) \\
\textsc{Nil:} & \Gamma \vdash [\,] \ : \ t \ \mathsf{list} & (t \quad \text{arbitrary}) \\
\textsc{Var:} & \Gamma \vdash x \ : \ \Gamma(x) & (x \quad \text{variable})
\end{array}$$

Each family of axioms is given a name for later reference. Furthermore, we assume that each basic value b has a uniquely determined basic type t_b, syntactically associated with it.

Rules are also given names. The preconditions of a rule are written above the line; the conclusion is written below.

$$\textsc{Op:} \quad \frac{\Gamma \vdash e_1 \ : \ \mathbf{int} \quad \Gamma \vdash e_2 \ : \ \mathbf{int}}{\Gamma \vdash e_1 + e_2 \ : \ \mathbf{int}}$$

$$\textsc{Comp:} \quad \frac{\Gamma \vdash e_1 \ : \ t \quad \Gamma \vdash e_2 \ : \ t}{\Gamma \vdash e_1 = e_2 \ : \ \mathbf{bool}}$$

$$\textsc{If:} \quad \frac{\Gamma \vdash e_0 \ : \ \mathbf{bool} \quad \Gamma \vdash e_1 \ : \ t \quad \Gamma \vdash e_2 \ : \ t}{\Gamma \vdash (\mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2) \ : \ t}$$

$$\textsc{Tupel:} \quad \frac{\Gamma \vdash e_1 \ : \ t_1 \quad \ldots \quad \Gamma \vdash e_m \ : \ t_m}{\Gamma \vdash (e_1, \ldots, e_m) \ : \ (t_1 * \ldots * t_m)}$$

$$\textsc{Cons:} \quad \frac{\Gamma \vdash e_1 \ : \ t \quad \Gamma \vdash e_2 \ : \ t \ \mathsf{list}}{\Gamma \vdash (e_1 :: e_2) \ : \ t \ \mathsf{list}}$$

$$\textsc{Match}_1: \quad \frac{\Gamma \vdash e_0 \ : \ (t_1 * \ldots * t_m) \quad \Gamma \oplus \{x_1 \mapsto t_1, \ldots, x_m \mapsto t_m\} \vdash e_1 \ : \ t}{\Gamma \vdash (\mathbf{match}\ e_0\ \mathbf{with}\ (x_1, \ldots, x_m) \to e_1) \ : \ t}$$

$$\textsc{Match}_2: \quad \frac{\Gamma \vdash e_0 \ : \ t_1 \ \mathsf{list} \quad \Gamma \vdash e_1 \ : \ t \quad \Gamma \oplus \{x \mapsto t_1, y \mapsto t_1 \ \mathsf{list}\} \vdash e_2 \ : \ t}{\Gamma \vdash (\mathbf{match}\ e_0\ \mathbf{with}\ [\,] \to e_1 \mid x :: y \to e_2) \ : \ t}$$

$$\textsc{App:} \quad \frac{\Gamma \vdash e_1 \ : \ t_1 \to t_2 \quad \Gamma \vdash e_2 \ : \ t_1}{\Gamma \vdash (e_1\ e_2) \ : \ t_2}$$

$$\textsc{Fun:} \quad \frac{\Gamma \oplus \{x \mapsto t_1\} \vdash e \ : \ t_2}{\Gamma \vdash \mathbf{fun}\ x \to e \ : \ t_1 \to t_2}$$

$$\textsc{Let:} \quad \frac{\Gamma \vdash e_1 \ : \ t_1 \quad \Gamma \oplus \{x_1 \mapsto t_1\} \vdash e_0 \ : \ t}{\Gamma \vdash (\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_0) \ : \ t}$$

$$\textsc{Letrec:} \quad \frac{\Gamma' \vdash e_1 \ : \ t_1 \quad \ldots \quad \Gamma' \vdash e_m \ : \ t_m \quad \Gamma' \vdash e_0 \ : \ t}{\Gamma \vdash (\mathbf{let\ rec}\ x_1 = e_1\ \mathbf{and} \ldots \mathbf{and}\ x_m = e_m\ \mathbf{in}\ e_0) \ : \ t}$$

$$\text{where} \quad \Gamma' = \Gamma \oplus \{x_1 \mapsto t_1, \ldots, x_m \mapsto t_m\}$$

In the rule Op has been displayed for the integral operator $+$. Analogous rules are provided for the other unary and binary operators. In the case of boolean operators the arguments and the result are of type **bool**. For comparison operators, the rule Comp s shown for the comparison operator $=$. Analogous rules are provided in Ocaml for the other comparison operators. Please note that according to the semantics of Ocaml, comparisons are allowed between arbitrary values as long as they have the same type.

Example 4.2.3 For the body of the function fac of Example 4.2.1 and the type environment

$$\Gamma = \{\mathsf{fac} \mapsto \textbf{int} \to \textbf{int}, x \mapsto \textbf{int}\}$$

we obtain the following type derivation:

$$\frac{\dfrac{\Gamma \vdash x : \textbf{int} \quad \Gamma \vdash 0 : \textbf{int}}{\Gamma \vdash x \leq 0 \;:\; \textbf{bool}} \qquad \Gamma \vdash 1 \;:\; \textbf{int} \qquad \dfrac{\Gamma \vdash x : \textbf{int} \quad \dfrac{\Gamma \vdash \mathsf{fac} : \textbf{int} \to \textbf{int} \quad \dfrac{\Gamma \vdash x : \textbf{int} \quad \Gamma \vdash 1 : \textbf{int}}{\Gamma \vdash x - 1 : \textbf{int}}}{\Gamma \vdash \mathsf{fac}\ (x - 1) \;:\; \textbf{int}}}{\Gamma \vdash x \cdot \mathsf{fac}\ (x - 1) \;:\; \textbf{int}}}{\Gamma \vdash \textbf{if}\ x \leq 0\ \textbf{then}\ 1\ \textbf{else}\ x \cdot \mathsf{fac}\ (x - 1) \;:\; \textbf{int}}$$

Under the assumption that fac has type $\textbf{int} \to \textbf{int}$, and x has type **int**, it can be inferred that the body of the function fac has type **int**. □

The rules are designed in such a way that the type of an expression is preserved in the evaluation of the expression. This property is called *subject reduction*. If the types of all variables are *guessed* correctly at their definition, the rules could be used to check whether the guesses are *consistent*. Note that an expression might have several types.

Example 4.2.4 The expression id, given by

$$\textbf{fun}\ x \to\ x$$

describes the identity function. In each type environment Γ and for each type t

$$\Gamma \vdash \mathsf{id} \;:\; t \to t$$

can be derived. □

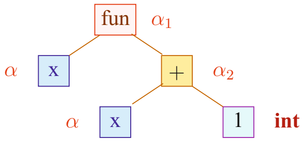
A system of equations is set up to systematically derive the set of all potential types of an expression. The solutions of this system of equations characterize the consistent bindings of variables and expressions to types. This system of equations can constructed by means of the following steps.

First, the names of variables in the program are made *unique*. Then we extend type terms by allowing additional *type variables* for unknown types of variables and subexpressions. Finally, equations between type variables are collected that must hold due to the axioms and rules of the type system when applied to the subexpressions of the program.

Example 4.2.5 Consider the function:

$$\textbf{fun } x \to x + 1$$

with the syntax tree:



As type variable for variable x we choose α, while α_1 and α_2 denote the types of the expressions **fun** $x \to x + 1$ and $x + 1$, respectively. The type rules for functions and operator applications then generate the following equations:

$$\begin{array}{lllll} \textsc{Fun} & : & \alpha_1 & = & \alpha \to \alpha_2 \\ \textsc{Op} & : & \alpha_2 & = & \textbf{int} \\ & & \alpha & = & \textbf{int} \\ & & \textbf{int} & = & \textbf{int} \end{array}$$

It follows that

$$\alpha = \textbf{int} \qquad \alpha_1 = \textbf{int} \to \textbf{int} \qquad \alpha_2 = \textbf{int}$$

must hold. □

Let $\alpha[e]$ denote the type variable for the expression e. Each rule application generates the following equations:

CONST:	$e \equiv b$	$\alpha[e] = t_b$
NIL:	$e \equiv []$	$\alpha[e] = \alpha$ list $(\alpha$ new$)$
OP:	$e \equiv e_1 + e_2$	$\alpha[e] = \textbf{int}$ $\alpha[e_1] = \textbf{int}$ $\alpha[e_2] = \textbf{int}$
COMP:	$e \equiv e_1 = e_2$	$\alpha[e_1] = \alpha[e_2]$ $\alpha[e] = \textbf{bool}$
TUPEL:	$e \equiv (e_1, \ldots, e_m)$	$\alpha[e] = (\alpha[e_1] * \ldots * \alpha[e_m])$
CONS:	$e \equiv e_1 :: e_2$	$\alpha[e_2] = \alpha[e_1]$ list $\alpha[e] = \alpha[e_1]$ list
IF:	$e \equiv \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2$	$\alpha[e_0] = \textbf{bool}$ $\alpha[e] = \alpha[e_1]$ $\alpha[e] = \alpha[e_2]$

MATCH1:	$e \equiv \textbf{match}\ e_0\ \textbf{with}\ (x_1, \ldots, x_k)\ \rightarrow\ e_1$	$\alpha[e_0] = (\alpha[x_1] * \ldots * \alpha[x_k])$ $\alpha[e] = \alpha[e_1]$
MATCH2:	$e \equiv \textbf{match}\ e_0\ \textbf{with}\ []\ \rightarrow\ e_1\ \mid\ x :: y\ \rightarrow\ e_2$	$\alpha[y] = \alpha[x]\ \mathsf{list}$ $\alpha[e_0] = \alpha[x]\ \mathsf{list}$ $\alpha[e] = \alpha[e_1]$ $\alpha[e] = \alpha[e_2]$
FUN:	$e \equiv \textbf{fun}\ x\ \rightarrow\ e_1$	$\alpha[e] = \alpha[x]\ \rightarrow\ \alpha[e_1]$
APP:	$e \equiv e_1\ e_2$	$\alpha[e_1]\ =\ \alpha[e_2]\ \rightarrow\ \alpha[e]$
LETREC:	$e \equiv$ $\textbf{let rec}\ x_1\ = e_1\ \textbf{and} \ldots\ \textbf{and}\ x_m\ = e_m\ \textbf{in}\ e_0$	$\alpha[x_1] = \alpha[e_1] \quad \ldots$ $\alpha[x_m] = \alpha[e_m]$ $\alpha[e] = \alpha[e_0]$

Example 4.2.6 For the expression id $\equiv$ **fun** $x \to x$ in Example 4.2.5 the following equation is obtained:

$$\alpha[\mathsf{id}] = \alpha[x] \to \alpha[x]$$

Different solutions of this equation are obtained if different types t are chosen for $\alpha[x]$. □

What is the relation between the system of equations for an expression e and the type judgments derivable for this expression? Let us assume that all variables occurring in e are unique. Let V be the set of variables occurring in e. In the following we only consider *uniform* derivations. These are derivations of type judgments all using the same type environment. Each type judgment $\Gamma \vdash e : t$ for a type environment Γ for the free variables in e can be converted into a uniform derivation of a type judgment $\Gamma' \vdash e : t$ for a Γ' that agrees with Γ on the free variables of e. It follows:

Theorem 4.2.1 Let e be an expression, V be the set of variables that occur in e, and let E be the equations system for the expression e. Then the following holds.

1. If σ is a solution of E, then there is a uniform derivation of the judgment:

$$\Gamma \vdash e : t$$

   for

$$\Gamma = \{x \mapsto \sigma(\alpha[x]) \mid x \in V\} \qquad \text{and} \qquad t = \sigma(\alpha[e])$$

2. Let A be a uniform derivation for a type judgment $\Gamma \vdash e : t$, in which for each subexpression e' of e a type judgment $\Gamma \vdash e' : t_{e'}$ is derived. Then the substitution σ defined by:

$$\sigma(\alpha[e']) = \begin{cases} \Gamma(x) & \text{if } e' \equiv x \in V \\ t_{e'} & \text{if } e' \text{ subterm of } e \text{ otherwise} \end{cases}$$

   is a solution of E.

Theorem 4.2.1 states that all valid type judgments can be read off the solutions of the system of equations to an expression. The systems of equations that occur here construct equalities between type *terms*. The process of solving such systems of equalities between terms is called *unification*.

Example 4.2.7

1. Consider the equation
$$Y = X \to X$$
where $\to$ is a binary constructor in infix notation. The set of solutions of this equation is the substitution
$$\{X \mapsto t, Y \mapsto (t \to t)\}$$
for each term t. One such possible term t is the variable X itself.
2. The equation:
$$X \to \mathbf{int} = \mathbf{bool} \to Z$$
has exactly one solution, namely the substitution
$$\{X \mapsto \mathbf{bool}, Y \mapsto \mathbf{int}\}$$
3. The equation
$$\mathbf{bool} = X \to Y$$
has *no* solution. □

We introduce some notions: A substitution σ is *idempotent* if $\sigma \circ \sigma = \sigma$ holds. This means that no variable X with $\sigma(X) \neq X$ occurs in the image of σ. Thus, the substitution $\{X \mapsto \mathbf{bool}, Y \mapsto Z\}$ is idempotent, but the substitution $\{X \mapsto \mathbf{bool}, Y \mapsto X\}$ is not.

For systems of term equations it suffices to consider idempotent solutions. An idempotent solution σ of a system of term equations E is *most general*, if for each other idempotent solution τ of E, $\tau = \tau' \circ \sigma$ holds for a suitable substitution τ'. Thus the substitution $\{Y \mapsto (X \to X)\}$ is a most general idempotent solution of the equation $Y = X \to X$ of Example 4.2.7. The following theorem characterizes the idempotent solutions of finite sets of term equations.

Theorem 4.2.2 Each system of equations $s_i = t_i, i = 1, \ldots, m$, between terms s_i, t_i either has *no* solution or a *most general idempotent* solution. □

By Theorem 4.2.2, computing all (idempotent) solutions of a system of term equations is reduced to computing one most general solution—given that the system has at least one solution. Most general solutions may become very large. Consider the following system of equations:

$$X_0 = (X_1 \to X_1), \ldots, X_{n-1} = (X_n \to X_n)$$

for $n \geq 1$. The most general solution of this system of equations maps X_0 to a term with 2^n occurrences of the variables X_n. According to the same principle, an OCAML program can be constructed that may lead to exponentially large type terms. In most practically useful programs those large type terms do not occur. Various techniques are known how to determine a most general idempotent solution to a finite set of term equations. One such method has been described in the volume *Compiler Design: Virtual Machines*, when we presented a virtual machine for the programming language PROLOG. In PROLOG, unification is a basic operation in the semantics. This chapter presents a functional program that for a list of pairs $(s_i, t_i), i = 1, \ldots, m$, computes a most general idempotent solution of the system of equations $s_i = t_i, i = 1, \ldots, m$.

This functional program consists of the following functions:

- The function occurs takes a pair, consisting of a variable X and a term t, and checks whether X occurs in t.
- The function unify takes a pair (s, t) of terms and an idempotent substitution θ and checks whether the solution $\theta(s) = \theta(t)$ has a solution. It returns Fail if that is not the case. Otherwise it returns a most general idempotent substitution θ' that satisfies this system of equations and additionally is a specialization of θ, that is, for which $\theta' = \theta' \circ \theta$ holds.
- The function unifyList takes a list $[(s_1, t_1); \ldots; (s_m, t_m)]$ of term pairs and an idempotent substitution θ and checks, whether the system of equations $\theta(s_i) = \theta(t_i), i = 1, \ldots, m$, has a solution. It returns Fail if this is not the case, otherwise it returns a most general idempotent substitution θ' that satisfies all equations and is a specialization of θ.

The function occurs is defined by:

```
let rec occurs $(X, t)$ = match $t$
            with $X$                     $\to$ true
              |  $f(t_1, \ldots, t_k)$   $\to$ occurs $(X, t_1) \vee \ldots \vee$ occurs $(X, t_k)$
              |  _                       $\to$ false
```

We defined the function occurs for one generic constructor f of arity $k \geq 1$ instead of all possible constructors that may occur in a program text. The functions unify and unifyList are simultaneously recursive. They are defined by:

```
let rec unify $(s, t)\ \theta$ = if $\theta\ s \equiv \theta\ t$ then $\theta$
              else match $(\theta\, s, \theta\, t)$
                   with $(X, t)$ $\to$ if occurs $(X, t)$ then Fail
                                          else $\{X \mapsto t\} \circ \theta$
                     |  $(t, X)$ $\to$ if occurs $(X, t)$ then Fail
                                          else $\{X \mapsto t\} \circ \theta$
                     |  $(f(s_1, \ldots, s_k), f(t_1, \ldots, t_k))$
                            $\to$ unifyList $[(s_1, t_1), \ldots, (s_k, t_k)]\ \theta$
                     |  _ $\to$ Fail
```

```
and unifyList list $\theta$ = match list
          with [ ] → $\theta$
             | ((s,t) :: rest) → let $\theta$ = unify (s,t) $\theta$
                                 in if $\theta$ = Fail then Fail
                                    else unifyList rest $\theta$
```

Besides a generic term constructor f we also used a generic atomic constant a. The algorithm starts with the call unifyList $[(s_1, t_1), \ldots, (s_m, t_m)]$ $\emptyset$, that is, with the system of equations together with an empty substitution. The algorithm terminates and returns either Fail, if the system of equations has no solution, or it returns an idempotent most general solution, if the system of equations has a solution.

4.2.1 Syntax-Directed Type Inference

The type-inference method described so far is not *syntax directed*. This is a disadvantage. If the system of equations to a program has no solution, no information is provided *where* the type error originates from. A precise localization of the error cause, however, is of utmost importance for the programmer. Therefore we modify the described method such that it closely follows the syntax of the program. This syntax-oriented algorithm is given as a functional program, which uses case distinction over the different possible forms of the program by pattern matching. To distinguish the syntax of the expression e from syntax of the algorithm we use capital letters for the keywords in e and put the operators in apostrophs.

A call of the function $\mathcal{W}$ is evaluated recursively over the structure of an expression e. A type environment Γ and a substitution of type variables θ is passed in additional accumulating parameters. The call returns as its result a type term t for e and a substitution accumulated during the evaluation. In the description, which now follows, the calls of the function unify are always emphasized. To increase readability we assume that the calls of unify always return substitutions. If unification at a call should fail an error message should be generated and the type inference either terminates or continues with a meaningful error recovery.

```
let rec $\mathcal{W}$ $e$ $(\Gamma, \theta)$ = match $e$
     with $c$                 → $(t_c, \theta)$
        | [ ]                 → let $\alpha$ = new()
                                in $(\alpha$ list$, \theta)$
        | $x$                 → $(\Gamma(x), \theta)$
        | $(e_1, \ldots, e_m)$ →    let $(t_1, \theta) = \mathcal{W}\ e_1\ (\Gamma, \theta)$
                                         ...
                                in let $(t_m, \theta) = \mathcal{W}\ e_m\ (\Gamma, \theta)$
                                in $((t_1 * \ldots * t_m), \theta)$
```

$$\begin{array}{lllrl}
\mid & (e_1 :: e_2) & \to & \textbf{let} & (t_1,\theta) = \mathcal{W}\; e_1\; (\Gamma,\theta) \\
 & & & \textbf{in let} & (t_2,\theta) = \mathcal{W}\; e_2\; (\Gamma,\theta) \\
 & & & \textbf{in let} & \theta = \boxed{\mathsf{unify}\; (t_1\; \mathsf{list}, t_2)\; \theta} \\
 & & & \textbf{in} & (t_2,\theta) \\
\mid & (e_1' +' e_2) & \to & \textbf{let} & (t_1,\theta) = \mathcal{W}\; e_1\; (\Gamma,\theta) \\
 & & & \textbf{in let} & (t_2,\theta) = \mathcal{W}\; e_2\; (\Gamma,\theta) \\
 & & & \textbf{in let} & \theta = \boxed{\mathsf{unify}\; (\textbf{int}, t_1)\; \theta} \\
 & & & \textbf{in let} & \theta = \boxed{\mathsf{unify}\; (\textbf{int}, t_2)\; \theta} \\
 & & & \textbf{in} & (\textbf{int},\theta) \\
\mid & (e_1' =' e_2) & \to & \textbf{let} & (t_1,\theta) = \mathcal{W}\; e_1\; (\Gamma,\theta) \\
 & & & \textbf{in let} & (t_2,\theta) = \mathcal{W}\; e_2\; (\Gamma,\theta) \\
 & & & \textbf{in let} & \theta = \boxed{\mathsf{unify}\; (t_1, t_2)\; \theta} \\
 & & & \textbf{in} & (\textbf{bool},\theta) \\
\mid & (e_1\; e_2) & \to & \textbf{let} & (t_1,\theta) = \mathcal{W}\; e_1\; (\Gamma,\theta) \\
 & & & \textbf{in let} & (t_2,\theta) = \mathcal{W}\; e_2\; (\Gamma,\theta) \\
 & & & \textbf{in let} & \alpha = \mathsf{new}\,() \\
 & & & \textbf{in let} & \theta = \boxed{\mathsf{unify}\; (t_1, t_2 \to \alpha)\; \theta} \\
 & & & \textbf{in} & (\alpha,\theta) \\
 & & \ldots & &
\end{array}$$

As an example for operators on values of basic types we only treat the binary operator $+$. As one case of a comparison operator we treat the equality operator. Note that the calls of unification directly realize the equations that belong to the associated expressions. The cases of constants, of the empty list, of applications of the tuple constructors, and the case of single variables require no unification to determine the type of the expression. The auxiliary function new() returns a *new* type variable for each call. The algorithm, however, avoids requesting new type variables whenever this is possible. The result type for the expression *e* is composed of the types of its components. This is not possible for the type of the empty list since the type of potential elements of the list is not known yet. This is also impossible for function applications where the result type is a *component* of the type of the function.

