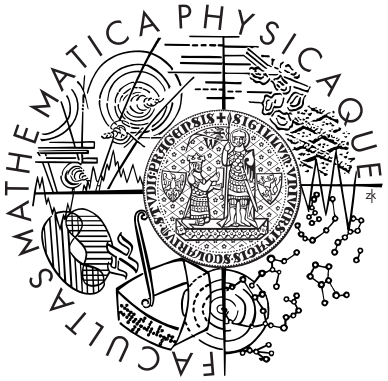
Charles University

Faculty of Mathematics and Physics



Principles of Kind Parsing

Ph.D. Thesis

RNDr. Michal Žemlička

Department of Software Engineering
Malostranské náměstí 25
Prague, Czech Republic

Supervisor: Prof. RNDr. Jaroslav Král, DrSc.

June 2006

Abstract:

A new modification of (semi)top-down parsing called kind parsing is presented in this work. Kind parsers with lookahead k, $k \geq 1$, work for a calls of kind grammars denoted K(k). These grammars are proper superclasses of SLL(k) grammars and proper subclasses of LR(k) grammmars for the same k. Moreover K(k) grammars generate LL(k) languages. Parsing based on kind grammars (kind parsing) is a parsing technique very close to top-down parsing; it can be viewe as an almost LL(k) parser able to parse some languages generated by grammars with specific left recursive rules. Kind parsers have many preferable properties of LL(k) parsers. Kind parsing works for a hierarchy of grammar classes that are strictly between SLL(k) and LR(k) and generates LL(k) languages. Kind parsing is probably the first top-down parsing technique (or very close to top-down parsing) that is able to handle effectively left recursive productions. Kind parsers can be very easily extended (even in parse-time) or modified by hand (in the case of persistent parsers). This parsing technique has many software engineering advantages: it is easy to learn, the parsers are easy to read and to modify manually. Manual modification/optimization is usually necessary for parsers of commercial quality, semantic actions can be called almost everywhere inside phrases. Parsers can be on-line extensible. This property is important if it is required e.g. to produce XML documents from plain texts.

Keywords: Parse-time extensible parsing, parser generator, kind grammars, compiler-compiler, automata with oracles.

Thanks to my supervisor, Prof. RNDr. Jaroslav Král, DrSc. for helpful hints, to RNDr. Leo Galamboš, Ph.D., RNDr. Pavel Váňa, Ph.D., and RNDr. David Bednárek for proofreading.

My thanks go also to the institutions that provided financial support for my research work. Through my doctoral study, my work was partially supported by grants 201/96/0195 and 201/99/0236 of the Grant Agency of Czech Republic and by project 1ET100300517 of the Program "Information Society".

Prague, 30th June 2006 RNDr. Michal Žemlička

Contents

Chapter 1

Introduction

This work describes new classes of deterministic context-free grammars and parsers[1] in two different views: the first view is the usual formal view and the second view is closer to the author's mind flow – it describes how some constructions were developed and also describes some ideas and principles.

It is possible to look only at the 'formal' chapters but we hope that using both views together can present the idea better.

1.1 Motivation

Run-time extensibility and adaptability of tools are more and more popular. Examples are e.g. TEX, SGML, XML, application plugins, programmable applications.

It is not only fashion, in many cases it is a necessity. For example, huge information systems cannot be implemented without software confederation architecture [KŽ00, KŽ03b]. This architecture expects that services are interconnected by *configurable* components called front-end gates.

This document was written using LATEX, one of the macro packets (extensions) for TEX system written by D. Knuth. This system is based on macro

[1] There are several meanings for parser and parsing: The most restrictive is *acceptor* that only determines whether an input string belongs to given language or not. For linguistics it can be useful to understand parser as a tool producing for given input (sentence) set of trees representing the sentence. For practical purposes there are built parsers that moreover produce some output (therefore can be also called *transducers*) that allow to reconstruct the original string, respective parsing process. Such output generating parsers can be further divided into three categories: the first ones produce syntax tree of the input string, the second ones produce sequences of messages or procedure calls called *semantic actions*, the third ones produce message(s) in an output language. Sometimes only the third type of transducers is mentioned as real transducers. If not specified otherwise, we will understand under the term 'parser' all the above mentioned variants of parsers.

processing, which is one of the techniques usable for extending abilities of applications (and enriching languages accepted by them).

There have existed extensible languages and tools for a long time. Most popular are probably SGML [ISO86] and newly XML [W3 99a]. They use some fixed constructs (tags) to structure the text and make the parsing easier (almost unnecessary).

Extensibility in programming languages follows several different paths (and uses different techniques):

1. identifiers for variables and subroutines (in all programming languages),
2. macros (e.g. in macroassemblers, TEX),
3. overloading of operators (C++, ...),
4. (semantic) hooks (most extensible languages), and
5. extensible syntax (to be discussed here).

These techniques may be (and often are) combined.

The idea of an extensible language/parser is not new – an example of a macro-based extensible (programming) language was proposed by Leavenworth [Lea66] reprinted e.g. in [AU72].

Our considerations develop the idea of Demner [Dem90] who proposed to handle built-in and user defined (new) data types in the same way. The given principle was extended by the author from data types to whole constructs. A prototype of such parser (part of a compiler) is in [Žem94]. The typical application of the compiler is shown in Ex. 1.1.

Example 1.1 (Run-time extended source code):

```
program powering;
function power(base,exponent);
  var result;
  begin
    result:=1;
    while exponent>0 do
      result:=result*base; exponent:=exponent-1
      end;
    power:=result
  end;
terminal 'doublestar','**';
rule 'power:<F>::=<f>(doublestar)<F>';
```

```
instance of 'power' is power;
var base, exponent;
begin
  write 'Enter base and exponent:  ';
  read base; read exponent;
  write 'Power is '; write base**exponent
end.
```

Explanation of less usual constructs	
Command	**Description**
`terminal` *name*, *shape*	Declares new terminal symbol or new shape for an existing one. The parameter *name* is used to specify a terminal (already existing in the grammar or a new one), and *shape* is the string used in a parsed text. More different shapes may be defined just by more terminal declarations with the same name and different shapes.
`rule` *production*	Adds new syntactic production to the grammar.
`instance of` *semantic action* `is` *subroutine*	Assigns semantics to given syntactic construction. The first parameter refers to production name (label in the front of a production) and the second should be a procedure/function identifier.

We expect 'f' and 'F' to be nonterminals. (They belong, together with 'E' and 'T', to the ones that define arithmetic expressions. They are ordered E, T, F, and f.)

The meaning of the production is: rewrite the nonterminal `F` by the nonterminal `f` followed by a terminal `doublestar` ("**") and by the nonterminal `F`. The semantic action `power` should be applied on reduction of the production. The parenthesis '`(`' and '`)`' delimit terminal having no semantic value, '`{`' and '`}`' delimit a nonterminal having a semantic value. The knowledge on semantic values is used to check parameters of a subroutine (procedure or function) assigned to the semantic action (production).

Other example of grammar extension is shown in Ex. 1.2, exact description is available in [Žem94]. □

The compiler from [Žem94] is able to handle only extended 1-kind grammars having semantic actions only at the end of productions. There is a new version under development with the ability to handle more complex grammars – with longer lookahead, with semantic actions placed quite everywhere

within productions (the only exception is the beginning of left recursive productions – this restriction is respected in all other compilers), with output terminal symbols and with an ability to generate source code of a parser for the actual language.

Parser generator for LL(1) languages was already released as [Žem02a].

The new system does not include back-end (code generation) as it is possible to add it in a standard way.

Example 1.2 (New loop declaration):

The following program construction:

LOOP
commands
ON *condition* EXIT;
commands
ENDLOOP;

is syntactically declared in following lines:

```
terminal 'loop-kwd','loop';
terminal 'endloop-kwd','endloop';
terminal 'on-kwd','on';
terminal 'exit-kwd','exit';
rule '[loop1]::=loop1:(loop-kwd)';
rule '[loop2]::=loop2:[loop1][cmds](on-kwd)<cond>(exit-kwd)';
rule '[cmd]::=loop3:[loop2][cmds](endloop-kwd)';
```

where

```
terminal 'loop-kwd','loop';
terminal keyword name, keyword shape;
```

declares new terminal with given name and shape, and

```
rule production;
```

introduces new production.

In newer versions of the compiler it will be possible to write the production as one whole with several semantic actions in it[2].

[2]This feature is already supported by parser/transducer generator KindCons [Žem02a] and by transducing interpreter KindTran [Žem02b].

Such program constructions cannot be composed from usual ones but may be important for the application. There is a whole hierarchy of more and more complex loops that cannot be simply constructed just using IF and WHILE constructs. They are programmed using conditions, labels and GOTOs. □

Programs in extensible programming systems are composed from two parts: one part handles metalevel and one part the application and data specific level.

Examples of the metalevel are macro specifications or production or terminal specifications in our short examples.

1.2 Structure of the document

The Thesis is divided into several chapters. Some of them are directed closer to the theory (Chap. 2 reminds notation and some known definitions, Chap. 5 deals with kind grammars, Chap. 7 introduces oracle and lookahead pushdown automata – modifications of usual pushdown automata that are closer to the programming reality, Chap. 6 places kind grammars and kind parsers into usual hierarchies, and Chap. 8 describe some implementations of kind parsers). Other chapters are meant as a less formal guide through the basic ideas of kind parsing (Chap. 3 builds up the basic data structures and shows its use for parsing or generation of text, and Chap. 4 shows the extension and modification abilities of the structure and presents parser types derived from the kind ones), or as a connection to the real world (motivation is given in Chap. 1, Chap. 10 is dedicated to kind constructor that gives opportunity to use advantages of static – non extensive – version of kind parsing, Chap. 11 presents kind transducer, and Chap. 12 gives a look at possible applications of introduced technique). Related results are collected in Chap. 13 and conclusion is in Chap. 14.

