Reinhard Wilhelm, Helmut Seidl, Sebastian Hack

Compiler Design

Syntactic and Semantic Analysis

November 11, 2011

Springer

Contents

1 The Structure of Compilers 1
1.1 Subtasks of compilation 2
1.2 Lexical Analysis 2
1.3 The Screener 3
1.4 Syntactic Analysis 4
1.5 Semantic Analysis 5
1.6 Machine-Independent Optimization 5
1.7 Memory Allocation 6
1.8 Generation of the Target Program 6
1.9 Specification and Generation of Compiler Components 8
1.10 Literature 9

2 Lexical Analysis 11
2.1 The Task of Lexical Analysis 11
2.2 Regular Expressions and Finite-State Machines 12
2.2.1 Words and Languages 12
2.3 Language for the Specification of Lexical Analyzers 23
2.3.1 Character classes 23
2.3.2 Non-recursive Parentheses 24
2.4 Scanner Generation 24
2.4.1 Character Classes 24
2.4.2 An Implementation of the *until*-Construct 25
2.4.3 Sequences of regular expressions 26
2.4.4 The Implementation of a Scanner 28
2.5 The Screener 29
2.5.1 Scanner States 30
2.5.2 Recognizing Reserved Words 31
2.6 Exercises 32
2.7 Literature 34

3 Syntactic Analysis 35
3.1 The Task of Syntactic Analysis 35
3.2 Foundations 37
3.2.1 Context-free Grammars 37
3.2.2 Productivity and Reachability of Nonterminals 42
3.2.3 Pushdown Automata 45
3.2.4 The Item-Pushdown Automaton to a Context-Free Grammar 47
3.2.5 first- and follow-Sets 51
3.2.6 The Special Case first_1 and follow_1 56
3.2.7 Pure Union Problems 58

3.3 *Top-down*-Syntax Analysis 60
3.3.1 Introduction 60
3.3.2 $LL(k)$: Definition, Examples, and Properties 62
3.3.3 Left Recursion 66
3.3.4 Strong $LL(k)$ Parsers 69
3.3.5 LL Parsers for Right-regular Context-free Grammars 71
3.4 Bottom-up Syntax Analysis 79
3.4.1 Introduction 79
3.4.2 $LR(k)$ Parsers 80
3.4.3 $LR(k)$: Definition, Properties, and Examples 89
3.4.4 Fehlerbehandlung in LR parsern 98
3.5 Literaturhinweise 104
3.6 "Ubungen 104

4 Semantic Analysis 109
4.1 Aufgabe der semantischen Analyse 109
4.1.1 G"ultigkeits- und Sichtbarkeitsregeln 113
4.1.2 "Uberpr"ufung der Kontextbedingungen 117
4.1.3 "Uberladung von Bezeichnern 121
4.2 Typinferenz 124
4.3 Attributgrammatiken 143
4.3.1 Die Semantik einer Attributgrammatik 146
4.3.2 Einige Attributgrammatiken 147
4.4 Die Generierung von Attributauswertern 153
4.4.1 Bedarfsgetriebene Auswertung der Attribute 153
4.4.2 Statische Vorberechnungen für Attributauswerter 154
4.4.3 Besuchsgesteuerte Attributauswertung 160
4.4.4 Parsergesteuerte Attributauswertung 164
4.5 "Ubungen 170
4.6 Literaturhinweise 171

Literatur 173

References 173

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 special about compilers such that they can be even implemented as a project accompanying a compiler course? A decomposition of the task into subtasks with clearly defined functionalities and clean interfaces between them makes this, in fact, possible. This is true about compilers; there is a more or less standard conceptual compiler structure composed of components solving a well-defined subtask of the compilation task. The interfaces between the components are representations of the input program.

The compiler structure described in the following is a *conceptual* structure. i.e. it identifies the subtasks of the translation of 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. But a component may also be split into several components if the realized subtask is very complex.

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

1. The *analysis phase*, realized by the *Frontend*. 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 what is often called the *Middleend*. The syntactically analysed and semantically checks 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 *Backend*. The program is being translated into an equivalent program in the target language. Machine-dependent optimizations might be performed, which exploit peculiarities of the target architecture.

This coarse compiler structure splits it 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 structure helps to adapt compiler components to new source languages and 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 Subtasks of compilation

Fig. 1.1 shows a conceptual compiler structure. 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 name of the subtask and maybe with the name of the module realizing this subtask.

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:

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

where $'='$ denotes the assignment operator.

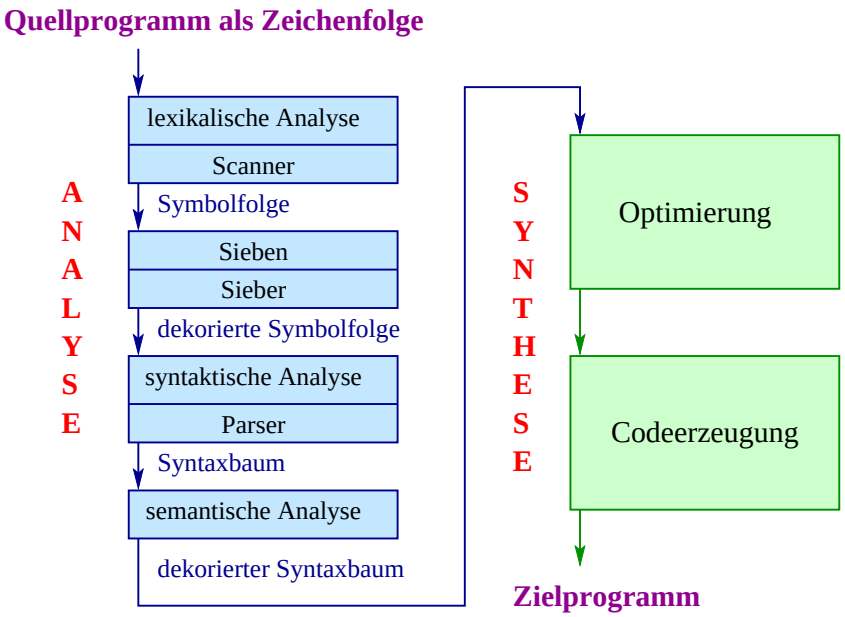

Fig. 1.1. Structure of a compiler together with the program representations during the analysis phase.

1.2 Lexical Analysis

The component performing lexical analysis of source programs is often called the *scanner*. This componen reads the source program represented 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*. Typial lexical units are keywords such as **if**, **else**, **while** or **switch** and special charactes 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 identifier and constants. Examples for identifiers are value42, abc,

Myclass, x, while the character sequences 42, 3.14159 and ″HalloWorld!″ represent constants. Something special to note is that there are, in principle, arbitrarily many such symbols. However, they can be categorized into finitely many *classes*. A symbol class consists of symbols that are equivalent as far as the syntactic structure of programs is concerned. Identifiers are 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 and need, therefore, be considered in code generation. However, there are symbols without semantics. Two symbols need a separator between them if their concatenation would also form a symbol. Such a separator can be a blank, a newline, or an indentation or a sequence of such characters. Such so-called white space can also be inserted into a program to make visible the structure of the program.

Another type of symbols, without meaning for the compiler, but helpful for the human reader, are comments and can be used by software development tools. A similar type of symbols are *compiler directives* (*pragmas*). Such directives 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 might look as follows:

$\mathsf{Int}(''int'')\ \mathsf{Sep}(''\ '')\ \mathsf{Id}(''a'')\ \mathsf{Com}(''{,}'')\ \mathsf{Sep}(''\ '')\ \mathsf{Id}(''b'')\ \mathsf{Sem}(''{;}'')\ \mathsf{Sep}(''\backslash n'')$
$\mathsf{Id}(''a'')\ \mathsf{Bec}(''{=}'')\ \mathsf{Intconst}(''42'')\ \mathsf{Sem}(''{;}'')\ \mathsf{Sep}(''\backslash n'')$
$\mathsf{Id}(''b'')\ \mathsf{Bec}(''{=}'')\ \mathsf{Id}(''a'')\ \mathsf{Mop}(''{*}'')\ \mathsf{Id}(''a'')\ \mathsf{Aop}(''{-}'')\ \mathsf{Intconst}(''7'')\ \mathsf{Sem}(''{;}'')\ \mathsf{Sep}(''\backslash n'')$

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

1.3 The Screener

The scanner delivers a sequence of symbols to the screener. These are substrings of the program text labeled with their symbol classes. It is the task of the screener to further process this sequence. Some symbols it will eliminate since they have served their prupose as separators. Others it will transform into a different representation. More precisely, it will perform the following actions, specific for different symbol classes:

Reserved symbols: These are typically identifiers, but have a special meaning in the programming language. e.g. begin, end, var, int etc.

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 some functional languages, e.g. HASKELL, where indentation is used to express program nesting. Comments will also not be needed later and can be removed.

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

Other types of symbols are typically preserved, but their textual representation may be converted into some more efficient internal representation.

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.

Identifier: 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.

3

Syntactic Analysis

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 cmbination of scanner and screener. It is its job to identify the syntactic structure in this sequence of symbols, that is the composition of syntactic units from other units.

Syntactic units in imperative languages are variables, expressions, declarations, statements and sequences of statements. Functional languages have variables, expressions, patterns, definitions and declarations. Logic languages such as [sc 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 *parse tree*. The parse tree maz later be decorated with more information about the program. Transformation may be applied to it, and 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 main function calling routines for semantic analysis and for code generation.

Many programs that are presented to a compiler contain errors, many of them syntax errors. Syntax errors consist in violations of the rules for forming valid programs. The compiler is expected to adequately react to errors. It should at least attempt to locate the error precisely. However, often only the localization of the error symptom is possible, not the localization of the error itself. The error symptom is the position where no continuation of the syntactic analysis is possible. The compiler should not give up after the first error found, but continue to analyze the rest of the program and maybe detect more errors.

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 unambiguity reasons, parsing methods are often restricted to deterministically analyzable context-free languages. For these, several automatic methods for parser generation exist. 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 be best 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 of the parse tree labelled with that symbol. Top-down parser are called *predictive* parser 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 non-confirmed 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 non-confirmed 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 it means that another symbol of the prediction is confirmed. Otherwise, the parser has detected an error.

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. To do this 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 if does a *shift*, that is, it reads the next input symbol. These are the only two actions a bottom-up parser can perform. It is, therefore, 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 difficult to diagnose. This chapter covers required and possible reactions to syntax errors by the parser. There are essentially four different types of reaction to syntax errors:

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 alternative is absolutely required. Later stages of the compiler assume that they are only given syntactically correct programs in the form of syntax trees. The programmer needs to be informed about syntax errors in his programs. There exist, however, two significant problems: Firstly, further syntax errors can remain undetected in the vicinity of a detected error. Second, the parser detects an error when it has no continuation out of its actual configuration under the next input symbol. This is, in general, only the error *symptom*, not the error itself.

Example 3.1.1 Consider the following erroneous assignment statement:

$$a = a * (b + c * d \quad ;$$
$$\uparrow$$
error symptom: $')'$ is missing

There are several potential errors: Either there is an extra open parenthesis, or a closing parenthesis is missing after c or after d. These three corrections lead to programs 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. The practically relevant parsing methods, $LL(k)$- and $LR(k)$ parsing, presented in the following sections, have the *viable-prefix* property.

When the parser for a context-free grammar G has analyzed the prefix u of a word without announcing an error then there exists a word w such that uw is a word of G.

Parsers possessing this property report error and error symptoms at the earliest possible time. We have said above that, in general, the parser will only discover an error symptom, not the error itself. 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 easily done.

The parser can attempt a diagnosis of the error symptom. It should at least provide the following information:

- the psition 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 know the intention of the programmer. This is, in general, impossible. Somewhat more realistic is the search for a globally optimal error correction. To realize this, 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 necessary effort.

Instead, most parsers do 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 and their notion of derivation and pushdown automata, their acceptors. A special non-deterministic 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. The corresponding grammar classes are characterized and parser-generation methods are presented. Error handling for both top-down and bottom-up parsers is described in detail.

3.2 Foundations

We have seen that lexical analysis is specified by regular expressions and implemented by finite-state machines. We will now see that syntax analysis is specified by context-free grammars and implemented by pushdown automata.

Regular expressions are not sufficient to describe the syntax of programming languages since they cannot *embedded recursion* as they occur in the nesting of expressions, statements, and blocks.

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

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 describes what the elementary components of programs are and how pieces of programs can be composed to form bigger pieces.

Example 3.2.1 A section of a grammar to describe a C-like programming language might look like follows:

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

The nonterminal symbol $\langle stat \rangle$ generates statements. We will use the meta-character | to combine several alternatives for one nonterminal.

According to this section of a grammar, a statement is either and *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.

if-statements in which the *else*-part may be missing. They 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 some possible alternatives are explicitly given. Other alternatives are indicated by .... □

Formally, a *context-free grammar* 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 programs are built. While we spoke of alphabets of *characters* in the section on lexical analysis, typically ASCII or Uni-code characters, we now speak of alphabets of *symbols* as they are returned from the scanner or the screener. Such symbols are reserved keywords of the language, identifiers, or symbol classes comprising sets of symbols.

The nonterminals of the grammar stand for sets of words that can be generated from them according to the production rules of the grammar. In the example grammar 3.2.1, they are enclosed in angle brackets. A production rules (in short: production) (A, α) in the relations P describes possible replacements: 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 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 conventions to talk about context-free grammars $G = (V_N, V_T, P, S)$. 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:

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

□

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

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

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

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

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

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

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

A word $x \in L(G)$ is called a *word* of G. A word $\alpha \in (V_T \cup V_N)^*$ where $S \overset{*}{\underset{G}{\Longrightarrow}} \alpha$ 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 $E \overset{*}{\underset{G_0}{\Longrightarrow}} \mathsf{Id} * \mathsf{Id} + \mathsf{Id}$ as well as $E \overset{*}{\underset{G_1}{\Longrightarrow}} \mathsf{Id} * \mathsf{Id} + \mathsf{Id}$ hold. □

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

The syntactic structure of a program, as it results from syntactic analysis, is the *parse tree*, which is an *ordered tree*, that is, a tree in which the outgoing edges of each node are ordered. The parse tree describes a set of derivations of the program according to the underlying grammar. It, therefore, allows

to define the notion *ambiguity* and to explain the differences between parsing strategies, see Sections 3.3 and 3.4. 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 Chapter 4, about semantic analysis, work on this tree structure.

Let $G = (V_N, V_T, P, S)$ be a context-free grammar. Let t be an ordered tree whose inner nodes are labeled with symbols from V_N and whose leaves are labeled with symbols from $V_T \cup \{\varepsilon\}$. t is a *parse tree* if the label X 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. $X \to X_1 \ldots X_k$ is a production from P.
2. Is $X_1 \ldots X_k = \varepsilon$, then $k = 1$, that is, node n has exactly one child and this child is labeled with ε.
3. Is $X_1 \ldots X_k \neq \varepsilon$ then $X_i \neq \varepsilon$ 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 Fig. 3.1 shows two parse trees according to grammar G_1 of Example 3.2.2 for the word $\mathsf{Id} * \mathsf{Id} + \mathsf{Id}$. □

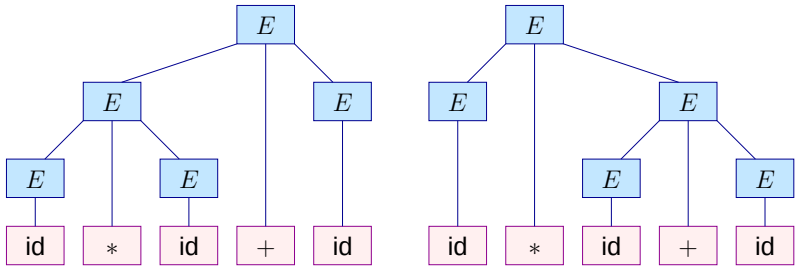

Fig. 3.1. Two syntax trees according to grammar G_1 of Example 3.2.2 for the word $\mathsf{Id} * \mathsf{Id} + \mathsf{Id}$.

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 were 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 context-free grammar that is not ambiguous is called *non-ambiguous*.

Example 3.2.6 The grammar G_1 is ambiguous because the word $\mathsf{Id} * \mathsf{Id} + \mathsf{Id}$ has more than one parse tree. The grammar G_0, on the other hand, is non-ambiguous. □

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 $\mathsf{Id} + \mathsf{Id}$ 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:

$$E \Rightarrow E + E \Rightarrow \mathsf{Id} + E \Rightarrow \mathsf{Id} + \mathsf{Id}$$
$$E \Rightarrow E + E \Rightarrow E + \mathsf{Id} \Rightarrow \mathsf{Id} + \mathsf{Id}$$

□

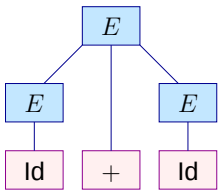

Fig. 3.2. The uniquely determined parse tree for the word Id + Id.

In Example 3.2.7 we saw that—even with non-ambiguous words— several derivations might correspond to one parse tree. This results from the different possibilities to chose a nonterminal in a sentential form for the next application of a production. One can chose essentially two different canonical replacement strategies, replacing the leftmost nonterminal or the rightmost nonterminal. In each case one obtains uniquely determined derivations, namely *leftmost* and *rightmost* derivations, resp.