$$\begin{array}{llrl}
\mid & \multicolumn{3}{l}{(\textsc{if}\; e_0\; \textsc{then}\; e_1\; \textsc{else}\; e_2)} \\
 & \to & \textbf{let} & (t_0,\theta) = \mathcal{W}\; e_0\; (\Gamma,\theta) \\
 & & \textbf{in let} & \theta = \boxed{\mathsf{unify}\; (\textbf{bool}, t_0)\; \theta} \\
 & & \textbf{in let} & (t_1,\theta) = \mathcal{W}\; e_1\; (\Gamma,\theta) \\
 & & \textbf{in let} & (t_2,\theta) = \mathcal{W}\; e_2\; (\Gamma,\theta) \\
 & & \textbf{in let} & \theta = \boxed{\mathsf{unify}\; (t_1, t_2)\; \theta} \\
 & & \textbf{in} & (t_1,\theta)
\end{array}$$

```
| $(\textsc{match}\ e_0\ \textsc{with}\ (x_1, \ldots, x_m) \rightarrow e_1)$
        $\rightarrow$     let $\alpha_1 = \mathsf{new}()$
                      $\ldots$
                  in let $\alpha_m = \mathsf{new}()$
                  in let $(t_0, \theta) = \mathcal{W}\ e_0\ (\Gamma, \theta)$
                  in let $\theta = \boxed{\mathsf{unify}\ (\alpha_1 * \ldots * \alpha_m, t_0)\ \theta}$
                  in let $(t_1, \theta) = \mathcal{W}\ e_1\ (\Gamma \oplus \{x_1 \mapsto \alpha_1, \ldots, x_m \mapsto \alpha_m\}, \theta)$
                  in $(t_1, \theta)$
| $(\textsc{match}\ e_0\ \textsc{with}\ [\,] \rightarrow e_1 \mid (x :: y) \rightarrow e_2)$
        $\rightarrow$     let $(t_0, \theta) = \mathcal{W}\ e_0\ (\Gamma, \theta)$
                  in let $\alpha = \mathsf{new}()$
                  in let $\theta = \boxed{\mathsf{unify}\ (\alpha\ \mathsf{list}, t_0)\ \theta}$
                  in let $(t_1, \theta) = \mathcal{W}\ e_1\ (\Gamma, \theta)$
                  in let $(t_2, \theta) = \mathcal{W}\ e_2\ (\Gamma \oplus \{x \mapsto \alpha, y \mapsto \alpha\ \mathsf{list}\}, \theta)$
                  in let $\theta = \boxed{\mathsf{unify}\ (t_1, t_2)\ \theta}$
                in $(t_1, \theta)$
  ...
```

The second group of cases realizes the rules for case distinctions and for pattern matching. If several alternative expressions may deliver the result, their types must agree. The types of the components into which a value is decomposed in the *match*-cases are determined by unification.

```
| $(\textsc{fun}\ x \rightarrow e)$
        $\rightarrow$     let $\alpha = \mathsf{new}()$
                  in let $(t, \theta) = \mathcal{W}\ e\ (\Gamma \oplus \{x \mapsto \alpha\}, \theta)$
                  in $(\alpha \rightarrow t, \theta)$
| $(\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_0)$
        $\rightarrow$     let $(t_1, \theta) = \mathcal{W}\ e_1\ (\Gamma, \theta)$
                  in let $\Gamma = \Gamma \oplus \{x_1 \mapsto t_1\}$
                  in let $(t_0, \theta) = \mathcal{W}\ e_0\ (\Gamma, \theta)$
                  in $(t_0, \theta)$
| $(\textsc{let rec}\ x_1 = e_1\ \textsc{and} \ldots \textsc{and}\ x_m = e_m\ \textsc{in}\ e_0)$
        $\rightarrow$     let $\alpha_1 = \mathsf{new}()$
                      $\ldots$
                  in let $\alpha_m = \mathsf{new}()$
                  in let $\Gamma = \Gamma \oplus \{x_1 \mapsto \alpha_1, \ldots, x_m \mapsto \alpha_m\}$
                  in let $(t_1, \theta) = \mathcal{W}\ e_1\ (\Gamma, \theta)$
                  in let $\theta = \boxed{\mathsf{unify}\ (\alpha_1, t_1)\ \theta}$
                      $\ldots$
                  in let $(t_m, \theta) = \mathcal{W}\ e_m\ (\Gamma, \theta)$
                  in let $\theta = \boxed{\mathsf{unify}\ (\alpha_m, t_m)\ \theta}$
                  in let $(t_0, \theta) = \mathcal{W}\ e_0\ (\Gamma, \theta)$
                in $(t_0, \theta)$
```

The last three cases treat functions and definitions of new variables. New type variables are created for the unknown type of the formal parameter as well as for the unknown types of the simultaneously recursively defined variables. Their bindings are determined during the processing of the expression e. No new type variable needs to be created for a variable x introduced by a *let*-expression; the type of variable x derives directly from the type of the defining expression for x.

The function $\mathcal{W}$ is called for an expression e with a type environment Γ_0 that associates a new type variable α_x with each free variable x occurring in e and the empty substition $\emptyset$. The call fails if and only if there is no type environment for which the expression e has a type. If on the other hand, the call delivers as return value a pair (t, θ), then for each derivable type judgment $\Gamma' \vdash e : t'$, there is a substitution σ such that:

$$t' = \sigma(\theta(t)) \qquad \text{and} \qquad \Gamma'(x) = \sigma(\theta(\Gamma(x))) \qquad \text{for all variables } x$$

The most general type for the expression e therefore is $\theta(t)$.

4.2.2 Polymorphism

A function with type $\alpha \to \alpha\,\mathbf{list}$ is expected to be *polymorphic*, that is, to be applicable to values of arbitrary types. The type system, as described so far, does not admit this.

Example 4.2.8 Consider the following program expression:

$$\begin{array}{l} \mathbf{let}\ \ \mathsf{single}\ \ =\ \ \mathbf{fun}\, y\ \to\ [y] \\ \mathbf{in}\ \ \mathsf{single}\ (\mathsf{single}\ 1) \end{array}$$

For the function single the type

$$\alpha[\mathsf{single}] \quad = \quad (\gamma \to \gamma\ \mathsf{list})$$

is derived. Because of the function application (single 1) the type variable γ is instantiated with the basic type **int**. For the function application (single 1) the following type is obtained:

$$\alpha[\mathsf{single}\ 1] \quad = \quad \mathbf{int}\,\mathsf{list}$$

The type equation for the outermost function application therefore requires the instantiation of γ with **int** list. Since unification of **int** with **int** list fails, a type error is reported. □

A possible solution to this problem consists in *copying* each *let*-definition for each use of the defined variable. In the example we obtain:

$$\begin{array}{l} (\mathbf{let}\ \mathsf{single} = \mathbf{fun}\ y\ \to\ [y]\ \mathbf{in}\ \mathsf{single})\ (\\ \qquad (\mathbf{let}\ \mathsf{single} = \mathbf{fun}\ y\ \to\ [y]\ \mathbf{in}\ \mathsf{single})\ 1) \end{array}$$

The two occurrences of the subexpression (**fun** $y \to [y]$) are now treated independently of each other and receive the types $\gamma \to \gamma$ **list** and $\gamma' \to \gamma'$ **list** for distinct type variables γ, γ'. The expanded program now has a type. In the example one type variable can be instantiated with **int** and the other with **int list**.

A solution by copying as just sketched, is not recommended because it imposes extra restrictions for the resulting program to have the same semantics as the original program. In addition, the program expanded in this way, may become *very* large. Also, type inference is no longer *modular*: for a function of another compilation unit that is used several times, the implementation must be known in order to copy it. A better idea therefore consists in copying not code but types. For this purpose we extend types to *type schemes*. A type scheme is obtained from a type t by *generalizing* some of the type variables that occur in t. Generalized type variables may be instantiated differently at different uses of the type. In the type scheme:

$$\forall \alpha_1, \ldots, \alpha_m.t$$

the variables $\alpha_1, \ldots, \alpha_m$ are generalized in t. All other type variables that occur in t must be identically instantiated at all uses of the type scheme. The quantifier $\forall$ only occurs at the outermost level: The expression t may not contain any other $\forall$. Type schemes are introduced for **let**-defined variables. At their occurrences type variables in the scheme can be independently instantiated wtih different types. For simplicity we regard normal type expressions as type schemes in which an *empty* list of variables has been generalized. Aa new rules we obtain:

$$\text{I}\textsc{nst}: \quad \frac{\Gamma(x) = \forall\, \alpha_1, \ldots, \alpha_k.t}{\Gamma \ \vdash \ x \ : \ t[t_1/\alpha_1, \ldots, t_k/\alpha_k]} \qquad (t_1, \ldots, t_k \quad \text{arbitrary})$$

$$\text{L}\textsc{et}: \quad \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \oplus \{x \mapsto \mathsf{close}\, t_1\, \Gamma\} \vdash e_0 : t_0}{\Gamma \ \vdash \ (\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_0) \ : \ t_0}$$

The operation close takes a type term t and a type environment Γ and generalizes in t all type variables that do not occur in Γ. The types of variables that are introduced in a recursive definition can also be generalized, but only for occurrences of the variables in the main expression:

$$\text{L}\textsc{etrec}: \quad \frac{\Gamma' \vdash e_1 : t_1 \quad \ldots \quad \Gamma' \vdash e_m : t_m \quad \Gamma'' \ \vdash e_0 : \ t}{\Gamma \ \vdash \ (\mathbf{let\ rec}\ x_1 = e_1\ \mathbf{and} \ldots \mathbf{and}\ x_m = e_m\ \mathbf{in}\ e_0) \ : \ t}$$

where

$$\begin{array}{lcl} \Gamma' & = & \Gamma \oplus \{x_1 \mapsto t_1, \ldots, x_m \mapsto t_m\} \\ \Gamma'' & = & \Gamma \oplus \{x_1 \mapsto \mathsf{close}\, t_1\, \Gamma, \ldots, x_m, \mapsto \mathsf{close}\, t_m\, \Gamma\} \end{array}$$

Thus, all variables in the type terms $t_1, \ldots, t_m$ are generalized that do not occur in the types of other variables of the type environment Γ that are visible in the main expression. It is mandatory that the types of *recursive* occurrences of the variables x_i in the right sides $e_1, \ldots, e_m$ may not be instantiated. Type systems permitting such *polymorphic recursion* are in general undecidable.

We now modify algorithm $\mathcal{W}$ such that it maintains type schemes in its type environments. For the case of variables we need the following auxiliary function that instantiates the type term in a type scheme with fresh type variables:

```
fun inst $(\forall\, \alpha_1, \ldots, \alpha_k . t)$ =
            let $\beta_1$ = new()
            ...
         in let $\beta_k$ = new()
         in $t[\beta_1/\alpha_1, \ldots, \beta_k/\alpha_k]$
```

Then we modify algorithm $\mathcal{W}$ for variables and *let*-expressions as follows:

```
  ...
| $x$            →  (inst $(\theta(\Gamma(x))), \theta)$
| (LET $x_1 = e_1$ IN $e_0$)
                 →      let $(t_1, \theta) = \mathcal{W}\ e_1\ (\Gamma, \theta)$
                     in let $s_1$ = close $(\theta\ t_1)\ (\theta \circ \Gamma)$
                     in let $\Gamma = \Gamma \oplus \{x_1 \mapsto s_1\}$
                     in let $(t_0, \theta) = \mathcal{W}\ e_0\ (\Gamma, \theta)$
                     in $(t_0, \theta)$
```

Correspondingly, we modify algorithm $\mathcal{W}$ for *letrec*-expressions. We must take care that the inferred types for newly introduced variables are only generalized for their occurrences in the main expression:

```
| (LET REC $x_1 = e_1$ AND ... AND $x_m = e_m$ IN $e_0$)
      →       let $\alpha_1$ = new()
              ...
           in let $\alpha_m$ = new()
           in let $\Gamma' = \Gamma \oplus \{x_1 \mapsto \alpha_1, \ldots, x_m \mapsto \alpha_m\}$
           in let $(t_1, \theta) = \mathcal{W}\ e_1\ (\Gamma', \theta)$
           in let $\theta = \boxed{\text{unify}\ (\alpha_1, t_1)\ \theta}$
              ...
           in let $(t_m, \theta) = \mathcal{W}\ e_m\ (\Gamma', \theta)$
           in let $\theta = \boxed{\text{unify}\ (\alpha_m, t_m)\ \theta}$
           in let $s_1$ = close $(\theta\, t_1)\ (\theta \circ \Gamma\})$
              ...
           in let $s_m$ = close $(\theta\, t_m)\ (\theta \circ \Gamma)$
           in let $\Gamma' = \Gamma \oplus \{x_1 \mapsto s_1, \ldots, x_m \mapsto s_m\}$
           in let $(t_0, \theta) = \mathcal{W}\ e_0\ (\Gamma', \theta)$
           in $(t_0, \theta)$
```

Example 4.2.9 Consider again the *let*-expression

```
let single = fun $y$ → $[y]$
in  single (single 1)
```

of Example 4.2.8. Algorithm $\mathcal{W}$ derives the type scheme $\forall\,\gamma.\gamma \to \gamma\,\textbf{list}$ for the function single. This type scheme is instantiated for both occurrences of single in the main expression with distinct type variables γ_1, γ_2, which are then instantiated to the types **int list** and **int**. Altogether algorithm $\mathcal{W}$ derives the type **int list list** for the *let*-expression. □

The extended algorithm $\mathcal{W}$ computes the *most general* type of an expression relative to a type environment with type schemes for the global variables of the expression. The instantiation of type schemes at all occurrences of variables allows us to define *polymorphic* functions that can be applied to values of different types. Type schemes admit *modular* type inference, since for functions of other program parts only their type (scheme) must be known to derive the types of expressions, in which they are used.

The possibility to instantiate variables in type schemes differently at different occurrences allows to construct program expressions whose types not only have exponential but even doubly exponential size! Such examples, however, are artificial and play no particular role in practical programming.

4.2.3 Side Effects

Variables whose values can be changed are sometimes useful even for essentially functional programming. To study the problems for type inference resulting from such *modifiable* variables, we extend our small programming language by *references*:

$$e \;::=\; \ldots \;\mid\; \textbf{ref}\; e \;\mid\; !e \;\mid\; e_1 := e_2$$

Example 4.2.10 A function new that returns a new value each time it is called, can elegantly be defined using references. Such a function is needed to implement algorithm $\mathcal{W}$. Consider the following program:

```
    let    count  =  ref 0
 in let    new    =  fun () →
                                 let   ret   =   !count
                              in let   _     =   count := ret + 1
                              in  ret
in  new() + new()
```

The empty tuple () is the only element of type **unit**. Assigning a value to a reference changes the contents of the reference as a *side effect*. The assignment itself is an expression whose value is (). Since this value is irrelevant, no dedicated variable is provided for it in our program but an *anonymous* variable _. □

Type expressions are extended by the special type **unit** and by introducing **ref** as new unary type constructor:

$$t \ ::= \ \ldots \mid \mathbf{unit} \mid t\, \mathbf{ref} \mid \ldots$$

Correspondingly we extend the rules of our type systems by

$$\textsc{Ref}: \quad \frac{\Gamma \ \vdash \ e \ : \ t}{\Gamma \ \vdash \ (\mathbf{ref}\, e) \ : \ t\, \mathbf{ref}}$$

$$\textsc{Deref}: \quad \frac{\Gamma \ \vdash \ e \ : \ t\, \mathbf{ref}}{\Gamma \ \vdash \ (!\, e) \ : \ t}$$

$$\textsc{Assign}: \quad \frac{\Gamma \ \vdash \ e_1 \ : \ t\, \mathbf{ref} \qquad \Gamma \ \vdash \ e_2 \ : \ t}{\Gamma \ \vdash \ (e_1 := e_2) \ : \ \mathbf{unit}}$$

These rules seem plausible. Interestingly, they are nonetheless incompatible with polymorphism.

Example 4.2.11 Consider the program expression

$$\begin{array}{rcccl} \mathbf{let} & y & = & \mathbf{ref}\, [\,] & \\ \mathbf{in\ let} & \_ & = & y \ := & 1 :: (!\, y) \\ \mathbf{in\ let} & \_ & = & y \ := & \mathbf{true} :: (!\, y) \\ \mathbf{in} \ \ 1 & & & & \end{array}$$

Type inference leads to no contradictions. For variable y, it returns the type scheme $\forall\, \alpha.\, \alpha\, \mathsf{list}\, \mathbf{ref}$. At run-time, though, a list is constructed that contains the *int*-value 1 together with the boolean value **true**. The construction of lists with elements of different types, though, should be prevented by the type system. □

The problem in Example 4.2.11 can be avoided if the types of modifiable values are never generalized. This is assured by the *value restriction*.

The set of *value expressions* contains all expressions without occurrences of references and function applications outside a functional abstraction. In particular *every* function **fun** $x \to e$ is a value expression. The *value restriction* expresses that in an expression

$$\mathbf{let}\ x = e_1\ \mathbf{in}\ e_0$$

type variables of the type of e_1 may only be generalized if e_1 is a value expression. A corresponding restriction applies to *letrec*-expressions.

Since the defining expression for variable y in Example 4.2.11 contains a reference, its type may not be generalized. The first application of y therefore fixes the type of y to **int ref**. Accordingly, the second application in the example therefore leads to a type error.

Polymorphism is a very useful mechanism in programming. In the form of *generics* it has been migrated into JAVA 1.5. Also, type inference is no longer restricted in functional languages; it has been incorporated into newer versions of C# and JAVA to avoid complicated and redundant type declarations.

The development of expressive type systems with powerful type inference has not stopped with the Hindley–Milner-type system. The programming language HASKELL has been experimentally equipped with a number of different extensions. One quite established idea consists in admitting constraints on the types that may be inserted for a generic type variable.

Example 4.2.12 Consider the following recursive function

```
fun member = fun x → fun list → match list
             with  [ ]   →  false
               |   h::t  →  if x = h then true
                            else member x t
```

In OCAML, the function member has type $\forall\,\alpha.\,\alpha \to \alpha\ \mathsf{list} \to \mathbf{bool}$. This follows from OCAML's design choice to assume equality to be defined for values of *all* types: For some types it throws an *exception*, though. This is different in the functional language SML: SML makes a difference between *equality types* providing equality and arbitrary types. Function types for example do not provide equality. In SML the type variable α in the SML-type for member may only be instantiated by *equality types*. □

4.2.4 Type Classes

The problem of Example 4.2.12 can be generalized. Often a function or a data structure is not generally polymorphic, but require data for which a certain operation must be supported. The function sort, for example, is only applicable to lists whose elements admit an $\leq$ operation.

In the following we sketch how the Hindley–Milner type system can be extended such that type parameteris can be associated with such constraints. The extensions which we pressent essentially are the *type classes* of the programming language HASKELL. A constraint to a type variable α lists the operations which all types must provide by which α can be instantiated. A *type class* C collects all types that implement the operations associated to C. Type classes together with the associated operations must be explicitly declared.

Example 4.2.13 Examples of type classes are:

Name	Operation	
Equality types	$(=)$	$: \alpha \to \alpha \to \mathbf{bool}$
Comparison types	$(\leq)$	$: \alpha \to \alpha \to \mathbf{bool}$
Printable types	to_string	$: \alpha \to \mathbf{string}$
Hashable types	hash	$: \alpha \to \mathbf{int}$

Here, $(\Box)$ denotes the binary function corresponding to the binary infix operator $\Box$. □

A constrained type scheme for an arbitrary expression has the form:

$$\forall \alpha_1 : S_1, \ldots, \alpha_m : S_m.\ s$$

where $S_1, \ldots, S_m$ are finite *sets* of type classes, and s is a polymorphic type scheme and thus may contain also generalized but unconstrained variables. A set S of type classes, which occurs as a constraint, is also called *sort*. When a sort $S = \{C'\}$ consists of a single element only, we will also omit the set brackets. For simpliity, we assume that each type class is assiciated with a single operation only. In order to declare a new type class C, the associated operation op_C must be specified together with the type of the operation op_C:

$$\mathbf{class}\ C\ \ \mathbf{where}\ \ \mathsf{op}_C\ :\ \forall\,\alpha : C\,.\,t$$

for some type expression t. The type scheme for op_C may containt exactly one generic variable, which is qualified by C.

Declarations of classes are complemented by *instance declarations*. An instance declaration for the class C specifies assumptions for the argument types of the application of a type constructor b, such that the resulting type is a member of class C, and provides an implementation of the operator op_C:

$$\begin{array}{l} \mathbf{inst}\ \ b(S_1, \ldots, S_k)\ :\ \ C \\ \mathbf{where}\ \ \mathsf{op}_C = e \end{array}$$

An operator which has different implementations for different types has been called *overloaded*. The case where the type constructor b has no parameters corresponds to base types.