In order to enhance the legibility of this work we included Chap. 2 containing the definitions and results from various sources used here in this work.

Chapter 2

Notation and Known Results

For the sake of legibility of this work this chapter collects known definitions, theorems, lemmas, etc. from various sources used in the rest of this work. The reader can skip this chapter and use it like a dictionary during the reading of the rest of this thesis.

Every time when some source has been used, it is cited and possible formal modifications (e.g. notation changes introduced to synchronize it) are mentioned.

Terminology and notation used in this work is usually inspired by the ones from [AU72, Chy84]. Some basic notation and definitions known from the literature are repeated here to minimize reader's searching for additional resources.

a, b, c, ...	alphabet symbols
A, B, C, ...	sets, nonterminal symbols
...X, Y, Z	grammar symbols (both terminal i nonterminal ones)
α, β, γ, ...	grammar symbol strings
...x, y, z	terminal symbol strings
ε	empty string
$x \in M$	(an item) x is an element of (a set) M
$A \subset B$	(a set) A is a subset of (a set) B
$A \cup B$	union of (sets) A and B
$A \cap B$	intersection of (sets) A and B
$\emptyset$	empty set
2^A	power set of A (the set of all subsets of A)

Table 2.1: Notation

Definition 2.1 (String):

(Taken from [AU72] pages 15–16; slightly modified.)

We formally define *strings over an alphabet* Σ in the following manner:

1. ε is a string over Σ.
2. If x is a string over Σ and a is in Σ, then xa is a string over Σ.
3. y is a string over Σ if and only if its being so follows from 1. and 2.

If x and y are strings over Σ, then the string xy is called the *concatenation* of x and y. In some cases it can be also written as $x \cdot y$. If M and N are set of strings over Σ, concatenation can be extended to the set of strings:

$M \cdot v =_{df} \{uv | u \in M\}$,
$u \cdot M =_{df} \{uv | v \in M\}$, and
$M \cdot N =_{df} \{uv | u \in M,\ v \in N\}$.

Let x, y, and z be arbitrary strings over some alphabet Σ. We call x a *prefix* of the string xy and y a *suffix* of xy. y is a *substring* of xyz. Both a prefix and suffix of a string x are substrings of x. Notice that the empty string is a substring, a prefix, and a suffix of every string.

If $x \neq y$, $x \neq \varepsilon$ and the string x is a prefix (suffix) of some string y, then x is called *proper* prefix (suffix) of y.

The *length* of a string is a number of symbols in the string. That is, if $x = a_1 \ldots a_n$, where each a_i is a symbol, then the length of x is n.

Definition 2.2 (Iterations):

For every nonempty alphabet Σ and $a \in \Sigma$ we define:

a^0 as empty string (ε);

a^1 as a;

$a^n, n \geq 2$ as aa^{n-1};

a^+ as $\bigcup_{i=1}^{\infty}\{a^n\}$;

a^* as $\bigcup_{i=0}^{\infty}\{a^n\}$.

Similarly $\Sigma^0 = \{\varepsilon\}$, $\Sigma^1 = \Sigma$, $\Sigma^n = \Sigma \cdot \Sigma^{n-1}$, $\forall n \geq 2$; $\Sigma^+ = \bigcup_{i=1}^{\infty} \Sigma^n$, and $\Sigma^* = \bigcup_{i=0}^{\infty} \Sigma^n$.

Definition 2.3 (Grammar):

(Taken from [AU72] page 85.)

A *grammar* is a 4-tuple $G = (\mathrm{N}, \Sigma, P, S)$ where

1. N is a finite set of *nonterminal symbols* (sometimes called variables or syntactic categories).

2. Σ is a finite set of *terminal symbols*, disjoint from N.

3. P is a finite subset of

$$(\mathrm{N} \cup \Sigma)^* \mathrm{N} (\mathrm{N} \cup \Sigma)^* \times (\mathrm{N} \cup \Sigma)^*$$

   An element (α, β) in P will be written $\alpha \to \beta$ and called a *production*.

4. S is a distinguished symbol in N called the *sentence* (or *start*) symbol.

Definition 2.4 (Sentential form):
(Taken from [AU72] page 86; modified.)
We define a special kind of string called *sentential form* of a grammar $G = (\mathrm{N}, \Sigma, P, S)$ recursively as follows:

- S is a sentential form.
- If $\alpha\beta\gamma$ is a sentential form and $\beta \to \delta$ is in P, then $\alpha\delta\gamma$ is also a sentential form ($\alpha, \gamma, \delta \in (\mathrm{N} \cup \Sigma)^*$, $\beta \in (\mathrm{N} \cup \Sigma)^* \mathrm{N} (\mathrm{N} \cup \Sigma)^*$).

A sentential form of G containing no nonterminal symbols is called a *sentence* (or *word*) generated by G.

Definition 2.5 (Language generated by a grammar):
(Taken from [AU72] page 86.)
The *language generated by a grammar* G denoted $L(G)$, is the set of all sentences generated by G.

Definition 2.6 *(Taken from [AU72] page 86; slightly modified)*:
Let $G = (\mathrm{N}, \Sigma, P, S)$ be a grammar. We define a relation $\Rightarrow_G$ (to be read as *directly derives*) on $(\mathrm{N} \cup \Sigma)^*$ as follows: If $\alpha\beta\gamma$ is a string in $(\mathrm{N} \cup \Sigma)^*$ and $\beta \to \delta$ is a production in P, then $\alpha\beta\gamma \Rightarrow_G \alpha\delta\gamma$.

We shall use $\Rightarrow_G^+$ (to be read *derives in a nontrivial way*) to denote the transitive closure of $\Rightarrow_G$, and $\Rightarrow_G^*$ (to be read *derives*) to denote reflexive and transitive closure of $\Rightarrow_G$. Whenever it is clear that G is understood, the subscript can be removed and we can write $\Rightarrow$, $\Rightarrow^+$, and $\Rightarrow^*$.

Obviously x is a sentential form iff $S \Rightarrow^* x$.

We shall also use the notation $\Rightarrow^k$ to denote the k-fold product of the relation $\Rightarrow$. That is to say, $\alpha \Rightarrow^k \beta$ if there is a sequence $\alpha_0, \alpha_1, \ldots, \alpha_k$

of $k+1$ strings (not necessarily distinct) such that $\alpha = \alpha_0$, $\alpha_{i-1} \Rightarrow \alpha_i$ for $1 \leq i \leq k$ and $\alpha_k = \beta$. This sequence of strings is called a *derivation of length* k of β from α in G. Thus, $L(G) = \{w | w \in \Sigma^* \text{ and } S \Rightarrow^* w\}$. Also notice that $\alpha \Rightarrow^* \beta$ if and only if $\alpha \Rightarrow^i \beta$ for some $i \geq 0$, and $\alpha \Rightarrow^+ \beta$ if and only if $\alpha \Rightarrow^i \beta$ for some $i \geq 1$.

Definition 2.7 (Context-free grammar):
(Taken from [AU72] page 91.)

A grammar G is said to be *context free* (CFG) if each production in P is of the form $A \to \alpha$, where $A \in \mathrm{N}$ and $\alpha \in (\mathrm{N} \cup \Sigma)^*$.

Definition 2.8 *(Taken from [AU72] page 143)*:

If there is always expanded the leftmost (rightmost) nonterminal in all the derivations in a succession of derivations, such derivation is called *leftmost* (*rightmost*).

If $S = \alpha_0, \alpha_1, \ldots, \alpha_n$ is a leftmost derivation in grammar G, then we shall write $S \Rightarrow^*_{G\ lm} \alpha_n$ (or $S \Rightarrow^*_{lm} \alpha_n$ if G is clear) to indicate the leftmost derivation. We call α_n a *left sentential form*. Likewise, if $S = \alpha_0, \alpha_1, \ldots, \alpha_n$ is a rightmost derivation, we shall write $S \Rightarrow^*_{rm} \alpha_n$, and call α_n a *right sentential form*. We use $\Rightarrow_{lm}$ and $\Rightarrow_{rm}$ to indicate single-step leftmost and rightmost derivations.

Definition 2.9 (Useless symbol):
(Taken from [AU72] page 144.)

We say that a symbol $X \in \mathrm{N} \cup \Sigma$ is *useless* in a CFG $G = (\mathrm{N}, \Sigma, P, S)$ if there does not exist a derivation of the form $S \Rightarrow^* wXy \Rightarrow^* wxy$. Note that w, x, and y are in Σ^*.

Definition 2.10 (Inaccessible symbol):
(Taken from [AU72] page 145.)

We say that a symbol $X \in \mathrm{N} \cup \Sigma$ is *inaccessible* in a CFG $G = (\mathrm{N}, \Sigma, P, S)$ if X does not appear in any sentential form.

Definition 2.11 (Reduced grammar):
(Taken from [WM95] page 273; synchronized terminology.)

A CFG G is said to be *reduced* if it contains neither inaccessible nor useless nonterminals.

Definition 2.12 (ε-free grammar):
(Taken from [AU72] pages 147–148.)

We say that a CFG $G = (\mathrm{N}, \Sigma, P, S)$ is *ε-free* if either

1. P has no ε-productions, or

2. There is exactly one ε-production $S \to \varepsilon$ and S does not appear on the right side of any production in P.

Definition 2.13 *(Taken from [AU72] page 150)*:
A CFG $G = (\mathrm{N}, \Sigma, P, S)$ is said to be *cycle-free* if there is no derivation of the form $A \to^+ A$ for any A in N.

G is said to be *proper* if it is cycle-free, is ε-free, and has no useless symbols.

Definition 2.14 **(A-production):**
(Taken from [AU72] page 150.)
An *A-production* in a CFG is a production of the form $A \to \alpha$ for some α.