A derivation $\varphi_1 \Longrightarrow \ldots \Longrightarrow \varphi_n$ of $\varphi = \varphi_n$ from $S = \varphi_1$ is a *leftmost derivation* of φ, denoted as $S \overset{*}{\underset{lm}{\Longrightarrow}} \varphi$, 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, we call a derivation $\varphi_1 \Longrightarrow \ldots \Longrightarrow \varphi_n$ a *rightmost derivation* of φ, denoted by $S \overset{*}{\underset{rm}{\Longrightarrow}} \varphi$, if the rightmost nonterminal in φ_i is replaced, i.e. $\varphi_i = \sigma Au$, $\varphi_{i+1} = \sigma\alpha u$ 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

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

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 difference is the order of application. 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 3.1. 1. If $S \underset{lm}{\overset{*}{\Longrightarrow}} uA\varphi$ 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 $S \underset{rm}{\overset{*}{\Longrightarrow}} \psi Av$ holds, then there exists a φ with $\varphi \overset{*}{\Longrightarrow} v$, such that for all u with $\psi \overset{*}{\Longrightarrow} u$ holds $S \underset{lm}{\overset{*}{\Longrightarrow}} uA\varphi$. □

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

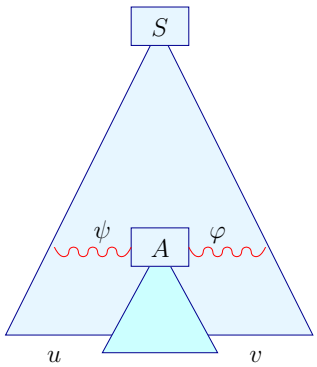

Fig. 3.3. Correspondence between leftmost and rightmost derivation.

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

3.2.2 Productivity and Reachability of Nonterminals

A context-free grammar might have superfluous nonterminals and productions. Eliminating them reduces the size of the grammar, but doesn't change the language. We will now introduce two properties of nonterminals that characterize them as useful and present methods to compute the subsets of nonterminals that have these properties. Grammars from which all nonterminals not having these properties are removed will be called *reduced*. We will later always assume that the grammars we deal with are reduced.

The first required property of useful nonterminals is *productivity*. A nonterminal X of a context-free grammar $G = (V_N, V_T, P, S)$ is called *productive*, if there exists a derivation $X \underset{G}{\overset{*}{\Longrightarrow}} w$ 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 :

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

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 on occurrence of Z on its right side. □

A two-level characterization of nonterminal productivity leading to an algorithm to compute it is the following:

(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 $X \to u \in P$ 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 context-free grammar G the set of all productive nonterminals. The method uses 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 manage the productions whose counter has sunk to 0 the algorithm uses a *worklist* W.