Example 4.2.14 The class Eq, which collects all equality types, together with two instance declarations for this class may look as follows:

```
class Eq where
    (=) :   $\forall\alpha : C.\, \alpha \to \alpha \to$ bool


inst (Eq * Eq) : Eq
where (=) = fun x → fun y →  match x with
                             $(x_1, x_2)$ →  match y with
                             $(y_1, y_2) \to (x_1 = y_1) \wedge (x_2 = y_2)$
inst Eq list : Eq
where (=) =  let rec eq_list = fun $l_1$ → fun $l_2$ → match $l_1$
                 with     []       →  (match $l_2$ with [] → true | _ → false)
                 |       x :: xs   →  (match $l_2$ with [] → false
                                          | y :: ys →  if (=) x y then eq_list xs ys
                                                       else false)
             in eq_list
```

The implementation of equaltiy for pairs is defined by means of the equalities for the component types. Accordingly, the equality for lists refers to the equality for the element type. The challenge for type inference is not only to check that the types of expressions are compatible, but also to identify for different occurrences of an operator the corresponding correct implementation. □

Often it is convenient to associate *several* operations with a single class. A class Number, for example, may consist of all types that support the standard four arithmetic operators. Together with the declaration of a class, also implementations of *derived* operations could be provided. In this way, an equality operation can be realized whenever a comparison operation $\leq$ is at hand. Such a generic implementation results in a generic subclass relationship. Additionally to usedful data types, the standard library should also provide a system of predefined classes, into which predefined types are organized. In our example the base types **int** and **bool** should also be instances of the class Eq.

The key question is: how to check that an overloaded operator op_C is only applied to values of types that provide implementations for op_C? First, we convince ourselves that it can be checked whether a given type expression t belongs to the class C or not. Let Σ denote an assignment of type variables to sorts. Type variables not mentioned by Σ are implicitly mapped to the *empty* set of classes: for these, Σ imposes no constraints what so ever. The mapping Σ is also called *sort environment*. For a sort environment Σ and a class C, the judgement

$$\Sigma \vdash t : C$$

expresses that the type t belongs to C whenever each type variable α occurring in t belongs to all classes from $\Sigma(\alpha)$.

For a given sort environment Σ, the set $S[t]\,\Sigma$ of all classes to which t belongs can be determined inductively over the structure of t. If t is a type variable α, then $S[t]\,\Sigma = \Sigma(\alpha)$. If t is of the form $b(t_1, \ldots, t_k)$ for some type constructor b of arity $k \geq 0$, then $S[t]\,\Sigma$ is the set of all classes for which an instance declaration **inst** $b(S_1, \ldots, S_k) : C\ \ldots$ has been provided with $S_i \subseteq S[t_i]\,\Sigma$ for all i.

If, on the other hand, for every class C and every type constructor there is at most one instance declaration, then a required sort constraint S for the root of t can be translated into required sort constraints for the subterms of t. This allows us to determine *minimal* sort constraints for the variables occurring in t, which must be satisfied in order to make t a member of each class in S.

Example 4.2.15 Assume that the base types **int** and **bool** belong to the class Eq.

- Then the types (**bool**, **int list**) and (**bool**, **int list**) **list** also belong to Eq.
- The type **bool** → **int** does not belong to the class Eq, as long as no instance declaration for Eq and and the type constructor → has been provided.
- The type expression (α, **int**) **list** denotes types of the class Eq, whenever α belongs to the class Eq. □

In order to infer types for functional programs with class and instance declarations, we first may ignore the constraints in typ schemes and just derive Hindley–Milner polymoriphic types. In a second phase, we then may determine the sorts of each type variable. The disadvantage of the procedure is that then type correctness of a program is verified, while it still remains unclear how the program is translated.

A better idea, therefore, consists in modifying polymorphic type inference by means of algorithm $\mathcal{W}$ in such a way that besides typing and sort information, it also provides a *translation* of e into an expression e' which makes the selection of the right operator implementation explicit.

The translation provides for every sort S a table $\alpha\,\mathsf{dict}_S$ that for every operator op of a class in S provides an *implementation* of op. The overloaded operator op_C of class C with type scheme $\forall\,\alpha : C.\ t$ then is translated into a look-up $\alpha.\mathsf{op}_C$ in a table α that contains a corresponding component op_C. The goal of the translation is to provide tables such that the right implementations of a given operator can be looked up at every use of this operator. A variable f for which the algorithm $\mathcal{W}$ provides a type scheme $\forall\,\alpha_1 : S_1, \ldots, \alpha_m : S_m.\ s$ therefore is translated into a *function* that receives m tables as extra actual parameters:

$$f\ :\ \forall \alpha_1. \ldots \alpha_m.\ \alpha_1\,\mathsf{dict}_{S_1} \to \ldots \alpha_m\,\mathsf{dict}_{S_m} \to s$$

In order to extend the algorithm $\mathcal{W}$, we require a unification procedure that additionally maintains the sort information:

```
sort_unify $(\tau_1, \tau_2)\ \Sigma$ = match unify $(\tau_1, \tau_2)\ \emptyset$
             with Fail → Fail
               | $\theta$  → (match $\theta^{-1}\ \Sigma$ with Fail → Fail
                                                  | $\Sigma'$   → $(\theta, \Sigma')$)
```

Thereby, $\theta^{-1}\ \Sigma$ returns the minimal sort requirement Σ' for the type variables occurring the image of θ that must be provided in order to satisfy the constraints given by Σ, i.e., such that $\Sigma' \vdash (\theta\,\alpha) : (\Sigma\,\alpha)$ holds for all type variables α.

Example 4.2.16 Consider the instance declarations that result in the following rules:

$$\begin{array}{lcl} \mathsf{Eq}\ \textbf{list} & : & \mathsf{Eq} \\ \mathsf{Comp}\ \textbf{set} & : & \mathsf{Eq} \end{array}$$

and assume that

$$\Sigma = \{\alpha \mapsto \mathsf{Eq}\} \qquad \theta = \{\alpha \mapsto\ \beta\,\mathsf{set\,list}\}$$

Propagating the sort constraint $\Sigma(\alpha) = \mathsf{Eq}$ for the type variable α w.r.t. the type substitution θ to sort constraints for the type variables occurring the type expression $\theta\,\alpha$ (here: just β), results in the sort constraint:

$$\theta^{-1}\ \Sigma = \{\beta \mapsto \mathsf{Comp}\}$$

Note that the substituted variable α no longer occurs in $\theta^{-1}\ \Sigma$. □

For the implementation of the extended algorithm $\mathcal{W}$, we also modify the auxiliary functions close and inst.

A call sort_close (t, e) (Γ, Σ) for a type t and and expression e w.r.t. a type environment Γ and a sort environment Σ makes all type variables of t generic that neither occur in Γ nor in Σ and makes all type variables of t constrained generic that do not occur in Γ, but in Σ. Besides the type scheme, the call additionally returns in a second component the sort environment Σ, where all variables that have been generalized in the type scheme are removed. As a third component, the functional expression is returned that is obtained from e by abstracting the constrained generic type variables of the type scheme as formal parameters:

```
sort_close $(t, e)$ $(\Gamma, \Sigma)$
        $=$      let    $\alpha'_1, \ldots, \alpha'_n = \mathsf{free}\,(t) \setminus (\mathsf{free}\,(\Gamma) \cup \mathsf{dom}\,(\Sigma))$
             in let  $s = \forall\,\alpha'_1, \ldots, \alpha'_n . t$
             in let  $\alpha_1, \ldots, \alpha_m = (\mathsf{free}\,(t) \setminus \mathsf{free}\,(\Gamma)) \cap \mathsf{dom}\,(\Sigma)$
             in let  $s = \forall\,\alpha_1 : \Sigma(\alpha_1), \ldots, \alpha_m : \Sigma(\alpha_m) . s$
             in let  $\Sigma = \Sigma \backslash \{\alpha_1, \ldots, \alpha_m\}$
             in $(s, \Sigma,$ fun $\alpha_1 \to \ldots$ fun $\alpha_m \to e)$
```

An instantiation by means of *fresh* type variables is realized by means of the following function:

```
fun sort_inst $(\forall\,\alpha_1 : S_1, \ldots, \alpha_m : S_m . s, x)$
                                    $=$      let  $t$ = inst $s$
                                          in let  $\beta_1$ = new()
                                                  ...
                                          in let  $\beta_m$ = new()
                                          in let  $t = t[\beta_1/\alpha_1, \ldots, \beta_m/\alpha_m]$
                                          in  $(t, \{\beta_1 \mapsto S_1, \ldots, \beta_m \mapsto S_m\},$
                                                 $x\ \beta_1 \ldots \beta_m)$
```

Note that the transformation insists to create functional parameters only for type variables that are constrained by sorts. The type variables that occur in output expressions may later be further instantiated by unification of type expressions. If a variable $\alpha : S$ is substituted by a type expression t, then an S-table corresponding to type t is generated and substituted for the program variable α. The table is generated by means of the transformation $\mathcal{T}$:

```
$\mathcal{T}[\beta]\ S$                $=$  $\beta$
$\mathcal{T}[b(t_1, \ldots, t_m)]\ S$  $=$  forall $C \in S$
                                   let op$'_C$ =  let $d_1 = \mathcal{T}[t_{i_1}]\ S_{C,i_1}$  in
                                                  ...
                                                  let $d_k = \mathcal{T}[t_{i_k}]\ S_{C,i_k}$  in
                                                  op$_{C,b}\ d_1 \ldots d_k$
                                   in {op$_C$ = op$'_C$ | $C \in S$}
```

whenever $(S_{C,1}, \ldots, S_{C,m})\, b\ :\ C$ holds and $S_{C,i_1}, \ldots, S_{C,i_k}$ is the subsequence of nonempty sorts among the $S_{C,i}$. The implementation of the operator op$_C$ mit $C \in S$ for type t then can be looked up in the table $\mathcal{T}[t]\ S$.

Now we are ready to extend the algorithm $\mathcal{W}$. The extended algorithm $\mathcal{W}$ receives as actual parameters the expression e, a type environment Γ, an idempotent type substitution θ together with a sort environment Σ. As result it returns the type of e, together with a type substitution θ', a sort environment Σ' as well as the translation of e. Here, we only spell out the most important cases. The complete algorithm should be developed in Exercise 4.

```
   ...
 | op$_C$     →     let  $\beta$ = new()
                 in    $(t_C[\beta/\alpha], \Sigma \oplus \{\beta \mapsto C\}, \theta, \beta$.op$_C)$
 | $x$        →     let  $(t, \Sigma', e')$ = sort_inst $(\Gamma(x), x)$
                 in    $(t, \Sigma \cup \Sigma', \theta, e')$
```

$$\begin{array}{lll}
\mid & (\text{LET } x = e_1 \text{ IN } e_0) & \\
\rightarrow & \textbf{let} & (t_1, \Sigma, \theta, e_1') = \mathcal{W}\ e_1\ (\Gamma, \Sigma, \theta) \\
& \textbf{in let} & e_1' = \mathcal{T}[e_1']\,(\theta, \Sigma) \\
& \textbf{in let} & (s_1, \Sigma, e_1') = \mathsf{sort\_close}\ (\theta\ t_1, e_1')\ (\theta\ \Gamma, \Sigma) \\
& \textbf{in let} & \Gamma = \Gamma \oplus \{x_1 \mapsto s_1\} \\
& \textbf{in let} & (t_0, \Sigma, \theta, e_0') = \mathcal{W}\ e_0\ (\Gamma, \Sigma, \theta) \\
& \textbf{in let} & e' = (\text{LET } x_1 = e_1' \text{ IN } e_0') \\
& \textbf{in} & (t_0, \Sigma, \theta, e')
\end{array}$$

where $\mathcal{T}[e]\,(\theta, \Sigma)$ replaces each occurrence of a variable β in e with $\Sigma(\beta) = S$ by the table $\mathcal{T}[\theta\,\beta]\ S$. Type inference/transformation starts with an empty sort environment $\Sigma_0 = \emptyset$ and an empty type environment $\Gamma_0 = \emptyset$.

Consider an instance declaration

$$\textbf{inst}\ \ b(S_1, \ldots, S_m) : C\ \ \textbf{where}\ \ \mathsf{op}_C = e$$

of a class C whose operator op_C satisfies the type scheme $\forall\,\alpha : C.t$ where $S_{i_1}, \ldots, S_{i_k}$ is the subsequence of nonempty sorts among the S_i. Then it must be checked whether the implementation has an appropriate type, i.e.,

$$\mathcal{W}\ e\ (\Gamma_0, \emptyset, \emptyset) = (t', \Sigma, \theta, e') \qquad \text{with} \qquad \theta\,t' = t[b(\beta_1, \ldots, \beta_m)/\alpha]$$

holds for suitable type variables β_i where:

$$\Sigma(\beta_i) \subseteq S_i$$

holds. The implementation of the operator op_C for the type constructor b then is given by:

$$\mathsf{op}_{C,b} = \textbf{fun}\ \beta_{i_1} \rightarrow \ldots \rightarrow \textbf{fun}\ \beta_{i_k} \rightarrow \mathcal{T}[e']\,(\theta, \Sigma)$$

Example 4.2.17 Consider the implementations of equality for pairs and lists. According to their declaration, they have the following types:

$$\begin{array}{lll}
(=)_{\mathsf{pair}} & : & \forall\,\alpha_1 : \mathsf{Eq}, \alpha_2 : \mathsf{Eq}.\ (\alpha_1 * \alpha_2) \rightarrow (\alpha_1 * \alpha_2) \rightarrow \textbf{bool} \\
(=)_{\mathsf{list}} & : & \forall\,\alpha : \mathsf{Eq}.\ \alpha\ \textbf{list} \rightarrow \alpha\ \textbf{list} \rightarrow \textbf{bool}
\end{array}$$

After transformation, we obtain the following types:

$$\begin{array}{lll}
(=)_{\mathsf{pair}} & : & \forall\,\alpha_1, \alpha_2.\ \alpha_1\ \mathsf{dict}_{\mathsf{Eq}} \rightarrow \alpha_2\ \mathsf{dict}_{\mathsf{Eq}} \rightarrow (\alpha_1 * \alpha_2) \rightarrow (\alpha_1 * \alpha_2) \rightarrow \textbf{bool} \\
(=)_{\mathsf{list}} & : & \forall\,\alpha.\ \alpha\ \mathsf{dict}_{\mathsf{Eq}} \rightarrow \alpha\ \textbf{list} \rightarrow \alpha\ \textbf{list} \rightarrow \textbf{bool}
\end{array}$$

For every constrained type parameter, an extra argument is provided. After the transformation, we obtain the following implementation:

$$
\begin{array}{ll}
(=)_{\mathsf{pair}} = & \textbf{fun}\ \beta_1 \to \textbf{fun}\ \beta_2 \to \textbf{fun}\ x \to \textbf{fun}\ y \to \\
& \qquad\qquad \textbf{match}\ x\ \textbf{with}\ (x_1, x_2)\ \to \\
& \qquad\qquad \textbf{match}\ y\ \textbf{with}\ (y_1, y_2)\ \to \\
& \qquad\qquad\qquad \beta_1.(=)\ x_1\ y_1\ \wedge\ \beta_2.(=)\ x_2\ y_2 \\
(=)_{\mathsf{list}}\ = & \textbf{fun}\ \beta \to \textbf{let rec}\ \mathsf{eq\_list} = \textbf{fun}\ l_1\ \to \textbf{fun}\ l_2 \to \textbf{match}\ l_1 \\
& \quad \textbf{with}\quad []\qquad \to \quad (\textbf{match}\ l_2\ \textbf{with}\ [] \to \textbf{true} \mid \_ \to \textbf{false}) \\
& \quad \mid\quad x :: xs\quad \to \quad (\textbf{match}\ l_2\ \textbf{with}\ [] \to \textbf{false} \\
& \qquad\qquad\qquad\qquad \mid\ y :: ys \to \\
& \qquad\qquad\qquad\qquad\quad \textbf{if}\ \beta.(=)\ x\ y\ \textbf{then}\ \mathsf{eq\_list}\ xs\ ys \\
& \qquad\qquad\qquad\qquad\quad \textbf{else false}) \\
& \textbf{in}\ \ \mathsf{eq\_list}
\end{array}
$$

The program variables β_1, β_2, and β are fresh program variables that have been generated from type variables. Their run-time values are tables that provide the actual implementations of the overloaded operator $(=)$. □

In our implementation, the component name op occurs in all tables of sorts that require an implementation of the operator op. In programming languages such as OCAML, components of different record types may not be equal. A practical solution therefore is to *rename* components for op in different records, once type inference and transformation has been finished and adapt the accesses to components accordingly.

The invention of type classes is by no means the end of the story. The programming language HASKELL has proven to be an ingenious test bed for various extensions of the Hindley–Milner type system. HASKELL thus not only provides type classes but also *type constructor* classes. These conveniently allow us to deal with *monads*. Monads have evolved into a central part of HASKELL since they allow us to realize input/output as well as various kinds of side effects in a purely functional way.

4.3 Attribute Grammars

We have described several tasks to be performed by semantic analysis in Sect. 4.1. Each of these essentially turned out to be computations on syntax trees. It would be nice to have a description mechanism also for these tasks, from which implementations could be generated automatically.

An elegant and powerful description mechanism which may serve this purpose is offered by *attribute grammars*. Attribute grammars extend context-free grammars by associating *attributes* with the symbols of the underlying context-free grammar. These attributes are containers for static semantic information. Their values are computed by computations performed on trees, with computations traversing the trees as needed. The set of attributes of a symbol X is denoted by $\mathcal{A}(X)$. With each attribute a is associated a type τ_a, which determines the set of possible values for the instances of the attribute. Consider a production $p : X_0 \longrightarrow X_1 \ldots X_k$ with $k \geq 0$ symbols occurring on the right side. To tell the different occurrences of symbols in production p apart, we number these from left to right. The left side nonterminal X_0 will be denoted by $p[0]$, the ith symbol X_i on the right side of p by $p[i]$ for $i = 1, \ldots, k$. The attribute a of a symbol X has an *attribute occurrence* at each occurrence of X in a production. The occurrence of an attribute a at a symbol X_i is denoted by $p[i].a$.

For every production, functional specifications are provided how attributes of occurring symbols may be determined from the values of further attributes of symbol occurrences of the same production. These specifications are called *semantic rules*. In our examples, semantic rules are realized by means of an OCAML-like programming language. This has the extra advantage that the explicit specification of types can be omitted.

A restricted instance of such a mechanism is already provided by standard LR-parsers such as YACC or BISON: Here, each symbol of the grammar is equipped with a single attribute. For every production then there is one semantic rule that determines how the attribute of the nonterminal occurring on the left side is determined from the attributes of the symbol occurrences on the right side.

Example 4.3.1 Consider a CFG with the nonterminals E, T, F for arithmetic expressions. The set of terminals consists of symbols for brackets, operators, and the symbols var and const, that represent *int*-variables and constants, respectively. The nonterminals should be equipped with an attribute *tree* that receives the internal representation of the expression.

In order to determine the values for the attributes, we extend the productions of the grammar by semantic rules as follows:

$$\begin{array}{llll}
p_1: & E & \longrightarrow & E + T \\
 & & & p_1[0].tree = \mathsf{Plus}\,(p_1[1].tree, p_1[2].tree) \\
p_2: & E & \longrightarrow & T \\
 & & & p_2[0].tree = p_2[1].tree \\
p_3: & T & \longrightarrow & T * F \\
 & & & p_3[0].tree = \mathsf{Mult}\,(p_3[1].tree, p_3[2].tree) \\
p_4: & T & \longrightarrow & F \\
 & & & p_4[0].tree = p_4[1].tree \\
p_5: & F & \longrightarrow & \mathsf{const} \\
 & & & p_5[0].tree = \mathsf{Int}\,(p_5[1].val) \\
p_6: & F & \longrightarrow & \mathsf{var} \\
 & & & p_6[0].tree = \mathsf{Var}\,(p_6[1].id) \\
p_7: & F & \longrightarrow & (\,E\,) \\
 & & & p_6[0].tree = p_6[2].tree
\end{array}$$

For the construction of the internal representation, the constructors

$$\mathsf{Plus}, \mathsf{Mult}, \mathsf{Int}, \mathsf{Var}$$

have been applied. Furthermore, we assumed that the symbol const has an attribute *val* containing the value of the constant, and the symbol var has an attribute *id* containing a unique identifier for the variable. □

Some parser generators address the different occurrences of symbols in a production by indexing the occurrences according to each symbol separately.

Example 4.3.2 Consider again the grammar from Example 4.3.1. According to the convention of indexing different occurrences of the same symbol in the production, the semantic rules are denoted as follows:

$$\begin{array}{lcll}
p_1: & E & \longrightarrow & E + T \\
 & & & E[0].tree = \mathsf{Plus}\,(E[1].tree, T.tree) \\
p_2: & E & \longrightarrow & T \\
 & & & E.tree = T.tree \\
p_3: & T & \longrightarrow & T * F \\
 & & & T[0].tree = \mathsf{Mult}\,(T[1].tree, F.tree) \\
p_4: & T & \longrightarrow & F \\
 & & & T.tree = F.tree \\
p_5: & F & \longrightarrow & \mathsf{const} \\
 & & & F.tree = \mathsf{Int}\,(\mathsf{const}.val) \\
p_6: & F & \longrightarrow & \mathsf{var} \\
 & & & F.tree = \mathsf{Var}\,(\mathsf{var}.id) \\
p_7: & F & \longrightarrow & (\,E\,) \\
 & & & F.tree = E.tree
\end{array}$$

The index is omitted if a symbol occurs only once. If a symbol occurs several times, the index 0 identifies an occurrence of the left side, while all occurrences on the productions's right side are successively indexed starting from 1. □

In concrete examples of attribute grammars, we will consistently use the convention from Example 4.3.2, while the convention to address the occurrences of symbols in a production $p: \; X_0 \rightarrow X_1 \dots X_k$ through $p[0], \dots, p[k]$ as in Example 4.3.1 is more convenient for conceptual reasoning.

The semantic attribute for each symbol as provided by LR-generators can be used by the parser itself to build up a representation of the syntax tree. Attribute grammars generalize this idea in two directions. First, every symbol may possess several attributes. Second, attributes on the left side are not necessarily defined by means of the attributes of symbols on the production's right side. And the values of right side attributes now may be determined by means of values of left side attributes or attributes of other right side symbols.