Definition 2.15 *(Taken from [AU72] page 153)*:
A nonterminal A in CFG $G = (\mathrm{N}, \Sigma, P, S)$ is said to be recursive if $A \Rightarrow^+ \alpha A \beta$ for some α and β. If $\alpha = \varepsilon$, then A is said to be *left-recursive*. Similarly, if $\beta = \varepsilon$, then A is *right-recursive*. A grammar with at least one left- (right-) recursive nonterminal is said to be *left- (right-) recursive*.

Definition 2.16 **(Nondeterministic finite automaton):**
(Taken from [AU72] page 113.)
A *nondeterministic finite automaton* is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set of *states*,

2. Σ is a finite set of permissible *input symbols*,

3. δ is a mapping $Q \times \Sigma \to 2^Q$ which dictates the behavior of the finite state control; δ is sometimes called *state transition function*,

4. $q_0 \in Q$ is the *initial state* of the final state control, and

5. $F \subseteq Q$ is the set of *final states*.

Definition 2.17 *(Taken from [AU72] pages 113–114, slightly modified)*:
If $M = (Q, \Sigma, \delta, q_0, F)$ is a finite automaton, then a pair (q, w) in $Q \times \Sigma^*$ is a *configuration* of M. A configuration of the form (q_0, w) is called an *initial configuration*, and one of the form (q, ε) where q is in F, is called a *final* (or *accepting*) configuration.

A *move* by M is represented by a binary relation $\vdash_M$ (or $\vdash$, where M is understood) on configurations. If $\delta(q, a)$ contains q', then $(q, aw) \vdash (q', w)$ for all $w \in \Sigma^*$.

We define $\overset{+}{\vdash}$ and $\overset{*}{\vdash}$ as transitive and reflexive-transitive closure of $\vdash$.

We shall say that an input string w is *accepted* by M if $(q_0, w) \overset{*}{\vdash} (q, \varepsilon)$ for some q in F. The *language defined by* M, denoted $L(M)$, is the set of input strings accepted by M, that is,

$$\{w | w \in \Sigma^* \text{ and } (q_0, w) \overset{*}{\vdash} (q, \varepsilon) \text{ for some } q \in F\}.$$

Definition 2.18 (Deterministic finite automaton):
(Taken from [AU72] pages 116–117.)

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a nondeterministic finite automaton. We say that M is *deterministic* if $\delta(q, a)$ has no more than one member for any $q \in Q$ and $a \in \Sigma$. If $\delta(q, a)$ always has exactly one member, we say that M is *completely specified.*

Definition 2.19 (Accessible and inaccessible state):
(Taken from [Med00] page 156, synchronized notation.)

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton. A state $q \in Q$ is *accessible* if there exists a word w in Σ^* such that $sw \overset{*}{\vdash} q$; otherwise, q is *inaccessible.*

Definition 2.20 (Terminating state):
(Taken from [Med00] page 161, synchronized notation.)

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite automaton. A state $q \in Q$ is *terminating* if there exists a word $w \in \Sigma^*$ such that $(q, w) \overset{*}{\vdash} (f, \varepsilon)$ for some $f \in F$; otherwise, q is *nonterminating.*

Definition 2.21 (Well-specified finite automaton):
(Taken from [Med00] page 165, synchronized notation, corrected.)

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite automaton satisfying these two properties:

1. M has no inaccessible state;
2. M has no more than one nonterminating state.

Then, M is a *well-specified* finite automaton.

Definition 2.22 (Distinguishable state):
(Taken from [Med00] page 171, synchronized notation.)

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a well-specified finite automaton, and let $p, q \in Q$ such that $p \neq q$. If there exists a word $w \in \Sigma^*$, so that

$$pw \stackrel{*}{\vdash} p' \text{ and } qw \stackrel{*}{\vdash} q'$$

where $p', q' \in Q$ and $|\{p', q'\} \cap F| = 1$, then w *distinguishes* p from q; at this point, p and q are *distinguishable*. If there exists no word that distinguishes p from q, p and q are *indistinguishable*.

Definition 2.23 (Minimum-state finite automaton):
(Taken from [Med00] page 172.)

Let M be a well-specified finite automaton. Let M_{min} be a well-specified finite automaton satisfying these two properties:

1. M_{min} contains only distinguishable states;
2. $L(M) = L(M_{min})$.

Then, M_{min} is a *minimum-state finite automaton* equivalent to M.

Definition 2.24 (First):
(Taken from [AU72] page 300.)

For a CFG $G = (\mathrm{N}, \Sigma, P, S)$,

$$\mathrm{First}_G^k(\alpha) = \{x \,|\alpha \Rightarrow_{lm}^* x\beta \wedge |x| = k \ \vee \alpha \Rightarrow^* x \wedge |x| < k\}$$

If it is clear what grammar is mentioned, the index representing grammar can be omitted (and then we can write $First_k(\alpha)$). We often omit k if $k = 1$.

Note 2.25 Let us apply the operator to arbitrary string:

$$First_k(a_1 \ldots a_l) = \begin{cases} a_1 \ldots a_l & l < k \\ a_1 \ldots a_k & l \geq k \end{cases}$$

For $k = 1$ the index 1 can be omitted.

Definition 2.26 (Follow):
(Taken from [AU72] page 343.)

Let $G = (\mathrm{N}, \Sigma, P, S)$ be a CFG. We define $\mathrm{Follow}_G^k(\beta)$, where k is an integer and β is in $(N \cup \Sigma)^*$, to be set $\{w \mid S \Rightarrow^* \alpha\beta\gamma$ and w is in $\mathrm{First}_G^k(\gamma)\}$. We shall omit k and G whenever they are understood.

Definition 2.27 (LL(k) grammar):
(Taken from [AU72] page 336.)
Let $G = (\mathrm{N}, \Sigma, P, S)$ be a CFG. We say that G is $LL(k)$, for some fixed integer k, if whenever there are two leftmost derivations

1. $S \Rightarrow^*_{lm} wA\alpha \Rightarrow_{lm} w\beta\alpha \Rightarrow^*_{lm} wx$ and
2. $S \Rightarrow^*_{lm} wA\alpha \Rightarrow_{lm} w\gamma\alpha \Rightarrow^*_{lm} wy$

such that $First_k(x) = First_k(y)$, it follows that $\beta = \gamma$.

Definition 2.28 (Strong LL(k) grammar):
(Taken from [AU72] pages 343–344.)
Let $G = (N, \Sigma, P, S)$ be a CFG such that it holds: If $A \to \beta$ and $A \to \gamma$ are distinct A-productions, then

$$First_k(\beta\ Follow_k(A)) \cap First_k(\gamma\ Follow_k(A)) = \emptyset.$$

Such grammar is called *strong* $LL(k)$ grammar.

Strong $LL(k)$ grammars are often called $SLL(k)$ grammars.

Note 2.29 As stated e.g. in [SSS90], every LL(1) grammar is a strong LL(1) grammar (Lemma 8.39, page 232) and for $k > 1$ there are LL(k) grammars that are not strong LL(k) grammars (Theorem 8.41, page 233).

Definition 2.30 (Pushdown machine):
(Taken from [ABB97] pages 138–139; synchronized notation.)
A *pushdown machine* (PDM) is a quadruple $\mathcal{M} = (Q, \Sigma, \Gamma, \delta)$ where

Q is a finite nonempty set of *states*,

Σ is a finite nonempty set of *input symbols* called *input alphabet*,

Γ is a finite nonempty set of *stack symbols* called *stack alphabet*, and

δ is a finite subset of $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \times Q \times \Gamma^*$ called *transition function* or the set of *transition rules*.

Any element $(q, a, Z, q', \alpha) \in \delta$ is called a *rule*, and if $a = \varepsilon$, it is called an *ε-rule*. The first three components of a rule can be viewed as a precondition and the last two components as a postcondition of the rule. To emphasize it, we can write $(q, a, Z) \to (q', \alpha) \in \delta$ instead of $(q, a, Z, q', \alpha) \in \delta$.

Definition 2.31 (Internal configuration, configuration):
(Taken from [ABB97] page 139; synchronized notation.)

Let $\mathcal{M} = (Q, \Sigma, \Gamma, \delta)$ be a PDM.

An *internal configuration* of $\mathcal{M}$ is a couple $(q, \alpha) \in Q \times \Gamma^*$, where q is the current state, and α is the string over Γ^* composed of the symbols in the stack, the first letter of α being the top-most symbol of the stack.

A *configuration* of $\mathcal{M}$ is a triple $(q, u, \alpha) \in Q \times \Sigma^* \times \Gamma^*$, where u is the input word to be read, and (q, α) is an internal configuration.

Definition 2.32 (Transition relation, transition, computation):
(Taken from [ABB97] page 139; synchronized notation, slightly modified.)

Let $\mathcal{M} = (Q, \Sigma, \Gamma, \delta)$ be a PDM.

The *transition relation* is a relation over configurations defined in the following way: let $(q, a, Z, q', \beta) \in \delta$ and let $c = (q, au, Z\alpha)$ and $c' = (q', u, \beta\alpha)$ be two configurations where a is in $\Sigma \cup \{\varepsilon\}$, u is in Σ^*, q and q' are in Q, Z is in Γ and α and β are in Γ^*. We say then that there is a *transition* between c and c', and we write $c \vdash c'$. If $a = \varepsilon$, the transition is called an ε-transition, and if $a \in \Sigma$, the transition is said to involve the reading of a letter.

A *valid computation* is an element of the reflexive and transitive closure of the transition relation. We denote a valid computation starting from c and leading to c' by $c \overset{*}{\vdash} c'$. A convenient notation is to introduce, for any word $x \in \Sigma^*$ the relation on internal configuration, denoted $\overset{x}{\models}$ and defined by:

$$(q, \alpha) \overset{x}{\models} (q', \alpha') \iff \exists z \in \Sigma^* : \ (q, xz, \alpha) \overset{*}{\vdash} (q', z, \alpha').$$

We clearly have $\overset{x}{\models} \circ \overset{y}{\models} \ = \ \overset{xy}{\models}$.