Further, for each nonterminal X a list $\mathsf{occ}[X]$ of occurrences of this nonterminal in right sides of productions is managed:

```
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 still $\mathsf{count}[p] = 0$ 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 once into the list W and so removes it at most once from the list. At the removal of a nonterminals 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 j-th 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 entered into W once.

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 context-free grammar $G = (V_N, V_T, P, S)$, 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:

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

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

Reachability can also be characterized in a two-level definition that leads to an algorithm:

(1) If a nonterminal X is reachable and $X \to \alpha \in P$, 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:

$$\begin{array}{l} \mathbf{set}\langle nonterminal\rangle \;\; reachable \leftarrow \emptyset; \\ \mathbf{list}\langle nonterminal\rangle \;\; W \leftarrow S :: [\,]; \\ nonterminal\; Y; \\ \mathbf{while}\; (W \neq [\,])\; \{ \\ \quad X \leftarrow \mathsf{hd}(W);\;\; W \leftarrow \mathsf{tl}(W); \\ \quad \mathbf{if}\; (X \notin reachable)\; \{ \\ \quad\quad reachable \leftarrow reachable \cup \{X\}; \\ \quad\quad \mathbf{forall}\; (Y \in \mathsf{rhs}[X])\; W \leftarrow W \cup \{Y\}; \\ \quad\quad \} \end{array}$$

To reduce a grammar G, first all non-productive nonterminals are removed from the grammar together with all productions in which they occur. Only in a second step are the non-reachable nonterminals eliminated, also 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

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

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:

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow Y \\ X &\rightarrow bS \mid aYbY \\ Y &\rightarrow ba \end{aligned}$$

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

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow Y \\ Y &\rightarrow ba \end{aligned}$$

□

We assume n the following that grammars are always reduced.

3.2.3 Pushdown Automata

This section treats the automata model corresponding to context-free grammars, pushdown automata. We need to describe how to realize a compiler component that performs syntax analysis according to a given context-free grammar. Section 3.2.4 describes such a method. The pushdown automaton constructed for a context-free grammar, however, has a problem: it is non-deterministic for most grammars. In Sections 3.3 and 3.4 we describe how for appropriate subclasses of context-free grammars the thus constructed pushdown automaton can be modified to become deterministic.

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

Formally, a *pushdown automaton* 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^*.

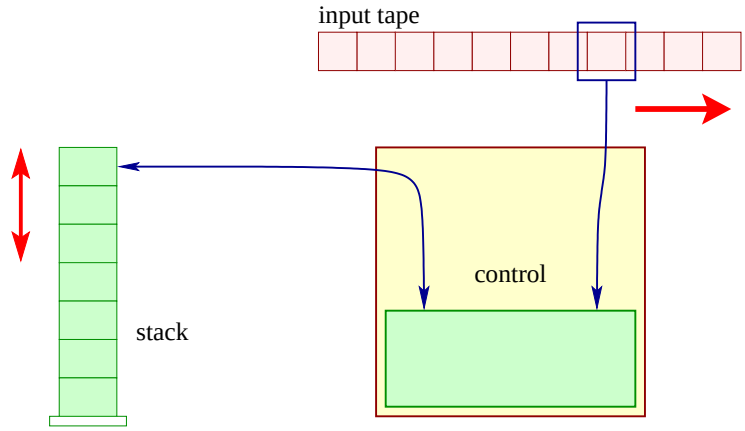

Fig. 3.4. Schematic representation of a pushdown automaton

Our definition of a pushdown automaton is somewhat unusual as it doesn't make a distinction between the states of the automaton and its stack symbols. It uses the same alphabet for both. In this way, the topmost stack symbol is interpreted as the *actual state*. The transition relation describes the possible computation steps of the pushdown automaton. It lists finitely many transitions. Executing the transition (γ, x, γ') replaces the upper section $\gamma \in Q^+$ of the stack contents 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 stack contents has at least the length 1. A transition that doesn't inspect the next input symbol is called an *ε-transition*.

Similarly as for finite-state machines, we introduce the notion of a *configuration* for pushdown automata. A configuration encompasses all components that may influence the future behavior of the automaton. With our kind of pushdown automata these are the stack contents and the remaining input. Formally, a *configuration* of the pushdown automaton P is a pair $(\gamma, w) \in Q^+ \times V_T^*$. In the linear representation the topmost position of the stack 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 $\vdash_P$ between configurations. This relation is defined by:

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

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

$$\vdash_P^+ \ = \bigcup_{n \geq 1} \vdash_P^n \quad \text{and} \quad \vdash_P^* \ = \bigcup_{n \geq 0} \vdash_P^n$$

A configuration (q_0, w) for a $w \in V_T^*$ is called an *initial configuration*, (q, ε), for $q \in F$, a *final configuration* of the pushdown automaton P. A word $w \in V_T^*$ is *accepted* by a pushdown automaton P if $(q_0, w) \vdash_P^* (q, \varepsilon)$ holds for a $q \in F$. The *language* $L(P)$ of the pushdown automaton P is the set of words accepted by P:

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

This means, a word w is accepted by a pushdown automaton if there exists at least one computation that goes from an initial configuration (q_0, w) to a final configuration. Such computations are called *accepting*. Several accepting computations may exist for one word, but also several computations that can only read a prefix of a word w or that can read w, but don't reach a final configuration.

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

A pushdown automaton 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 context-free grammar a pushdown automaton that accepts the language defined by the grammar. This automaton is non-deterministic and therefore not overly useful for a practical application. However, we can derive the LL-parsers of Section 3.3, as well as the LR-parsers of Section 3.4 by appropriate design decisions.

The notion of context-free *item* plays a decisive role. Let $G = (V_N, V_T, P, S)$ be a context-free grammar. 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. Is ρ the sequence of items

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

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

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

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

Before we show how to construct the item-pushdown automaton to a grammar, we want to extend the grammar G is such a way that termination of the pushdown automaton can be recognized by looking at the actual state. Is S the start symbol of the grammar, candidates for final states of the item-pushdown automaton 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 stack but still the automaton need not terminate since below it there may be incomplete items. 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 item-pushdown automaton for the extended grammar we chose the item $[S' \to .S]$ and as single final state the complete item $[S' \to S.]$. The *item-pushdown automaton* to the grammar G is the pushdown automaton

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

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 stack contents in the computation of an item-pushdown automaton satisfies the following invariant (I):

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

This invariant is an essential part of the proof that the item-pushdown automaton 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 firstly 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 non-deterministically 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 next case, we assume that the last transition was a shifting transition. Before this transition, a configuration $(\rho[X \to \beta.a\gamma], av)$ was reached from the initial configuration $([S' \to .S], uav)$. This configuration 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 transitions, a configuration $(\rho[X \to \beta.Y\gamma][Y \to \alpha.], v)$ was reached from the initial configuration $([S' \to .S], uv)$. This configuration satisfies the invariant (I) according to the induction hypothesis, that is, $\mathsf{hist}(\rho)\beta\alpha \underset{G}{\overset{*}{\Longrightarrow}} u$ 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 was 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 $\mathsf{hist}(\rho)\beta Y \underset{G}{\overset{*}{\Longrightarrow}} u$. □

Taken together, the following theorem holds:

Theorem 3.2.1 For each context-free grammar 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 $\rho \in \mathsf{lt}_G^*$ 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

$$\begin{aligned} S &\rightarrow E \\ E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \mathsf{Id} \end{aligned}$$

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 $\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$. □

Pushdown Automata with Output

Pushdown automata as such are only acceptors, that is, they decide whether or not an input string is a word of the language. To use a pushdown automaton for the syntactic analysis in a compiler needs more than a yes/no answer. The automaton should output the syntactic structure of accepted input words. This can have one of several forms, a parse tree or the sequence of productions as they were applied in a leftmost or rightmost derivation. We, therefore, extend pushdown automata by a means to produce output.

A pushdown automaton *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 a normal pushdown automaton and O is a finite output alphabet. Δ is a finite relation between $Q^+ \times (V_T \cup \{\varepsilon\})$ and $Q^* \times (O \cup \{\varepsilon\})$. A *configuration* consists of the actual stack content, 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. If a pushdown automaton with output is used as a parser its output alphabet consists of the productions of the context-free grammar or their numbers.

The item-pushdown automaton can be extended by a means to produce output 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 pushdown automaton with this output discipline is called a *left-parser*.

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

Deterministic Parsers

In Theorem 3.2.1 we proved that the item-pushdown automaton P_G to a context-free grammar G accepts the grammar's language $L(G)$. However, the non-deterministic way of working of the pushdown automaton is unsuitable for practice. The source of non-determinism lies in the transitions of type (E): the item-pushdown automaton can choose between several alternatives for a nonterminal at expanding transitions, With a non-ambiguous grammar at most one is the correct choice to derive a prefix of the remaining input. The other alternatives lead sooner or later into dead ends. The item-pushdown automaton can only *guess* the right alternative.

In Sections 3.3 and 3.4, we describe two different ways to replace guessing. The LL-parsers of Section 3.3 deterministically choose one alternative for the actual nonterminal using a bounded lookahead 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. All the time during the analysis they pursue all possible derivations 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, by which production. Basis for this decision is again the actual stack contents and

top of the stack	input	new top of the stack
$[S \to .E]$	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 3.1. Tabular representation of the transition relation of Example 3.2.12. The middle column shows the consumed input.

stack contents	remaining input
$[S \to .E]$	$\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$
$[S \to .E][E \to .E + T]$	$\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$
$[S \to .E][E \to .E + T][E \to .T]$	$\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$
$[S \to .E][E \to .E + T][E \to .T][T \to .F]$	$\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$
$[S \to .E][E \to .E + T][E \to .T][T \to .F][F \to .\mathsf{Id}]$	$\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$
$[S \to .E][E \to .E + T][E \to .T][T \to .F][F \to \mathsf{Id}.]$	$+\mathsf{Id} * \mathsf{Id}$
$[S \to .E][E \to .E + T][E \to .T][T \to F.]$	$+\mathsf{Id} * \mathsf{Id}$
$[S \to .E][E \to .E + T][E \to T.]$	$+\mathsf{Id} * \mathsf{Id}$
$[S \to .E][E \to E. + T]$	$+\mathsf{Id} * \mathsf{Id}$
$[S \to .E][E \to E + .T]$	$\mathsf{Id} * \mathsf{Id}$
$[S \to .E][E \to E + .T][T \to .T * F]$	$\mathsf{Id} * \mathsf{Id}$
$[S \to .E][E \to E + .T][T \to .T * F][T \to .F]$	$\mathsf{Id} * \mathsf{Id}$
$[S \to .E][E \to E + .T][T \to .T * F][T \to .F][F \to .\mathsf{Id}]$	$\mathsf{Id} * \mathsf{Id}$
$[S \to .E][E \to E + .T][T \to .T * F][T \to .F][F \to \mathsf{Id}.]$	$*\mathsf{Id}$
$[S \to .E][E \to E + .T][T \to .T * F][T \to F.]$	$*\mathsf{Id}$
$[S \to .E][E \to E + .T][T \to T. * F]$	$*\mathsf{Id}$
$[S \to .E][E \to E + .T][T \to T * .F]$	Id
$[S \to .E][E \to E + .T][T \to T * .F][F \to .\mathsf{Id}]$	Id
$[S \to .E][E \to E + .T][T \to T * .F][F \to \mathsf{Id}.]$	
$[S \to .E][E \to E + .T][T \to T * F.]$	
$[S \to .E][E \to E + T.]$	
$[S \to E.]$	

Table 3.2. The accepting computation of P_G for the word $\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$.

a bounded lookahead into the remaining input. LR-parsers signal reductions, and therefore are right-parsers. There does not exist an LR-parser for each context-free grammar, but only for grammars of the class $LR(k)$, where k again is the number of necessary lookahead symbols.

3.2.5 first- and follow-Sets

Let us consider the item-pushdown automaton P_G to a context-free grammar 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 pushdown automaton P_G must select non-deterministically one of the alternatives $Y \to \alpha_1 \mid \ldots \mid \alpha_n$ for the nonterminal Y. A good aid for this selection is the knowledge of 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 also produce short words or even ε the set of words that may follow Y becomes relevant.

It is wise to only consider *prefixes* of such words of a given length k since the sets of words that can be derived from an alternative are in general infinite. The sets of prefixes, in contrast, are 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, resp. For an alphabet V_T, we write $V_T^{\leq k}$ for $\bigcup_{i=0}^{k} V_T^i$ and $V_{T,\#}^{\leq k}$ for $V^{\leq k} \cup (V_T^{\leq k-1}\{\#\})$, 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

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

Further, we introduce the operator $\odot_k : V_T \times V_T \to V_T^{\le k}$ 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 $L_1, L_2 \subseteq V^{\le k}$ 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 context-free grammar. For $k \ge 1$, we define the function $\mathsf{first}_k : (V_N \cup V_T)^* \to 2^{V_T^{\le k}}$ 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 $\mathsf{follow}_k : V_N \to 2^{V_{T,\#}^{\le k}}$ 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:

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

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

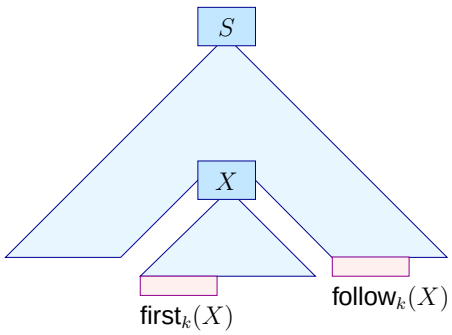

Fig. 3.5. first_k and follow_k in a parse tree

describes some properties of k-concatenation and the function first_k.

Lemma 3.2. Let $k \ge 1$, and let $L_1, L_2, L_3 \subseteq V^{\le k}$ be given. We then have:

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

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 $X_1 \dots X_n \stackrel{*}{\Longrightarrow} u$ holds if and only if $u = u_1 \dots u_n$ for suitable words u_i with $X_i \stackrel{*}{\Longrightarrow} u_i$.

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, the first_k-sets satisfy the equation system (fi):

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

Example 3.2.13 Let G_2 be the context-free grammar with the productions:

$$\begin{array}{lllllllll}
0: & S & \to & E & 3: & E' & \to & +E & 6: & T' & \to & *T \\
1: & E & \to & TE' & 4: & T & \to & FT' & 7: & F & \to & (E) \\
2: & E' & \to & \varepsilon & 5: & T' & \to & \varepsilon & 8: & F & \to & \mathsf{Id}
\end{array}$$

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:

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

□

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 one corresponding to the first_k-sets?
- How does one compute this solution?

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

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

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

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

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

Let us assume, $\mathbb{D}$ contained a distinctive element d_0 that would offer itself as start value for the calculation of a solution. A simple idea to determine a solution 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 might associate 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 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 properties of the domains $\mathbb{D}$ that occur in our applications.

- There always 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 $\perp$ (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 *compete 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 $d_0 = \perp$, 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 j, such that $I^{(j)} = I^{(j+1)}$ holds. This means that the algorithm in fact finds a solution. One can even show that this solution is the *least* solution. Such a least solution does even exist 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 i-ten 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:

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

□

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 context-free grammar, $\mathbb{D}$ the complete lattice of the subsets of $V_T^{\leq k}$, and $I : V_N \to \mathbb{D}$ 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 i-th iteration of the algorithm to find solutions for (fi). One shows by induction over i 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 $X \overset{*}{\underset{lm}{\Longrightarrow}} w$, 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. Is $n = 1$ the grammar has a production $X \to w$. We then have

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

and the claim follows with $i = 1$. Is $n > 1$, ther exists a production $X \to u_0 X_1 u_1 \ldots X_m u_m$ with $u_0, \ldots, u_m \in V_T^*$ and $X_1, \ldots, X_m \in V_N$ and leftmost derivations $X_i \overset{*}{\underset{lm}{\Longrightarrow}} w_j, j = 1, \ldots, k$ who 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

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

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, which are used to show the applicability of program transformations, which 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 just sketched is not necessarily the best method to solve systems of equations. In the volume *Compiler Design: Analysis and Transformation* we describe more efficient methods.

Let us now consider how to compute follow_k-sets for an extended context-free grammar G. Again, we start with an adequate recursive property. For a word $w \in V_T^k \cup V_T^{\leq k-1}\{\#\}$ holds $w \in \mathsf{follow}_k(X)$ 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 $\mathsf{follow}_k(X)$ satisfy the following system of equations :

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

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

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

□

The system of equations (fo) has again to 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. We want to ascertain 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 context-free grammar, $\mathbb{D}$ 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). We then 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 $\emptyset$ 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:

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

□

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 for more efficient methods, the computation of first_k- and follow_1-sets needs a large effort when k gets larger. Therefore, practical parsers only use lookahead of length $k = 1$. In this case, the computation of the first- and follow-sets can be performed particularly efficient. The following lemma is the base for our further treatment.

Lemma 3.3. Let $L_1, L_2 \subseteq V_T^{\leq 1}$ be non-empty 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 non-productive nor unreachable nonterminals. It holds for all $X \in V_N$ that $\mathsf{first}_1(X)$ as well as $\mathsf{follow}_1(X)$ are non-empty. Taken together with Lemma 3.3, it allows us to simplify the transfer functions for first_1 and follow_1 in such a way that the 1-concatenation can be (essentially) replaced by *unions*. We want to eliminate the case distinction of whether ε is contained in the first_1-sets or not. This done 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-sets. The *ε-free* first_1-sets are 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

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

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

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

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 $X \to X_1 \ldots X_m$. Its contribution to $\mathsf{eff}(X)$ can be written as

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

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 context-free grammar G_2 of Example 3.2.13. The following system of equations serves to compute the ε-free first_1-sets.

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

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

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

□

Nonterminals that occur to the right of terminals do not contribute to the ε-free first_1-sets. 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 simplifies to

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

Example 3.2.19 The simplified system of equations for the computation of the follow_1-sets of the context-free grammar G_2 of Example 3.2.13 becomes

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

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

The next section presents a method that solves arbitrary systems of equations very 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 had a system of equations

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

over an arbitrary complete lattice $\mathbb{D}$. On the right side of the equations were expressions e_i that are built only from constants in $\mathbb{D}$, variables $\mathbf{x}_j$, and applications of the operator $\sqcup$ (least upper bound of the complete lattice $\mathbb{D}$). The problem is to efficiently determine the least solution of this system of equations. Such a problem is called a *pure union problem*.

The computation of the set of reachable nonterminals of a context-free grammar 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 context-free grammar are pure union problems. In these cases, the complete lattices are 2^{V_T} and $2^{V_T \cup \{\#\}}$, 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

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

□

We construct a variable-dependency graph to a pure union problem. The nodes of this graph are the variables $\mathbf{x}_i$ of the system of equations. 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 $\mathbf{x}_j$. Fig. 3.6 shows the variable-dependency graph for the system of equations of Example 3.2.20

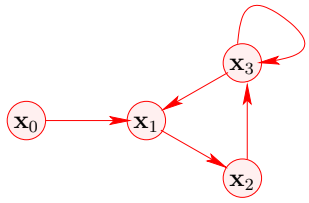

Fig. 3.6. The variable-dependency graph for the system of equations of Example 3.2.20.

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-dependency graph. In consequence, the values of all variables in each *strongly-connected component* of the variable-dependency graph are the same.

We label each variable $\mathbf{x}_i$ with the least upper bound of all constants that occur on the right sides of equations for variable $\mathbf{x}_i$. Let us call this value $I_0(\mathbf{x}_i)$. We have 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 to compute the least solution I of the system of equations. First, the strongly-connected components of the variable-dependency 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 entering edges coming 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:

$$\begin{aligned} &\mathbb{D}\ t \leftarrow \bot; \\ &\textbf{forall } (\mathbf{x}_i \in Q) \\ &\qquad t \leftarrow t \sqcup I_0(\mathbf{x}_i); \\ &\textbf{forall } (\mathbf{x}_i \in Q) \\ &\qquad I(\mathbf{x}_i) \leftarrow t; \end{aligned}$$

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 propagated along the outgoing edges. Let E_Q be the set of edges $(\mathbf{x}_i, \mathbf{x}_j)$ of the variable-dependency 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:

$$\begin{aligned}&\textbf{forall } ((\mathbf{x}_i, \mathbf{x}_j) \in E_Q)\\ &\qquad I_0(\mathbf{x}_j) \leftarrow I_0(\mathbf{x}_j) \sqcup I(\mathbf{x}_i);\end{aligned}$$

The number of steps for the propagation is proportional to the number of edges in E_Q.

The strongly-connected component Q together with the set E_Q of outgoing edges is removed from the graph and one continues with the next strongly-connected component without ingoing edges. This is repeated until no more strongly-connected component remains. Altogether, we have a method that performs a linear number of operations $\sqcup$ on the complete lattice $\mathbb{D}$.

Example 3.2.22 (Continuation of Example 3.2.20) The dependency 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 one obtains the value $I_0(\mathbf{x}_0) = \{a\}$. After removal of Q_0 and the edge $(\mathbf{x}_0, \mathbf{x}_1)$, the new assignment is:

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

The value of all variables in the strongly-connected component Q_1 arise 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 best be made intuitively clear 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 context-free grammar; 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. By attaching symbols of the right side of a production terminal symbols can appear in the leaf word of a tree fragment. If there is no nonterminal to the left of a terminal symbol in the leaf word the top-down *top-down* parser compares them with the next symbol in the input. If they agree the parser will consume these symbols in the input. Otherwise, the parser will report 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.

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

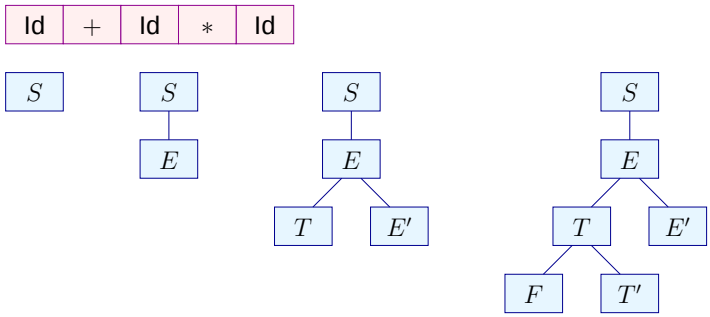

Fig. 3.7. The first parse-tree fragments of a *top-down* analysis of the word $\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$ according to grammar G_2. They are constructed without reading any symbol from the input.

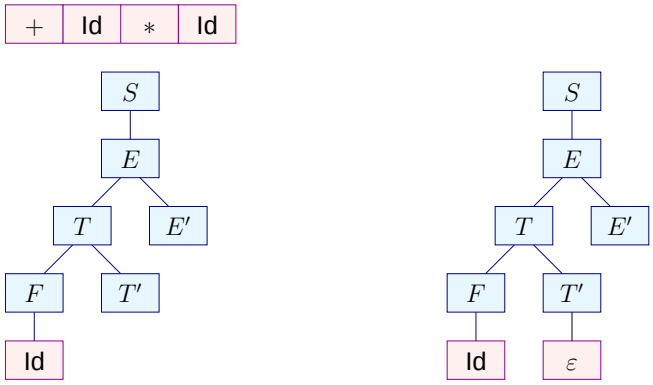

Fig. 3.8. The parse tree fragments after reading of the symbol Id and before the terminal symbol $+$ is attached to the fragment.

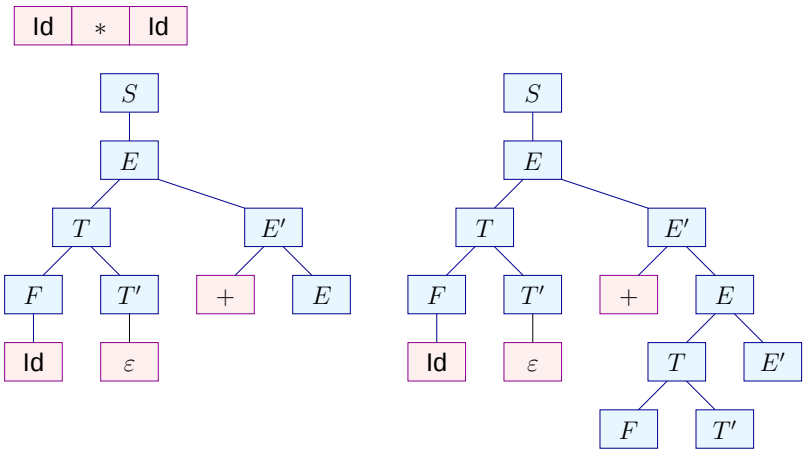

Fig. 3.9. The first and the last parse tree after reading of the symbols $+$ and before the second symbol Id appears in the parse tree.

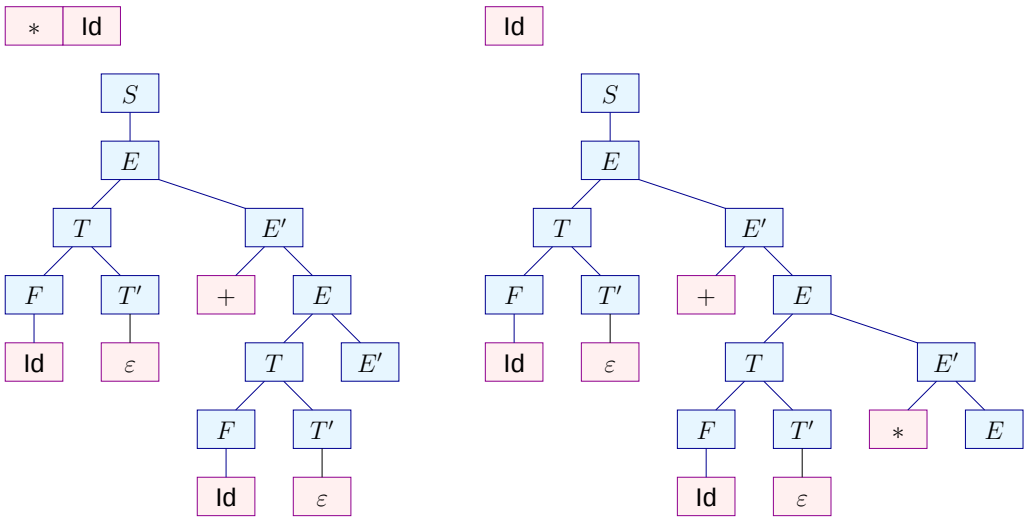

Fig. 3.10. The parse tree after the reduction for the second occurrence of T' and the parse tree after reading the symbol $*$, together with the remaining input.

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

The Item-pushdown automaton P_G to a context-free grammar G works in principle like a *top-down* parser; its (E)-transitions make a predictions which alternative to select for the actual nonterminal to derive the input word. The trouble is that the item pushdown-automaton 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 item pushdown-automaton P_G we equip the automaton with a *bounded lookahead* into the remaining input. We fix a natural number $k \geq 1$ and allow the item pushdown-automaton to inspect the k first symbols of the remaining input at each (E) transition to aid in its decision. If this lookahead of depth k always suffices to select the right alternative we call the grammar $LL(k)$ grammar.

Let us regard a configuration that the item pushdown-automaton 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 Section **??** it holds $\mathsf{hist}(\rho)\beta \overset{*}{\Longrightarrow} u$.

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 $S' \overset{*}{\underset{lm}{\Longrightarrow}} uY\gamma\delta$ has been found. If this derivation can be extended to a derivation of the terminal word uv, that is, $S' \overset{*}{\underset{lm}{\Longrightarrow}} uY\gamma\delta \overset{*}{\underset{lm}{\Longrightarrow}} uv$, then in an $LL(k)$ 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 context-free grammar G is a $LL(k)$-*grammar* if for every two leftmost derivations:

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

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 context-free grammar with the productions:

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

The grammar G_1 is an $LL(1)$ grammar. If $\langle \textit{stat} \rangle$ 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

$$\langle \textit{stat} \rangle \underset{lm}{\overset{*}{\Longrightarrow}} w\,\langle \textit{stat} \rangle\,\alpha \underset{lm}{\Longrightarrow} w\,\beta\,\alpha \underset{lm}{\overset{*}{\Longrightarrow}} w\,x$$
$$\langle \textit{stat} \rangle \underset{lm}{\overset{*}{\Longrightarrow}} w\,\langle \textit{stat} \rangle\,\alpha \underset{lm}{\Longrightarrow} w\,\gamma\,\alpha \underset{lm}{\overset{*}{\Longrightarrow}} w\,y$$

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

Definition 3.3.1 (simple LL(1)-grammar)
Let G be a context-free grammar without ε-productions. If for each nonterminal N, each of its alternatives begins with a different terminal symbol, then G is called a simple LL(1) grammar. □

This is a first, easily checked criterion for a special case. The grammar G_1 of Example 3.3.1 is a simple LL(1) grammar.

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

$$\begin{array}{lcll} \langle \textit{stat} \rangle & \rightarrow & \mathsf{Id}\ :\ \langle \textit{stat} \rangle \ \mid & // \quad \text{labeled statement} \\ & & \mathsf{Id}\ (\mathsf{Id});\ & // \quad \text{procedure call} \end{array}$$