We introduce the following terminology. The individual attribute $p[i].a$ of an occurrence $p[i]$ of a symbol in the production p is called *occurrence* of the attribute a in p. An attribute occurrence belongs to the attribute grammar and is used for the *specification* of the local behavior at a node. If the defining rule for an attribute occurrence o accesses another attribute occurrence o', then *o functionally depends* on o'.

The attributes of occurrences of symbols in a syntax tree, on the other hand, are called *instances* of attributes. Attribute instances exist at compile time after syntactic analysis has produced the parse tree.

The functional dependences between attribute occurrences determine in which order the attribute instances at nodes of the parse tree may be evaluated. Arguments of semantic rules need to be evaluated before the rules can be applied to compute the value of the associated attribute (instance). Suitable *constraints on the functional dependences* ensure that the *local* semantic rules of the attribute grammar for the attribute occurrences in the productions can be composed to a *global* computation of all attribute instances in a parse tree. The values of attribute instances at the individual nodes of the parse tree are computed by a global algorithm, which is generated from the attribute grammar, and which at each node n adheres to the local semantic rules of the production applied at n. This chapter is concerned with the questions how such an algorithm can be automatically generated from a given attribute grammar.

Attributes also have *directions*, which can be better understood if we think of production applications in parse trees. The attributes of a symbol X are either *inherited* or *synthesized*. The values of (instances of) synthesized attributes at a node are computed from (values of attribute instances in) the subtree at this node. The values of (instances of) inherited attributes at a node are computed from the context of the node, see Fig. 4.6. All the attribute instances with ingoing yellow edges are computed within the production. These are the occurrences of synthesized attributes of the left side of the production and the occurrences of inherited attributes of the right side of productions. Together we call them *defining occurrences* of attributes in this production. All other occurrence of attributes in the production are called *applied attribute occurrences*. Each production has *semantic rules*, which describe how the values of defining attribute occurrences in the production are computed from the values of other attribute occurrences of the same production. So, semantic rules need to be given for each inherited attribute occurrence on the right side of a production and each synthesized attribute occurrence on the left side. The set of inherited and synthesized attributes of an attribute grammar are denoted by $\mathcal{I}$ and $\mathcal{S}$, respectively, and the set of inherited and synthesized attributes of a symbols X correspondingly by $\mathcal{I}(X)$ and $\mathcal{S}(X)$.

An attribute grammar is in *normal form* if all defining attribute occurrences in productions only depend on applied occurrences in the same productions. If not explicitly stated otherwise, we assume that attribute grammars are in normal form.

Our definition also allows synthesized attributes for terminal symbols of the grammar. In compiler design, attribute grammars are used to specify semantic analysis. This phase follows lexical and syntactic analysis. In semantic analysis, the synthesized attributes of terminal symbols play an important role. Typical synthesized attributes of terminal symbols are values of constants, external representations or unique encodings of names, and the addresses of string constants. The values of these attributes are produced by the scanner, at least if it is extended by semantic

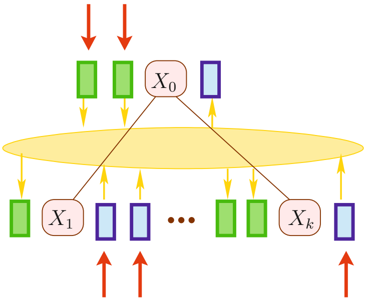


Fig. 4.6 An attributed node in the parse tree with its attributed successors. Instances of inherited attributes are drawn as *boxes* to the *left* of syntactic symbols, instances of synthesized attributes as *boxes* to the *right* of symbols. *Red (darker) arrows* show the information flow into the production instance from the outside, *yellow (lighter) arrows* symbolize functional dependences between attribute instances that are given through the semantic rules associated with the production

functionality. Also for inherited attribute instances at the root of the parse tree, no semantic rules are provided by the grammar to compute their values. Here, the application must provide meaningful values for their initialization.

4.3.1 The Semantics of an Attribute Grammar

The semantics of an attribute grammar determines for each parse tree t of the underlying CFG which values the attributes at each node in t should have.

For each node n in t, let $\mathsf{symb}(n)$ denote the symbol of the grammar labeling n. If $\mathsf{symb}(n) = X$ then n is associated with the attributes in $\mathcal{A}(X)$. The attribute $a \in \mathcal{A}(n)$ of the node n is addressed by $n.a$. Furthermore, we need an operator to navigate from a node to its successors. Let $n_1, \ldots, n_k$ be the sequence of successors of node n in parse tree t. Then $n[0]$ denotes the node n itself, and $n[i] = n_i$ for $i = 1, \ldots, k$ denotes the i-th successor of n in the parse tree t.

If $X_0 = \mathsf{symb}(n)$ and, if $X_i = \mathsf{symb}(n_i)$ for $i = 1, \ldots, k$ are the labels of the successors n_i of n then $X_0 \longrightarrow X_1 \ldots X_k$ is the production of the CFG that has been applied at node n. From the semantic rules of this production p, semantic definitions of attributes at the nodes $n, n_1, \ldots, n_k$ are generated by instantiating p with the node n. The semantic rule

$$p[i].a = f(p[i_1].a_1, \ldots, p[i_r].a_r)$$

for the production p gives rise to the semantic rule

$$n[i].a = f(n[i_1].a_1, \ldots, n[i_r].a_r)$$

for the node n in the parse tree. Hereby, we assume that the semantic rules specify *total* functions. For a parse tree t, let

$$V(t) = \{n.a \mid n \text{ node in } t, a \in \mathcal{A}(\mathsf{symb}(n))\}$$

denote the set of all attribute instances in t. The subset $V_{in}(t)$ of inherited attribute instances at the root together with the set of all synthesized attribute instances at the leaves are called the set of *input attribute instances* of t. Instantiating the semantic rules of the attribute grammar at all nodes in t produces a system of equations in the unknowns $n.a$ that has for all but the input attribute instances exactly one equation. Let $\mathsf{AES}(t)$ be this attribute equation system. Now consider any assignment σ to the input attribute instances. If $\mathsf{AES}(t)$ is recursive (cyclic), it can have several solutions or no solution (relative to σ). If $\mathsf{AES}(t)$ is not recursive then for every assignment σ of the input attribute instance, there is exactly one assignment to the noninput attribute instances of the parse trees t such that all equations are satisfied. Accordingly, the attribute grammar is called *well-formed*, if the system of equations $\mathsf{AES}(t)$ is not recursive for any parse tree t of the underlying CFG. In this case, we define the semantics of the attribute grammar as the function mapping each parse tree t and each assignment σ of the input attribute instances to an assignment of all attribute instances of t that agrees with σ on the input attribute instances and additionally satisfies all equations of the system $\mathsf{AES}(t)$.

4.3.2 Some Attribute Grammars

In the following we present some (fragments of) attribute grammars that solve essential subtasks of semantic analysis. The first attribute grammar shows how types of expressions can be computed using an attribute grammar.

Example 4.3.3 (Type checking) The attribute grammar AG_{types} realizes type inference for expressions containing assignments, nullary functions, operators $+, -, *, /$ as well as variable and constants of type **int** or **float** for a C-like programming language with explicit type declarations for variables. The attribute grammar has an attribute *type* for the nonterminal symbols E, T, and F, and for the terminal symbol const, which may take values Int and Float. This grammar can be easily extended to more general expressions with function application, component selection in composed values, and pointers.

$$\begin{array}{lcl}
E \longrightarrow \mathsf{var}\ '='\ E & & \\
\quad E[1].env & = & E[0].env \\
\quad E[0].typ & = & E[0].env\ \mathsf{var}.id \\
\quad E[0].ok & = & \quad \mathbf{let}\ x = \mathsf{var}.id \\
 & & \mathbf{in\ let}\ \tau = E[0].env\, x \\
 & & \mathbf{in}\quad (\tau \neq \mathsf{error}) \wedge (E[1].type \sqsubseteq \tau)
\end{array}$$

$$\begin{array}{l} E \longrightarrow E\ \mathsf{aop}\ T \\ \quad \begin{array}{lcl} E[1].\mathit{env} & = & E[0].\mathit{env} \\ T.\mathit{env} & = & E[0].\mathit{env} \\ E[0].\mathit{typ} & = & E[1].\mathit{typ} \sqcup T.\mathit{typ} \\ E[0].\mathit{ok} & = & (E[1].\mathit{typ} \sqsubseteq \mathsf{float}) \\ & & \wedge\ (T.\mathit{typ} \sqsubseteq \mathsf{float}) \end{array} \end{array}$$

$$\begin{array}{l} E \longrightarrow T \\ \quad \begin{array}{lcl} T.\mathit{env} & = & E.\mathit{env} \\ E.\mathit{typ} & = & T.\mathit{typ} \\ E.\mathit{ok} & = & T.\mathit{ok} \end{array} \end{array}$$

$$\begin{array}{l} T \longrightarrow T\ \mathsf{mop}\ F \\ \quad \begin{array}{lcl} T[1].\mathit{env} & = & T[0].\mathit{env} \\ F.\mathit{env} & = & T[0].\mathit{env} \\ T[0].\mathit{typ} & = & T[1].\mathit{typ} \sqcup F.\mathit{typ} \\ T[0].\mathit{ok} & = & (T[1].\mathit{typ} \sqsubseteq \mathsf{float}) \\ & & \wedge\ (F.\mathit{typ} \sqsubseteq \mathsf{float}) \end{array} \end{array}$$

$$\begin{array}{l} T \longrightarrow F \\ \quad \begin{array}{lcl} F.\mathit{env} & = & T.\mathit{env} \\ T.\mathit{typ} & = & F.\mathit{typ} \\ T.\mathit{ok} & = & F.\mathit{ok} \end{array} \end{array}$$

$$\begin{array}{l} F \longrightarrow (E) \\ \quad \begin{array}{lcl} E.\mathit{env} & = & F.\mathit{env} \\ F.\mathit{typ} & = & E.\mathit{typ} \\ F.\mathit{ok} & = & E.\mathit{ok} \end{array} \end{array}$$

$$\begin{array}{l} F \longrightarrow \mathsf{const} \\ \quad \begin{array}{lcl} F.\mathit{typ} & = & \mathsf{const}.\mathit{typ} \\ F.\mathit{ok} & = & \mathsf{true} \end{array} \end{array}$$

$$\begin{array}{l} F \longrightarrow \mathsf{var} \\ \quad \begin{array}{lcl} F.\mathit{typ} & = & F.\mathit{env}\ \mathsf{var}.\mathit{id} \\ F.\mathit{ok} & = & (F.\mathit{env}\ \mathsf{var}.\mathit{id} \neq \mathsf{error}) \end{array} \end{array}$$

$$\begin{array}{l} F \longrightarrow \mathsf{var}\,() \\ \quad \begin{array}{lcl} F.\mathit{typ} & = & (F.\mathit{env}\ \mathsf{var}.\mathit{id})\ () \\ F.\mathit{ok} & = & \begin{array}{l} \mathbf{match}\ F.\mathit{env}\ \mathsf{var}.\mathit{id} \\ \mathbf{with}\ \tau\,() \ \rightarrow\ \mathsf{true} \\ \quad\ \mid\ \ \_\ \ \ \rightarrow\ \mathsf{false} \end{array} \end{array} \end{array}$$

The attribute *env* of the nonterminals E, T, and F is inherited while all other attributes of grammar AG_{types} are synthesized. □

Attribute grammars refer to some underlying CFG. If, e.g., operator precedences have been coded into the grammar, a larger number of chain rules may occur forcing the values of attibutes to be copied from the upper node to the single child (in case of inherited attributes) or from the single child to the ancestor (in case of synthesized attributes). This phenomenon can be nicely observed already for the attribute grammar AG_{types}. Therefore, we introduce a convention for writing attribute grammars that reduces the overhead of specifying copying attribute values:

> If the semantic rule for an occurrence of an inherited attribute on the right side (synthesized attribute on the left side) is missing, an identical assignment from an occurrence of an equally named inherited attibute on the left side (synthesized attribute on the right side) is assumed.

In case of a missing semantic rule for an occurrence of a synthesized attribut on the left side, this rule can only be applied if there is *exactly one* occurrence of a synthesized attribute on the right with that particular name. The following examples use this convention, at least for chain productions of the form $A \longrightarrow B$.

Example 4.3.4 (Managing Symbol Tables) The attribute grammar AG_{scopes} manages symbol tables for a fragment of a C-like imperative language with parameterless procedures. Nonterminals for declarations, statements, blocks, and expressions are associated with an inherited attribute *env* that will contain the actual symbol table.

The redeclaration of an identifier within the same block is forbidden while it is allowed in a new block. To check this a further inherited attribute *same* is used to collect the set of identifiers that are encountered so far in the actual block. The synthesized attribute *ok* signals whether all used identifiers are declared and used in a type-correct way.

$$\langle decl\rangle \longrightarrow \langle type\rangle\ \ \mathsf{var};$$

$$\begin{aligned}
\langle decl\rangle\ .new &= (\mathsf{var}.id, \langle type\rangle\ .typ)\\
\langle decl\rangle\ .ok &= \mathsf{true}
\end{aligned}$$

$$\langle decl\rangle \longrightarrow \mathsf{void\ var\ ()}\ \{\ \langle block\rangle\ \}$$

$$\begin{aligned}
\langle block\rangle\ .same &= \emptyset\\
\langle block\rangle\ .env &= \langle decl\rangle\ .env \oplus\\
&\qquad \{\mathsf{var}.id \mapsto \mathsf{void}\ ()\}\\
\langle decl\rangle\ .new &= (\mathsf{var}.id, \mathsf{void}\ ())\\
\langle decl\rangle\ .ok &= \langle block\rangle\ .ok
\end{aligned}$$

$$\langle stat\rangle \longrightarrow \mathsf{var}\ =\ E;$$

$$\begin{aligned}
E.env &= \langle stat\rangle\ .env\\
\langle stat\rangle\ .ok &= E.ok\ \wedge\\
&\qquad \langle stat\rangle\ .env(\mathsf{var}.id)\\
&\qquad = E.typ
\end{aligned}$$

$$\langle stat\rangle \longrightarrow \{\ \langle block\rangle\ \}$$

$$\begin{aligned}
\langle block\rangle\ .env &= \langle stat\rangle\ .env\\
\langle block\rangle\ .same &= \emptyset\\
\langle stat\rangle\ .ok &= \langle block\rangle\ .ok
\end{aligned}$$

$$\langle block\rangle \longrightarrow \langle decl\rangle\ \ \langle block\rangle$$

$$\begin{aligned}
&\langle decl\rangle\ .env = \langle block\rangle\ [0].env\\
&\langle block\rangle\ [1].same\\
&= \quad \mathbf{let}\ (x, \_) = \langle decl\rangle\ .new\\
&\qquad \mathbf{in}\ \ \langle block\rangle\ [0].same \cup \{x\}\\
&\langle block\rangle\ [1].env\\
&= \quad \mathbf{let}\ (x, \tau) = \langle decl\rangle\ .new\\
&\qquad \mathbf{in}\ \ \langle block\rangle\ [0].env\\
&\qquad\quad \oplus\{x \mapsto \tau\}\\
&\langle block\rangle\ [0].ok\\
&= \quad \mathbf{let}\ (x, \_) = \langle decl\rangle\ .new\\
&\qquad \mathbf{in}\ \mathbf{if}\ \neg(x \in \langle block\rangle\ [0].same)\\
&\qquad\quad \mathbf{then}\ \ \langle decl\rangle\ .ok \wedge \langle block\rangle\ [1].ok\\
&\qquad\quad \mathbf{else}\ \mathsf{false}
\end{aligned}$$

$$\langle block\rangle \longrightarrow \langle stat\rangle\ \ \langle block\rangle$$

$$\begin{aligned}
&\langle stat\rangle\ .env = \langle block\rangle\ [0].env\\
&\langle block\rangle\ [1].env = \langle block\rangle\ [0].env\\
&\langle block\rangle\ [1].same = \langle block\rangle\ [0].same\\
&\langle block\rangle\ [0].ok\\
&= \quad (\langle stat\rangle\ .ok \wedge \langle block\rangle\ [1].ok)
\end{aligned}$$

$$\langle block\rangle \longrightarrow \epsilon$$

$$\langle block\rangle\ .ok = \mathsf{true}$$

This grammar only contains a minimalistic set of productions for the nonterminal symbol $\langle stat\rangle$. To obtain a more complete grammar, further productions for expressions like those in 4.3.3 are needed. For the case that the programming language also contains type declarations, another attribute is required that manages the actual type environment.

Since the given rules collect declarations from left to right, the use of a procedure *before* its declaration is excluded. This formalizes the intended scoping rule for the language, namely that the scope of a procedure declaration begins at the end of the declaration. Let us now change this scoping rule to allow the use of procedures starting with the beginning of the block in which they are declared. The modified attribute grammar reflecting the modified scoping rule is called $AG_{scopes+}$. In the attribute grammar $AG_{scopes+}$ the semantic rule of the attribute *env* is modified such that all procedures declared in the block are added to *env* already at the beginning of a block. The nonterminal $\langle block \rangle$ therefore receives an additional synthesized attribute *procs*, and the productions for the nonterminal $\langle block \rangle$ obtain the additional rules:

$$\begin{array}{l} \langle block \rangle \quad \longrightarrow \quad \varepsilon \\ \langle block \rangle\ .procs = \emptyset \end{array}$$

$$\begin{array}{l} \langle block \rangle \quad \longrightarrow \quad \langle stat \rangle\ \langle block \rangle \\ \langle block \rangle\ [0].procs = \langle block \rangle\ [1].procs \end{array}$$

$$\begin{array}{llll} \langle block \rangle \quad \longrightarrow \quad \langle decl \rangle\ \langle block \rangle & & & \\ \quad \langle block \rangle\ [0].procs & = & \mathbf{match}\ \ \langle decl \rangle\ .new & \\ & & \mathbf{with}\ (x, \mathsf{void}()) \ \rightarrow & \langle block \rangle\ [1].procs \oplus \{x \mapsto \mathsf{void}()\} \\ & & \quad \mid \ \_ \qquad\qquad\quad \rightarrow & \langle block \rangle\ [1].procs \end{array}$$

The procedures collected in $\langle block \rangle\ .procs$, are added to the environment $\langle block \rangle\ .env$ in the productions that introduce new blocks. The attribute grammar $AG_{scopes+}$ then has the following semantic rules:

$$\begin{array}{l} \langle stat \rangle \longrightarrow \{\ \langle block \rangle\ \} \\ \langle block \rangle\ .env = \langle stat \rangle\ .env \oplus \langle block \rangle\ .procs \end{array}$$

$$\begin{array}{l} \langle decl \rangle \longrightarrow \mathsf{void\ var}\ ()\ \{\ \langle block \rangle\ \} \\ \langle block \rangle\ .env = \langle decl \rangle\ .env \oplus \langle block \rangle\ .procs \end{array}$$

The rest of attribute grammar $AG_{scopes+}$ agrees with attribute grammar AG_{scopes}. Note that the new semantic rules induce an interesting functional dependency: inherited attributes of a nonterminal on the right side of a production depend on synthesized attributes of the same nonterminal. □

Attribute grammars can be used to generate intermediate code or even code for machines like the virtual machine as presented in the first volume *Wilhelm/Seidl: Compiler Design – Virtual Machines*. The functions realizing code-generation as described in that volume are recursively defined over the structure of programs. They use information about the program such as the types of identifiers visible in a program fragment whose computation can be described by attribute grammars as well. As an example for this kind of application, we consider a nontrivial subproblem of code generation, namely, *short-circuit evaluation* of boolean expressions.

Example 4.3.5 We consider code generation for a virtual machine like the CMA in *Wilhelm/Seidl: Compiler Design – Virtual Machines*. The code generated for a boolean expression according to attribute grammar AG_{bool} should have the following properties:

- The generated code consists only of load-instructions and conditional jumps. In particular, no boolean operations are generated.
- Subexpressions are evaluated from left to right.
- Of each subexpression as well of the whole expression only the smallest subexpressions are evaluated that uniquely determine the value of the whole (sub)expression. So, each subexpression is left as soon as its value determines the value of its containing expression.

The following code is generated for the boolean expression $(a \wedge b) \vee \neg c$ with the boolean variables a, b and c:

```
      load a
      jumpf l1        //  jump-on-false
      load b
      jumpt l2        //  jump-on-true
l1 :  load c
      jumpt l3
l2 :  //  continuation if the expression evaluates to true
l3 :  //  continuation if the expression evaluates to false
```

The attribute grammar AG_{bool} generates labels for the code for subexpressions, and it transports these labels to atomic subexpressions from which the evaluation jumps to these labels. Each subexpression E and T receives in *fsucc* the label of the successor if the expression evaluates to false, and in *tsucc* the label of the successor if it evaluates to true. A synthesized attribute *jcond* contains the relation of the value of the whole (sub)expression to its rightmost identifier.

- If *jcond* has the value true for an expression this means that the value of the expression is the same as the value of its rightmost identifier. This identifier is the last one that is loaded during the evaluation.
- If *jcond* has the value false the value of expression is the negation of the value of its rightmost identifier.