Definition 2.33 (Accessible (internal) configuration):
(Taken from [ABB97] page 139; synchronized notation, extended.)

Let $\mathcal{M} = (Q, \Sigma, \Gamma, \delta)$ be a PDM. An internal configuration (q', α') is *accessible* from an internal configuration (q, α) in $\mathcal{M}$, or equivalently, (q, α) is *co-accessible* from (q', α') in $\mathcal{M}$ if there is some $u \in \Sigma^*$ such that $(q, \alpha) \overset{u}{\models} (q', \alpha')$. Similarly, a configuration $c' = (q', v, \alpha')$ of $\mathcal{M}$ is said to be *accessible* from a configuration $c = (q, uv, \alpha)$ of $\mathcal{M}$ if $c \overset{*}{\vdash} c'$ in $\mathcal{M}$.

Definition 2.34 (Pushdown automaton):
(Taken from [ABB97] pages 139–140; synchronized notation, simplified.)

A *pushdown automaton* (a *PDA* for short) is composed of a pushdown machine $(Q, \Sigma, \Gamma, \delta)$, the *initial state* $q_0 \in Q$, *initial pushdown symbol* $Z_0 \in \Gamma$ (forming with a word x to be recognized an *initial configuration* (q_0, x, Z_0)), and a set $F \subset Q$ of *accepting states*. It is therefore a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$. The pushdown machine (Q, Σ, Γ, P) is called the *PDM associated* to $\mathcal{A}$.

For a PDA, an (internal) configuration is *accessible* if it is accessible from the initial (internal) configuration, and is *co-accessible* if it is co-accessible from an accepting (internal) configuration.

A word $x \in \Sigma^*$ is *recognized* by a PDA $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ if there is $q \in F$ and $\alpha \in \Gamma^*$ such that $(q_0, x, Z_0) \stackrel{*}{\vdash} (q, \varepsilon, \alpha)$.

There is no transition starting with empty pushdown – hence when the pushdown is empty, the computation terminates. Unless an accepting configuration is reached, the computation terminates with an error.

The *language accepted* by a PDA is the set of all words recognized by this PDA. For any PDA $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ we note $L(\mathcal{A})$ the language recognized by $\mathcal{A}$.

Language accepted by a pushdown automaton is called *pushdown automaton language*.

Definition 2.35 (Deterministic pushdown automaton):
(Taken from [AU72] pages 184–185.)

A PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is said to be *deterministic* (a DPDA for short) if for each $q \in Q$ and $Z \in \Gamma$ either

1. $\delta(q, a, Z)$ contains at most one element for each a in Σ and $\delta(q, \varepsilon, Z) = \emptyset$ or

2. $\delta(q, a, Z) = \emptyset$ for all $a \in \Sigma$ and $\delta(q, \varepsilon, Z)$ contains at most one element.

Definition 2.36 (Augmented grammar):
(Taken from [AU72] page 372.)

Let $G = (\mathrm{N}, \Sigma, P, S)$ be a CFG. We define the *augmented grammar derived from* G as $G' = (\mathrm{N} \cup \{S'\}, \Sigma, P \cup \{S' \to S\}, S')$. The augmented grammar G' is merely G with a new staring production $S' \to S$, where S' is a new start symbol, not in N. We assume that $S' \to S$ is the zeroth production in G' and that the other productions of G are numbered $1, 2, \ldots, p$. We add the starting production so that when a reduction using the zeroth production is called for, we can interpret this "reduction" as a signal to accept.

Definition 2.37 (LR(k) grammar):
(Taken from [AU72] page 372.)

Let $G = (\mathrm{N}, \Sigma, P, S)$ be a CFG and let $G' = (\mathrm{N}', \Sigma, P', S')$ be its augmented grammar. We say that G is LR(k), $k \geq 0$, if the three conditions

1. $S' \Longrightarrow^*_{G'rm} \alpha A w \Longrightarrow_{G'rm} \alpha\beta w$,

2. $S' \Longrightarrow^*_{G'rm} \gamma B x \Longrightarrow_{G'rm} \alpha\beta y$,

3. $First_k(w) = First_k(y)$

imply that $\alpha Aw = \gamma Bx$. (That is, $\alpha = \gamma$, $A = B$, and $x = y$.)

A grammar is LR if it is LR(k) for some k.

Definition 2.38 (SLR(k) grammar):
(Taken from [SSS90] – adapted.)

Grammar $G = (\mathrm{N}, \Sigma, P, S)$ is *SLR(k)* if its parser[1] is deterministic and $S \Rightarrow^+ S$ is impossible in G. A language over alphabet Σ is *SLR(k)* if it is the language generated by an SLR(k) grammar.

Definition 2.39 (graph, node, edge):

An ordered 2-tuple $G = (V, E)$ where $E \subseteq V \times V$ is called *oriented graph*, elements from V are called *nodes* (or *vertices*), elements from E are called *edges*.

It is said that an edge (x, y) *starts* with (in) x and *ends* by (in) y. Then x is an *predecessor* of y and y is an *successor* of x (according to the edge (x, y)).

Definition 2.40 (path):

It is said that there is a *path* between nodes x_1 and x_{k+1} iff there are nodes $x_2, \ldots, x_k$ such that there is a sequence of edges (x_i, x_{i+1}), $i \in \{1, \ldots, k\}$. k is called *length of the path from* x_1 *to* x_{k+1}.

Definition 2.41 (cycle):

Path of length > 0 from some node x to the same node x is called *cycle*.

Definition 2.42 (tree, root, leaf):

Let $G = (V, E)$ is an oriented graph. G is called *(split) tree* if $|E| = |V| - 1$ and if $\exists v \in V\ \forall v' \in V$, $v' \neq v$: there is a path in G from v to v'. The node v is called the *root* of G. The tree nodes with no successors are called *leaves*.

Definition 2.43 (forest):

Graph consisting only from trees is called *forest*.

Definition 2.44 (Left corner):
(Taken from [AU72] page 278.)

The *left corner* of a non-ε-production is the leftmost symbol (terminal or nonterminal) on the right side. A *left-corner parse* of a sentence is the sequence of productions used at the interior nodes of a parse tree in which have been ordered as follows. If a node n has p direct descendants $n_1, n_2, ..., n_p$,

[1] For the definition of SLR(k) parser see e.g. [SSS90].

then all nodes in the subtree with root n_1 precede n. Node n precede all its other descendants. The descendants of n_2 precede those of n_3, which precede those of n_4, and so forth.

Roughly speaking, in left corner parsing the left corner of a production is recognized bottom-up and the remainder of the production is recognized top-down.

Definition 2.45 *(Taken from [AU72] page 363)*:
Let G be a CFG. We say that $S \Rightarrow^*_{\text{lc}} wA\delta$ if $S \Rightarrow^*_{lm} wA\delta$ and the nonterminal A is not the left corner of the production which introduced it into a left sentential form of the sequence represented by $S \Rightarrow^*_{lm} wA\alpha$.

Definition 2.46 (LC(k) grammar):
(Taken from [AU72] pages 363–364.)
CFG $G = (\mathrm{N}, \Sigma, P, S)$ is $LC(k)$ if the following conditions are satisfied: Suppose that $S \Rightarrow^*_{\text{lc}} wA\delta$. Then for each lookahead string u and derivation $A \Rightarrow^* B\gamma$, there is at most one production $B \to \alpha$ such that

1. (a) if $\alpha = C\beta$, $C \in \mathrm{N}$, then $u \in First_k(\beta\gamma\delta)$, and
   (b) In addition, if $C = A$, then u is not in $First_k(\delta)$;
2. If α does not begin with a nonterminal, then $u \in First_k(\alpha\gamma\delta)$.

Definition 2.47 (Ch(k)-grammars):
(Taken from [NSS79] page 393.)
Let k be a non-negative integer. A grammar $G = (\mathrm{N}, \Sigma, P, S)$ is said to be a *Ch(k) grammar* if, for a terminal string w, a nonterminal A, and strings γ, α, δ_1, and δ_2 in $(\mathrm{N} \cup \Sigma)^*$ such that $A \to \alpha\delta_1$ and $A \to \alpha\delta_2$, $\delta_1 \neq \delta_2$, the condition $S \Rightarrow^*_{lm} wA\gamma$ implies that $First_k(\delta_1\gamma) \cap First_k(\delta_2\gamma) = \emptyset$.

Theorem 2.48 *(Taken from [NSS79] Theorem 3.2; page 394)*:
Every LL(k) grammar is a Ch(k) grammar.

Theorem 2.49 *(Taken from [NSS79] Theorem 3.5; page 395)*:
A reduced Ch(k) grammar is not left-recursive.

Definition 2.50
Let $a \in \Sigma$, $v = \Sigma^*$. For each word $u = av$ we say that $head(u) = a$ and $tail(u) = v$.

Chapter 3

Principles of Kind Parsing

3.1 Introduction

The tour of kind parsing will be guided by the running example of a simple arithmetic expression. Let the tour start with Ex. 3.1.

Example 3.1 (A grammar of simple arithmetic expressions):
$\mathrm{N} = \{S, E, T, F\}$
$\Sigma = \{id, num, (,), +, *\}$
$$P = \left\{ \begin{array}{l} S \to E, \\ E \to E + T, E \to T, \\ T \to T * F, T \to F, \\ F \to id, F \to num, F \to (E) \end{array} \right\}$$
$S = S$ □

The grammar is directly left recursive due the productions $E \to E + T$ and $T \to T * F$, so it cannot be LL(k) for any k (compare Def. 2.27). It implies that the usual method of construction of recursive descent parsers for LL-grammars cannot be used. Such a method can be used only in the combination with grammar conversion that would remove the left recursiveness. It is a quite simple algorithm, but after that processing the syntactic structure of sentences given by the original grammar is substantially changed. If the LL parser for the grammar from Ex. 3.1 (respective for a grammar transformed to LL) is implemented as a set of recursive parsing procedures, we need 6 subroutines for the grammar after left factoring. Techniques discussed below produce only 4 subroutines.