The grammar G_2 thus obtained is no longer an $LL(1)$ grammar because it holds

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

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

However, G_2 is a $LL(2)$ grammar. For the three leftmost derivations given above holds,

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

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

Example 3.3.3 G_3 possesses the productions

$$\begin{array}{lcl}
\langle \textit{stat} \rangle & \rightarrow & \mathsf{if}\ (\langle \textit{var} \rangle)\ \langle \textit{stat} \rangle\ \mathsf{else}\ \langle \textit{stat} \rangle \mid \\
& & \mathsf{while}\ (\langle \textit{var} \rangle)\ \langle \textit{stat} \rangle \mid \\
& & \{\ \langle \textit{stats} \rangle\ \} \mid \\
& & \langle \textit{var} \rangle\ '='\ \langle \textit{var} \rangle; \mid \\
& & \langle \textit{var} \rangle; \\
\langle \textit{stats} \rangle & \rightarrow & \langle \textit{stat} \rangle\ \langle \textit{stats} \rangle \mid \\
& & \varepsilon \\
\langle \textit{var} \rangle & \rightarrow & \mathsf{Id} \mid \\
& & \mathsf{Id}() \mid \\
& & \mathsf{Id}(\langle \textit{vars} \rangle) \\
\langle \textit{vars} \rangle & \rightarrow & \langle \textit{var} \rangle,\ \langle \textit{vars} \rangle \mid \\
& & \langle \textit{var} \rangle
\end{array}$$

The grammar G_3 is for no $k \geq 1$ an $LL(k)$ grammar. To derive a contradiction assume G_3 were an $LL(k)$ grammar for a $k > 0$.

Let $\langle \textit{stat} \rangle \Rightarrow \beta \underset{lm}{\overset{*}{\Longrightarrow}} x$ and $\langle \textit{stat} \rangle \Rightarrow \gamma \underset{lm}{\overset{*}{\Longrightarrow}} y$ with
$x = \mathsf{Id}\ (\underbrace{\mathsf{Id}, \mathsf{Id}, \ldots, \mathsf{Id}}_{k})\ '='\ \mathsf{Id};$ and $y = \mathsf{Id}\ (\underbrace{\mathsf{Id}, \mathsf{Id}, \ldots, \mathsf{Id}}_{k});$

We have $x|_k = y|_k$, but

$$\beta = \langle \textit{var} \rangle\ '='\ \langle \textit{var} \rangle \qquad \gamma\ =\ \langle \textit{var} \rangle;$$

and therefore $\beta \neq \gamma$. $\square$

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. The different continuations can be derived from this nonterminal. The productions

$$\langle \textit{stat} \rangle \rightarrow \langle \textit{var} \rangle\ '='\ \langle \textit{var} \rangle;\ \mid\ \langle \textit{var} \rangle;$$

are replaced by

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

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 no $LL(k)$ grammar for any $k \geq 1$. To see this we consider the two leftmost derivations

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

G_4 is for no $k \geq 1$ an $LL(k)$ grammar since 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 one can show that for no $k \geq 1$ there exists an $LL(k)$-grammar for the language $L(G_4)$. □

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

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

Proof. To prove the direction, " ⇒ ", we assume, G were an $LL(k)$ grammar, but there existed 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

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

where in in the case $|x| < k$ it must hold $y = z = \varepsilon$. $\beta \neq \gamma$ implies that G can not be an $LL(k)$ grammar—a contradiction to our assumption.

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

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

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 the application of two different productions to a left-sentential form always leads to different k-prefixes of the remaining input. Theorem 3.3.1 allows to derive useful criteria for membership of 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-produktiv.

Theorem 3.3.2 Let G be an ε-free context-free grammar, that is, without productions of the form $X \to \varepsilon$. Then G is an $LL(1)$ grammar if and only if for each nonterminal X with the alternatives $X \to \alpha_1 \mid \ldots \mid \alpha_n$ the sets $\mathsf{first}_1(\alpha_1), \ldots, \mathsf{first}_1(\alpha_n)$ are pairwise disjoint.

In practice, it would be too hard a restriction to forbid ε-productions. Consider the case that one of the two right sides β or γ would produce the empty word. If both β as well as γ produce the empty word G can not be an $LL(1)$ grammar. Let us, therefore, assume that $\beta \overset{*}{\Longrightarrow} \varepsilon$, but that ε can not be derived from γ. However, then 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

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

We, hereby, obtain the following theorem:

Theorem 3.3.3 A reduced context-free grammar G is an $LL(1)$ grammar if and only if for each two different productions $A \to \beta$ and $A \to \gamma$ holds

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

□

The characterization of Theorem 3.3.3 is easily checked in contrast to the characterization of Theorem 3.3.1. An even more easily checkable formulation is obtained by exploiting properties of 1-concatenation.

Corollary 3.3.3.1 A reduced context-free grammar G is an $LL(1)$ grammar if and only if for all alternatives $A \to \alpha_1 \mid \ldots \mid \alpha_n$ holds

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. $\varepsilon \in \mathsf{first}_1(\alpha_i)$ implies $\mathsf{first}_1(\alpha_j) \cap \mathsf{follow}_1(A) = \emptyset$ for all $1 \leq j \leq n,\ j \neq i$. □

We extend the property of Theorem 3.3.3 to arbitrary lengths $k \geq 1$ of lookaheads.

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

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

According to this definition and Theorem 3.3.3 every $LL(1)$ grammar is a strong $LL(1)$ grammar. However, an $LL(k)$ grammar for $k > 1$ is not automatically a strong $LL(k)$ grammar. The reason is that the set $\mathsf{follow}_k(A)$ contains the follow words of *all* left sentential forms with occurrences of A. In contrast, the $LL(k)$ condition only refers to follow words of *one* left sentential form.

Example 3.3.5 Let G be the context-free grammar with the productions

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

We check:

1. Fall: The derivation starts with $S \Rightarrow aAaa$. It holds $\mathsf{first}_2(baa) \cap \mathsf{first}_2(aa) = \emptyset$.
2. Fall: The derivation starts with $S \Rightarrow bAba$. It holds $\mathsf{first}_2(bba) \cap \mathsf{first}_2(ba) = \emptyset$.

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

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

In the example, $\mathsf{follow}_1(A)$ is too undifferentiated because it collects terminal follow 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 context-free grammar G is called left recursive if there exists a derivation $A \overset{+}{\Longrightarrow} A\beta$.

Theorem 3.3.4 Let G be a reduced context-free grammar. 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 $X \to X\beta$. G is reduced. So, there must exist another production $X \to \alpha$. If X occurs in a left sentential form, that is, $S \overset{*}{\underset{lm}{\Longrightarrow}} uX\gamma$, the alternative $X \to X\beta$ can be applied arbitrarily often. For each $n \geq 1$ there exists a leftmost derivation

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

Let us assume that grammar G were an $LL(k)$ grammar. Theorem 3.3.1 implies

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

Due to $X \to \alpha$ we have

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

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 can not 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 us assume for simplicity that the grammar G has no ε-productions (see Exercise **??**) and no recursive chain productions, that is, there is no nonterminal A with $A \overset{+}{\underset{G}{\Longrightarrow}} A$. Let $G = (V_N, V_T, P, S)$. We construct for G a context-free grammar $G' = (V'_N, V_T, P', S)$ 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'

- Is $B \to a\beta \ \in \ P$ for a terminal symbol $a \in V_T$, then $A \to a\beta\,\langle A, B\rangle \ \in \ P'$ for each $A \in V_N$;
- Is $C \to B\beta \ \in \ P$ 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

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

we obtain after removal of non-productive nonterminals

$$\begin{aligned} E &\to (E)\,\langle E, F\rangle \mid \mathsf{Id}\,\langle E, F\rangle \\ \langle E, F\rangle &\to \langle E, T\rangle \\ \langle E, T\rangle &\to *\,F\,\langle E, T\rangle \mid \langle E, E\rangle \\ \langle E, E\rangle &\to +T\,\langle E, E\rangle \mid \varepsilon \\ T &\to (E)\,\langle T, F\rangle \mid \mathsf{Id}\,\langle E, F\rangle \\ \langle T, F\rangle &\to \langle T, T\rangle \\ \langle T, T\rangle &\to *\,F\,\langle T, T\rangle \mid \varepsilon \\ F &\to (E)\,\langle F, F\rangle \mid \mathsf{Id}\,\langle F, F\rangle \\ \langle F, F\rangle &\to \varepsilon \end{aligned}$$

Grammar G_0 has three nonterminals and six productions, grammar G_1, needs nine nonterminals and 15 productions.

The parse tree for Id + Id according to grammar G_0 is shown in Fig. 3.11 (a), the one according to grammar G_1 in Fig. 3.11 (b). The latter one has a definitely different structure. Intuitively, the grammar generates directly the first possible terminal symbol and then in a backward fashion collects the remainders of the right sides, which follow the left-side nonterminal symbol. The nonterminal $\langle A, B\rangle$ stands for the job to return from B back 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

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

if and only there exists a rightmost derivation

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

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

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 might be not advisable to perform this transformation *manually*. A parser generator however, could do the transformation automatically and also generate a program that would automatically convert parse trees of the transformed grammar back into parse trees of the original grammar (see Exercise **??** of the next section). The user wouldn't even see the grammar transformation.

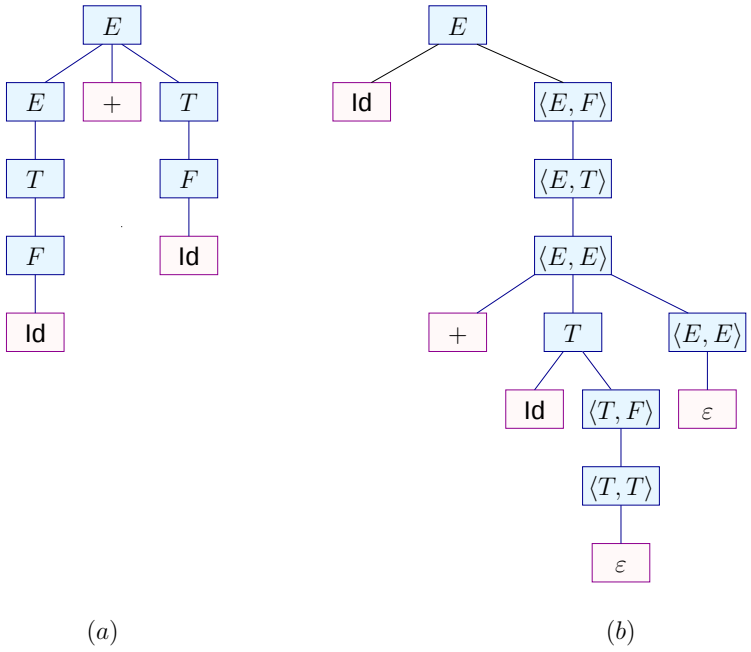

Fig. 3.11. Parse trees for Id + Id according to grammar G_0 of Example 3.3.6 and according to the grammar after removal of left recursion.

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 later.

3.3.4 Strong $LL(k)$ Parsers

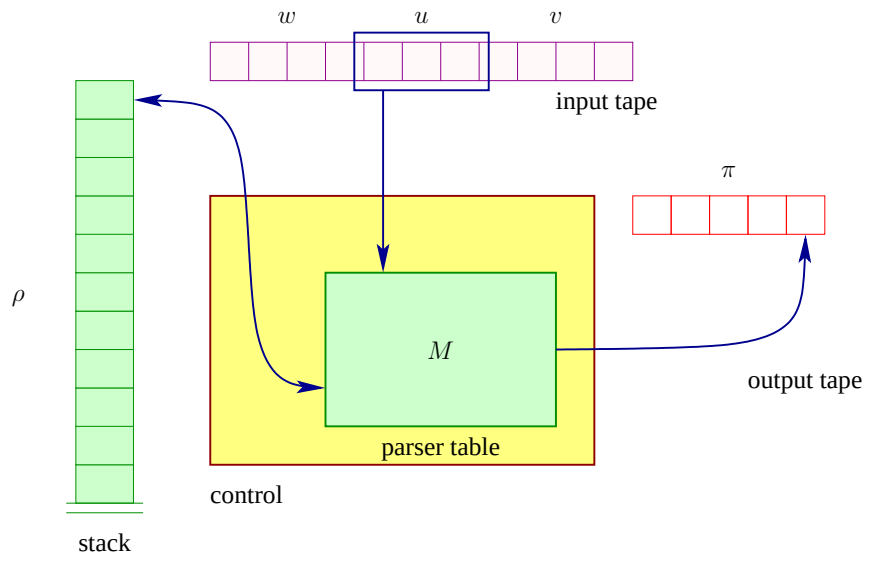

Fig. 3.12. Schematic representation of a strong $LL(k)$-Parser.

Fig. 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 stack contains a sequence of items of the context-free grammar. The topmost item, the 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 2-dimensional array whose rows are indexed by the 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 one of its alternatives that should be applied based on the given lookahead. It could also signal an error if no alternative exists for the combination of actual state and lookahead. Let $[X \rightarrow \beta.Y\gamma]$ be the topmost item on the stack 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 stack 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 $u \in \mathsf{first}_k(\alpha_1) \odot_k \mathsf{follow}_k(Y) \cup \ldots \cup, \mathsf{first}_k(\alpha_r) \odot_k \mathsf{follow}_k(Y)$ is therefore

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

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 don't go together. This means that a syntax error has been found. A

error-diagnosis and error-handling routine is started, which will attempt to continue the analysis. Such approaches will be described in Section **??**.

For $k = 1$, the construction of the parser table is particularly simple. Because of Corollary 3.3.3.1, it works without 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 $\mathsf{Id} * \mathsf{Id}\#$. □

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

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

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:

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

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

Our construction of $LL(k)$ parser 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 lookahead $k > 1$ is needed, and is the grammar $LL(k)$, but not strong $LL(k)$, a general transformation can be applied converting the grammar into a strong $LL(k)$ grammar that accepts the same language. (see Exercise 7).

We do, therefore, not describe a parsing method for arbitrary $LL(k)$ grammars.

3.3.5 LL Parsers for Right-regular Context-free Grammars

Left-recursive nonterminals destroy the LL property of context-free grammars. Left recursion is mostly used to describe sequences and lists of syntactic objects, like parameter lists and sequences of operands connected by an associative operator. These can also be described by regular expressions. Thus, we want to offer the best description comfort by admitting regular expressions on the right side of productions.

A *right-regular* context-free grammar is a tuple $G = (V_N, V_T, p, S)$, where V_N, V_T, S are as usual the set of nonterminals, the set of terminals, and the start symbol. $p : V_N \to RA$ is now a function from the set of nonterminals into the set *RA* of regular expressions over $V_N \cup V_T$. A pair (X, r) with $p(X) = r$ is written as $X \to r$.

Example 3.3.8
A right-regular context-free grammar for arithmetic expressions is

$$G_e = (\{S, E, T, F\}, \{\mathbf{id}, (,), +, -, *, /\}, p, S),$$

where p is the following function ($'\{'$ and $'\}'$ are used as meta-characters to avoid the conflict with the terminal symbols $'('$ and $')'$):
$S \to E$
$E \to T\{\{+\,|\,-\}\,T\}^*$
$T \to F\{\{*\,|\,/\}\,F\}^*$
$F \to (E)\,|\,\mathbf{id}$ □

Definition 3.3.2 (regular derivation)
Let G be a right-regular context-free grammar. The relation $\underset{R,lm}{\Longrightarrow}$ on *RA*, *directly derives leftmost, regular*, is defined by:

(a)	$w\,X\,\beta$	$\underset{R,lm}{\Longrightarrow}\ w\,\alpha\,\beta$	mit $\alpha = p(X)$
(b)	$w\,(r_1 \,|\, \ldots \,|\, r_n)\,\beta$	$\underset{R,lm}{\Longrightarrow}\ w\,r_i\,\beta$	f"ur $1 \le i \le n$
(c)	$w\,(r)^*\,\beta$	$\underset{R,lm}{\Longrightarrow}\ w\,\beta$	
(d)	$w\,(r)^*\,\beta$	$\underset{R,lm}{\Longrightarrow}\ w\,r\,(r)^*\,\beta$	

Let $\overset{*}{\underset{R,lm}{\Longrightarrow}}$ be the die reflexive, transitive closure of $\underset{R,lm}{\Longrightarrow}$. The language defined by G is $L(G) = \{w \in V_T^* \mid S \overset{*}{\underset{R,lm}{\Longrightarrow}} w\}$ □

Example 3.3.9
A regular leftmost derivation for the word **id** + **id** ∗ **id** of grammar G_e of Example 3.3.8 is:
$S \underset{R,lm}{\Longrightarrow} E \underset{R,lm}{\Longrightarrow} T\{\{+|-\}T\}^*$
$\underset{R,lm}{\Longrightarrow} F\{\{*|/\}F\}^*\{\{+|-\}T\}^*$
$\underset{R,lm}{\Longrightarrow} \{(E)|\mathbf{id}\}\{\{*|/\}F\}^*\{\{+|-\}T\}^*$
$\underset{R,lm}{\Longrightarrow} \mathbf{id}\{\{*|/\}F\}^*\{\{+|-\}T\}^*$

$\underset{R,lm}{\Longrightarrow}$ **id**$\{\{+|-\}T\}^*$
$\underset{R,lm}{\Longrightarrow}$ **id**$\{+|-\}T\{\{+|-\}T\}^*$
$\underset{R,lm}{\Longrightarrow}$ **id** $+ T\{\{+|-\}T\}^*$
$\underset{R,lm}{\Longrightarrow}$ **id** $+ F\{\{*|/\}F\}^*\{\{+|-\}T\}^*$
$\underset{R,lm}{\Longrightarrow}$ **id** $+ \{(E)|$**id**$\}\{\{*|/\}F\}^*\{\{+|-\}T\}^*$
$\underset{R,lm}{\Longrightarrow}$ **id** $+$ **id**$\{\{*|/\}F\}^*\{\{+|-\}T\}^*$
$\underset{R,lm}{\Longrightarrow}$ **id** $+$ **id**$\{*|/\}F\{\{*|/\}F\}^*\{\{+|-\}T\}^*$
$\underset{R,lm}{\Longrightarrow}$ **id** $+$ **id** $* F\{\{*|/\}F\}^*\{\{+|-\}T\}^*$
$\underset{R,lm}{\Longrightarrow}$ **id** $+$ **id** $* \{(E)|$**id**$\}\{\{*|/\}F\}^*\{\{+|-\}T\}^*$
$\underset{R,lm}{\Longrightarrow}$ **id** $+$ **id** $*$ **id**$\{\{*|/\}F\}^*\{\{+|-\}T\}^*$
$\underset{R,lm}{\Longrightarrow}$ **id** $+$ **id** $*$ **id**$\{\{+|-\}T\}^*$
$\underset{R,lm}{\Longrightarrow}$ **id** $+$ **id** $*$ **id** □

Our goal is to develop an RLL parser, that is, a deterministic top down parser for right-regular context-free grammars. This is the method of choice to implement a parser as long as no powerful and comfortable tools offer an attractive alternative.

The RLL parser will produce a regular leftmost derivation for any correct input word. Looking at the definition above makes clear that the case of expansion (a)—a nonterminal is replaced by its only right side—is no longer critical. Instead, the cases (b), (c) and (d) need to be made deterministic.

We will call a parser for a right-regular context-free grammar an RLL(1) parser if it

- for each regular left-sentential form $w(r_1 \mid \ldots \mid r_n)\beta$ can take the decision for the right alternative,
- for each regular left-sentential form $w(r)^*\beta$ can take the decision for the continuation or the termination of the iteration

based on the next input symbol of the remaining input. We transfer some notions to the case of right-regular context-free grammars.

Definition 3.3.3 (regular subexpression)
r_i, $1 \leq i \leq n$, is **direct regular subexpression** of $(r_1 \mid \ldots \mid r_n)$ and $(r_1 \ \ldots \ r_n)$; r is **direct regular subexpression** von $(r)^*$ and of r; r_1 ist **regular subexpression** of r_2, if $r_1 = r_2$ or if r_1 is a direct regular subexpression of r_2 or regular subexpression of a direct regular subexpression of r_2. □

Definition 3.3.4 (extended context-free item)
A tuple $(X, \alpha, \beta, \gamma)$ is an **extended context-free item** of a right-regular context-free grammar $G = (V_N, V_T, p, S)$ if $X \in V_N$, $\alpha, \beta, \gamma \in (V_N \cup V_T \cup \{(,), {}^*, |, \varepsilon\})^*$, $p(X) = \beta\alpha\gamma$ and α is regular subexpression of $\beta\alpha\gamma$. This item is written as $[X \to \beta.\alpha\gamma]$. □

Realizing an RLL(1) parser for a right-regular context-free grammar uses again first_1- and follow_1 sets, this time of regular subexpressions of right sides of productions.

first_1- and follow_1- Computation for Right-regular Context-free Grammars

The computations of first_1- and follow_1-sets for right-regular context-free grammars can again be represented as pure union-problems, and can, therefore, be efficiently solved. In the same way as in the conventional case, this starts with the computation of ε-productivity. The equations for ε-productivity can be defined over the structure of regular expressions. The ε-productivity of right sides transfers to the nonterminal of the left side.

$$
\begin{array}{l}
\mathit{eps}(a) = \mathit{false}, \quad \text{for } a \in V_T \\
\mathit{eps}(\varepsilon) = \mathit{true} \\
\mathit{eps}(r^*) = \mathit{true} \\
\mathit{eps}(X) = \mathit{eps}(r), \text{ if } p(X) = r \text{ for } X \in V_N \\
\mathit{eps}((r_1 | \ldots | r_n)) = \bigvee_{i=1}^{n} \mathit{eps}(r_i) \\
\mathit{eps}((r_1 \ldots r_n)) = \bigwedge_{i=1}^{n} \mathit{eps}(r_i)
\end{array}
\qquad (\mathit{eps})
$$

Example 3.3.10 (Continuation of Example 3.3.8)
For all nonterminals of G_e holds: $\mathit{eps}(X) = \mathit{false}$ □

After ε-productivity is computed, the ε-free first-function can be computed. This is specified by the following equations:

$$
\begin{array}{l}
\mathsf{eff}(\varepsilon) = \emptyset \\
\mathsf{eff}(a) = \{a\} \\
\mathsf{eff}(r^*) = \mathsf{eff}(r) \\
\mathsf{eff}(X) = \mathsf{eff}(r), \text{ if } p(X) = r \\
\mathsf{eff}((r_1 | \ldots | r_n)) = \bigcup_{1 \le i \le n} \mathsf{eff}(r_i) \\
\mathsf{eff}((r_1 \ldots r_n)) = \bigcup_{1 \le j \le n} \{\mathsf{eff}(r_j) \mid \bigwedge_{1 \le i < j} \mathit{eps}(r_i)\}
\end{array}
\qquad (\mathsf{eff})
$$

Example 3.3.11 (Continuation of Example 3.3.8)
The eff- and, therefore, also the first_1-sets for the nonterminals of grammar G_e are
$\mathsf{first}_1(S) = \mathsf{first}_1(E) = \mathsf{first}_1(T) = \mathsf{first}_1(F) = \{(, \mathbf{id}\}$ □

ε-productivity and ε-free first-functions could be defined recursively over the structure of regular expressions. The first_1-set of a regular expression is independent of the context in which it occurs.

This is different for the follow_1-set; two different occurrences of a regular (sub-) expression have in general different follow_1-sets. In realizing RLL(1) parsers, we are interested in the follow_1-sets of occurrences of regular (sub-) expressions. A particular occurrence of a regular expression in a right side corresponds to exactly one extended regular item in which the dot is positioned in front of this regular expression. The following equations for follow_1 assume that concatenations and lists of alternatives are surrounded on the outside by parentheses, but have no superfluous parentheses inside.

(1) $\mathsf{follow}_1([S' \to .S]) = \{\#\}$ The eof symbol '#' follows after each input word.

(2) $\mathsf{follow}_1([X \to \cdots (r_1 | \cdots | .r_i | \cdots | r_n) \cdots]) =$
$\quad \mathsf{follow}_1([X \to \cdots .(r_1 | \cdots | r_i | \cdots | r_n) \cdots])$ for $1 \le i \le n$

(3) $\mathsf{follow}_1([X \to \cdots (\cdots .r_i r_{i+1} \cdots) \cdots]) =$
$$\mathsf{eff}(r_{i+1}) \cup \begin{cases} \mathsf{follow}_1([X \to \cdots (\cdots r_i . r_{i+1} \cdots) \cdots]), \\ \quad \text{if } \mathit{eps}(r_{i+1}) = \mathit{true} \\ \emptyset \quad \text{otherwise} \end{cases}$$

(4) $\mathsf{follow}_1([X \to \cdots (r_1 \cdots r_{n-1} . r_n) \cdots]) =$ $\qquad (\mathsf{follow}_1)$
$\quad \mathsf{follow}_1([X \to \cdots .(r_1 \cdots r_{n-1} r_n) \cdots])$

(5) $\mathsf{follow}_1([X \to \cdots (.r)^* \cdots]) =$
$\quad \mathsf{eff}(r) \cup \mathsf{follow}_1([X \to \cdots .(r)^* \cdots])$

(6) $\mathsf{follow}_1([X \to .r]) = \bigcup \mathsf{follow}_1([Y \to \cdots .X \cdots])$

Example 3.3.12 (Continuation of Example 3.3.8)
The follow_1-sets for some items to grammar G_e are:
$\mathsf{follow}_1([S \to .E]) = \{\#\}$
$\mathsf{follow}_1([E \to T.\{\{+|-\}T\}^*]) \stackrel{(4)}{=}$
$\mathsf{follow}_1([E \to .T\{\{+|-\}T\}^*]) \stackrel{(6)}{=}$
$\mathsf{follow}_1([S \to .E]) \cup \mathsf{follow}_1([F \to (.E)]) =$
$(\{\#\} \cup \mathsf{follow}_1([F \to (.E)])) \stackrel{(3)}{=} \{), \#\}$
$\mathsf{follow}_1([T \to F.\{\{*|/\}F\}^*]) = \{+, -,), \#\}$ □

To compute solutions for eff and follow_1 as efficiently as possible, that is, in linear time, these equation systems need to be brought into the form

$$f(X) = g(X) \cup \bigcup \{f(Y) \mid X\ R\ Y\}$$

with a known set-valued function g and a binary relation R.

In the computation of eff the base set of R and the set of nodes of the directed graph induced by R is the set of regular (sub-) expressions occurring in the production. A directed edge from X to Y exists if and only if either Y is a direct subexpression of X and Y contributes to the first_1-set of X, or if X is a nonterminal (occurrence) and Y its right side. The function g is only defined to be non-empty for the case of a terminal symbols.

In the computation of follow_1 the base set is the set of extended items, and the relation associates such items j with an item i, that contribute to the follow_1-set of i. The function g is defined using the already computed eff-sets.

Definition 3.3.5 (RLL(1)-grammar)
A right-regular context-free grammar $G = (V_N, V_T, p, S)$ is called **RLL(1) grammar** if for all extended context-free items
$[X \to \cdots .(r_1|\cdots|r_n)\cdots]$ holds:
$\mathsf{first}_1(r_i) \oplus_1 \mathsf{follow}_1([X \to \cdots .(r_1|\cdots|r_n)\cdots]) \cap$
$\mathsf{first}_1(r_j) \oplus_1 \mathsf{follow}_1([X \to \cdots .(r_1|\cdots|r_n)\cdots]) = \emptyset$ for all $i \neq j$,
and for all extended context-free items $[X \to \cdots .(r)^*\cdots]$ holds:
$\mathsf{first}_1(r) \cap \mathsf{follow}_1([X \to \cdots .(r)^*\cdots]) = \emptyset$ and $\mathit{eps}(r) = \mathit{false}$. □

Once the first_1- and follow_1-sets for a right-regular context-free grammar are computed, and the check for the RLL(1)-property has been successful, an RLL(1) parser for the grammar can be generated. Two different representations are popular. The first consists of a driver, fixed for all grammars, and a table specifically generated for each grammar. The driver indexes the table with the actual item and the next input symbol, more precisely, some integer codes for these two. The selected entry in the table indicates the next item or signals a syntax error.

The second representation is by a program. This program consists essentially of a set simultaneously recursice procedures, one per nonterminal. The procedure for nonterminal X is in charge of analyzing words for X. We first introduce the table version of RLL(1) parsers.

RLL(1) Parser for Right-regular Context-free Grammars (Table Version)

The RLL(1) parser is a deterministic pushdown automaton. The parser table M represents a selection function $m : \mathsf{It}_G \times V_T \rightsquigarrow \mathsf{It}_G \cup \{\mathit{error}\}$. The parser table is consulted when a decision has to be taken by considering lookahead into the remaining input. Therefore, M has only rows for

- items in which an alternative needs to be chosen, and
- items in which an iteration needs to processed;

i.e. the function m is defined for items of the form $[X \to \cdots .(r_1|\cdots|r_n)\cdots]$ and of the form $[X \to \cdots .(r)^*\cdots]$.

The RLL(1) parser is started in an initial configuration $(\#[S' \to .S], w\#)$. The actual item, the topmost on the stack, determines whether the parser table should be consulted. If the table needs to be consulted $M[\rho, a]$ – if not error – indicates the next item for the actual item ρ and the actual input symbol a. If $M[\rho, a] =$ error , a syntax error has been discovered. In the configuration $(\#[S' \to S.], \#)$, the parser accepts the input word.
The other transitions are:

$\delta([X \to \cdots .a \cdots], a) = [X \to \cdots a. \cdots]$
$\delta([X \to \cdots .Y \cdots], \varepsilon) = [X \to \cdots .Y \cdots][Y \to .p(Y)]$
$\delta([X \to \cdots .Y \cdots][Y \to p(Y).], \varepsilon) = [X \to \cdots Y. \cdots]$

In addition, there were some transitions, for example from $[X \to \cdots(\cdots|r_i.|\cdots)\cdots]$ to $[X \to \cdots(\cdots|r_i|\cdots).\cdots]$, which neither read symbols, nor expand nonterminals, nor reduce to nonterminals. They can be avoided by modifying the transition function in the following way:

$(1)\ [X \to \cdots(\cdots|r_i.|\cdots)\cdots] \Rightarrow (2)\ [X \to \cdots(\cdots|r_i|\cdots).\cdots]$
$(3)\ [X \to \cdots(r.)^*\cdots] \Rightarrow (4)\ [X \to \cdots .(r)^*\cdots]$
$(5)\ [X \to \cdots .(r_1 \cdots r_n)\cdots] \Rightarrow (6)\ [X \to \cdots(.r_1 \cdots r_n)\cdots]$

If a transition of δ leads to (1), it is made to lead to the context-free item (2). If it leads to (3), it is made to lead to (4), and from (5) directly to (6).

We present now the algorithm for the generation of the RLL(1) parser tables.

Algorithm RLL(1)-GEN

Input: RLL(1)-grammar G, first_1 and follow_1 for G.
Output: parser table M for RLL(1) parser for G.
Method: For all items of the form $[X \to \cdots .(r_1|\cdots|r_n)\cdots]$ set
$M([X \to \cdots .(r_1|\cdots|r_n)\cdots], a) = [X \to \cdots(\cdots|.r_i|\cdots)\cdots]$, for $a \in \mathsf{first}_1(r_i)$ and if in addition $\varepsilon \in \mathsf{first}_1(r_i)$ then also for $a \in \mathsf{follow}_1([X \to \cdots .(r_1|\cdots|r_n)\cdots])$.
For all items of the form $[X \to \cdots .(r)^*\cdots]$ set

$$M([X \to \cdots .(r)^*\cdots], a) = \begin{cases} [X \to \cdots(.r)^*\cdots] & \text{if } a \in \mathsf{first}_1(r) \\ [X \to \cdots(r)^*.\cdots] & \text{if } a \in \mathsf{follow}_1([X \to \cdots .(r)^*\cdots]) \end{cases}$$

Set all not yet filled entries to *error*.

Example 3.3.13 (Continuation of Example 3.3.8)
The parser table to grammar G_e. (Rows and columns are exchanged for layout reasons.)

	$[E \to T.\{\{+\|-\}T\}^*]$	$[T \to F.\{\{*\|/\}F\}^*]$
$+$	$[E \to T\{\{.+\|-\}T\}^*]$	$[T \to F\{\{*\|/\}F\}^*.]$
$-$	$[E \to T\{\{+\|.-\}T\}^*]$	$[T \to F\{\{*\|/\}F\}^*.]$
$\#$	$[E \to T\{\{+\|-\}T\}^*.]$	$[T \to F\{\{*\|/\}F\}^*.]$
$)$	$[E \to T\{\{+\|-\}T\}^*.]$	$[T \to F\{\{*\|/\}F\}^*.]$
$*$	error	$[T \to F\{\{.*\|/\}F\}^*]$
$/$	error	$[T \to F\{\{*\|./\}F\}^*]$

Note that the construction of the table uses compression; from the item $[E \to T.\{\{+|-\}T\}^*]$ a direct transition under $+$ into the item $[E \to T\{\{.+|-\}T\}^*]$ was entered. Analogously for $-$ and for the item $[T \to F.\{\{*|/\}F\}^*]$ under $*$ and $/$. Thereby, all items of the form $[E \to T\{.\{+|-\}T\}^*]$ and $[T \to F\{.\{*|/\}F\}^*]$ can be eliminated, and at compile time, the corresponding transitions can be saved. □

Recursive descent RLL(1) Parser (Program Version)

A popular implementation method of RLL(1) parsers is in the form of a program. This implementation can be automatically generated from an RLL(1)-grammar and its first_1- and follow_1-sets, but it can also

be written in the programming language of one's choice. The latter is the implementation method as long as no generator tool is available.

Let a right-regular context-free grammar $G = (V_N, V_T, p, S)$ with $V_N = \{X_0, \ldots, X_n\}$, $S = X_0$, $p = \{X_0 \mapsto \alpha_0, X_1 \mapsto \alpha_1, \ldots, X_n \mapsto \alpha_n\}$ be given. We present recursive functions *p_progr* and *progr* that generate a so-called *recursive descent parser* from the grammar G and the computed first_1- und follow_1-sets.

For each production, this also means for each nonterminal X, a procedure with the name X is generated. The constructors for regular expressions on the right sides are translated into programming language constructs such as switch-, while-, do-while-statements, into checks for terminal symbols, and into recursive calls of procedures for nonterminals. The first_1- and follow_1-sets of occurrences of regular expressions are needed, for instance, to select the right one of several alternatives. Such an occurrence of a regular (sub-) expression corresponds exactly to an extended context-free item. The function *progr* is, therefore, recursively defined over the structure of context-free items of the grammar G. The following function *FiFo* is used in the case distinction for alternatives. $\mathit{FiFo}([X \to \cdots .\beta \cdots]) = \mathsf{first}_1(\beta) \oplus_1 \mathsf{follow}_1([X \to \cdots .\beta \cdots])$.