Correspondingly, a *load* instruction for the last identifier is followed by a jumpt to the label in *tsucc*, if $jcond = \text{true}$, and it is followed by a jumpf if $jcond = \text{false}$. This selection is performed by the function:

$$\text{gencjump}\,(jc, l) = \textbf{if}\ jc\ \textbf{then}\ (\text{jumpt}\,l)\ \textbf{else}\ (\text{jumpf}\,l)$$

As a context for boolean expressions we add a production for a two-sided conditional statement. The labels *tsucc* and *fsucc* of the condition quite naturally correspond to the start addresses of the code for the *then* and the *else* parts of the conditional statements. The code for the condition ends in a conditional jump to the *else* part. It tests the condition E for the value false. Therefore the function gencjump receives $\neg \mathit{jcond}$ as first parameter. We obtain:

$$\begin{array}{lll}
\langle \mathit{if\_stat}\rangle \longrightarrow \mathsf{if}\ (E)\ \langle \mathit{stat}\rangle\ \mathsf{else}\ \langle \mathit{stat}\rangle & & \\
\quad E.\mathit{tsucc} & = \mathsf{new}() & \\
\quad E.\mathit{fsucc} & = \mathsf{new}() & \\
\quad \langle \mathit{if\_stat}\rangle\,.\mathit{code} = & \quad\ \ \mathbf{let}\ t\ =\ E.\mathit{tsucc} & \\
 & \mathbf{in\ let}\ e\ =\ E.\mathit{fsucc} & \\
 & \mathbf{in\ let}\ f\ =\ \mathsf{new}() & \\
 & \mathbf{in}\ \ E.\mathit{code}\ \hat{}\,\mathsf{gencjump}\,(\neg E.\mathit{jcond}, e)\,\hat{} & \\
 & \quad t\ :\ \hat{}\ \langle \mathit{stat}\rangle\,[1].\mathit{code}\,\hat{}\,\mathsf{jump}\ f\,\hat{} & \\
 & \quad e\ :\ \hat{}\ \langle \mathit{stat}\rangle\,[2].\mathit{code}\,\hat{} & \\
 & \quad f\ : &
\end{array}$$

$$\begin{array}{lllll}
E & \to & T & & \\
E & \to & E\ \mathsf{or}\ T & & \\
 & & E[1].\mathit{tsucc} = E[0].\mathit{tsucc} & & T.\mathit{tsucc} = E[0].\mathit{tsucc} \\
 & & E[1].\mathit{fsucc} = \mathsf{new}() & & T.\mathit{fsucc} = E[0].\mathit{fsucc} \\
 & & E[0].\mathit{jcond} = T.\mathit{jcond} & & \\
 & & E[0].\mathit{code}\ = \mathbf{let}\ t\ =\ E[1].\mathit{fsucc} & & \\
 & & \qquad\qquad\quad \mathbf{in}\ \ E[1].\mathit{code}\,\hat{}\,\mathsf{gencjump}\,(E[1].\mathit{jcond}, E[0].\mathit{tsucc})\,\hat{} & & \\
 & & \qquad\qquad\qquad t\ :\ \hat{}\,T.\mathit{code} & & \\
T & \to & F & & \\
T & \to & T\ \mathsf{and}\ F & & \\
 & & T[1].\mathit{tsucc} = \mathsf{new}() & & F.\mathit{tsucc} = T[0].\mathit{tsucc} \\
 & & T[1].\mathit{fsucc} = T[0].\mathit{fsucc} & & F.\mathit{fsucc} = T[0].\mathit{fsucc} \\
 & & T[0].\mathit{jcond} = F.\mathit{j\,cond} & & \\
 & & T[0].\mathit{code}\ = \mathbf{let}\ f\ =\ T[1].\mathit{tsucc} & & \\
 & & \qquad\qquad\quad \mathbf{in}\ \ T[1].\mathit{code}\,\hat{}\,\mathsf{gencjump}\,(\neg T[1].\mathit{jcond}, T[0].\mathit{fsucc})\,\hat{} & & \\
 & & \qquad\qquad\qquad f\ :\ \hat{}\,F.\mathit{code} & &
\end{array}$$

$$\begin{array}{lll}
F \to (E) & & \\
F \to \mathsf{not}\ F & & \\
\quad F[1].\mathit{tsucc} & = & F[0].\mathit{fsucc} \\
\quad F[1].\mathit{fsucc} & = & F[0].\mathit{tsucc} \\
\quad F[0].\mathit{code} & = & F[1].\mathit{code} \\
\quad F[0].\mathit{jcond} & = & \neg F[1].\mathit{jcond} \\
F \to \mathsf{var} & & \\
\quad F.\mathit{jcond} & = & \mathsf{true} \\
\quad F.\mathit{code} & = & \mathsf{load\ var}.\mathit{id}
\end{array}$$

Here, the infix operator ˆ denotes the concatenation of code fragments. This attribute grammar is not in normal form: The semantic rule for the synthesized attribute *code* of the left side ⟨*if_stat*⟩ in the first production uses the inherited attributes *tsucc* and *fsucc* of the nonterminal E on the right side. The reason is that the two inherited attributes are computed using a function new() that generates a new label every time it is called. Since it implicitly changes a global state, calls to the function new() are, puristically viewed, not admitted in semantic rules of attribute grammars.

Here, at least two solutions are conceivable:

- The global state, that is, the counter of already allocated labels, is propagated through the parse tree in dedicated auxiliary attributes. Generating a new label then accesses these local attributes without referring to any global state. The disadvantage of this procedure is that the esssential flow of computation within the attribute grammar is blurred by the auxiliary attributes.
- We do allow functions that access a global state such as the auxiliary function new() as described in the example grammar. Then, however, we have to abandon normalization of some semantic rules, since function calls referring to the global state may not be dublicated. Furthermore, we must convince ourselves that distinct sequences of attribute evaluations, while not always returning identical results, will at least always return *acceptable* results.

□

4.4 The Generation of Attribute Evaluators

This section treats attribute evaluation, more precisely the evaluation of attribute instances in parse trees, and the generation of the corresponding evaluators. An attribute grammar defines for each parse tree t of the underlying CFG a system of equations AES(t), the attribute evaluation system. The unknowns in this system of equations are the attribute instances at the nodes of t. Let us assume that the attribute grammar is well-formed. In this case, the system of equations is not recursive and therefore can be solved by elimination methods. Each elimination step selects one attribute instance to be evaluated next that must only depend on attribute instances whose values have already been determined. Such an attribute evaluator is purely *dynamic* if it does not exploit any information about the dependences in the attribute gammar. One such evaluator is described in the next section.

4.4.1 Demand-Driven Attribute Evaluation

A reasonably efficient dynamic attribute evaluator for well-formed attribute grammars is obtained if attribute instances are evaluated *on demand*. Demand-driven evaluation means that not all attribute instances necessarily will receive their values. Instead, values of attribute instances are evaluated only when their values are *queried*. This demand-driven evaluation is performed by a recursive function solve, which is called for a node n and one of the attributes a of the symbol that labels n. The evaluation starts by checking whether the queried attribute instance $n.a$ has already received its value. If this is the case, the function returns with the value that already has been computed. Otherwise the evaluation of $n.a$ is triggered. This evaluation may in turn query the values of other attribute instances, whose evaluation is triggered recursively. This strategy has the consequence that for each attribute instance in the parse tree the right side of its semantic rule is evaluated at most once. The evaluation of attribute instances that are never demanded is avoided.

To realize this idea all attribute instances that are not initialized are set to the value Undef before the first value enquiry. Each attribute instance initialized with a non-Undef value d is set to the value Value d. For navigation in the parse tree we use the postfix operators $[i]$ to go from a node n to its ith successor. For $i = 0$ the navigation stays at n. Furthermore, we need an operator father that, when given a node n, returns the pair (n', j) consisting of the father n' of node n and the information in which direction, seen from n', to find n. This latter information says which child of its father n' the argument node is. To implement the function solve for the recursive evaluation, we need a function eval. If p is the production that was applied at node n, and if

$$f(p[i_1].a_1, \ldots, p[i_r].a_r)$$

is the right side of the semantic rule for the attribute occurrence $p[i].a$, eval n (i, a) returns the value of f, where for each demanded attribute instance the function solve is called. Therefore, we define:

$$\mathsf{eval}\ n\ (i, a) = f(\mathsf{solve}\ n[i_1]\ a_1, \ldots, \mathsf{solve}\ n[i_r]\ a_r)$$

In a simultaneous recursion with the function eval the function solve is implemented by:

```
solve $n$ $a$ = match $n.a$
      with Value $d$  →  $d$
       |   Undef      →  if $b \in S(\mathsf{symb}(n))$
                         then  let $d$ = eval $n$ $(0, a)$
                               in let _ = $n.a$ ← Value $d$
                               in $d$
                         else  let $(n', j')$ = father $n$
                               in let $d'$ = eval $n'$ $(j', a)$
                               in let _ = $n.a$ ← Value $d'$
                               in $d'$
```

The function solve checks whether the attribute instance $n.a$ in the parse tree already has a value. If this is the case, solve returns this value. If the attribute instance $n.a$ does not yet have a value, $n.a$ is labeled with Undef. In this case the semantic rule for $n.a$ is searched for.

If a is a synthesized attribute of the symbol at node n, a semantic rule for a is supplied by the production p at node n. The right side f of this rule is modified such that it does not directly attempt to access its argument attribute instances, but instead calls the function solve recursively for these instances for node n. If a value d for the attribute instance $n.a$ is obtained, it is assigned to the attribute instance $n.a$ and in addition returned as result.

If, on the other hand, a is an inherited attribute of the symbol at node n, the semantic rule for $n.a$ is not supplied by the production at n, but instead by the production at the father of n. Let n' be the father of n and n be the j'th child of n'. The semantic rule for the attribute occurrence $p'[j'].a$ is chosen if the production p' is applied at node n'. Its right side is again modified in the same way such that before any access to attribute values the function solve is called. The computed value is again stored in the attribute instance $n.a$ and returned as result.

If the attribute grammar is well-formed, the demand-driven evaluator always computes for each parse tree and for each attribute instance the correct value. If the attribute grammar is not well-formed, the attribute evaluation systems for some parse trees may be recursive. If t is such a parse tree, there are a node n and an attribute a at n in t such that $n.a$ depends, directly or indirectly, on itself, implying that the call solve n a may not terminate. To avoid nontermination, attribute instances are labeled with Called if their evaluation has started, but not yet terminated. Furthermore, the function solve is modified to terminate and return some error value whenever it meets an attribute instance labeled with Called (see Exercise 8).

4.4.2 Static Precomputations of Attribute Evaluators

Dynamic attribute evaluation does not exploit information about the attribute grammar to improve the efficiency of attribute evaluation. More efficient attribute-evaluation methods are possible if knowledge of the functional dependences in productions is taken into account. An attribute occurrence $p[i].a$ in production p functionally depends on an occurrence $p[j].b$ if $p[j].b$ is an argument of the semantic rule for $p[i].a$. The production-local dependences determine the dependences in the system of equation $\mathsf{AES}(t)$. Based on the functional dependences, sometimes attributes can be evaluated according to statically determined *visit sequences*. The visit sequences guarantee that an attribute instance is only scheduled for evaluation when the argument instances for the corresponding semantic rule are already evaluated. Consider again Fig. 4.6. Attribute evaluation requires a cooperation of the *local* computations at a node n and and its successors $n_1, \ldots, n_k$, and those in the context of this production instance. A local computation of an instance of a synthesized attribute at a node n labeled with X_0 provides an attribute value to be used by local computation at the ancestor of n in the upper context. The computation of the value of an inherited attribute instance at the same node n takes place at the ancestor of n and may enable further evaluations according to the semantic rules of the production corresponding to n. A similar exchange of data takes place through the attribute instances at the nodes $n_1, \ldots, n_k$ with the computations within the subtrees. To schedule this interaction of computations, *global* functional

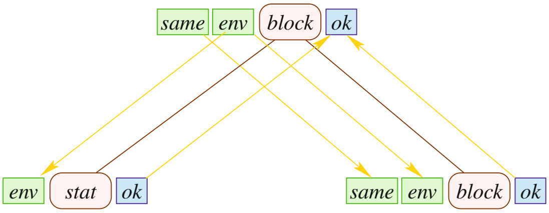


Fig. 4.7 The production-local dependence relation to production $\langle block\rangle \rightarrow \langle stat\rangle\ \langle block\rangle$ in AG_{scopes}

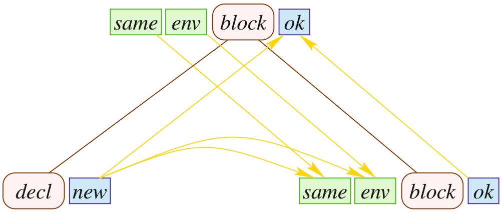


Fig. 4.8 The production-local dependence relation to production $\langle block\rangle \rightarrow \langle decl\rangle\ \langle block\rangle$ in AG_{scopes}

dependences between attribute instances need to be derived from production-local dependences. We introduce the following notions.

For a production p let $O(p)$ be the set of attribute occurrences in p. The semantic rules for production p define a relation $D(p) \subseteq O(p) \times O(p)$ of *production-local* functional dependences on the set $O(p)$. The relation $D(p)$ contains a pair $(p[j].b, p[i].a)$ of attribute occurrences if and only if $p[j].b$ occurs as an argument in the semantic rule for $p[i].a$.

Example 4.4.1 (Continuation of Examples 4.3.4 and 4.3.3) To increase readability we always represent attribute-dependence relations together with the underlying syntactic structure, that is, the production of the parse tree. The dependence relations for the productions $\langle block\rangle \rightarrow \langle stat\rangle\ \langle block\rangle$ and $\langle block\rangle \rightarrow \langle decl\rangle\ \langle block\rangle$ in AG_{scopes} are shown in Figs. 4.7 and 4.8. While here the dependences are always directed from left to right, the dependences of the extra attribute *procs* of the extended attribute grammar $AG_{scopes+}$ in production $\langle stat\rangle \rightarrow \{\ \langle block\rangle\ \}$ is oriented in the opposite direction (Fig. 4.9). The production-local dependence relations for attribute grammar AG_{types} are all very simple: There are dependences between the occurrence of the inherited attribute *env* on the left side and the inherited occurrence of attribute *env* on the right side, and between the synthesized attributes *type* and *op* on the right side to the synthesized attribute *type* on the left side (see Fig. 4.10). Only in production $F \rightarrow \texttt{var}$ is there a dependence between the attributes *env* and *type* of nonterminal F (see Fig. 4.11). □

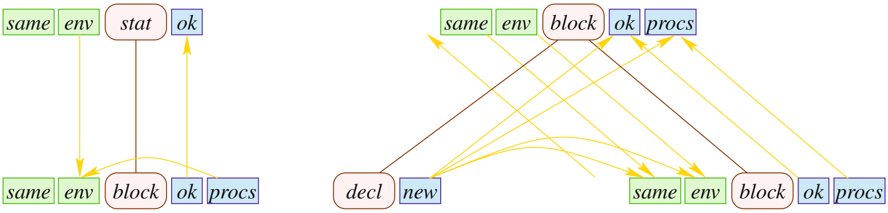


Fig. 4.9 The production-local dependence relation to productions $\langle stat \rangle \rightarrow \{ \langle block \rangle \}$ and $\langle block \rangle \rightarrow \langle decl \rangle \; \langle block \rangle$ in $AG_{scopes+}$. Here, the terminal leaves for opening and closing brackets have been omitted

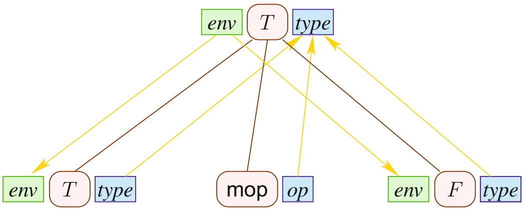
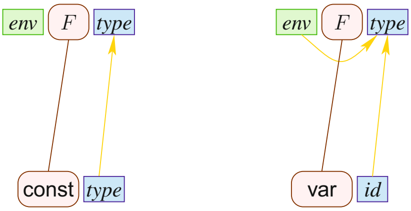


Fig. 4.10 The production-local dependence relation to production $T \rightarrow T$ mop F in AG_{types}

Fig. 4.11 The production-local dependence relation to productions $F \rightarrow$ const and $F \rightarrow$ var in AG_{types}.



In attribute grammars in normal form the arguments of semantic rules for defining occurrences are always applied attribute occurrences. Therefore the paths in production-local dependence relations all have length 1, and there exist no cycles of the form $(p[i].a, p[i].a)$. The adherence to normal form therefore simplifies some considerations.

The production-local dependences between attribute occurrences in productions induce dependences between attribute instances in the parse trees of the grammar. Let t be a tree of the CFG underlying an attribute grammar. The *individual* dependence relation on the set $I(t)$ of attribute instances of t, $D(t)$ is obtained by *instantiating* the production-local dependence relations of productions applied in t. For each node n in t at which production p has been applied, the relation $D(t)$ consists of exactly the pairs $(n[j].b, n[i].a)$ with $(p[j].b, p[i].a) \in D(p)$.

Example 4.4.2 (Continuation of Example 4.3.4) The dependence relation for the parse tree of statement $\{$ **int** $x; \; x = 1; \}$ according to attribute grammar $AG_{scopes+}$ is shown in Fig. 4.12. For simplicity we assume that the nonterminal *type* directly

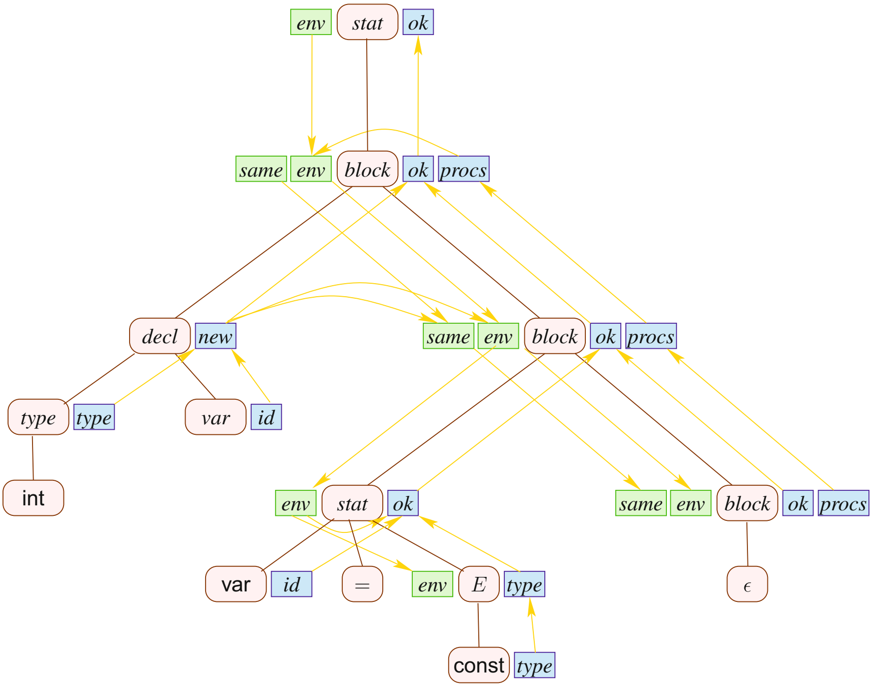


Fig. 4.12 The individual dependence relation for the parse tree to $\{$ int $x;\ x = 1;\ \}$ according to attribute grammar AG_{scopes}

derives the base type int, and that nonterminal E for expressions directly derives the terminal const. □

A relation R on a set A is called *cyclic* if its transitive closure contains a pair (a, a). Otherwise we call the relation R *acyclic*. An attribute grammar is called *noncircular*, if all its individual dependence relations are acyclic. An individual dependence relation $D(t)$ is acyclic if and only if the system of equations AES(t) that was introduced in Sect. 4.3.1 is not recursive. Attribute grammars that satisfy the latter condition are called well-formed. Thus, an attribute grammar is well-formed if and only it is noncircular.

Consider a parse tree t with root label X as in Fig. 4.13. The instances of the inherited attributes at the root are viewed as input to t, and the instances of the synthesized attributes at the root as output of t. The instance of d at the root (transitively) depends only on the instance of c at the root. If the value of the instance of c is known, an attribute evaluator can descend into t and return with the value for the instance of d since there are no other dependences of instances external to t that do not pass through c. The instance of e at the root depends on the instances of a and b at the root. When both values are available the evaluation of the instance of e

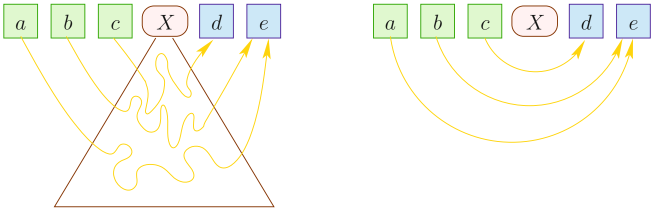
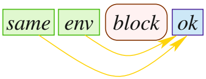


Fig. 4.13 Attribute dependences in a parse tree for X and the induced lower characteristic dependence relation

Fig. 4.14 Lower characteristic dependence relation for *block*



can be triggered. This situation is described by the *lower characteristic dependence relation* of X induced by t. This is a relation over the set $\mathcal{A}(X)$.

Let t be a parse tree for a symbol X with root n. The lower characteristic dependence relation $L_t(X)$ for X induced by t consists of all pairs (a, b) of attributes for which the pair $(n.a, n.b)$ of attribute instances at the root n of t is in the transitive closure of the individual dependence relation $D(t)$. In particular is

$$L_t(X) \subseteq \mathcal{I}(X) \times \mathcal{S}(X).$$

Example 4.4.3 (Continuation of Example 4.4.2) The lower characteristic dependence relation for the nonterminal *block* induced by the subtree of t with root *block* in Example 4.4.2 is shown in Fig. 4.14. □

While the set of dependence relations of an attribute grammar is in general infinite, the set of *lower* dependence relations is always finite since they refer to the finitely many attributes of single nonterminals only. The set of all lower dependence relations therefore can be computed and used to decide whether the dependence relations $D(t)$ for each parse tree t is acyclic and thus, whether the demand-driven attribute evaluator always terminates.