These techniques simplify the interface to compiler back-ends (insertion of semantic procedures/symbols, etc.). As the productions may be written in their logical scopes (instead being modified to fit into LL requirements or

broken into parts for parsing systems where semantic actions may be only at the end of the productions), it is easier to specify the logical structure of the defined language within the formal specification of the grammar.

This chapter is focused to the development process of parsers suitable for recursive descent parsing designed without any automated tools like a constructor. According to the intended application the explanations will be rather intuitive. It is not too complicated to formalize below introduced process using results introduced in subsequent chapters but it can make the process less understandable without having significant advantages.

It is important to emphasize here that the below described process reaches the goal (construction of a parser) only for a specific grammar class further called *kind grammars*.

A similar approach (description of the parser creation process) is used also by Rechenberg and Mössenböck in the description of their compiler constructor CoCo ([RM85]). Describing languages using graphs for individual nonterminals or language constructs is also common (syntax diagrams for Pascal).

3.2 Parser Structure

Any set of productions can be viewed as a *forest of production trees* (see Def. 6.17). Focusing the principles and their understanding we describe the process of the structure creation on a set of examples. The grammar G from Ex. 3.1 will be used.

The aim is to describe a grammar G as a forest of production trees F in the sense of graph theory. The forest can be easily transformed into a data structure defining an (extensible) parser for $L(G)$ or to a program structure for a persistent parser for the same language.

The productions of G are first divided using the nonterminal on their left-hand sides and by the presence of direct left recursiveness (i.e. if the same nonterminal is presented at the beginning of the right hand-side of the production as on the left-hand side – see Fig. 3.1). Each row of the resulting table contains only the productions with the same nonterminal on their left-hand sides. This nonterminal can be separated and written in a special column. Afterwards the nonterminal can be omitted at the beginnings of the production descriptions. Similarly it is not needed to explicitly write the recurring nonterminal at the beginning of right-hand sides of recursive productions – its presence is obvious from the position of the production description in the table – Fig. 3.2.

The same can be written also in another style. A path through the

Productions	
without left recursion	with left recursion
$S \to E$	
$E \to T$	$E \to E + T$
$T \to F$	$T \to T * F$
$F \to id$ $F \to num$ $F \to (E)$	

Figure 3.1: Productions divided by defined nonterminals and by the presence of left recursion

Nonterminal	right-hand sides of productions having	
	no left recursion	direct left recursion
S	E	
E	T	$+T$
T	F	$*F$
F	id num (E)	

Figure 3.2: A short description of productions divided by defined nonterminals and by the presence of left recursion

production tree can be assigned to each production. The edges along the path are labeled by the symbols from the productions. For left recursive production the leading edge labeled by recurring nonterminal is omitted. Figure 3.3 shows/implies production as paths in a directed graph. The part, where they are placed, shows/imply to what group of production it belongs (i.e. the defined nonterminal and presence of left recursion).

It is practical to put at the end of production paths (i.e. to add them as new lists to the production trees) special nodes signaling which production ends at given position. It is important especially in a situation when one production is a prefix of another one.

Figures 3.4 and above use this notation – only the production nodes are not labeled by full productions but only by the nonterminal from their left-hand side (let it be A) and with an index of the position of the production within A-productions as given in the beginning – Fig. 3.1. Now we can join the common beginnings of all the production paths so that they will make trees – *production trees*, like in Fig. 3.4.

As the example of a simple arithmetic expression does not fully show the

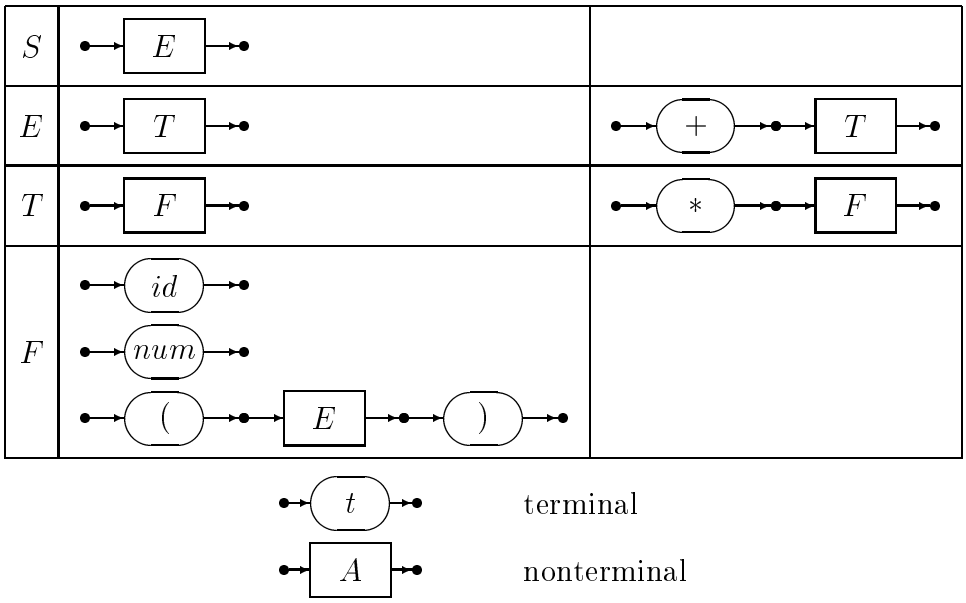


Figure 3.3: Productions written as paths in a directed acyclic graph

joining of production paths (the grammar does not contain production with common left-hand sides and common nonempty prefixes also from their right-hand sides), this detail is shown on a set of productions of a nonterminal of a simple command – see Fig. 3.5 and Ex. 3.2.

Example 3.2 (Productions of a simple command):

The part of grammar that is dealt with here consists only of cmd-productions (i.e. productions with a nonterminal cmd on their left-hand side). Other nonterminals (E, cond, cmds) are only referred here.

cmd → *begin* cmds *end*
cmd → *id* := E
cmd → *id* (E)
cmd → *read id*
cmd → *write* E
cmd → *write str*
cmd → *while* cond *do* cmd

where *begin*, *do*, *end*, *read*, *while*, *write*, :=, (,), *id*, and *str* are terminals, E, cmd, cond, and cmds are nonterminals. □

Figures 3.4 and 3.6 can be used also as a definition of a parser that can be implemented as recursive descent parser or as pushdown automaton or lookahead pushdown automaton (see Chap. 7).

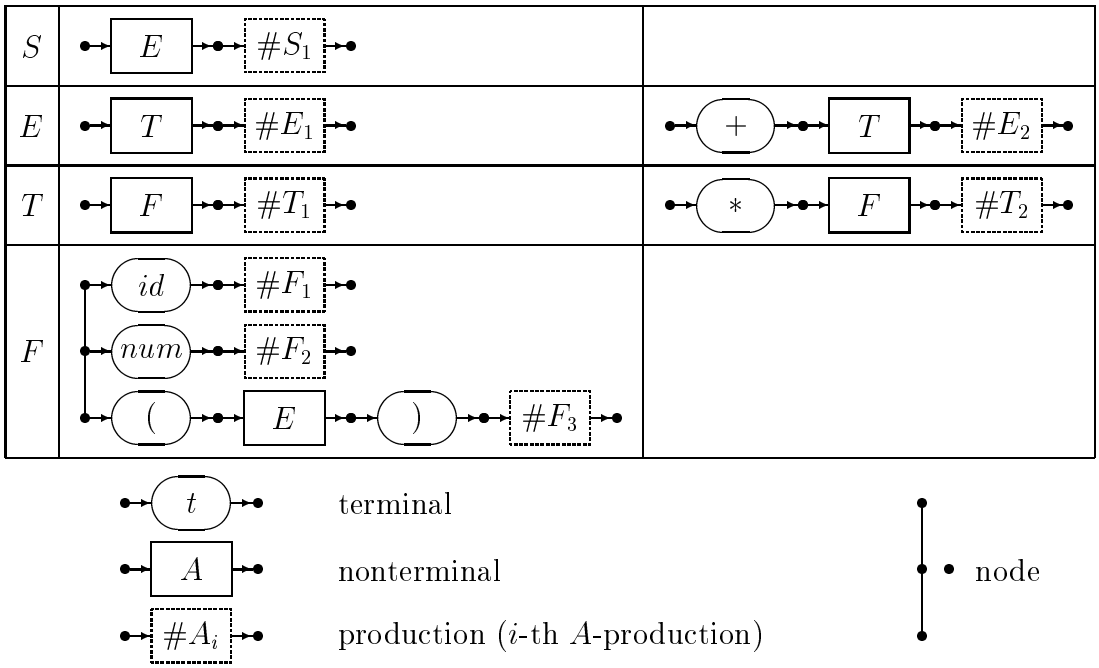


Figure 3.4: Productions written as paths in production trees

Pushdown symbols can be implemented as pointers or references to nodes or edges of production trees. Such reference will be graphically represented as corresponding tree with a suitably marked node. The mark is in Fig. 3.6 represented by a circle.

Such reference can be represented by dotted notation as shown in Fig. 3.6. Reference to the tree T of nonterminal A with referenced node is bijectively assigned to the dotted production (item) set

$$\{A \to u.hv \,|A \to uhv \in P_G\}\,.$$

The use of both notations is dependent on the focus: when more detailed view is needed, the graphical notation is used, when more global view is needed (to keep track with changes, or to get more situations in one view), the dotted notation is used.

Both the notations can be useful to build up the parser structure.

3.3 Usage of Production Trees

In the parser construction it is common to use *dotted productions*[1] – productions divided into two parts: one corresponding to already "processed"

[1]Dotted productions are described e.g. in [NL94] or [AU72].