```
struct symbol nextsym;

/* Stores next input symbol in nextsym */
void scan();

/* Prints the error message and
   stops the run of the parser */
void error(String errorMessage);

/* Announces the end of the analysis and
   stops the run of the parser */
void accept();

/* Translating the input grammar */
p_progr(X_0 → α_0);
p_progr(X_1 → α_1);
        ⋮
p_progr(X_n → α_n);

void parser() {
  scan();
  X_0();

  if(nextsym == "#")
    accept();
  else
    error("...");
}

/* For all rules like this... */
p_progr(X → .α)

/* ...we create an according method like this.*/
void X() {
  progr([X → .α]);
}

void progr([X → ···.(α_1|α_2|···|α_{k-1}|α_k)···]) {
```

```
  switch () {
    case (FiFo([X → ···(.α1|α2|···|αk−1|αk)···])
                .contains(nextsym)):
      progr([X → ···(.α1|α2|···|αk−1|αk)···]);
    break;
    case (FiFo([X → ···(α1|.α2|···|αk−1|αk)···])
                .contains(nextsym)):
      progr([X → ···(α1|.α2|···|αk−1|αk)···]);
    break;
        ⋮
    case (FiFo([X → ···(α1|α2|···|.αk−1|αk)···])
                .contains(nextsym)):
      progr([X → ···(α1|α2|···|.αk−1|αk)···]);
    break;
    default:
      progr([X → ···(α1|α2|···|αk−1|.αk)···]);
  }
}

void progr([X → ···.(α1α2···αk)···]) {
  progr([X → ···(.α1α2···αk)···]);
  progr([X → ···(α1.α2···αk)···]);
        ⋮
  progr([X → ···(α1α2···.αk)···]);
}

void progr([X → ···.(α)*···]) {
  while(FIRST1(α).contains(nextsym)) {
    progr([X → ···.α···]);
  }
}

void progr([X → ···.(α)+···]) {
  do {
    progr([X → ···.α···]);
  } while(FIRST1(α).contains(nextsym));
}

void progr([X → ···.ϵ···]) {}
```

For $a \in V_T$ is

```
void progr([X → ···.a···]) {
  if(nextsym == a)
    scan();
  else
    error("...");
}
```

For $Y \in V_N$ is

```
void progr([X → ···.Y···]) = void Y()
```

How does such a parser work? Procedure X for a nonterminal X is in charge of recognizing words for X. When it is called, the first symbol of the word to recognize has already been read by the combi-

nation scanner/screener, the procedure *scan*. When procedure X has found a word for X and returns, it has already read the symbol following the found word.

The next section describes several modifications for the handling of syntax errors.

We now present the recursive descent parsers for the right-regular context-free grammar G for arithmetic expressions.

Example 3.3.14 (Continuation of Example 3.3.8)
The following parser results from the schematic translation or the extended expression grammar. For terminal symbols their string representation is used.