Let X be a symbol with a set $\mathcal{A}$ of attributes. For a relation $L \subseteq \mathcal{A}^2$ and $i \geq 0$ and a production p we define the relation $L[p, i]$ as

$$L[p, i] = \{(p[i].a,\, p[i].b) \mid (a, b) \in L\}$$

Consider a production $p: \;\; X \longrightarrow X_1 \ldots X_k$. For a binary relation $S \subseteq O(p)^2$ over the set of attribute occurrence of production p we define the following two operations

$$\begin{array}{lcll} S^+ & = & \bigcup\{S^j \mid j \geq 1\} & \text{(transitive closure)} \\ \pi_i(S) & = & \{(a, b) \mid (p[i].a,\, p[i].b) \in S\} & \text{(projection)} \end{array}$$

Projection allows us to extract the induced dependences between the attributes of a symbol occurring in a production p from a dependence relation for attribute occurrences in p. We can thereby define the effect $[\![p]\!]^\sharp$ of the application of production p on the dependence relations $L_1, \ldots, L_k$ for symbol occurrences $p[i]$ on the right side of p by:

$$[\![p]\!]^\sharp(L_1, \ldots, L_k) = \pi_0((D(p) \cup L_1[p,1] \cup \ldots \cup L_k[p,k])^+)$$

The operation $[\![p]\!]^\sharp$ takes the local dependence relation of production p and adds the instantiated dependence relations for the symbol occurrences of the right side. The transitive closure of this relation is computed and then projected to the attributes of the left-side nonterminal of p. If production p is applied at the root of a parse trees t, and if the relations $L_1, \ldots, L_k$ are the lower dependence relations for the subtrees under the root of t the characteristic dependence relation for t is obtained by

$$L_t(X) = [\![p]\!]^\sharp(L_1, \ldots, L_k)$$

The sets $\mathcal{L}(X), X \in V$ of all lower dependence relations for nonterminal symbols X result as the least solution of the system of equations

$$\begin{aligned} (\mathcal{L}) \quad \mathcal{L}(a) &= \{\emptyset\}, \quad a \in V_T \\ \mathcal{L}(X) &= \{[\![p]\!]^\sharp(L_1, \ldots, L_k) \mid p : X \to X_1 \ldots X_k \in P, L_i \in \mathcal{L}(X_i)\}, \\ & \quad X \in V_N \end{aligned}$$

Here, V_T, V_N, and P are the sets of terminal and nonterminal symbols, and productions, respectively, of the underlying CFG. Each right side of these equations is *monotonic* in each unknown $\mathcal{L}(X_i)$ on which it depends. The set of all transitive binary relations over a finite set is finite. Therefore the set of its subsets is also finite. Hence, the least solution of this system of equations, i.e., the set of all lower dependence relations for each X, can be iteratively determined. The sets of all lower dependence relations $\mathcal{L}(X)$ allow for an alternative characterization of noncircularity of attribute grammars. We have:

Lemma 4.4.1 For an attribute grammar the following statements are equivalent:

1. For every parse tree t, the lower dependence relation $L_t(X)$ (X label of the root of t) is acyclic;
2. For each production $p : X \to X_1 \ldots X_k$ and all dependence relations $L_i \in \mathcal{L}(X_i)$, the relation

$$D(p) \cup L_1[p,1] \cup \ldots \cup L_k[p,k]$$

   is acyclic. □

Since the sets $\mathcal{L}(X)$ are finite and can be effectively computed, the lemma provides us with a decidable characterization of well-formed attribute grammars. We obtain:

Theorem 4.4.1 It is decidable whether an attribute grammar is well-formed. □

In order to decide well-formedness, the sets $\mathcal{L}(X)$ of all lower dependence relations of the attribute grammar for symbols X must be computed. These sets are finite, but their sizes may grow exponentially in the number of attributes. The check for noncircularity is thus only practically feasible if either the number of attributes is small, or the symbols have only few lower dependence relations. In general, though, the exponential effort is inevitable since the problem to check for noncircularity of an attribute grammar is *EXPTIME*-complete.

In many attribute grammars a nonterminal X may have several lower characteristic dependence relations, but these are all contained in one common transitive acyclic dependence relation.

Example 4.4.4 Consider the attribute grammar $AG_{scopes+}$ in Example 4.3.4. For nonterminal $\langle block \rangle$ there are the following lower characteristic dependence relations:

$$\begin{array}{ll} (1) & \emptyset \\ (2) & \{(same, ok)\} \\ (3) & \{(env, ok)\} \\ (4) & \{(same, ok), (env, ok)\} \end{array}$$

The first three dependence relations are all contained in the fourth. □

To compute for each symbol X a transitive relation that contains all lower characteristic dependences for X we set up the following system of equations over transitive relations:

$$\begin{array}{ll} (\mathcal{R}) & \mathcal{R}(a) = \emptyset\,, \quad a \in V_T \\ & \mathcal{R}(X) = \bigsqcup \{[\![p]\!]^\sharp(\mathcal{R}(X_1), \ldots, \mathcal{R}(X_k)) \mid p : X \to X_1 \ldots X_k \ \in P\}\,, \\ & \qquad X \in V_N \end{array}$$

The partial order on transitive relations is the subset relation $\subseteq$. Note that the least upper bound of the transitive relations $R \in S$ is not just their union. Instead we have:

$$\bigsqcup S = (\bigcup S)^+$$

i.e., following the union of the relations the transitive closure must be recomputed. For each production p the operation $[\![p]\!]^\sharp$ is monotonic in each of its arguments. Therefore the system of equations possesses a least solution. Since there are only finitely many transitive relations over the set of attributes this solution can again be determined by iteration. Let $\mathcal{L}(X), X \in V$, and $\mathcal{R}(X), X \in V$, be the least solutions of the systems of equations $(\mathcal{L})$ and $(\mathcal{R})$. By induction over the iterations of the fixed-point algorithm, it can be proved that for all $X \in V$,

$$\mathcal{R}(X) \ \supseteq \ \bigcup \mathcal{L}(X)$$

holds. We conclude that all characteristic lower dependence relations of the attribute grammar are acyclic if all relations $\mathcal{R}(X), X \in V$, are acyclic.

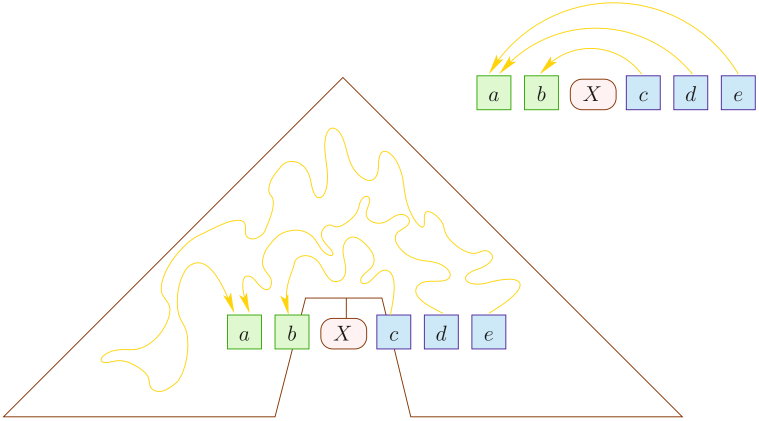


Fig. 4.15 Attribute dependences in an upper tree fragment for X and the induced upper characteristic dependence relation

An attribute grammar is called *absolutely noncircular* if all relations $\mathcal{R}(X)$, $X \in V$, are acyclic as well as for each production $p : X_0 \to X_1 \dots X_k$ the relation $D(p) \cup \mathcal{R}(X_1)[p, 1] \cup \mathcal{R}(X_k)[p, k]$. Each absolutely noncircular attribute grammar is thereby also well-formed. This means that for absolutely noncircular attribute grammars the algorithm for demand-driven attribute evaluation always terminates. Solving the system of equations $(\mathcal{R})$ provides us with a polynomial criterion to guarantee the applicability of demand-driven attribute evaluation.

Similar to the *lower* characteristic dependence relations of a symbol X the *upper* characteristic dependence relation for X can be defined. It is derived from attribute dependences of upper tree fragments for X. Recall that the upper tree fragment of a parse tree t at n is the tree that is obtained by replacing the subtree at n with the node n. This upper tree fragment is denoted by $t \backslash n$. Let $D(t \backslash n)$ be the individual dependence relation of the upper tree fragment, i.e., the set of all pairs $(n_1.a, n_2, b)$ of the individual dependence relation $D(t)$ for which n_1 as well as n_2 lie in the upper tree fragment $t \backslash n$. The upper characteristic dependence relation $U_{t,n}(X)$ for X at node n in t consists of all pairs $(a, b) \in \mathcal{A}(X) \times \mathcal{A}(X)$, for which the pair $(n.a, n.b)$ lies in the transitive closure of $D(t \backslash n)$ (see Fig. 4.15). A system of equations over sets of transitive relations can also be constructed for the set $\mathcal{U}(X)$ of all possible upper characteristic dependence relations of symbol X (see Exercise 9).

4.4.3 Visit-Oriented Attribute Evaluation

The advantage of a statically generated attribute evaluator over the demand-driven dynamic evaluator is that the behavior of the evaluator at each node is already statically fixed at generation time. No checks of whether attribute instances are already evaluated are required at evaluation time. The largest class of attribute grammars for which we describe the generation of attribute evaluators is the class of *l-ordered* or *simple-multivisit* attribute grammars. An attribute grammar is called *l-ordered* if there is a function $\mathcal{T}$ that maps each symbol X to a *total* order $\mathcal{T}(X) \subseteq \mathcal{A}^2$ on the set $\mathcal{A}$ of attributes of X that is compatible with all productions. This means that for each production $p : X_0 \longrightarrow X_1 \ldots X_k$ of the underlying grammar the relation

$$D_{\mathcal{T}}(p) = (D(p) \cup \mathcal{T}(X_0)[p,0] \cup \ldots \cup \mathcal{T}(X_k)[p,k])^+$$

is acyclic. This property holds if and only if

$$\mathcal{T}(X_i) = \pi_i((D(p) \cup \mathcal{T}(X_0)[p,0] \cup \ldots \cup \mathcal{T}(X_k)[p,k])^+)$$

holds for all i. Therefore in particular:

$$\mathcal{T}(X_0) \quad \supseteq \quad [\![p]\!]^\sharp(\mathcal{T}(X_1), \ldots, \mathcal{T}(X_k))$$

By comparing this inequality with the equation for the unknown X_0 in the system of equations $(\mathcal{R})$ in the last section, we conclude that the total order $\mathcal{T}(X_0)$ contains the dependence relation $\mathcal{R}(X_0)$. Since $\mathcal{T}(X_0)$ is a total order and therefore acyclic, the attribute grammar is absolutely noncircular, and all local lower dependence relations at X_0 are contained in $\mathcal{T}(X_0)$. In analogy, it can be shown that $\mathcal{T}(X_0)$ contains all *upper* dependence relations at X_0.

Example 4.4.5 (Continuation of Example 4.3.4) In the attribute grammar $AG_{scopes+}$ the following total orders on the sets of attributes offer themselves for the symbols *stat*, *block*, *decl*, E, and var:

$$\begin{array}{llllllll}
\langle stat \rangle & env & \rightarrow & ok & & & & \\
\langle block \rangle & procs & \rightarrow & same & \rightarrow & env & \rightarrow & ok \\
\langle decl \rangle & new & & & & & & \\
E & env & \rightarrow & ok & & & & \\
\mathsf{var} & id & & & & & &
\end{array}$$

□

Let $B_{\mathcal{T}}(X) \in \mathcal{A}(X)^*$ be the sequence of the attributes of X according to the total order $\mathcal{T}(X)$. This linear sequence can be factored into subsequences of only inherited and of only synthesized attributes. In our example this factorization is very simple for all considered symbols: All inherited attributes occur before all synthesized attributes. The only notable exception is the nonterminal $\langle block \rangle$, where the synthesized attribute *procs* precedes all other attributes. Thus in general, some inherited attributes may depend on some synthesized attributes. We then obtain a factorization:

$$B_{\mathcal{T}}(X) = I_{X,1} S_{X,1} \ldots I_{X,r_X} S_{X,r_X}$$

where $I_{X,i} \in \mathcal{I}(X)^*$ and $S_{X,i} \in \mathcal{S}(X)^*$ holds for all $i = 1, \ldots, r_X$, and furthermore $I_{X,i} \neq \epsilon$ for $i = 2, \ldots, r_X$, and $S_{X,i} \neq \epsilon$ for $i = 1, \ldots, r_X - 1$.

Intuitively, this factorization of the sequence $B_{\mathcal{T}}(X)$ means that the synthesized attributes at each node of a parse tree labeled with X can be evaluated in at most r_X visits; at the first visit of the node, coming from the parent node, the values of the inherited attributes in $I_{X,1}$ are available, at the return to the parent node, the values of the synthesized attributes in $S_{X,1}$ are evaluated. Correspondingly, at the ith visit of the node, the values of the inherited attributes in $I_{X,1} \ldots I_{X,i}$ are available, and the synthesized attributes in $S_{X,i}$ are computed. A subsequence $I_{X,i} S_{X,i}$ of $B_{\mathcal{T}}(X)$ is called a *visit* of X. To determine which evaluations may be performed during the ith visit at a node n and at the successors of the node n, one considers the dependence relation $D_{\mathcal{T}}(p)$ for the production $X_0 \longrightarrow X_1 \ldots X_k$ that is applied at n. Since the relation $D_{\mathcal{T}}(p)$ is acyclic, $D_{\mathcal{T}}(p)$ can be arranged into a linear order. In our case, we choose the order $B_{\mathcal{T}}(p)$, which can be factorized into *visits*. Altogether we obtain for the relation $D_{\mathcal{T}}(p)$ a visit sequence:

$$B_{\mathcal{T}}(p) = B_{\mathcal{T},1}(p) \ldots B_{\mathcal{T},r_{X_0}}(p)$$

The ith subsequence $B_{\mathcal{T},i}(p)$ describes what happens during the ith visit of a node n at which the production $p : X_0 \longrightarrow X_1 \ldots X_k$ is applied. For each occurrence of inherited attributes of the X_j $(j > 0)$ in the subsequence, the corresponding attribute instances are computed one after the other. After the computation of the listed inherited attribute instances of the i'th visit of the jth successor this successor is recursively visited to determine the values of the synthesized attributes associated with the i'th visit. When the values of the synthesized attributes of all successors are available that are directly or indirectly needed for th computation of the synthesized attributes of the ith visit of the left side X_0 the values of these synthesized attributes are computed.

To describe the subsequence $B_{\mathcal{T},i}(p)$ in an elegant way we introduce the following abbreviations. Let $w = a_1 \ldots a_l$ be a sequence of attributes of nonterminal X_j. $p[j].w = p[j].a_1 \ldots p[j].a_l$ shall denote the associated sequence of attribute occurrences in p. The i'th visit $I_{X_j,i'} S_{X_j,i'}$ of the jth symbol of the production p is denoted by the sequence $p[j].(I_{X_j,i'}\ S_{X_j,i'})$. The sequence $B_{\mathcal{T},i}(p)$, interpreted as a sequence of attribute occurrences in p, has the form:

$$\begin{array}{rl} B_{\mathcal{T},i}(p) = & p[0].I_{X_0,i} \\ & \quad p[j_1].(I_{X_{j_1},i_1}\ S_{X_{j_1},i_1}) \\ & \quad \ldots \\ & \quad p[j_r].(I_{X_{j_r},i_r}\ S_{X_{j_r},i_r}) \\ & p[0].S_{X_0,i} \end{array}$$

for an appropriate sequence of pairs $(j_1, i_1), \ldots, (j_r, i_r)$. It consists of the visits of the nonterminal occurrences $X_{j_1}, \ldots, X_{j_r}$ of the right side of production p that are embedded in the ith visit of the left side of p.

Let p be a production and $f(p[j_1].a_1, \ldots, p[j_r].a_r)$ be the right side of the semantic rule for the attribute occurrence $p[j].a$ for a total function f. For a node

n in the parse tree at which production p has been applied, we define

$$\mathsf{eval}_{p,j,a}\ n = f(n[j_1].a_1, \ldots, n[j_r].a_r)$$

The functions $\mathsf{eval}_{p,j,a}$ are used to generate a function $\mathsf{solve}_{p,i}$ from the ith subsequence $B_{\mathcal{T},i}(p)$ of production p:

```
$\mathsf{solve}_{p,i}\ n$   =   forall $(a \in I_{X_{j_1},i_1})$
                          $n[j_1].a \leftarrow \mathsf{eval}_{p,j_1,a}\ n$;
                      $\mathsf{visit}_{i_1}\ n[j_1]$;
                      ...
                      forall $(a \in I_{X_{j_r},i_r})$
                          $n[j_r].a \leftarrow \mathsf{eval}_{p,j_r,a}\ n$;
                      $\mathsf{visit}_{i_r}\ n[j_r]$;
                      forall $(a \in S_{X_0,i})$
                          $n.a \leftarrow \mathsf{eval}_{p,0,a}\ n$;
```

Example 4.4.6 (Continuation of Example 4.3.4) The production-local dependence relation for the production $block \rightarrow decl\ block$ of the attribute grammar $AG_{scopes+}$ is obtained from the relation in Fig. 4.8 by adding the total orders on the attribute occurrences of the symbols occurring in the production. Altogether, the resulting relation can be embedded into the following total order:

$$\begin{array}{l}
\quad\quad \langle decl\rangle\ .new \rightarrow \\
\quad\quad \langle block\rangle\ [1].procs \rightarrow \\
\langle block\rangle\ [0].procs \rightarrow \\
\langle block\rangle\ [0].same \rightarrow \langle block\rangle\ [0].env \rightarrow \\
\quad\quad \langle block\rangle\ [1].same \rightarrow \langle block\rangle\ [1].env \rightarrow \langle block\rangle\ [1].ok \rightarrow \\
\langle block\rangle\ [0].ok
\end{array}$$

According to this total order, the evaluator for the attribute grammar $AG_{scopes+}$ first descends into the subtree for the nonterminal *decl* to determine the value of the attribute *new*. Then the second subtree must be visited in order to determine the synthesized attribute *procs* of the nonterminal $\langle block\rangle$ on the right and then also on the left side of the production. For the second visit, the inherited attributes *procs*, *same* Und *env* of the left side have already been computed. The evaluation then again descends into the subtree of the nonterminal $\langle block\rangle$ on the right side during which the value of the synthesized attribute *ok* is determined. Then all values are available in order to determine the value of the synthesized attribute *ok* of the left side $\langle block\rangle$ of the production.

In the simpler attribute grammar AG_{scopes}, the attribute *procs* is not necessary. There, a single visit suffices to evaluate all attributes. A meaningful ordering is given by:

$$\begin{array}{l} \langle block\rangle\,[0].same \to \langle block\rangle\,[0].env \to \\ \quad \langle decl\rangle\,.new \to \\ \quad \langle block\rangle\,[1].same \to \langle block\rangle\,[1].env \to \langle block\rangle\,[1].ok \to \\ \langle block\rangle\,[0].ok \end{array}$$

□

The evaluation orders in visit_i are chosen in such a way that the value of each attribute instance $n[j'].b$ is computed before any attempt is made to read its value. The functions $\mathsf{solve}_{p,i}$ are simultaneously recursive with themselves and with the functions visit_i. For a node n let $\mathsf{get\_prod}\ n$ be the production that was applied at n or Null if n is a leaf that is labeled with a terminal symbol or ϵ. If $p_1, \ldots, p_m$ is a sequence of the productions of the grammar, the function visit_i is given by:

$$\begin{array}{lclccl} \mathsf{visit}_i\ n & = & \mathbf{match}\ \mathsf{get\_prod}\ n & & & \\ & & \mathbf{with} & \mathsf{Null} & \to & () \\ & & \mid & p_1 & \to & \mathsf{solve}_{p_1,i}\ n \\ & & & & \ldots & \\ & & \mid & p_m & \to & \mathsf{solve}_{p_m,i}\ n \end{array}$$

For a node n the function visit_i checks whether n is a leaf or whether it was generated by the application of a production. If n is a leaf the evaluator doesn't need to do anything, if we assume that the synthesized attributes of the leaves were properly initialized. The evaluator recognizes that n is a leaf if the call $\mathsf{get\_prod}\ n$ returns the value Null. If n is not a leaf the call $\mathsf{get\_prod}\ n$ returns the production p_j at n. In this case the function $\mathsf{solve}_{p_j,i}$ is called for n.

Let S be the start symbol of the context-free grammar underlying the attribute grammar. If S has no inherited attributes then $B_{\mathcal{T}}(X)$ consists of one order of only synthesized attributes, which one can evaluate in a single visit. The evaluation of all attribute instances in a parse tree t with root n_0 for the start symbol S is then performed by the call $\mathsf{visit}_1\ n_0$.

The given evaluator can be generated in polynomial time from the attribute grammar together with the total orders $\mathcal{T}(X), X \in V$. Not every attribute grammar, though, possesses such a system of compatible total orders. The question whether an attribute grammar is l-attributed is certainly in *NP*, since total orders $\mathcal{T}(X), X \in V$, can be guessed and then checked for compatibility in polynomial time. A significantly better algorithm is not known, as this problem is not only in *NP*, but it is *NP*-complete.