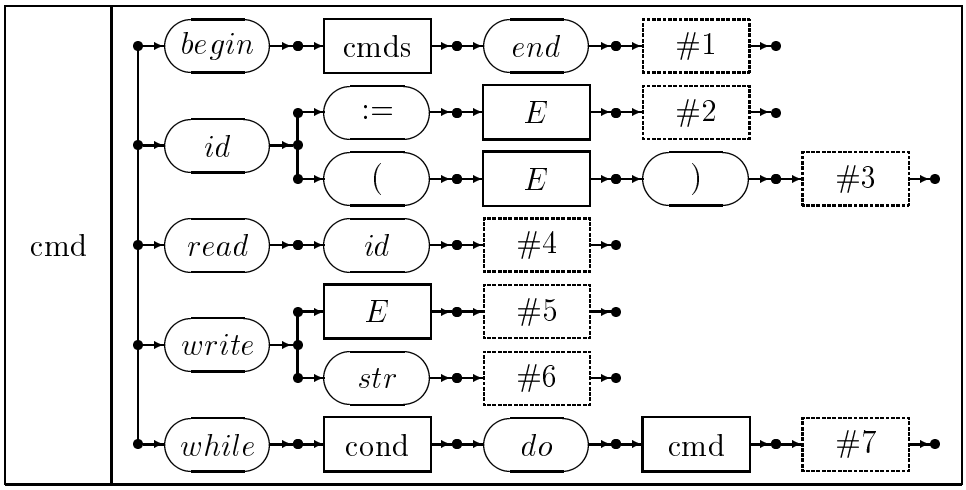


Figure 3.5: Simple command production tree

part of the productions and the second for the rest of them. This division is signalized by a dot within the right-hand side of the production description. For any production $A \to \alpha\beta \in P$ there is a set of dotted productions starting by '$A \to .\alpha\beta$' and ending by '$A \to \alpha\beta.$'.

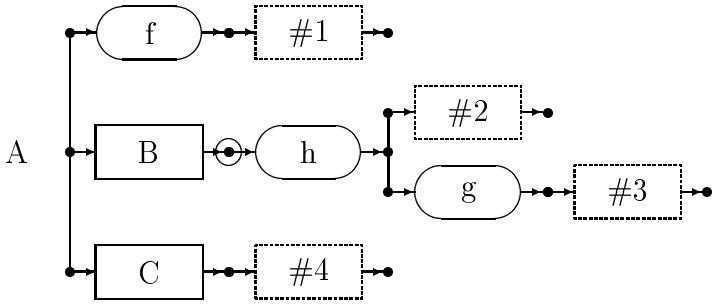
To show the work of a parser constructed this way it is necessary to draw the entire automaton again and again and show every time the change of its state and the situation on its pushdown.

To be able to compress this "animation" into a few pages, a new kind of a description of a configuration of the automata is used: Any node (drawn as a point or column of points connected with a vertical line – see Fig. 3.4) represents a position within some production. In some cases (at the production beginnings and when more productions from the same group have some common prefix) can the same position in the production tree represent several different dotted productions ($[A \to .\alpha]$, $[A \to .\beta]$). These parsing situations form equivalence classes and any of them can be used to represent its class[2].

The initial situation is therefore $[S \to .\alpha]$ for any of the S-productions $S \to \alpha \ \in P_G$ if S is a starting symbol of the grammar G.

This notation corresponds to the one used in the pictures: any node in the tree is represented by one equivalence class determined by the nonterminal on the left-hand side of the production and by the part of the right-hand side of the production before the dot. The nonterminal (let us denote it A)

[2]It is better to concentrate on actual location within the structure. The equivalence classes over dotted productions serve to this purpose well.



Position marked by a circle corresponds to the following dotted position:

$$[A \to B.h]_{=_G} = \left\{ \begin{array}{ll} A & \to B.h \\ A & \to B.hg \end{array} \right\}$$

Figure 3.6: Position within production tree in graphical and dotted notations

determines a pair of production trees (one for A-productions without left recursion and one for the left recursive ones – if there are any). The part before the dot determines the path to the position in the tree. The rest of the production (if written) only shows that the prefix belongs to some existing production.

The root of a production tree for left recursive A-production can be denoted as [A→A.α] if $A \to A\alpha$ is some production in the grammar.

How such a parser/generator works? As an example a parsing/generation of a simple arithmetic expression "3+2" is used. For better understanding every time the proper representatives of the equivalence classes are used.

input/output	state	pushdown
start of the computation		
$.3 + 2$	$[S \to .E]$	ε
descent to the nonterminal E		
$.3 + 2$	$[E \to .T]$	$[S \to E.]$
descent to the nonterminal T		
$.3 + 2$	$[T \to .F]$	$[S \to E.]$ $[E \to T.]$
descent to the nonterminal F		
$.3 + 2$	$[F \to .num]$	$[S \to E.]$ $[E \to T.]$ $[T \to F.]$

input/output	state	pushdown
generation/reading of '3'		
$3. + 2$	$[F \to num.]$	$[S \to E.]$ $[E \to T.]$ $[T \to F.]$
finishing of F (reduction by $\#F \to num$)		
$3. + 2$	$[T \to F.]$	$[S \to E.]$ $[E \to T.]$
End of expansion by the production $T \to F$. Now it is possible to try whether it is possible to apply some of the left-recursive productions for the nonterminal T. At this moment it is none, and therefore the expansion of the nonterminal T ends.		
$3. + 2$	$[E \to T.]$	$[S \to E.]$
End of expansion by the production $E \to T$. Now it is possible to try whether it is possible to apply some of the left-recursive E-productions. Such production is $E \to E + T$.		
$3. + 2$	$[E \to E. + T]$	$[S \to E.]$
Reading/generation of the terminal '+'		
$3. + 2$	$[E \to E + .T]$	$[S \to E.]$
descent to the nonterminal F		
$3 + .2$	$[T \to .F]$	$[S \to E.]$ $[E \to E + T.]$
descent to the nonterminal F		
$3 + .2$	$[F \to .num]$	$[S \to E.]$ $[E \to E + T.]$ $[T \to F.]$
generation/reading of '2'		
input/output	state	pushdown
$3 + 2.$	$[F \to num.]$	$[S \to E.]$ $[E \to E + T.]$ $[T \to F.]$
finishing of F (reduction by $\#F \to num$)		
$3 + 2.$	$[T \to F.]$	$[S \to E.]$ $[E \to E + T.]$
finishing of $T \to F \wedge$ it is not possible/we do not want to apply any left recursive production $\Rightarrow$ finishing of T		

$3+2.$	$[E \to E+T.]$	$[S \to E.]$
finishing of $E \to E+T \wedge$ it si not possible/we do not want to apply any left recursive production $\Rightarrow$ finishing of E		
$3+2.$	$[S \to E.]$	ε
finishing of $S \to E$, the final position was reached $\Rightarrow$ parsing/generation ends.		

It can be useful for the needs of parsing to have some hints which computational (parsing) path leads to the result if there are more ones possible. Therefore existing nodes are extended so that there can be written the lookaheads. They will be used to determine which continuation of computation leads to the result (i.e. it allows us to select proper edge along which the parsing should continue). In our example the lookahead of the length 1 is satisfying. To cover all possible situations we need a special terminal symbol representing end of input (*eoi*). Precomputed values of all lookahead functions are available in Tab. 3.1.

Nonterminal	First	Follow	DLRF	NLRF
S	*num*, *id*, (	*eoi*		*eoi*
E	*num*, *id*, (	+,), *eoi*	+	), *eoi*
T	*num*, *id*, (	+, ∗,), *eoi*	∗	+,), *eoi*
F	*num*, *id*, (	+, ∗,), *eoi*		+, −,), *eoi*

Table 3.1: Precomputed lookaheads for a simple arithmetic expression

Now it is possible to write down the lookaheads into extended nodes. These lookaheads can help to determine which branch of possible ones matches the input (or at least has a chance to match it). As the size of the resulting picture for a whole parser is too big, we restrict ourselves to the parsing tree for the nonterminal F:

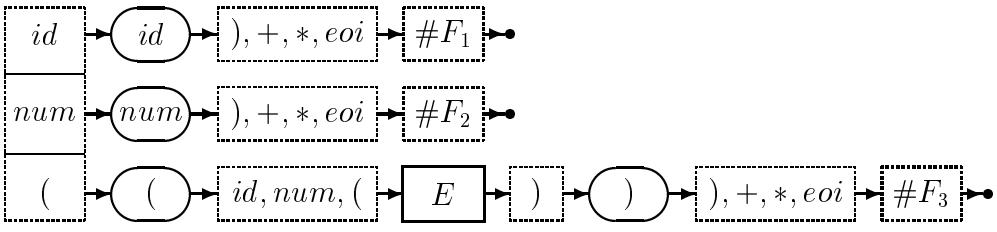


Figure 3.7: Nonterminal F with precomputed lookaheads

Chapter 4

Changes in Parser Structures

We control/implement the extensions of parsing structures by changes of production tree forests. Processing of extensible languages needs extensible/modifiable parsers. In this chapter we will show that the ability of forests of production trees (and kind parsers based on them) to be extended/modified:

4.1 Extension

Accepted language extension implemented via adding of new productions to the already existing grammar, occasionally also on the adding of new terminal and nonterminal symbols. In the case of translational grammars also output terminal symbols and for compiler grammars also semantic symbols can be added.

Such changes are in some sense additive: the existing parts are left intact and the new parts are only added to the existing ones. It allows us to extend parsers also at parse-time – that means that language extensions may be part of the parsed text.

A structure (or language or grammar – that is just point of view) extension can be demonstrated by an example. A new terminal symbol "$-$" and two productions $E \to E-T$ and $E \to -T$ will be added to our initial example of a simple arithmetic expression. The original and modified structure of the nonterminal E are shown in Fig. 4.1.