```
symbol nextsym;

/* Returns next input symbol */
symbol scan();

/* Prints the error message and
   stops the run of the parser */
void error(String errorMessage);

/* Announces the end of the analysis and
   stops the run of the parser */
void accept();

void S() {
  E();
}

void E() {
  T();
  while(nextsym == "+" || nextsym == "-") {
    switch (nextsym) {
      case "+":
        if(nextsym == "+")
          scan();
        else
          error("+ expected");
      break;
      default:
        if(nextsym == "-")
          scan();
        else
          error("- expected");
    }

    T();
  }
}

void T() {
  F();
  while(nextsym == "*" || nextsym == "/") {
    switch (nextsym) {
      case "*":
        if(nextsym == "*")
          scan();
```

```
        else
          error("* expected");
      break;
      default:
        if(nextsym == "/")
          scan();
        else
          error("/ expected");
    }

    F();
  }
}

void F() {
  switch (nextsym) {
    case "(":
      E();
      if(nextsym == ")")
        scan();
      else
        error(") expected");
    default:
      if(nextsym == "id")
        scan();
      else
        error("id expected");
  }
}

void parser() {
  scan();
  S();
  if(nextsym == "#")
    accept();
  else
    error("# expected");
}
```

Some inefficiencies result from the schematic generation of this parser program. A more sophisticated generation scheme will avoid most of these inefficiencies.

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 do two kinds of operations:

- Read the next input symbol (*shift*), and
- Reduce the right side of a production $X \rightarrow \alpha$ at the top of the stack 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. The result of the successful

analysis of an input word is a rightmost derivation in reverse order because *shift-reduce* parsers always reduce at the top of the stack.

A *shift-reduce* parser must never miss a *required* reduction, that is, cover it in the stack by a newly read input symbol. A reduction is *required*, if no rightmost derivation to the start symbol is possible without it. A right side covered by an input symbol will never reappear at the top of the stack and can, therefore, never be reduced. A right side at the top of the stack 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 stack are handles. Some reductions performed at the top of the stack lead into dead ends, that is, they can not 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:

$$\begin{aligned} S &\rightarrow E \\ E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \mathsf{Id} \end{aligned}$$

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. □

Stack	input	Erroneous alternative actions
	$\mathsf{Id} * \mathsf{Id}$	
Id	$* \mathsf{Id}$	
F	$* \mathsf{Id}$	Reading of $*$ misses a required reduction
T	$* \mathsf{Id}$	reduction of T to E leads into a dead end
$T *$	Id	
$T * \mathsf{Id}$		
$T * F$		reduction of F to T leads into a dead end
T		
E		
S		

Table 3.5. A successful analysis of the word $\mathsf{Id} * \mathsf{Id}$ together with potential dead ends.

Bottom-up parsers construct the parse tree from the *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 by attaching the subtrees for the right side α of a production $X \rightarrow \alpha$ below a newly created X node upon a reduction by this production. The analysis is successful if a parse tree with root label S, the start symbol of the grammar, has been constructed for the whole input word.

Fig. 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 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*, $LR(k)$ analysis. The letter L says that the parsers of this class read their input from left to right, The R characterizes

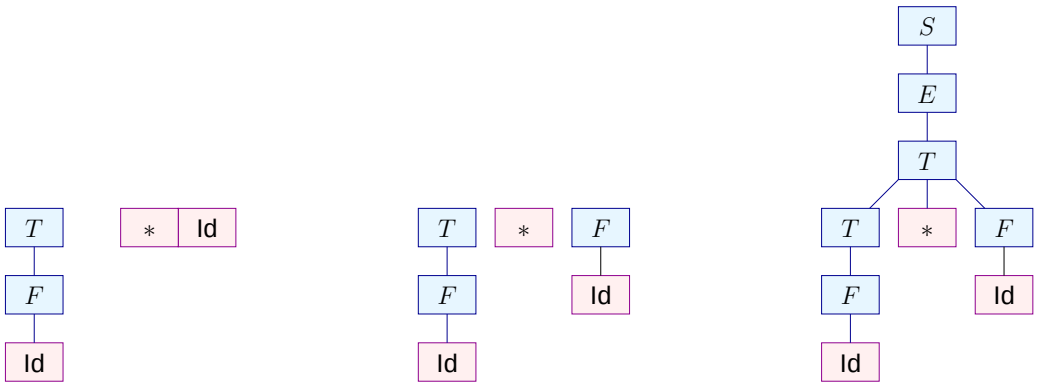

Fig. 3.13. Construction of the parse tree after reading the first symbol, Id, together with the remaining input, before the reduction of the handle $T * F$, and the complete parse tree.

them as Right parser; k is the length of the considered lookahead.

We start again with the item-pushdown automaton P_G for a context-free grammar G and transform it into a *shift-reduce* parser. Let us look back at what we did in the case of *top-down* analysis. Sets of lookahead words were computed from the grammar, which were 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 could be several productions by which to reduce, and a shift could 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 consider any lookahead. Section 3.4.3 presents the *canonical* $LR(k)$ parser. In Section 3.4.3, less powerful variants of $LR(k)$ are described, which are often powerful enough for practice. Finally, Section 3.4.4 describes a error recovery method for $LR(k)$. Note that all context-free grammars are assumed to be reduced of non-productive and unreachable nonterminals and extended by a new start symbol.

The Characteristic Finite-state Machine to a Context-free Grammar

We attempt to represent P_G by a non-deterministic finite-state machine, its *characteristic finite-state machine*, $ch(G)$. Since P_G is a pushdown automaton, this cannot easily work. An additional specification of actions on the stack is necessary. These are associated with some states and some transitions of $ch(G)$.

Our goal is to arrive at a pushdown automaton who pursues all potential expansion and read transitions of the item pushdown-automaton in parallel and only at reduction decides which production is the one to select. We define the *characteristic* finite-state machine $ch(G)$ to a reduced context-free grammar G. The states of the characteristic finite-state machine $ch(G)$ are the items $[A \to \alpha.\beta]$ of the grammar G, that is, the states of the item pushdown-automaton P_G. The set of input symbols of the characteristic finite-state machine $ch(G)$ is $V_T \cup V_N$, its initial state is the start item $[S' \to .S]$ of the item pushdown-automaton P_G. The final states of the characteristic finit-state machine are the complete items $[X \to \alpha.]$. Such a final state signals that the word just read corresponds to a stack contents of the item pushdown-automaton in which a reduction with the production $A \to \alpha$ can be performed. The transition relation Δ of the characteristic finite-state machine consists of the transitions:

$$\begin{array}{ll} ([X \to \alpha.Y\beta], \varepsilon, [Y \to .\gamma]) & \text{for} \quad X \to \alpha Y\beta \in P, \quad Y \to \gamma \in P \\ ([X \to \alpha.Y\beta], Y, [X \to \alpha Y.\beta]) & \text{for} \quad X \to \alpha Y\beta \in P, \quad Y \in V_N \cup V_T \end{array}$$

Reading a terminal symbols a in $\mathsf{char}(G)$ corresponds to a *shift* transition of the item pushdown-automaton under a. ε transitions of $\mathsf{char}(G)$ correspond to the expansion transitions of the item

pushdown-automaton. When $\mathsf{char}(G)$ reaches a final state $[X \to \alpha.]$ P_G undertakes the following actions: it removes the item $[X \to \alpha.]$ on top of its stack and makes a transition under X from the new state that has appears on top of the stack. This is a reduction move of the item pushdown-automaton P_G.

Example 3.4.2 Let G_0 again be the grammar for arithmetic expressions with the productions

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

Fig. 3.14 shows the characteristic finite-state machine to grammar G_0. □

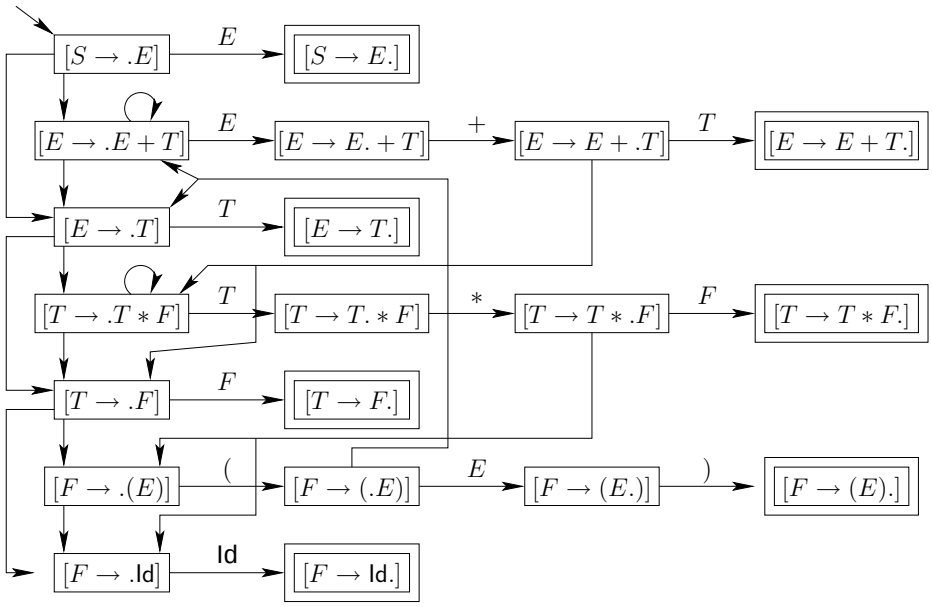

Fig. 3.14. The characteristic finite-state machine $\mathsf{char}(G_0)$ for the grammar G_0.

The following theorem clarifies the exact relation between the characteristic finite-state machine and the item pushdown automaton:

Theorem 3.4.1 Let G be a context-free grammar 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-state machine $\mathsf{char}(G)$.
2. There exists a computation $(\rho\,[A \to \alpha.\beta], w) \vdash^*_{P_G} ([S' \to S.], \varepsilon)$ of the item pushdown-automaton 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-state machine $\mathsf{char}(G)$ are exactly the histories of stack contents of the item pushdown-automaton 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 item pushdown-automaton for an input word w that starts with a stack contents ρ corresponds to a rightmost derivation that leads to a sentential form αw where α is the history of the stack contents ρ.

We introduce some terminology before we prove Theorem 3.4.1. For a rightmost derivation $S' \overset{*}{\underset{rm}{\Longrightarrow}} \gamma' Av \underset{rm}{\Longrightarrow} \gamma\alpha v$ and a production $A \to \alpha$ we call α the *handle* of the right sentential form $\gamma\alpha v$. Is the right side $\alpha = \alpha'\beta$, the prefix $\gamma = \gamma'\alpha'$ is called a *reliable prefix* of G for the item $[A \to \alpha'.\beta]$. The item $[A \to \alpha.\beta]$ is *valid* for γ. Theorem 3.4.1, thus, means, that the set of words under which the characteristic finite-state machine reaches an item $[A \to \alpha'.\beta]$ 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 a non-ambiguous 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.

relaible prefix	valid item	reason
$E+$	$[E \to E + .T]$	$S \underset{rm}{\Longrightarrow} E \underset{rm}{\Longrightarrow} E + T$
	$[T \to .F]$	$S \overset{*}{\underset{rm}{\Longrightarrow}} E + T \underset{rm}{\Longrightarrow} E + F$
	$[F \to .\mathsf{Id}]$	$S \overset{*}{\underset{rm}{\Longrightarrow}} E + F \underset{rm}{\Longrightarrow} E + \mathsf{Id}$
$(E + ($	$[F \to (.E)]$	$S \overset{*}{\underset{rm}{\Longrightarrow}} (E + F) \underset{rm}{\Longrightarrow} (E + (E))$
	$[T \to .F]$	$S \overset{*}{\underset{rm}{\Longrightarrow}} (E + (.T) \underset{rm}{\Longrightarrow} (E + (F))$
	$[F \to .\mathsf{Id}]$	$S \overset{*}{\underset{rm}{\Longrightarrow}} (E + (F) \underset{rm}{\Longrightarrow} (E + (\mathsf{Id}))$

□

Has, in the attempt to construct a rightmost derivation for a word, the prefix u of the word been reduced to a reliable prefix γ, then each item $[X \to \alpha.\beta]$, 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 $X \to \alpha\beta$ is one of the possibly just processed productions. All such productions are candidates for later reductions.

Consider the rightmost derivation

$$S' \overset{*}{\underset{rm}{\Longrightarrow}} \gamma Aw \underset{rm}{\Longrightarrow} \gamma\alpha\beta w$$

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,

$$S' \overset{*}{\underset{rm}{\Longrightarrow}} \gamma Aw \underset{rm}{\Longrightarrow} \gamma\alpha\beta w \overset{*}{\underset{rm}{\Longrightarrow}} \gamma\alpha v w \overset{*}{\underset{rm}{\Longrightarrow}} \gamma u v w \overset{*}{\underset{rm}{\Longrightarrow}} x u v w$$

We now consider this rightmost derivation in the direction of reduction, that is, in the direction in which a *bottom-up* parser constructs it. 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 which language is accepted by the characteristic finite-state machine of P_G. Theorem 3.4.1 says that chG goes under a reliable prefix into 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 $(1) \Rightarrow (2) \Rightarrow (3) \Rightarrow (1)$. Let us first assume $([S' \to .S], \gamma) \vdash^*_{\mathsf{char}(G)} ([A \to \alpha.\beta], \varepsilon)$. By induction over the number n of ε transitions we construct a rightmost derivation $S' \underset{*}{\overset{rm}{\Longrightarrow}} \gamma A w \overset{rm}{\Longrightarrow} \gamma\alpha\beta w$.

Ist $n = 0$, dann ist $\gamma = \varepsilon$ und $[A \to \alpha.\beta] = [S' \to .S]$. Da $S' \underset{*}{\overset{rm}{\Longrightarrow}} S'$ gilt, ist die Behauptung in diesem Fall erf"ullt. Ist $n > 0$, betrachten wir den letzten ε-"Ubergang. Dann l"asst sich die Berechnung of the characteristic automaton zerlegen in:

$$([S' \to .S], \gamma) \vdash^*_{\mathsf{char}(G)} ([X \to \alpha'.A\beta'], \varepsilon) \vdash_{\mathsf{char}(G)} ([A \to .\alpha\beta], \alpha) \vdash^*_{\mathsf{char}(G)} ([A \to \alpha.\beta], \varepsilon)$$

where $\gamma = \gamma'\alpha$. Nach Induktionsannahme gibt es eine rightmost derivation $S' \underset{*}{\overset{rm}{\Longrightarrow}} \gamma'' X w' \overset{rm}{\Longrightarrow} \gamma''\alpha' A\beta' w'$ mit $\gamma' = \gamma''\alpha'$. Da die grammar G reduziert ist, gibt es ebenfalls eine rightmost derivation $\beta' \underset{*}{\overset{rm}{\Longrightarrow}} v$. Deshalb haben wir:

$$S' \underset{*}{\overset{rm}{\Longrightarrow}} \gamma' A v w' \overset{rm}{\Longrightarrow} \gamma'\alpha\beta w$$

mit $w = vw'$. Damit ist die Richtung $(1) \Rightarrow (2)$ bewiesen.

Nehmen wir an, wir h"atten eine rightmost derivation $S' \underset{*}{\overset{rm}{\Longrightarrow}} \gamma' A w \overset{rm}{\Longrightarrow} \gamma'\alpha\beta w$. Diese Ableitung l"asst sich zerlegen in:

$$S' \overset{rm}{\Longrightarrow} \alpha_1 X_1 \beta_1 \underset{*}{\overset{rm}{\Longrightarrow}} \alpha_1 X_1 v_1 \underset{*}{\overset{rm}{\Longrightarrow}} \ldots \underset{*}{\overset{rm}{\Longrightarrow}} (\alpha_1 \ldots \alpha_n) X_n (v_n \ldots v_1) \overset{rm}{\Longrightarrow} (\alpha_1 \ldots \alpha_n)\alpha\beta(v_n \ldots v_1)$$

for $X_n = A$. Mit Induktion nach n folgt, dass $(\rho, vw) \vdash^*_{K_G} ([S' \to S.], \varepsilon)$ gilt for

$$\begin{aligned} \rho &= [S' \to \alpha_1.X_1\beta_1] \ldots [X_{n-1} \to \alpha_n.X_n\beta_n] \\ w &= vv_n \ldots v_1 \end{aligned}$$

sofern $\beta \underset{rm}{\overset{*}{\Longrightarrow}} v$, $\alpha_1 = \beta_1 = \varepsilon$ and $X_1 = S$. Damit ergibt sich der Schluss $(2) \Rightarrow (3)$.

F"ur den letzten Schluss betrachten wir einen Kellerinhalt $\rho = \rho'\ [A \to \alpha.\beta]$ mit $(\rho, w) \vdash^*_{K_G} ([S' \to S.], \varepsilon)$. Zuerst "uberzeugen wir uns mit Induktion nach der Anzahl der "Uberg"ange in einer solchen Berechnung, dass ρ' notwendigerweise von der Form:

$$\rho' = [S' \to \alpha_1.X_1\beta_1] \ldots [X_{n-1} \to \alpha_n.X_n\beta_n]$$

ist for ein $n \geq 0$ and $X_n = A$. Mit Induktion nach n folgt aber, dass $([S' \to .S], \gamma) \vdash^*_{\mathsf{char}(G)} ([A \to \alpha.\beta], \varepsilon)$ gilt for $\gamma = \alpha_1 \ldots \alpha_n\alpha$. Da $\gamma = \mathsf{hist}(\rho)$, gilt auch die Behauptung (1). Damit ist der Beweis vollst"andig. □

The Canonical $LR(0)$ Automaton

In Chapter 2, we presented an algorithm which takes a non-deterministic finite-state machine and constructs an equivalent deterministic finite-state machine. This deterministic finite-state machine pursues all paths in parallel which the non-deterministic automaton could take for a given input. Its states are sets of states of the non-deterministic automaton. This *subset construction* is now applied to the characteristic finite-state machine $\mathsf{char}(G)$ of a context-free grammar G. The resulting deterministic finite-state machine is called the *canonical* $LR(0)$ automaton for G and denote it by $LR_0(G)$.

Example 3.4.5 The canonical $LR(0)$ automaton for the context-free grammar G_0 of Example 3.2.2 on page 39 is obtained by the application of the subset construction to the characteristic finite-state machine $\mathsf{char}(G_0)$ of Fig. 3.14 on page 82. It is shown in Fig. 3.15 on page 85. It states are:

$$
\begin{aligned}
S_0 &= \{\ [S \to .E], \\
&\qquad [E \to .E + T], \\
&\qquad [E \to .T], \\
&\qquad [T \to .T * F], \\
&\qquad [T \to .F], \\
&\qquad [F \to .(E)], \\
&\qquad [F \to .\mathsf{Id}]\ \} \\
S_1 &= \{\ [S \to E.], \\
&\qquad [E \to E. + T]\ \} \\
S_2 &= \{\ [E \to T.], \\
&\qquad [T \to T. * F]\ \} \\
S_3 &= \{\ [T \to F.]\ \}
\end{aligned}
\qquad
\begin{aligned}
S_4 &= \{\ [F \to (.E)], \\
&\qquad [E \to .E + T], \\
&\qquad [E \to .T], \\
&\qquad [T \to .T * F] \\
&\qquad [T \to .F] \\
&\qquad [F \to .(E)] \\
&\qquad [F \to .\mathsf{Id}]\ \} \\
S_5 &= \{\ [F \to \mathsf{Id}.]\ \} \\
S_6 &= \{\ [E \to E + .T], \\
&\qquad [T \to .T * F], \\
&\qquad [T \to .F], \\
&\qquad [F \to .(E)], \\
&\qquad [F \to .\mathsf{Id}]\ \}
\end{aligned}
\qquad
\begin{aligned}
S_7 &= \{\ [T \to T * .F], \\
&\qquad [F \to .(E)], \\
&\qquad [F \to .\mathsf{Id}]\ \} \\
S_8 &= \{\ [F \to (E.)], \\
&\qquad [E \to E. + T]\ \} \\
S_9 &= \{\ [E \to E + T.], \\
&\qquad [T \to T. * F]\ \} \\
S_{10} &= \{\ [T \to T * F.]\} \\
S_{11} &= \{\ [F \to (E).]\ \} \\
S_{12} &= \emptyset
\end{aligned}
$$

□

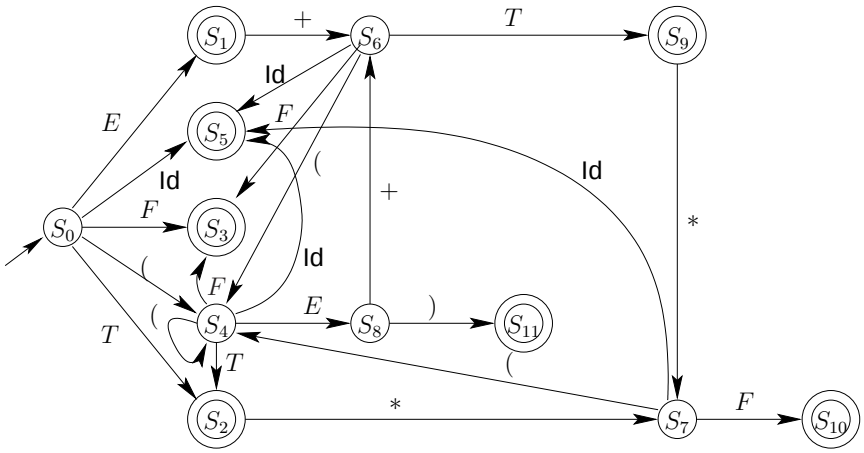

Fig. 3.15. The transition diagram of the $LR(0)$ automaton for the grammar G_0 obtained from the characteristic finite-state machine$\mathsf{char}(G_0)$ in Fig. 3.14. The error state $S_{12} = \emptyset$ and all transitions into it are left out.

The canonical $LR(0)$ automaton $LR_0(G)$ to a context-free grammar G has some 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. When a reduction is possible that will again lead to a right sentential-form This 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:

$$\begin{array}{llll} F\,, & (F\,, & ((F\,, & (((F\,, \quad \ldots \\ T*(F\,, & T*((F\,, & T*(((F\,, & \ldots \\ E+F\,, & E+(F\,, & E+((F\,, & \ldots \end{array}$$

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 $E+$ is a prefix of the right sentential form $E+T$:

$$\begin{array}{lccccccc} S \underset{rm}{\Longrightarrow} E \underset{rm}{\Longrightarrow} & E+T & \underset{rm}{\Longrightarrow} & E+F & \underset{rm}{\Longrightarrow} & E+\mathsf{Id} \\ & \uparrow & & \uparrow & & \uparrow \\ \text{Valid are for instance} & [E \to E+.T] & & [T \to .F] & & [F \to .\mathsf{Id}] \end{array}$$

□

The canonical $LR(0)$ automaton $LR_0(G)$ to a context-free grammar G is a deterministic finite-state machine 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 item-pushdown automaton) this parser stores on its stack states of the canonical $LR(0)$ automaton, that is *sets* of items. The underlying pushdown automata 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 stack. It can only be taken if at least one item of the form $[X \to \alpha.a\beta]$ is contained in q.

Reduce: $(qq_1 \ldots q_n, \varepsilon, q\,\delta_G(q,X)) \in \Delta$ if $[X \to \alpha.] \in q_n$ holds with $|\alpha| = n$. The complete item $[X \to \alpha.]$ in the topmost stack entry signals a potential reduction. As many entries are removed from the top of the stack as the length of the right side indicates. After that, the X successor of the new topmost stack entry is pushed onto the stack.
Fig. 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 stack. These entries are removed at reduction. The new actual state, previously below these removed entries, has a transition under X, which is now taken.

Finish: $(q_{G,0}\,q, \varepsilon, f)$ 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 stack and inserts the final state f.

The special case $[X \to .\,]$ merits special consideration. According to our description, $|\varepsilon| = 0$ topmost stack entries need to be removed from the stack 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 stack. This transition is possible since by construction it holds that with the item $[\cdots \to \cdots .X \cdots]$ also the item $[X \to .\alpha]$ is contained in state q for each right side α of nonterminal X. In the special case of a ε production, the actual state q contains together with the item $[\cdots \to \cdots .X \cdots]$ also the complete item $[X \to .\,]$. This latter reduction transition *extends the length* of the stack.

The construction of $LR_0(G)$ guarantees that for each non-initial and non-final state q there exists exactly one entry symbol under which the automaton can make a transition into q. The stack contents $q_0, \ldots, q_n$ mit $q_0 = q_{G,0}$ 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.

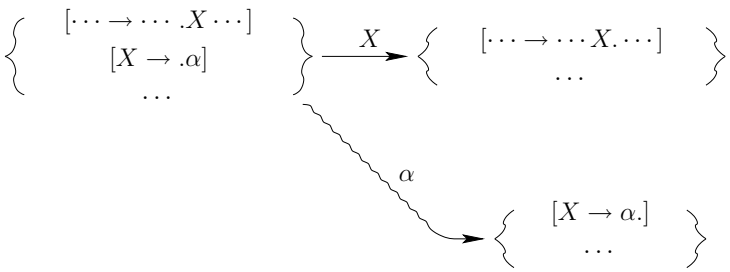

Fig. 3.16. Part of the transition diagram of a canonical $LR(0)$ automaton.

The pushdown automaton P_0 just constructed is not necessarily deterministic. There are two kinds of conflicts that cause non-determinism:

shift-reduce conflict: a state q allows a read transition under a symbol $a \in V_T$ as well as a reduce or finish transition, and

reduce-reduce conflict: a state q permits reduction transitions according to two different productions.

In the first case, the actual state contains at least one item $[X \to \alpha.a\beta]$ and at least one complete item $[Y \to \gamma.]$; in the second case, 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*. Es gilt:

Lemma 3.4. 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 non-complete 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 $w \in V_T^*$. $\square$

Inadequate states of the canonical $LR(0)$ automaton make the pushdown automata P_0 non-deterministic. We obtain deterministic parsers by permitting the parser to look ahead into the remaining input 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 $[S \to E.]$) and it can read the terminal symbol $+$ in the input (item $[E \to E. + T]$). In state S_2 the parser can reduce the right side T to E (complete item $[E \to T.]$) and it can read the terminal symbol $*$ (item $[T \to T. * F]$). In state S_9 finally, the parser can reduce the right side $E + T$ to E (complete item $[E \to E + T.]$), and it can read the terminal symbol $*$ (item $[T \to T. * F]$). $\square$

Direct Construction of the Canonical $LR(0)$ Automaton

The canonical $LR(0)$ automaton $LR_0(G)$ to a context-free grammar G needs not be derived through the construction of the characteristic finite-state machine $\mathsf{char}(G)$ and the subset construction. It can be constructed directly from G. The construction uses 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-state machine. The set $\Delta_{G,\varepsilon}(q)$ is the least solution of the following equation

$$I = q \cup \{[A \to .\gamma] \mid \exists X \to \alpha A\beta \in P :\ [X \to \alpha.A\beta] \in I\}$$

Similar to the function $\mathsf{closure}()$ 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. To implement it we specialize the function nextState() of the subset construction:

```
set⟨item⟩ nextState(set⟨item⟩ q, symbol X) {
    set⟨item⟩ q' ← ∅;
    nonterminal A; string⟨symbol⟩ α, β;
    forall (A, α, β : ([A → α.Xβ] ∈ q))
        q' ← q' ∪ {[A → αX.β]};
    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⟩ q0 ← closure({[S' → .S]});
states ← {q0}; W ← [q0];
trans ← ∅;
set⟨item⟩ q, q';
while (W ≠ []) {
    q ← hd(W); W ← tl(W);
    forall (symbol X) {
        q' ← nextState(q, X);
        trans ← trans ∪ {(q, X, q')};
        if (q' ∉ states) {
            states ← states ∪ {q'};
            W ← q' :: W;
        }
    }
}
```

3.4.3 $LR(k)$: Definition, Properties, and Examples

We call a context-free grammar 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

$$
\begin{array}{ll}
S' \overset{*}{\underset{rm}{\Longrightarrow}} \alpha X w \underset{rm}{\Longrightarrow} \alpha\beta w \text{ and} & \\
S' \overset{*}{\underset{rm}{\Longrightarrow}} \gamma Y x \underset{rm}{\Longrightarrow} \alpha\beta y \quad \text{and} & \\
w|_k = y|_k \qquad \text{implies} \qquad \alpha = \gamma \wedge X = Y \wedge x = y. &
\end{array}
$$

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 $L(G) = \{a^n 0 b^n \mid n \geq 0\} \cup \{a^n 1 b^{2n} \mid n \geq 0\}$. We know already that G is for no $k \geq 1$ an $LL(k)$-grammar. Grammar G is an $LR(0)$-grammar, though.

The right sentential forms of G have the form

$$S, \quad \underline{A}, \quad \underline{B}, \quad a^n \underline{aAb} b^n, \quad a^n \underline{aBbb} b^{2n}, \quad a^n a \underline{0} b b^n, \quad a^n a \underline{1} b b b^{2n}$$

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 $a^n aBbbb^{2n}$ One could reduce $a^n aAbb^n$ to $a^n Ab^n$ and to $a^n aSbb^n$. The first choice belonged to the rightmost derivation

$$S \overset{*}{\underset{rm}{\Longrightarrow}} a^n Ab^n \underset{rm}{\Longrightarrow} a^n aAbb^n$$

the second to no 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. The prefix $aAbb$ uniquely determines this. For the right sentential form $abb^{2n}c$, b is the handle, and the prefix ab uniquely determines this. □

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 $1 : w = b$, the handle lies in w; if $1 : w = c$, 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 $k \geq 0$. For, let k be arbitrary, but fix. Consider the two rightmost derivations

$$S \underset{rm}{\overset{*}{\Longrightarrow}} ab^n Ab^n c \underset{rm}{\Longrightarrow} ab^n bb^n c$$
$$S \underset{rm}{\overset{*}{\Longrightarrow}} ab^{n+1} Ab^{n+1} c \underset{rm}{\Longrightarrow} ab^{n+1} bb^{n+1} c$$

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 $w|_k = y|_k = b^k$. $\alpha \neq \gamma$ implies that G_3 can be no $LR(k)$-grammar. □

The following theorem clarifies the relation between the definition of $LR(0)$-grammar and the properties of the canonic $LR(0)$ automaton.

Theorem 3.4.2 A context-free grammar G is an $LR(0)$-grammar if and only if the canonical $LR(0)$ automaton for G has no $LR(0)$-inadequate states.

Proof: " $\Rightarrow$ " Let G eine $LR(0)$-grammar, and nehmen wir an, der canonical $LR(0)$ automaton $LR_0(G)$ habe einen einen $LR(0)$-inadequaten state p.

Fall 1: The state p hat einen *reduce-reduce*-conflict, d.h. p enth"alt zwei verschiedene items $[X \to \beta.]$, $[Y \to \delta.]$. Dem state p zugeordnet ist eine nichtleere Menge von reliable prefixesn. Let $\gamma = \gamma'\beta$ ein solches reliable prefix. Weil beide items valid for γ sind, gibt es rightmost derivations

$$S' \underset{rm}{\overset{*}{\Longrightarrow}} \gamma' X w \underset{rm}{\Longrightarrow} \gamma'\beta w \qquad \text{und}$$
$$S' \underset{rm}{\overset{*}{\Longrightarrow}} \nu Y y \underset{rm}{\Longrightarrow} \nu\delta y \qquad \text{mit} \quad \nu\delta = \gamma'\beta = \gamma$$

Das ist aber ein Widerspruch zur $LR(0)$-Eigenschaft.

Fall 2: state p hat einen *shift-reduce*-conflict, d.h. p enth"alt items $[X \to \beta.]$ and $[Y \to \delta.a\alpha]$. Let γ ein reliable prefix for beide item Weil beide items valid for γ sind, gibt es rightmost derivations

$$S' \underset{rm}{\overset{*}{\Longrightarrow}} \gamma' X w \underset{rm}{\Longrightarrow} \gamma'\beta w \qquad \text{und}$$
$$S' \underset{rm}{\overset{*}{\Longrightarrow}} \nu Y y \underset{rm}{\Longrightarrow} \nu\delta a\alpha y \qquad \text{mit} \quad \nu\delta = \gamma'\beta = \gamma$$

Ist $\beta' \in V_T^*$, erhalten wir sofort einen Widerspruch. Andernfalls gibt es eine rightmost derivation

$$\alpha \underset{rm}{\overset{*}{\Longrightarrow}} v_1 X v_3 \underset{rm}{\Longrightarrow} v_1 v_2 v_3$$

Weil $y \neq av_1v_2v_3y$ gilt, ist die $LR(0)$-Eigenschaft verletzt.

" $\Leftarrow$ " Nehmen wir an, der canonical $LR(0)$ automaton $LR_0(G)$ habe keine $LR(0)$-inadequaten states. Betrachten wir die zwei rightmost derivations:

$$S' \underset{rm}{\overset{*}{\Longrightarrow}} \alpha X w \underset{rm}{\Longrightarrow} \alpha\beta w$$
$$S' \underset{rm}{\overset{*}{\Longrightarrow}} \gamma Y x \underset{rm}{\Longrightarrow} \alpha\beta y$$

Zu zeigen ist, dass $\alpha = \gamma,\ X = Y,\ x = y$ gelten. Let p der state of the canonical $LR(0)$ automaton nach Lesen von $\alpha\beta$. Dann enth"alt p alle for $\alpha\beta$ valid items . Nach Voraussetzung ist p $LR(0)$-geeignet. Wir unterscheiden zwei F"alle:

Fall 1: $\beta \neq \varepsilon$. Wegen Lemma 3.4 ist $p = \{[X \to \beta.]\}$, d.h. $[X \to \beta.]$ ist das einzige valid item for $\alpha\beta$. Daraus folgt, dass $\alpha = \gamma,\ X = Y$ and $x = y$ sein muss.

Fall 2: $\beta = \varepsilon$. Nehmen wir an, die zweite rightmost derivation widerspreche der $LR(0)$-Bedingung. Dann gibt es ein weiteres item $[X \to \delta.Y'\eta] \in p$, so dass $\alpha = \alpha'\delta$ ist. The letzte Anwendung einer production in der unteren rightmost derivation ist die letzte Anwendung einer production in einer terminalen rightmost derivation for Y'. Nach Lemma 3.4 folgt daraus, dass die untere Ableitung gegeben ist durch:

$$S' \underset{rm}{\overset{*}{\Longrightarrow}} \alpha'\delta Y'w \underset{rm}{\overset{*}{\Longrightarrow}} \alpha'\delta Xvw \underset{rm}{\Longrightarrow} \alpha'\delta vw$$

wobei $y = vw$ ist. Damit gilt $\alpha = \alpha'\delta = \gamma$, $Y = X$ and $x = vw = y$ – im Widerspruch zu unserer Annahme. □

Let us conclude. We have seen how to construct the $LR(0)$ automaton $LR_0(G)$ from a given context-free grammar G. This can be done either directly of through the characteristic finite-state machine $\mathsf{char}(G)$. From the deterministic finite-state machine $LR_0(G)$ one can construct a pushdown automata P_0. This pushdown automaton P_0 is deterministic if $LR_0(G)$ does not contain $LR(0)$-inadequate states. Theorem 3.4.2 states this is exactly the case if the grammar G is an $LR(0)$-grammar. We have thereby met a method to generate parsers for $LR(0)$-grammars.

In real life, $LR(0)$-grammars are rather rare. Often lookahead of length $k > 0$ needs to be used to select between the different choices of a parsing situation. In an $LR(0)$ parser, the actual state determines what the next action is, independently of the next input symbols. $LR(k)$ parsers for $k > 0$ have states consisting of sets of items. A different kind of items are used, though, so-called $LR(k)$-items. $LR(k)$-items are context-free items, extended by lookahead 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 *lookahead* of the $LR(k)$-items 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 lookahead ε.

Example 3.4.12 Consider again grammar G_0. We have:

(1) $[E \to E + .T,)]$
$[E \to E + .T, +]$ 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 $E*$ can occur in no right sentential form. □

The folllowing theorem gives a characterization of the $LR(k)$-property based on valid $LR(k)$-items.

Theorem 3.4.3 Let G be a context-free grammar. 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 $[A \to \alpha., x] \in It(\gamma)$ holds:

1. if there is another $LR(k)$-item $[X \to \delta., y] \in It(\gamma)$, then $x \neq y$.
2. is there another $LR(k)$-item $[X \to \delta.a\beta, y] \in It(\gamma)$, then $x \notin \mathsf{first}_k(a\beta) \odot_k \{y\}$. $\square$

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 deterministic finite-state machine. Its states are sets of $LR(k)$-items. For each reliable prefix γ the deterministic finite-state machine $LR_k(G)$ 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 lookahead 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 stack, the $LR(k)$ parser stores states of the canonical $LR(k)$-automaton on is stack. The selection of the right of several possible actions of the $LR(k)$ parser is controlled by the *action*-table. This table contains for each combination of state and lookahead 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 stack. 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 stack symbols that belong to α. These two tables for $k = 1$ are shown in Fig. 3.17.

The $LR(k)$ parser for a grammar G needs a program that interprets the *action*- and *goto*-table, the *driver*. 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:

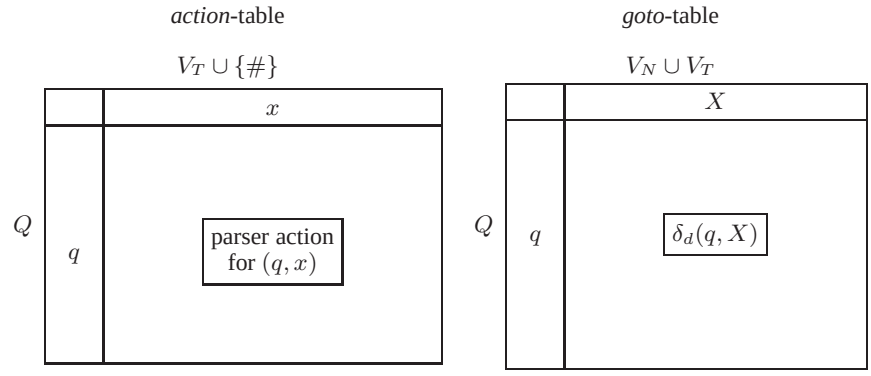

Fig. 3.17. Schematic representation of *action*- and *goto*-table of an $LR(1)$ parser with set of states Q.