In practice, therefore, only the subclass of l-ordered attribute grammars are used where a simple method delivers compatible total orders $\mathcal{T}(X), X \in V$. The starting point for the construction is the system of equations:

$$(\mathcal{R}') \quad \mathcal{R}'(X) = \bigsqcup \{\pi_i((D(p) \cup \mathcal{R}'(X_0)[p,0] \cup \ldots \cup \mathcal{R}'(X_k)[p,k])^+) \mid \\ p : X_0 \to X_1 \ldots X_k \in P, \quad X = X_i\}, \quad X \in V$$

over the *transitive* relations on attributes, ordered by the subset relation $\subseteq$. Recall that the least upper bound of transitive relations $R \in S$ is given by:

$$\bigsqcup S = \left(\bigcup S\right)^+$$

The least solution of the system of equation $(\mathcal{R}')$ exists since the operators on the right side of the equations are monotonic. The least solution can be determined by the iterative method that we used in Chapt. 3.2.5 for the computation of the first_ksets. Termination is guaranteed since the number of possible transitive relations is finite.

Let $\mathcal{R}'(X)$, for $X \in V$, be the least solution of the system of equations. Each system $\mathcal{T}(X), X \in V$, of compatible *total* orders is a solution of the system of equations $(\mathcal{R}')$. Therefore $\mathcal{R}'(X) \subseteq \mathcal{T}(X)$ holds for all symbols $X \in V$. If there exists such a system $\mathcal{T}(X), X \in V$, of compatible total orders the relations $\mathcal{R}'(X)$ are all acyclic. The relations $\mathcal{R}'(X)$ are therefore a good starting point to construct total orders $\mathcal{T}(X)$.

The construction is attempted in a way that for each X a sequence with a minimal number of visits is obtained. For a symbol X with $\mathcal{A}(X) \neq \emptyset$ a sequence $I_1 S_1 \ldots I_r S_r$ is computed, where I_i and S_i are sequences of inherited and synthesized attributes, respectively. All already listed attributes are collected in a set D, which is initialized with the empty set. Let us assume, $I_1, S_1, \ldots I_{i-1}, S_{i-1}$ are already computed, and D would contain all attributes that occur in these sequences. Two steps are executed:

1. First, a maximally large set of *inherited* attributes of X is determined that are not in D, and which only depend on each other or on attributes in D. This set is topologically sorted, delivering some sequence I_i. This set is added to D.
2. Next, a maximally large set of *synthesized* attributes is determined that are not in D, and that only depend on each other or on attributes in D. This set is added to D, and a topologically sorted sequence is produced as S_i.

This procedure is iterated producing more subsequences $I_i S_i$ until all attributes are listed, that is, until D is equal to the whole set $\mathcal{A}(X)$ of attributes of the nonterminal X.

Let $\mathcal{T}'(X), X \in V$, be the total orders on the attributes of the symbols of X that are computed this way. We call the attribute grammar *ordered* if the total orders $\mathcal{T}'(X), X \in V$, are already compatible, that is, satisfy the system of equations $(\mathcal{R}')$. In this method the relations $\mathcal{R}'(X)$ are expanded *one by one* to total orders, without checking whether the added artificial dependences generate cycles in the productions. The price to be paid for the polynomial complexity of the construction therefore is a restriction in the expressivity of the accepted attribute grammars.

In Examples 4.3.4, 4.3.3, and 4.3.5 from Sect. 4.3.2 attribute evaluators are generated by our method that visit each node of a parse tree exactly once. For the attribute grammar $AG_{scopes+}$, on the other hand, an evaluator is required that uses two visits. Several visits are also required when computing the assignments of types to identifier occurrences in JAVA. Here, the body of a class must be traversed several times because in JAVA methods can be called, although they are declared only later. A similar problem occurs in functional languages such as OCAML, when simultaneous recursive functions are introduced (see Exercise 12).

4.4.4 Parser-Directed Attribute Evaluation

In this section we consider classes of attribute grammars that are severely restricted in the types of attribute dependences they admit, but that are still useful in practice. The introductory Example 4.3.1 belongs to one of these classes. For attribute grammars in these classes, attribute evaluation can be performed in parallel to syntax analysis and directed by the parser. Attribute values are administered in a stack-like fashion either on a dedicated attribute stack or together with the parser states on the parse stack. The construction of the parse tree, at least for the purpose of attribute evaluation, is unnecessary. Attribute grammars in these classes are therefore interesting for the implementation of highly efficient compilers for not-too-complicated languages. Since attribute evaluation is directed by the parser, the values of synthesized attributes at terminal symbols need to be obtained by the scanner when the symbol is passed on to the parser.

L-Attributed Grammars

All parsers that we consider as possibly directing attribute evaluation process their input from left to right. This suggests that attribute dependences going from right to left are not acceptable. The first class of attribute grammars that we introduce, the L-attributed grammars, excludes exactly such dependences. This class properly contains all grammars subject to parser-directed attribute evaluation. It consists of those attribute grammars in normal form where the attribute instances in each parse tree can be evaluated in one *left-to-right* traversal of the parse tree. Formally, we call an attribute grammar *L-attributed* (abbreviated an $L - AG$), if for each production $p : \ X_0 \rightarrow X_1 \dots X_k$ of the underlying grammar the occurrence $p[j].b$ of an inherited attribute only depends on attribute occurrences $p[i].a$ with $i < j$. Attribute evaluation in one *left-to-right* traversal can be performed using the algorithm of Sect. 4.4.3, which visits each node in the parse tree only once and visits the children of a node in a fixed *left-to-right* order. For a production $p : \ X_0 \rightarrow X_1 \dots X_k$ a function solve_p is generated, which is defined by:

```
solve_p n  =  forall (a ∈ I_{X_1})
                  n[1].a ← eval_{p,1,a} n;
              visit n[1];
              ...
              forall (a ∈ I_{X_k})
                  n[k].a ← eval_{p,k,a} n;
              visit n[k];
              forall (a ∈ S_{X_0})
                  n.a ← eval_{p,0,a} n;
```

Here, I_X and S_X are the sets of inherited and synthesized attributes of symbol X, and the call $\mathsf{eval}_{p,j,a}\; n$ returns the value of the right side of the semantic rule for the attribute instance $n[j].a$. The visit of a node n is realized by the function visit:

$$\begin{array}{lcllll} \mathsf{visit}\; n & = & \textbf{match}\; \mathsf{get\_prod}\; n & & & \\ & & \textbf{with} & \mathsf{Null} & \rightarrow & () \\ & & \quad\mid & p_1 & \rightarrow & \mathsf{solve}_{p_1}\; n \\ & & & & \ldots & \\ & & \quad\mid & p_m & \rightarrow & \mathsf{solve}_{p_m}\; n \end{array}$$

Again the function call get_prod n returns the production that was applied at node n (or Null if n is a leaf). The attribute grammars AG_{scopes}, AG_{types}, and AG_{bool} of Examples 4.3.4, 4.3.3, and 4.3.5 are all L-attributed, where the last one is not in normal form.

LL Attributed Grammars

Let us consider the actions that are necessary for parser-directed attribute evaluation:

- When reading a terminal symbol a: Receiving the synthesized attributes of a from the scanner;
- When expanding a nonterminal X: Evaluation of the inherited attributes of X;
- When reducing to X: Evaluation of the synthesized attributes of X;

An $LL(k)$-parser as it was described in Chapt. 3 can trigger these actions at the reading of a terminal symbol, at expansion, and at reduction, respectively. An attribute grammar in normal form is called *LL-attributed*,

- if it is L-attributed, and
- if the underlying CFG is an $LL(k)$-grammar (for some $k \geq 1$).

The property of an attribute grammar to be LL-attributed means that syntax analysis can be performed by an LL-parser, and that whenever the LL-parser expands a nonterminal, all arguments for its inherited attributes are available.

In Sect. 3.3 we described how to construct a parser for an $LL(k)$-grammar. This parser administers items on its pushdown, which describe productions together with the parts of the right sides that has already been processed. We now extend this PDA such that it maintains for every item ι for a production $p : \; X_0 \rightarrow X_1 \ldots X_k$ a structure $S(\iota)$ that may receive all inherited attributes of the left side X_0 together with all synthesized attributes of the symbols $X_1, \ldots, X_k$ of the right side. If the dot of the item ι is in front of the ith symbol, all values of the inherited attributes of X_0 as well as all values of the synthesized attributes of the symbols $X_1, \ldots, X_{i-1}$ have already been computed.

Figure 4.16 visualizes the actions of the LL parser-directed attribute evaluation. Assume that the item ι corresponds to the production $p : \; A \rightarrow \alpha X \beta$ and the dot is positioned behind the prefix α of length $i - 1$, which has already been processed.

- A *shift*-transition of ι for a terminal $X = a$ moves the dot in ι across the symbol a. Additionally, the new attribute structure is obtained from $S(\iota)$ by storing the values of the synthesized attributes of a as provided by the scanner.

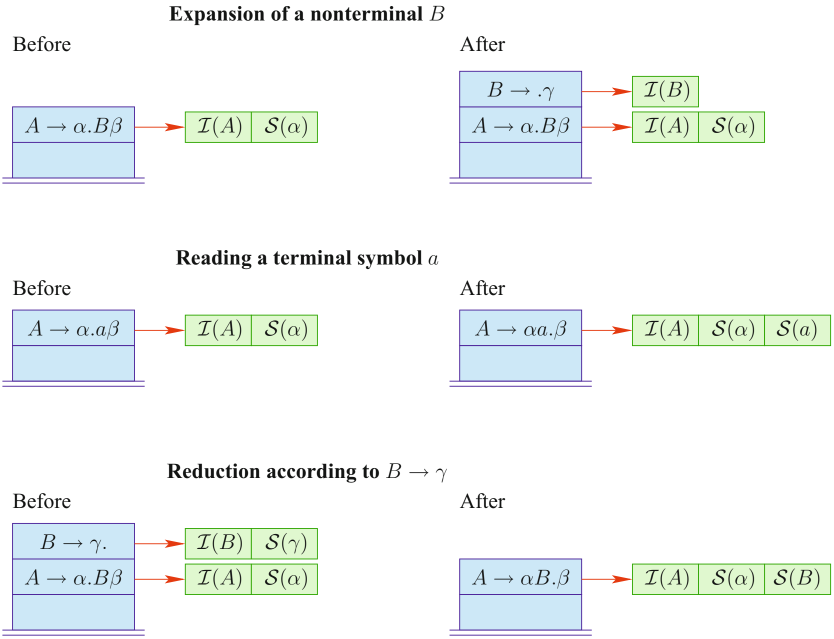


Fig. 4.16 Actions of LL parser-directed attribute evaluation, where $\mathcal{I}(A)$ and $\mathcal{S}(\alpha)$ denote the sequences of the values of the inherited attributes of a symbol A and of the synthesized attributes of the symbols in α, respectively

- An *expand*-transition for symbol $X = B$, pushes the item $\iota' = [B \to .\gamma]$ for one of the alternatives $B \to \gamma$ onto the pushdown. For the item ι', a new structure $S(\iota')$ is allocated in which each inherited attribute b of the left side B receives a value by means of the semantic rules of the production p for the attribute occurrence $p[i].b$.
- A *reduce*-transition is applied to a complete item $\iota' = [B \to \gamma.]$ on top of the pushdown. For such an item, the values of all synthesized attributes of symbols in γ as well as the values of all inherited attributes of the left side B are available in the associated structure $S(\iota')$. Therefore, the values of the synthesized attributes of the left side B can be computed according to the semantic rules of $B \to \gamma$ and stored in the structure $S(\iota)$ for B to by the item $\iota = [A \to \alpha.B\beta]$. The complete item $[B \to \gamma.]$ together with the associated structure $S(\iota')$ is removed, and the dot in $[A \to \alpha.B\beta]$ is moved over the B.

The attribute grammars AG_{types} and AG_{bool} are both L-attributed. However, both are not LL-attributed. In both cases the underlying CFG is left-recursive and therefore not $LL(k)$ for any k. For the attribute grammar AG_{bool} it can be shown that there exists no LL-attributed grammar that solves the code-generation problem in this way, that is, by propagation of two jump targets to each subexpression.

LR-Attributed Grammars

We now present a method by which an LR-parser can direct the evaluation of attributes. An LR-parser maintains states on its pushdown. States consist of sets of items, possibly extended by lookahead sets. With each such state q we associate an attribute structure $S(q)$. The attribute structure of the initial state is empty. For any other state $q \notin \{q_0, f\}$ with entry symbol X the structure $S(q)$ contains the values of the synthesized attributes of the symbol X. We extend the LR parser with a (global) attribute structure $\mathcal{I}$ which holds the value of each inherited attribute b or $\perp$ if the value of the attributes b is not available. Initially, the global attribute structure $\mathcal{I}$ contains the values of the inherited attributes of the start symbol.

The values of the synthesized attributes of a terminal symbol are made available by the scanner. Two problems need to be solved if the values of the attributes for the attribute structure $S(q)$ of a state q are computed:

- The semantic rule needs to be identified by which the attribute values should be evaluated.
- The values of the attribute occurrences that are arguments of the semantic rule need to be accessed.

The values of the synthesized attributes of a nonterminal X_0 can be computed when the LR-parser makes a *reduce*-transition: The production $p : \; X_0 \to X_1 \ldots X_k$ is known by which the reduction to X_0 is done. To compute a synthesized attribute b of X_0 the semantic rule for the attribute occurrence $p[0].b$ of this production is used. Before the reduction a sequence $q'q_1, \ldots, q_k$ of states is on top of the pushdown, where $q_1, \ldots, q_k$ have entry symbols $X_1, \ldots, X_k$ of the right side of p. Let us assume the values for the attribute structures $S(q_1), \ldots, S(q_k)$ have already been computed. The semantic rule for a synthesized attribute of X_0 can be applied by accessing the values for the occurrences $p[0].b$ of inherited attributes of the left side X_0 in $\mathcal{I}$ and the values for occurrences $p[j].b$ of synthesized attributes of X_j of the right side in $S(q_j)$. Before the *reduce*-transition, the values of the synthesized attributes of X_0 can be computed for the state $q = \Delta(q', X_0)$ that is entered under X_0. Still unsolved is the question how the values of the inherited attributes of X_0 can be determined.

For the case that there are no inherited attributes, though, we already have obtained a method for attribute evaluation. An attribute grammar is called S-*attributed* if it has only synthesized attributes. Example 4.3.1 is such a grammar. Despite the restriction to have only synthesized attributes one could describe how trees for expressions are constructed. More generally, the computation of some semantic value can be specified by an S-grammar. This mechanism is offered by parser generators such as YACC or BISON. Each S-attributed grammar is also L-attributed. If an LR-grammar is S-attributed, the attribute structures of the states can be maintained on the pushdown, and thus allow us to determine the values of synthesized attributes of the start symbol.

Attribute grammars with synthesized attributes alone are not expressive enough for more challenging compilation tasks. Even the inference of types of expressions relative to a symbol table *env* in Example 4.3.3 requires an inherited attribute, which is passed down the parse tree. Our goal therefore is to extend the approach for S-attributed grammars to deal with inherited attributes as well. The LR-parser does not in general know the upper tree fragment in which the transport paths for inherited attribute values lie. If a grammar is left-recursive, the application of an arbitrary number of semantic rules may be required to compute the value of an inherited attribute. We observe, however, that the values of inherited attributes are often passed down unchanged through the parse tree. This is the case in the attribute grammar AG_{types} of Example 4.3.3, which computes the type of an expression, where the value of the attribute *env* is copied from the left side of productions in attributes of the same name to occurrences of nonterminals on the right side. This is also the case in production $\langle block\rangle \rightarrow \langle stat\rangle\ \langle block\rangle$ of the attribute grammar AG_{scopes} in Example 4.3.4, where the inherited attribute *same* of the left side is copied to an attribute of the same name of the nonterminal occurrence $\langle block\rangle$ of the right side, and the inherited attribute *env* of the left side is copied to attributes of the same name at nonterminal occurrences of the right side.

Formally we call an occurrence $p[j].b$ of an inherited attribute b at the jth symbol of a production $P : X_0 \rightarrow X_1 \ldots X_k$ *copying* if there exists an $i < j$, such that the following holds:

1. The semantic rule for $p[j].b$ is $p[j].b = p[i].b$, *or* the right side of the semantic rule for $p[j].b$ is *semantically* equal to the right side of the semantic rule for $p[i].b$; and
2. $p[i].b$ is the last occurrence of the attribute b before $p[j].b$, that is, $b \notin \mathcal{A}(X_{i'})$ for all $i < i' < j$.

Clearly, semantic equality of right sides is in general undecidable. In a practical implementation, though, it is sufficient to refer to *syntactic* equivalence instead. At least, this covers the important case where both $p[j].b$ and $p[i].b$ are copies of the same inherited attribute b of the left side.

In this sense all occurrences of the inherited attributes *env* on the right side of the attribute grammar AG_{types} are copying. The same holds for the occurrences of the inherited attributes *same* and *env* of the attribute grammar AG_{scopes} in the production $\langle block\rangle \rightarrow \langle stat\rangle\ \langle block\rangle$.

Let us assume for a moment that the occurrences of inherited attributes in right sides were all copying. This means that the values of inherited attributes will never change. Once the global attribute structure $\mathcal{I}$ contains the right value of an inherited attribute, it therefore need not be changed throughout the whole evaluation.

Sadly enough, certain occurrences of inherited attributes of L-attributed grammars are not copying. For a noncopying occurrence $p[j].b$ of an inherited attribute b, the attribute evaluator needs to know the production $p : X_0 \rightarrow X_1 \ldots X_k$ and the position j in the right side of p, to select the correct semantic rule for the attribute occurrence. We use a trick to accomplish this. A new nonterminal $N_{p,j}$ is introduced with the only production $N_{p,j} \rightarrow \epsilon$. This nonterminal $N_{p,j}$ is inserted before the symbol X_j in the right side of p. The nonterminal symbol $N_{p,j}$ is associated with all inherited attributes b of X_j that are noncopying in p. Each attribute b of $N_{p,j}$ is equipped with a semantic rule that computes the same value as the semantic rule for $p[j].b$. Note that the insertion of auxiliary symbols $N_{p,j_1}, \ldots, N_{p,j_r}$ affects the positions of the original symbol occurrences in the right side of production p.

Example 4.4.7 Consider the production $\langle block\rangle \rightarrow \langle decl\rangle\ \langle block\rangle$ of the attribute grammar AG_{scopes} of Example 4.3.4. The attribute occurrences $\langle block\rangle\,[1].same$ and $\langle block\rangle\,[1].env$ on the right side of the production are not copying. Therefore a new nonterminal N is inserted before $\langle block\rangle$:

$$\begin{array}{lcl} \langle block\rangle & \longrightarrow & \langle decl\rangle\ \ N\ \ \langle block\rangle \\ N & \longrightarrow & \epsilon \end{array}$$

The new nonterminal symbol N has inherited attributes $\{same, env\}$. It does not need any synthesized attributes. The new semantic rules for the transformed production:

$$\begin{array}{lcl} N.same & = & \textbf{let}\ (x,\tau) = \langle decl\rangle\,.new \\ & & \textbf{in}\ \ \langle block\rangle\,[0].same \cup \{x\} \\ N.env & = & \textbf{let}\ (x,\tau) = \langle decl\rangle\,.new \\ & & \textbf{in}\ \ \langle block\rangle\,[0].env \oplus \{x \mapsto \tau\} \\ \langle block\rangle\,[1].same & = & N.same \\ \langle block\rangle\,[1].env & = & N.env \\ \langle block\rangle\,[1].ok & = & \textbf{let}\ (x,\tau) = \langle decl\rangle\,.new \\ & & \textbf{in}\ \ \textbf{if}\ x \notin \langle block\rangle\,[0].same \\ & & \qquad \textbf{then}\ \ \langle block\rangle\,[1].ok \\ & & \qquad \textbf{else}\ \mathsf{false} \end{array}$$

Since N has only inherited attributes, it does not need any semantic rules. We observe that the inherited attributes *same* and *env* of the nonterminal $\langle block\rangle$ are both copying after the transformation. □

Insertion of the nonterminal $N_{p,j}$ does not change the accepted language. It may, however, destroy the $LR(k)$-property. In Example 4.4.7 this is not the case. If the underlying context-free grammar is still an $LR(k)$-grammar after the transformation, we call the attribute grammar *LR-attributed*.

After the transformation, the inherited attributes at the new nonterminals $N_{p,j}$ are the only occurrences of inherited attributes that are noncopying. At a *reduce*-transition for $N_{p,j}$, the LR-parser has identified the production p and the position j in the right side of p at which $N_{p,j}$ has been positioned. At reduction the new value for the inherited attribute b therefore can be computed and stored in the global attribute structure $\mathcal{I}$. The states q' which the parser may reach by a transition under nonterminal $N_{p,j}$, are now associated with a dedicated attribute structure $old(q')$ which does not contain the values of synthesized attributes. Instead, the *previous* values of inherited attributes of $\mathcal{I}$ are stored that have been overwritten during the reduction. These previous values are required to reconstruct the original values of the inherited attributes before the descent into the subtree for X.