The ability of run-time extension is probably the main advantage of kind parsing.

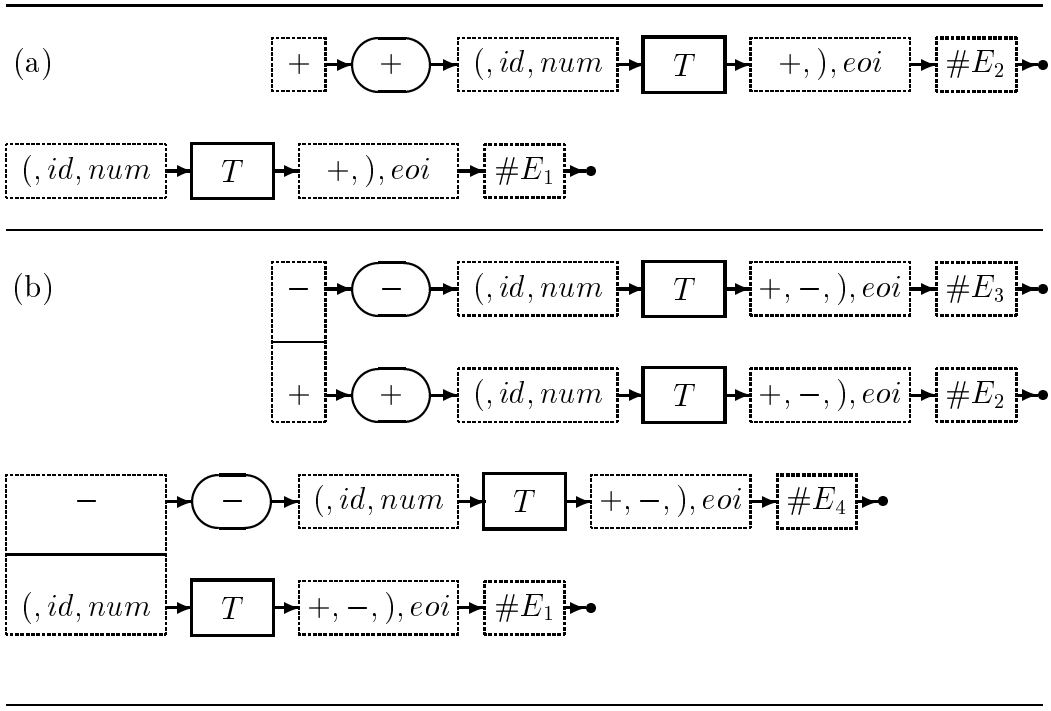


Figure 4.1: Original (a) and extended (b) production tree of the nonterminal E of a simple arithmetic expression. (In both cases the tree in the bottom and left represents the structure for non-recursive productions, the tree on top and right represents the left recursive productions.)

4.2 Restriction

Sometimes it can be useful to restrict the recognized language. It can be used for removal of some previous extension or to remove some parts that can conflict with some planned future extension. It is also possible to remove some language parts from preinstalled configuration according to payment.

The language/grammar/parser restriction is more dangerous than the extension: during extension no part is removed, no opportunity for successful termination of the parse is removed. By restriction some problems arise: it is possible, that if there are no limitations for language/grammar restrictions, some live parts of the parser that are needed to finish the computation would be removed.

As it is not very good to cut a bough we are sitting on, therefore we do not recommend to do the similar thing also in the restricting of the here presented parsing structure: It is possible to cut paths in the production trees only in the cases when all the situation that are unsolved on the pushdown are

preserved and also if for all the nonterminals that must be gone through at least one path is preserved – they also cannot be removed from the grammar.

Such restrictions ensure that the computation can be (for some proper input) successfully finished.

Chapter 5

Kind Grammars

5.1 Production Types

Productions with given (A) nonterminal on its left-hand side are usually denoted as A-productions [AU72].

Each production of an arbitrary proper context-free grammar can be assigned to exactly one of following five groups formally defined below:

1. productions without left recursion (NLRP),
2. productions with direct left recursion (DLRP),
3. productions with indirect left recursion (ILRP),
4. productions with hidden left recursion (HLRP), and
5. productions with hidden indirect recursion (HILRP).

This categorization is fixed for any context-free grammar.

If there are in some grammar productions from the third, fourth or fifth group, the grammar is usually harder to understand, and the language defined by it tends to be hard to read. For a description of an easy-to-read language it is better when also the grammar is easy to understand. Therefore we restrict ourselves to grammars with productions only from the first two groups.

Definition 5.1 (Productions with direct left recursion[1]):

$$DLRP_G(A) = \{A \to A\alpha \mid A \to A\alpha \in P_G \ \wedge \ \alpha \neq \varepsilon\}$$

[1] DLRP = directly left recursive productions

$$DLRP_G = \bigcup_{A \in \mathrm{N}_G} DLRP_G(A)$$

Definition 5.2 (Productions without left recursion[2]):

$$NLRP_G(A) = \{A \to \alpha \mid A \to \alpha \in P_G \wedge (\forall \beta)\alpha \not\Rightarrow_G^* A\beta\}$$
$$NLRP_G = \bigcup_{A \in \mathrm{N}_G} NLRP_G(A)$$

Definition 5.3 (Productions with indirect left recursion[3]):

$$ILRP_G(A) = \left\{ A \to \alpha \;\middle|\; \begin{array}{l} A \to \alpha \in P_G \wedge (\exists \beta)\alpha \Rightarrow_G^+ A\beta \\ \wedge\ A \to \alpha \notin DLRP_G(A) \end{array} \right\}$$
$$ILRP_G = \bigcup_{A \in \mathrm{N}_G} ILRP_G(A)$$

Definition 5.4 (Productions with hidden left recursion[4]):

$$HLRP_G(A) = \{A \to \alpha A\beta \mid A \to \alpha A\beta \in P_G \wedge \alpha \neq \varepsilon \wedge \alpha \Rightarrow_G^+ \varepsilon\}$$
$$HLRP_G = \bigcup_{A \in \mathrm{N}_G} HLRP_G(A)$$

Definition 5.5 (Productions with hidden indirect left recursion[5]):

$$HILRP_G(A) = \left\{ A \to \alpha\beta \;\middle|\; \begin{array}{l} A \to \alpha\beta \in P_G \ \wedge\ \alpha \neq \varepsilon \ \wedge\ \alpha \Rightarrow_G^+ \varepsilon \ \wedge \\ (\exists \gamma)(\beta \to_G^+ A\gamma) \ \wedge\ (\forall \delta)(\beta \neq A\delta) \end{array} \right\}$$
$$HILRP_G = \bigcup_{A \in \mathrm{N}_G} HILRP_G(A)$$

Note 5.6 The index G by $NLRP$, $DLRP$, $ILRP$, and $HILRP$ can be omitted whenever it is clear which grammar G is mentioned.

[2] NLRP = non-left-recursive productions
[3] ILRP = indirectly left recursive productions
[4] HLRP = hiddenly left-recursive productions
[5] HILRP = hiddenly indirectly left-recursive productions

Example 5.7 (Production types):
Let us have grammar with productions $p_1 = A \to a$, $p_2 = B \to \varepsilon$, $p_3 = A \to Ab$, $p_4 = A \to BAc$, $p_5 = A \to Cd$, $p_6 = C \to Ae$, and $p_7 = A \to BCf$. Then

$$\begin{aligned} NLRP &= \{p_1, p_2\} \\ DLRP &= \{p_3\} \\ HLRP &= \{p_4\} \\ ILRP &= \{p_5, p_6\} \\ HILRP &= \{p_7\} \end{aligned}$$

□

5.2 Kind Grammars

Kind grammars are a superset of strong LL grammars allowing direct left recursion. Kind grammars possess many good properties of LL grammars and include the grammars for arithmetical expressions and lists – the best known cases where left recursive productions are usually used.

Definition 5.8 (Follow without left recursion):
Let CFG G has only productions from NLRP and DLRP. A set of terminal strings of the length k that can occur immediately after given nonterminal A except its left recursion is called *follow of A without (its) left recursion (NLRF(A) – non-left-recursive follow)*. Formally:

$$NLRF_G^k(A) = \left\{ \begin{array}{l|l} a_1 \dots a_m \in & m \leq k \ \wedge B \to \alpha A \beta \in P_G \wedge \\ First_G^k(\beta \cdot Follow_G^k(B)) & ((\alpha \not\Rightarrow_G^* \varepsilon) \vee (B \neq A)) \end{array} \right\}$$

If it is obvious what grammar is mentioned, it is possible to write $NLRF_k(A)$. If $k = 1$, it is also possible to write just $NLRF(A)$.

Definition 5.9 (Follow after (direct) left recursion):
A set of terminal strings of the length k that may follow after given nonterminal A in the beginning of its left recursion is called *follow of A after its (direct) left recursion (DLRF(A) – direct left recursion follow)*.

$$DLRF_G^k(A) = \left\{ \begin{array}{l|l} a_1 \dots a_m \in & m \leq k \ \wedge A \to A\alpha A\beta \in P_G \\ First_G^k(\alpha \cdot Follow_G^k(A)) & \wedge \ \alpha \neq \varepsilon \end{array} \right\}$$

If it is obvious which grammar is mentioned, it is possible to write $DLRF_k(A)$. If $k = 1$, it is also possible to write just $DLRF(A)$.

Definition 5.10 (Common prefix of A-productions):
We say that two productions $A \to \alpha\beta$ and $A \to \gamma\delta$ from a CFG G share *common A-prefix* α iff $(\alpha = \gamma) \wedge (\beta \neq \delta)$.

We say that a set of productions $Q = \{p_1, \ldots, p_m\} \subset P_G$ has a common A-prefix α, if α is a common A-prefix for any two productions $p_i, p_j \in Q; i \neq j$.