```
list⟨state⟩ stack ← [q0];
terminal buffer ← scan();
state q; nonterminal X; string⟨symbol⟩ α;
while (true) {
    q ← hd(stack);
    switch (action[q, buffer]) {
    case shift :              stack ← goto[q, buffer] :: stack;
                              buffer ← scan();
                              break;
    case reduce(X → α) :      output(X → α);
                              stack ← tl(|α|, stack); q ← hd(stack);
                              stack ← goto[q, X] :: stack;
                              break;
    case accept :             stack ← f :: tl(2, stack);
                              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 context-free grammar 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 described 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 lookahead 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 lookahead 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

$$I = q \cup \{[A \to .\gamma, y] \mid \exists X \to \alpha A \beta \in P : \ [X \to \alpha.A\beta, x] \in I, y \in \mathsf{first}_1(\beta) \odot_1 \{x\}\}$$

It is computed by the following function

```
set⟨item_1⟩ closure(set⟨item_1⟩ q) {
    set⟨item_1⟩ result ← q;
    list⟨item_1⟩ W ← list_of(q);
    nonterminal X; string⟨symbol⟩ α, β; terminal x, y;
    while (W ≠ []) {
        item_1 i ← hd(W); W ← tl(W);
        switch (i) {
        case [_ → _ .Xβ, x] :
                forall (α : (X → α) ∈ P)
                    forall (y ∈ first_1(β) ⊙_1 {x})
                        if ([X → .α, y] ∉ result) {
                            result ← result ∪ {[X → .α, y]};
                            W ← [X → .α, 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

$$q_0 = \mathsf{closure}(\{[S' \to .S, \#]\})$$

We need a function $\mathsf{nextState}()$ that computes the successor state to a given set q of LR_1-items and a symbol $X \in V = V_N \cup V_T$. The corresponding function for the construction of $LR_0(G)$ needs to be extended by the compute the lookahead symbols:

```
set⟨item_1⟩ nextState(set⟨item_1⟩ q, symbol X) {
    set⟨item_1⟩ q' ← ∅;
    nonterminal A; string⟨symbol⟩ α, β; terminal x;
    forall (A, α, β, x : ([A → α.Xβ, x] ∈ q))
        q' ← q' ∪ {[A → αX.β, 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 exists 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 will decide to reduce with the production whose

lookahead symbol is the next input symbol. If state q contains at the same time a complete $LR(1)$-item $[X \to \alpha., x]$ and an $LR(1)$-item $[Y \to \beta.a\gamma, y]$, it still has no *shift-reduce*-conflict if $a \neq x$. In state q the generated parser will reduce if the next next input symbol is x and shift if it is a. Therefore, the *action*-table can be computed by the following iteration:

$$
\begin{array}{l}
\textbf{forall } (\mathit{state}\ \ q)\ \{ \\
\quad \textbf{forall } (\mathit{terminal}\ \ x)\ \ \mathit{action}[q,x] \leftarrow \mathit{error}; \\
\quad \textbf{forall } ([X \to \alpha.\beta, x] \in q) \\
\quad\quad \textbf{if } (\beta = \varepsilon) \\
\quad\quad\quad \textbf{if } (X = S' \wedge \alpha = S \wedge x = \#)\ \ \mathit{action}[q,\#] \leftarrow \mathit{accept}; \\
\quad\quad\quad \textbf{else } \ \mathit{action}[q,x] \leftarrow \mathit{reduce}(X \to \alpha); \\
\quad\quad \textbf{else if } (\beta = a\beta')\ \ \mathit{action}[q,a] \leftarrow \mathit{shift}; \\
\}
\end{array}
$$

Example 3.4.13 We consider some states of the canonical $LR(1)$-automaton for the context-free grammar 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 lookahead symbols in $LR(1)$-items from S with the same kernel $[A \to \alpha.\beta]$ are collected in one lookahead set

$$L = \{x \mid [A \to \alpha.\beta, x] \in q\}$$

We represent subsets $\{[A \to \alpha.\beta, x] \mid x \in L\}$ as $[A \to \alpha.\beta, L]$ and obtain

$$
\begin{array}{lll}
S'_0 = & \mathsf{closure}(\{[S \to .E, \{\#\}]\}) & S'_6 = \quad \mathsf{nextState}(S'_1, +) \\
= \{ & [S \to .E, \{\#\}] & = \{\ [E \to E + .T, \{\#, +\}], \\
& [E \to .E + T, \{\#, +\}], & \quad [T \to .T * F, \{\#, +, *\}], \\
& [E \to .T, \{\#, +\}], & \quad [T \to .F, \{\#, +, *\}], \\
& [T \to .T * F, \{\#, +, *\}], & \quad [F \to .(E), \{\#, +, *\}], \\
& [T \to .F, \{\#, +, *\}], & \quad [F \to .\mathsf{Id}, \{\#, +, *\}]\ \} \\
& [F \to .(E), \{\#, +, *\}], & \\
& [F \to .\mathsf{Id}, \{\#, +, *\}]\ \} & S'_9 = \quad \mathsf{nextState}(S'_6, T)) \\
& & = \{\ [E \to E + T., \{\#, +\}], \\
S'_1 = & \mathsf{nextState}(S'_0, E) & \quad [T \to T. * F, \{\#, +, *\}]\ \} \\
= \{ & [S \to E., \{\#\}], & \\
& [E \to E. + T, \{\#, +\}]\ \} & \\
& & \\
S'_2 = & \mathsf{nextState}(S'_1, T) & \\
= \{ & [E \to T., \{\#, +\}], & \\
& [T \to T. * F, \{\#, +, *\}]\ \} &
\end{array}
$$

After the extension by lookahead symbols, the states S_1, S_2 and S_9, which were $LR(0)$ inadequate, have no longer conflicts. In state S'_1 the next input symbol $+$ indicates to shift, the next input symbol $\#$ indicates to reduce. In state S'_2 lookahead symbol $*$ indicates to shift, $\#$ and $+$ to reduce; similarly in state S'_9.

The 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 $LALR(1)$ parser

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

	Id	(	)	$*$	$+$	$\#$
S_0'	s	s				
S_1'					s	*acc*
S_2'				s	$r(3)$	$r(3)$
S_6'	s	s				
S_9'				s	$r(2)$	$r(2)$

The used numbering of the productions:

$1 : S \to E$
$2 : E \to E + T$
$3 : E \to T$
$4 : T \to T * F$
$5 : T \to F$
$6 : F \to (E)$
$7 : F \to \mathsf{Id}$

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*.

methods are the $SLR(1)$- (*simple* LR-) and $LALR(1)$- (*lookahead* LR-)methods. Ist $SLR(1)$ 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. We denote by $\lambda(q, [X \to \alpha.\beta])$ the lookahead set that is added to the item $[X \to \alpha.\beta]$ in q. The $SLR(1)$-method is different from the $LALR(1)$-method in the definition of the function

$$\lambda : Q \times \mathcal{I}_G \to 2^{V_T \cup \{\#\}}$$

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 lookahead sets $\lambda(q, [X \to \alpha.])$ suffice to construct an *action*-table zu.

In $SLR(1)$ parsers, the lookahead sets for items are independent of the states in which they occur; the lookahead only depends on the left side of the production in the item:

$$\lambda_S(q, [X \to \alpha.\beta]) = \{a \in V_T \cup \{\#\} \mid S'\# \stackrel{*}{\Longrightarrow} \gamma X a w\} = \mathsf{follow}_1(X)$$

for alle states q mit $[X \to \alpha.] \in q$. 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. 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. Its canonical $LR(0)$ automaton $LR_0(G_0)$ has the inadequate states S_1, S_2 and S_9. We extend the complete items in the states by the follow_1-sets of their left sides to represent the function λ_S in a readable way. Since $\mathsf{follow}_1(S) = \{\#\}$ and $\mathsf{follow}_1(E) = \{\#, +,)\}$ we obtain:

$$S_1'' = \{\ [S \to E., \{\#\}], \quad [E \to E. + T]\} \qquad \text{conflict eliminated, da } + \notin \{\#\}$$

$$S_2'' = \{\ [E \to T., \{\#, +,)\}], \quad [T \to T. * F]\ \} \qquad \text{conflict eliminated, da } * \notin \{\#, +,)\}$$

$$S_9'' = \{\ [E \to E + T., \{\#, +,)\}], \quad [T \to T. * F]\ \} \qquad \text{conflict eliminated, da } * \notin \{\#, +,)\}$$

So, G_0 i an $SLR(1)$-grammar and it 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* lookahead 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 canonic $LR(0)$ 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 sets of right sentential forms appear in it that are in general infinite. The sets $\lambda_L(q, [A \to \alpha.\beta])$ can be characterized as the least solution of the following system of equations:

$$\begin{aligned}
\lambda_L(q_0, [S' \to .S]) &= \{\#\} \\
\lambda_L(q, [A \to \alpha X.\beta]) &= \bigcup\{\lambda_L(p, [A \to \alpha.X\beta]) \mid \Delta_G(p, X) = q\}\,, \qquad X \in (V_T \cup V_N) \\
\lambda_L(q, [A \to .\alpha]) &= \bigcup\{\mathsf{first}_1(\beta) \odot_1 \lambda_L(q, [X \to \gamma.A\beta]) \mid [X \to \gamma.A\beta] \in q'\}
\end{aligned}$$

The system of equations describes how sets of successor symbols of items in states originate. The first equation says that only $\#$ can follow the start symbol S'. The second class of equations describes that the follow symbols of an item $[A \to \alpha X.\beta]$ in a state q result from the follow symbols after the dot in an 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 after the dot, that is, from 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 the computation of least solutions. Considering which nonterminal may produce ε allows us to replace the occurrences of 1-concatenation by unions. We so obtain an equivalent pure union problem that can be solved by the efficient method of Section 3.2.7.

$LALR(1)$ parsers can be constructed in the following, not very efficient way: One constructs a canonical $LR(1)$ parser. If its states have no conflicts such states p and q are merged to a new state p' where the cores of the items in p are the same as the cores in the items of q, that is, where the difference of the two sets of items consists only in the lookahead sets. The lookahead sets in the new state p' are obtained as the union of the lookahead sets of items with the same core. The grammar is an $LALR(1)$-grammar if the new states have no conflicts.

A further possibility consists in the modification of Algorithm *LR(1)-GEN*. The conditional statement

if q' **not in** Q **then** $Q := Q \cup \{q'\}$ **fi**;

is replaced by

if *exist.* q'' **in** Q *mit kerngleich*(q', q'') **then** *verschmelze*(Q, q', q'') **fi**;

where

function *samecore*$(p, p'$: **set of item**$)$:**bool**;
  if *set of cores of* p = *set of cores of* p'
    then return (**true**)
    else return (**false**)

fi;

proc *merge*(Q : **set of set of item,** p, p' : **set of item**);
$Q := Q \cup \{[X \to \alpha.\beta, L_1 \cup L_2] \mid [X \to \alpha.\beta, L_1] \in p$ *und* $[X \to \alpha.\beta, L_2] \in p'\}$.

Example 3.4.15 The following grammar taken from [ASU86] describes a simplified version of the C assignment statement:

$$\begin{aligned} S' &\to S \\ S &\to L = R \mid R \\ L &\to *R \mid \mathsf{Id} \\ R &\to L \end{aligned}$$

This grammar is not an $SLR(1)$-grammar, but t is 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 lookahead 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 lookahead set. Therefore, the grammar is not an $SLR(1)$-grammar.

The grammar however is a $LALR(1)$-grammar. The transition diagram of its $LALR(1)$ parser is shown in Fig. 3.18. To increase readability, the lookahead sets $\lambda_L(q, [A \to \alpha.\beta])$ were directly associated with the item $[A \to \alpha.\beta]$ of state q. In state S_2, the item $[R \to L.]$ has now the lookahead set $\{\#\}$. The conflict is resolved since this set does not contain the next input symbol $=$. □

3.4.4 Fehlerbehandlung in LR parsern

LR parser besitzen ebenso wie LL parser die Eigenschaft of the fortsetzungsf"ahigen Pr"afixes. Das bedeutet, dass jedes durch einen LR parser fehlerfrei analysierte prefix der input zu einem korrekten inputwort, einem Satz der Sprache, fortgesetzt werden kann. Trifft ein LR parser in einer Konfiguration auf ein input symbol a mit $action[q, a] = \mathsf{error}$, ist dies die fr"uhestm"ogliche Situation, in der ein Fehler entdeckt werden kann. Diese Konfiguration nennen wir *Fehlerkonfiguration* und q den *Fehlerzustand* dieser Konfiguration. Auch for LR parser gibt es ein Spektrum von Fehlerbehandlungsverfahren:

- Vorw"artsfehlerbehandlung. Modifikationen werden in der restlichen input, nicht aber auf dem Parserkeller vorgenommen.
- R"uckw"artsfehlerbehandlung. Modifikationen werden auch auf dem Parserkeller vorgenommen.

Nehmen wir an, q sei der aktuelle state und a das next Symbol in der input. Als mögliche Korrekturen bieten sich die Aktionen ein verallgemeinertes *shift*(βa) für ein item $[A \to \alpha.\beta a\gamma]$ aus q, ein *reduce* für unvollständige items aus q oder *skip* an:

- The Korrektur *shift*(βa) nimmt an, dass das Teilwort zu β ausgefallen ist. Es kellert deshalb die Zustände, die der item-pushdown automaton bei Lesen der Symbolfolge β von q aus durchläuft. Anschließend wird das Symbol a gelesen and der entsprechende *shift*-Übergang des Parsers ausgeführt.