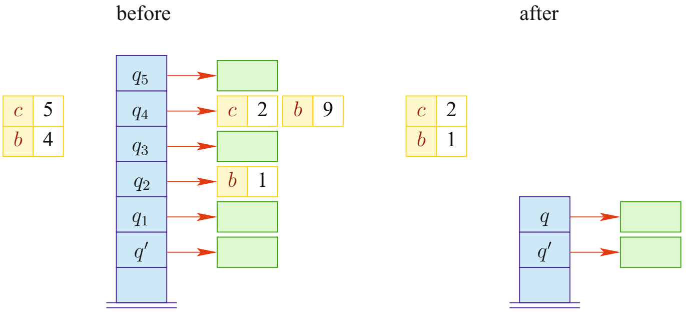


Fig. 4.17 The reconstruction of the inherited attributes at a *reduce*-transition for a production $X \to \gamma$ with $|\gamma| = 5$ and $\Delta(q', X) = q$. The attribute structures $old(q_2)$ and $old(q_4)$ contain the overwritten inherited attributes b and c in $\mathcal{I}$

Let us consider in detail how the value of an inherited attribute b of the nonterminal $N_{p,j}$ can be computed. Let $\bar{p}: X \to \alpha N_{p,j} \beta$ be the production that results from the transformation applied to p, where α has length m. Before the *reduce*-transition for $N_{p,j}$ there is a sequence $q'q_1 \ldots q_m$ on top of the pushdown, where the states in the sequence $q_1, \ldots, q_m$ correspond to the occurrences of symbols in α. The evaluation of the semantic rules for the inherited attribute b of $N_{p,j}$ therefore may access the values of the synthesized attributes of the symbols in α in the attribute structures of the states $q_1, \ldots, q_m$. The values of the inherited attribute a of the left side X, on the other hand, can be found in the global structure $\mathcal{I}$ – given that the attribute a has not been redefined by some $N_{p,i}$ with $i < j$ during the evaluation of the production $\bar{p}$ so far. If, however, that has been the case, the original value of a has been recorded in the structure $old(q_{i'})$ to state $q_{i'}$, which corresponds to the first redefinition of a in the right side of p.

Let us consider in detail what happens at a *reduce*-transition for a transformed production $\bar{p}$. Let $N_{p,j_1}, \ldots, N_{p,j_r}$ be the sequence of new nonterminals that were inserted by the transformation in the right side of the production p, and let m be the length of the transformed right side. Before the *reduce*-transition there is a sequence $q'q_1 \ldots q_m$ of states on top of the pushdown where the states $q_{j_1}, \ldots, q_{j_r+r-1}$ correspond to the nonterminals $N_{p,j_1}, \ldots, N_{p,j_r}$. Using the attribute structures $old(q_{j_1}), \ldots, old(q_{j_r+r-1})$ the values of the inherited attributes before the descent into the parse tree for X are reconstructed. If an attribute b occurs in no structure $old(q_{j_i+i-1})$, $\mathcal{I}$ already contains the correct value of b. Otherwise the value of b is set to the value of b in the first structure $old(q_{j_i+i-1})$ in which b occurs. This reconstruction of the global structure $\mathcal{I}$ for the inherited attributes is shown in Fig. 4.17. Once the former values in the structure $\mathcal{I}$ have been reconstructed, the semantic rules of the synthesized attributes of the left side X can be evaluated, where any required synthesized attribute of the ith symbol occurrence of the right side of $\bar{p}$ can be accessed in the attribute structure of q_i. In this way, the values for the attribute structure $\Delta(q', X)$ can be determined.

The method we have presented enables LR-parsers not only to evaluate synthesized attributes by means of their pushdown, but also to maintain and update inherited attributes – given that the grammar is LR-attributed.

Example 4.4.8 The attribute grammar *BoolExp* of Example 4.3.5 is L-attributed, but neither LL- nor LR-attributed. In general, new ε-nonterminals must be inserted at the beginning of the right side of the left-recursive production for the nonterminal E since the inherited attribute *fsucc* of nonterminal E of the right side is noncopying. Correspondingly, a new ε-nonterminal must be inserted at the beginning of the right side of left-recursive production for the nonterminal T since the inherited attribute *tsucc* of the nonterminal T is noncopying. This leaves the grammar no longer $LR(k)$ for any $k \geq 0$. □

4.5 Exercises

1. **Symbol Tables**
   What are the contents of the symbol table of the body of the procedure q after the decaration of the procedure r in Example 4.1.6?
2. **Overloading**
   Consider the following operators:

$$\begin{array}{ll}
+ & : \mathbf{integer} \longrightarrow \mathbf{integer} \\
+ & : \mathbf{real} \longrightarrow \mathbf{integer} \\
+ & : \mathbf{integer} \times \mathbf{integer} \longrightarrow \mathbf{integer} \\
+ & : \mathbf{real} \times \mathbf{real} \longrightarrow \mathbf{real} \\
/ & : \mathbf{integer} \times \mathbf{integer} \longrightarrow \mathbf{integer} \\
/ & : \mathbf{integer} \times \mathbf{integer} \longrightarrow \mathbf{real} \\
/ & : \mathbf{real} \times \mathbf{real} \longrightarrow \mathbf{real}
\end{array}$$

   Apply the algorithm of Sect. 4.1.3 in order to resolve the overloading in the assignment $A \leftarrow 1/2 + 3/4$ to the *real*-variable A.
3. **Type Inference**
   Apply the rules for type inference in order to infer the type of the following OCAML-expression:

$$\begin{array}{lcl}
\mathbf{let\ rec}\ \mathit{length} & = & \mathbf{fun}\ l \rightarrow \mathbf{match}\ l \\
 & & \quad \mathbf{with} \quad [\,] \quad \rightarrow \quad 0 \\
 & & \quad\quad | \quad x :: xs \quad \rightarrow \quad 1 + \mathit{length}\ xs \\
\mathbf{in}\ \ \mathit{length} & &
\end{array}$$

4. **Type Inference with Overloading**
   Complete the algorithm of Sect. 4.2.4. Add a rule for treating mutually recursive definitions.
5. **Attibute Grammar for Forward Declarations**
   Extend the grammar of Example 4.1.5 to an attribute grammar that checks for each occurrence of an identifier, whether or not it has been declared, and provides types for its applied occurrences. The attribute grammar should also check *forward* declarations of identifiers and verify whether the declared types coincide.
6. **Attribute Grammar for Resolving Overloading**
   Reformulate the algorithm for resolving overloading in ADA from Sect. 4.1.3 by means of an attribute grammar.
7. **Transformation of Grammars**
   Consider the removal of left recursion from Sect. 3.3. Assume that the grammar G to be transformed is equipped with attributes and semantic rules. Transform these rules in such a way into semantic rules for the transformed grammar G' such that the attribute instances at corresponding nodes in the parse trees of G and G' receive identical values.
   (a) For that, equip the new nonterminals $\langle A, B\rangle$ with *inherited* attributes in order to propagate the values of inherited attributes of A to the production $\langle A, A\rangle \to \varepsilon$, and also with *synthesized* attributes in order to computed the values of the inherited attributes of the formerly left recursive occurrences of nonterminals B.
   (b) Also equip the nonterminals $\langle A, B\rangle$ with *inherited* attributes in order to compute the values of the *synthesized* attributes of the formerly left-recursive occurrences of nonterminals B, as well as with *synthesized* attributes in order to propagate the values of the *synthesized* attributes of A from production $\langle A, A\rangle \to \varepsilon$ back to the occurrence of the nonterminal A.
8. **Demand-driven Attribute Evaluation**
   Modify the the demand-driven evaluation of attribute instances of Sect. 4.4.1 so that the evaluation itself observes when the evaluation wants to access the value of an attribute instance that is still under evaluation.
9. **Upper Dependence Graphs**
   Set up a system of equations in order to compute the sets of upper dependence graphs for each terminal and nonterminal symbol. Let $\mathcal{U}(X)$, $X \in V \cup T$, denote the least solution of your system of equations. Show that an attribute grammar is noncircular if and only if for each symbol X, all relations $l \cup u$ for $l \in \mathcal{L}(X)$ and $u \in \mathcal{U}(X)$ are acyclic.

10. **Absolutely Noncircular Attribute Grammars**
Consider the following attribute grammar with nonterminals L, A, B:

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
L & \longrightarrow & A & A & \longrightarrow & sB & B & \longrightarrow & u \\
 & & L.z = A.z & & & B.a = B.y & & & B.x = B.a \\
 & & A.c = 0 & & & B.b = A.c & & & B.y = B.b \\
 & & & & & A.z = B.x & B & \longrightarrow & v \\
 & & & A & \longrightarrow & tB & & & B.x = B.a \\
 & & & & & B.a = A.c & & & B.y = 0 \\
 & & & & & B.b = B.x & & & \\
 & & & & & A.z = B.y & & &
\end{array}$$

where x, y, z denote synthesized and a, b, c inherited attributes.
(a) Determine the dependence relations for the various productions of the grammar.
(b) Construct parse trees with the corresponding dependence relations for the inputs su, sv, tu, and tv.
(c) Determine the induced lower characteristic dependence relations for B.
(d) Determine an evaluation ordering for one of the trees.
(e) Is the given grammar absolutely noncircular if the first B-production is omitted?
If yes, then construct an evaluator according to the methods in Sect. 4.4.3.
(f) Is the given grammar also absolutely noncircular in presence of both B-productions?
If yes, then construct an evaluator according to the methods in Sect. 4.4.3.

11. **Regular Expressions**
Write a CFG for regular expressions; take operator precedences into account. Extend your CFG to an attribute grammar that constructs a finite automaton for the regular expression.

12. **Type Inference**
Implement the algorithm $\mathcal{W}$ by means of an attribute grammar.
(a) First, consider the fragment with the single datatype **int**. For a start also consider monomorphic types only, i.e., not type schemes.
(b) Extend your grammar by productions for the construction of structured datatypes tuples and lists as well as pattern matching. For that, also extend the attribute rules.
(c) Now modify your grammar to infer polymorphic types by introducing type schemes for *let*- and *let-rec*-expressions.

4.6 Bibliographic Notes

The presentation of context conditions partly follows [68]. The datastructures for symbol tables in Sect. 4.1.2 were independently invented by various compiler writers. An early source is [40].

The algorithm for resolving overloading in ADA is inspired by Pennello et al. [56]. Type inference for polymorphically typed languages goes back to the work of Hindley and Milner [24], [47], [10]. The extension to expressions with side effects by means of the value restriction was proposed by Wright [70], an interesting extension, which has been adopted by OCAML, is provided by Garrigue [19]. In a famous POPL paper, Wadler came up with the idea of realizing overloading by means of type classes [66]. Their integration into the programming language HASKELL is described in [21]. Type classes alone are not sufficiently expressive for applications such as monads in HASKELL. Therefore, Mark P. Jones extended the underlying concept to *type constructor classes* [30], which also have made it into the programming language HASKELL. Further possibilities to enhance the expressiveness of the type system are discussed in [31].

The concept of attribute grammars was introduced by Knuth [37], [38]. Jazayeri showed that noncircularity of attribute grammars is *DEXPTIME*-hard [28]. The class of absolutely noncircular attribute grammars together with the static generation of evaluators was proposed by Kennedy and Warren [35]. Ordered attribute grammars were defined by Kastens [34], visit-oriented evaluation of attributes by Riis Nielson [52]. The attribute grammar for short-circuit evaluation of Boolean expressions is taken from [20]. The LR-parser driven evaluation of attributes was developed by Watt [67].

[44] presented an implementation of attribute grammars with demand-driven evaluation and incremental reevaluation after tree transformations. Furthermore, these techniques suggest to use *uninterpreted* attribute values for realizing complex datastructures such as symbol tables. Thereby, semantic rules are taken for term constructors. Attributes thus may have values that are constructed according to the attribute dependences, while other semantic rules may operate on these terms by means of pattern matching.

The following authors give overviews over attribute grammars, classes of grammars, evaluation techniques, and/or implemented systems: Courcelle [8], Engelfriet [17], Alblas [3] as well as Deransart, Jourdan, and Lorho [12]. A more recent presentation (in German) is in [41].

Outside of compiler construction, connections are studied between attribute grammars and query languages for searching XML documents [51]. The application of generating spreadsheet-like tools is considered in [58]. Functional attribute grammars were studied in the context of the programming language HASKELL [65, 7].

References

1. Thomas H Cormen and Charles E Leiserson and Ronald L Rivest and Clifford Stein (2001) Introduction to Algorithms (second edition). MIT Press and McGraw-Hill
2. Aho AV, Sethi R, Ullman JD (1986) Principles of Compiler Design. Addison Wesley
3. Alblas H (1991) Attribute evaluation methods. In: Henk Alblas BM (ed) Proc. International Summer School on Attribute Grammars, Applications and Systems, Springer, LNCS 545
4. Ammann U (1978) Error recovery in recursive descent parsers and run–time storage organization, rep. No. 25, Inst. für Informatik der ETH Zürich
5. Baars AI, Swierstra SD, Viera M (2010) Typed transformations of typed grammars: The left corner transform. Electr Notes Theor Comput Sci 253(7):51–64
6. Blum N (2010) On *lr*(*k*)-parsers of polynomial size. In: Abramsky S, Gavoille C, Kirchner C, Meyer auf der Heide F, Spirakis PG (eds) ICALP (2), Springer, Lecture Notes in Computer Science, vol 6199, pp 163–174
7. Bransen J, Middelkoop A, Dijkstra A, Swierstra SD (2012) The kennedy-warren algorithm revisited: Ordering attribute grammars. In: Russo CV, Zhou NF (eds) PADL, Springer, Lecture Notes in Computer Science, vol 7149, pp 183–197
8. Courcelle B (1984) Attribute grammars: Definitions, analysis of dependencies. In: [45]
9. Courcelle B (1986) Equivalences and transformations of regular systems—applications to program schemes and grammars. Theoretical Computer Science 42:1–122
10. Damas L, Milner R (1982) Principal type schemes for functional programms. In: 9th ACM Symp. on Principles of Programming Languages, pp 207–212
11. Dencker P, Dürre K, Heuft J (1984) Optimization of parser tables for portable compilers. ACM Transactions on Programming Languages and Systems 6(4):546–572
12. Deransart P, Jourdan M, Lorho B (1988) Attribute Grammars, Definitions, Systems and Bibliography. Springer, LNCS 323
13. DeRemer F (1969) Practical translators for *LR*(*k*) languages. PhD thesis, Massachusetts Institute of Technology
14. DeRemer F (1971) Simple *LR*(*k*) grammars. Communications of the ACM 14:453–460
15. DeRemer F (1974) Lexical analysis. In: F.L. Bauer, J. Eickel (Hrsg.), Compiler Construction, An Advanced Course, Springer, LNCS 21
16. Dijkstra EW (1961) Algol-60 translation. Tech. rep., Stichting, Mathematisch Centrum, Amsterdam, rekenafdeling, MR 35. Algol Bulletin, supplement nr. 10

R. Wilhelm, H. Seidl, S. Hack, *Compiler Design*, DOI 10.1007/978-3-642-17540-4,

17. Engelfriet J (1984) Attribute grammars: Attribute evaluation methods. In: [45]
18. Floyd RW (1963) Syntactic analysis and operator precedence. J ACM 10(3):316–333
19. Garrigue J (2004) Relaxing the value restriction. In: Kameyama Y, Stuckey PJ (eds) Proc. of Functional and Logic Programming, 7th International Symposium, FLOPS 2004, Nara, Japan, April 7–9, 2004, Springer, LNCS 2998, pp 196–213
20. Giegerich R, Wilhelm R (1978) Counter–one–pass features in one–pass compilation: a formalization using attribute grammars. Information Processing Letters 7(6):279–284
21. Hall CV, Hammond K, Jones SLP, Wadler P (1994) Type classes in HASKELL. In: Sannella D (ed) ESOP, Springer, LNCS 788, pp 241–256
22. Harrison MA (1983) Introduction to Formal Language Theory. Addison Wesley
23. Heckmann R (1986) An efficient $ell(1)$-parser generator. Acta Informatica 23:127–148
24. Hindley JR (1969) The principal type scheme of an object in combinatory logic. Transactions of the AMS 146:29–60
25. Hopcroft J, Ullman JD (1979) Introduction to Automata Theory, Languages and Computation. Addison-Wesley
26. II PML, Stearns RE (1966) Syntax directed transduction. In: IEEE 7. Annual Symposium on Switching and Automata Theory, pp 21–35
27. II PML, Stearns RE (1968) Syntax directed transduction. Journal of the ACM 15:464–488
28. Jazayeri M, Ogden WF, Rounds WC (1975) The intrinsically exponential complexity of the circularity problem for attribute grammars. Communications of the ACM 18(12):697–706
29. Johnson WL, Porter JH, Ackley SI, Ross DT (1968) Automatic generation of efficient lexical analyzers using finite state techniques. Communications of the ACM 11(12):805–813
30. Jones MP (1995) A system of constructor classes: Overloading and implicit higher-order polymorphism. J Funct Program 5(1):1–35
31. Jones SP, Jones MP, Meijer E (1997) HASKELL type classes: an exploration of the design space. In: Proceedings of the 2nd HASKELL Workshop
32. Jourdan JH, Pottier F, Leroy X (2012) Validating lr(1) parsers. In: Seidl H (ed) ESOP, Springer, Lecture Notes in Computer Science, vol 7211, pp 397–416
33. Kannapinn S (2001) Eine rekonstruktion der *LR*-theorie zur elimination von redundanz mit anwendung auf den bau von *ELR*-parsern. PhD thesis, Fachbereich 13 – Informatik
34. Kastens U (1980) Ordered attribute grammars. Acta Informatica 13(3):229–256
35. Kennedy K, Warren SK (1976) Automatic generation of efficient evaluators for attribute grammars. In: Proc. 3rd ACM Symp. on Principles of Programming Languages, pp 32–49
36. Knuth DE (1965) On the translation of languages from left to right. Information and Control 8:607–639
37. Knuth DE (1968) Semantics of context-free languages. Math Systems Theory 2:127–145
38. Knuth DE (1971) Semantics of context–free languages, correction in Math. Systems Theory **5**, pp. 95-96
39. Knuth DE (1977) A generalization of dijkstra's algorithm. Information Processing Letters 6(1):1–5
40. Krieg B (1971) Formal definition of the block concept and some implementation models, mS. Thesis, Cornell University
41. Kühnemann A, Vogler H (1997) Attributgrammatiken. Eine grundlegende Einführung. Vieweg+Teubner

42. Lesk M (1975) Lex – a lexical analyzer generator, cSTR 39, Bell Laboratories, Murray Hill, N.J.

43. Lewi J, DeVlaminck K, Huens J, Steegmans E (1982) A Programming Methodology in Compiler Construction, part 2. North Holland

44. Lipps P, Olk M, Möncke U, Wilhelm R (1988) Attribute (re)evaluation in the optran system. Acta Informatica 26:213–239

45. Lorho B (ed) (1984) Methods and Tools for Compiler Construction. Cambridge University Press

46. Mayer O (1986) Syntaxanalyse, 3. Aufl. Bibliographisches Institut

47. Milner R (1978) A theory of type polymorphism in programming. Journal of Computer and System Sciences 17:348–375

48. Möncke U (1985) Generierung von systemen zur transformation attributierter operatorbäume; komponenten des systems und mechanismen der generierung. PhD thesis, Informatik

49. Möncke U, Wilhelm R (1982) Iterative algorithms on grammar graphs. In: Proc. 8th Conference on Graphtheoretic Concepts in Computer Science, Hanser, pp 177–194

50. Möncke U, Wilhelm R (1991) Grammar flow analysis. In: H. Alblas, B. Melichar (Hrsg.), Attribute Grammars, Applications and Systems, Springer, LNCS 545

51. Neven F, den Bussche JV (1998) Expressiveness of structured document query languages based on attribute grammars. In: Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1–3, 1998, Seattle, Washington, ACM Press, pp 11–17

52. Nielson HR (1983) Computation sequences: A way to characterize classes of attribute grammars. Acta Informatica 19:255–268

53. Pager D (1977) Eliminating unit productions from lr parsers. Acta Inf 9:31–59

54. Pager D (1977) The lane-tracing algorithm for constructing *LR*(*k*) parsers and ways of enhancing its efficiency. Inf Sci 12(1):19–42

55. Pennello TJ, DeRemer F (1978) A forward move for *LR* error recovery. In: Proc. 5th ACM Symp. on Principles of Programming Languages, pp 241–254

56. Pennello TJ, DeRemer F, Myers R (1980) A simplified operator identification scheme for ADA. ACM SIGPLAN Notices 15(7,8):82–87

57. Pratt VR (1973) Top down operator precedence. In: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp 41–51

58. Saraiva J, Swierstra SD (2003) Generating spreadsheet-like tools from strong attribute grammars. In: Pfenning F, Smaragdakis Y (eds) GPCE, Springer, Lecture Notes in Computer Science, vol 2830, pp 307–323

59. Sippu S, Soisalon-Soininen E (1990) Parsing Theory. Vol.1: Languages and Parsing. Springer

60. Sippu S, Soisalon-Soininen E (1990) Parsing Theory. Vol.2: *LR*(*k*) and *LL*(*k*) Parsing. Springer

61. Tarjan RE, Yao ACC (1979) Storing a sparse table. Communications of the ACM 22(11)

62. Tomita M (1984) *LR* parsers for natural languages. In: 10th International Conference on Computational Linguistics (COLING), pp 354–357

63. Tomita M (1985) An efficient context-free parsing algorithm for natural languages. In: International Joint Conference on Artificial Intelligence (IJCAI), pp 756–764

64. Van De Vanter ML (1975) A formalization and correctness proof of the cgol language system (master's thesis). Tech. Rep. MIT-LCS-TR-147, MIT Laboratory for Computer Science. Cambridge, MA

65. Viera M, Swierstra SD, Swierstra W (2009) Attribute grammars fly first-class: how to do aspect oriented programming in haskell. In: Hutton G, Tolmach AP (eds) ICFP, ACM, pp 245–256

66. Wadler P, Blott S (1989) How to make ad-hoc polymorphism less ad-hoc. In: POPL, pp 60–76

67. Watt DA (1977) The parsing problem for affix grammars. Acta Informatica 8:1–20

68. Watt DA (1984) Contextual constraints. In: [45]

69. Wirth N (1978) Algorithms + Data Structures = Programs, Chapter 5. Prentice Hall

70. Wright AK (1995) Simple imperative polymorphism. Lisp and Symbolic Computation 8(4):343–355

Index