If moreover $\alpha \neq \varepsilon$, we say, that the common A-prefix is *nonempty*.

We say that the common A-prefix α is *longest* A-prefix of a set of productions iff for any string α', such that α' is a proper prefix of α, α' is no common prefix of the given set of productions.

Definition 5.11 ((Strong) k-kind grammar):
A context-free grammar $G = (\mathrm{N}, \Sigma, P, S)$ having only productions without left recursion or with direct left recursion is called *(strong) k-kind* ($\mathrm{SK}(k)$ or $\mathrm{K}(k)$ for short) if for any $A \in \mathrm{N}$ the following two conditions hold:

1. $DLRF_k(A) \cap NLRF_k(A) = \emptyset$;
2. For any two productions being either both from $NLRP(A)$ ($A \to \alpha\beta$, $A \to \alpha\gamma$ and sharing the longest common A-prefix α) or both from $DLRP(A)$ ($A \to A\alpha\beta$, $A \to A\alpha\gamma$ and sharing the longest common A-prefix $A\alpha$) it holds that

$$First_k(\beta \cdot Follow_k(A)) \cap First_k(\gamma \cdot Follow_k(A)) = \emptyset.$$

The above two conditions together may be referred as *(strong) kind condition*.

Definition 5.12 (Kind grammar):
A context-free grammar G is called *kind* if it is k-kind for some $k > 0$. The class of kind grammars is denoted KG.

The name "kind" for the above defined class of grammars was taken for the reason that they are not so restrictive but they allow to describe rather restrictive class of languages. Similar situations in real life can be described e.g. by a Czech word "přívětivý".

Example 5.13 Let us show that grammar of a simple arithmetic expression (in its usual textbook form) $G = (\{\mathrm{N}, \Sigma, P, S\}, \{num, +, -, *, /, (,)\}, P, S\}$ where

$$\begin{aligned} P = \{ & S \to E, \\ & E \to T,\ E \to -T,\ E \to E + T,\ E \to E - T, \\ & T \to F,\ T \to T * F,\ T \to T/F, \\ & F \to num,\ F \to (E)\} \end{aligned}$$

is a 1-kind grammar.

Kind condition has two parts: the first expects two A-productions in DLRP or NLRP for some nonterminal A, the second is non-trivially applicable when there is at least one production in both groups. Therefore the S-production trivially fulfills this condition.

Let us compute the lookahead functions:

Nonterminal	First	DLRF	NLRF
S	$\{num, -, (\}$	$\emptyset$	$\{\varepsilon\}$
E	$\{num, -, (\}$	$\{+, -\}$	$\{), \varepsilon\}$
T	$\{num, (\}$	$\{*, /\}$	$\{+, -,), \varepsilon\}$
F	$\{num, (\}$	$\emptyset$	$\{*, /, +, -,), \varepsilon\}$

The second part of the kind condition ($\forall A: \ DLRF(A) \cap NLRF(A) = \emptyset$) is fulfilled for all nonterminals of grammar G.

The first part of the kind condition is applicable only on following pairs of productions:

1.	$E \to T$	$E \to -T$
2.	$E \to E + T$	$E \to E - T$
3.	$T \to T * F$	$T \to T/F$

Let us check these pairs one by one:

1. $First(T) = \{num, (\}$, $First(-T) = \{-\}$ ... OK,
2. $First(+T) = \{+\}$, $First(-T) = \{-\}$ ... OK, and
3. $First(*F) = \{*\}$, $First(/F) = \{/\}$ ... OK.

The kind conditions are fulfilled for grammar G and for $k = 1$, therefore G is a 1-kind grammar. □

Note 5.14 For static parsing purposes we expect kind grammars to be without inaccessible and useless symbols.

Presence of inaccessible or useless symbols in grammar can be advantageous in parse-time extensible grammars when the symbols may become accessible and useful (see Ex. 9.10).

5.3 Weak Kind Grammars

The class of k-kind grammars can be naturally extended to weak k-kind grammars so that it can contain Ch(k) grammars (and therefore also LL(k) grammars).

Idea of this extension is coming from the relation of LL(k) and strong LL(k) grammars.

Definition 5.15 (Weak k-kind grammar):
A context-free grammar $G = (\mathrm{N}, \Sigma, P, S)$ having only productions without left recursion or with direct left recursion is called *weak k-kind* (WK(k) for short) if for any $A \in \mathrm{N}$ the following two conditions hold:

1. $DLRF_k(A) \cap NLRF_k(A) = \emptyset$;
2. If there exist $w \in \Sigma^*$, $\delta \in (\mathrm{N} \cup \Sigma)^*$ such that $S \Rightarrow_{lm}^* wA\delta$, it holds for any two productions being either both from $NLRP(A)$ ($A \to \alpha\beta$, $A \to \alpha\gamma$ and sharing the longest common A-prefix α) or both from $DLRP(A)$ ($A \to A\alpha\beta$, $A \to A\alpha\gamma$ and sharing the longest common A-prefix $A\alpha$) that $First_k(\beta\delta) \cap First_k(\gamma\delta) = \emptyset$.

The above two conditions together may be referred as *weak kind condition*.

Definition 5.16 (Weak kind grammar):
A context-free grammar G is called *weak kind* if it is weak k-kind for some $k > 0$. The class of weak kind grammars is denoted WKG.

Chapter 6

Power of Kind Grammars

We show here the relations of k-kind and weak k-kind grammars to some selected grammar classes (LL(k), strong LL(k), Ch(k), LC(k), and SLR(k). We prove here that k-kind grammars generate LL(k) languages.

6.1 Relation to Other Grammar Classes

Lemma 6.1 Any reduced strong LL(k) grammar is a k-kind grammar.

Idea: Comparison of definitions of strong LL(k) grammars and k-kind grammars shows that the reduced strong LL(k) grammars are a special case of k-kind grammars. If $DLRP = \emptyset$ and any two productions to be diststinguished have no common prefix, we obtain exactly what is required in the definition of strong LL(k) grammars.

Proof: The Def. 2.28 implies that if $A \to \beta$, $A \to \gamma$ are two distinct A-productions of a strong LL(k) grammar, then

$$First_k(\beta \cdot Follow_k(A)) \cap First_k(\gamma \cdot Follow_k(A)) = \emptyset.$$

Let us test the kind condition on arbitrary two k-kind productions: As there are no left-recursive productions in reduced strong LL(k) grammars, $NLRF(A) \cap DLRF(A) = \emptyset$ holds trivially.

Now we need to test, whether $\forall\ A \to \alpha\beta$, $A \to \alpha\gamma \in$ DLRP $\vee$ $A \to \alpha\beta$, $A \to \alpha\gamma \in$ NLRP such that α is the longest common prefix of the productions implies that

$$First_k(\beta \cdot Follow_k(A)) \cap First_k(\gamma \cdot Follow_k(A)) = \emptyset. \qquad (6.1)$$

Both productions are from $NLRP(A)$. ($DLRP(A)$ is empty as there are no left recursive productions in reduced strong LL(k) grammars). Only the two possible situations may occur:

1. the two tested LL(k) productions have some nonempty common A-prefix – but of a finite size of $m < k$ or

2. the two tested LL(k) productions have no nonempty common A-prefix.

In the first case (6.1) is fulfilled, as the two productions can be distinguished already using shorter lookahead. In the second case (6.1) is fulfilled trivially. Hence the kind condition must be fulfilled. □

Theorem 6.2 For $k \geq 1$ the class of k-kind grammars is a proper superclass of reduced SLL(k) grammars.

Proof: The class of k-kind grammars is a superclass of reduced SLL(k) grammars – it was shown in Lemma 6.1. The grammar from Ex. 3.1 is 1-kind and is not SLL(k) grammar for any k because there are left recursive productions in it what is not allowed in SLL(k) grammars. □

Corollary 6.3 The class of LL(1) grammars is a proper subclass of 1-kind grammars.

Proof: Follows directly from Th. 6.2 and from the fact that the classes of LL(1) and SLL(1) grammars are equal (see e.g. [SSS90, Lemma 8.39, page 232]). □

Lemma 6.4 Every k-kind grammar is a weak k-kind grammar.

Proof: Let $G = (\mathrm{N}, \Sigma, P, S)$ be an arbitrary k-kind grammar. We will show that it is also a weak k-kind grammar. The general settings (the grammar can have only productions being direct left recursive or being without any left recursion) and the first part of weak kind condition holds from the definition of k-kind grammars:

$$\forall A \in \mathrm{N} : \; DLRF_k(A) \cap NLRF_k(A) = \emptyset.$$

We should therefore concentrate on the second part of the weak kind condition only. It states that if there exist $w \in \Sigma^*$ and $\delta \in (\mathrm{N} \cup \Sigma)^*$ such that $S \Rightarrow_{lm}^* wA\delta$, it holds for any two productions being either both from $NLRP(A)$ ($A \to \alpha\beta$, $A \to \alpha\gamma$ and sharing the longest common A-prefix α) or both from $DLRP(A)$ ($A \to A\alpha\beta$, $A \to A\alpha\gamma$ and sharing the longest common A-prefix $A\alpha$) that $First_k(\beta\delta) \cap First_k(\gamma\delta) = \emptyset$.

Let us show that a fulfilling of the (strong) kind condition implies a fulfilling of the weak kind condition. The second part of the kind condition

states that for any $A \in$ N any two A-productions $A \to \alpha\beta$, $A \to \alpha\gamma$ being both from DLRP(A) or both from NLRP(A) and having the longest common A-prefix α holds that

$$First_k(\beta \cdot Follow_k(A)) \cap First_k(\gamma \cdot Follow_k(A)) = \emptyset. \tag{6.2}$$

Let us show that any two productions (let them be $p_1 = A \to \alpha\beta$ and $p_2 = A' \to \alpha\gamma$) fulfilling the (strong) kind condition fulfill also the weak kind condition.

If $A \neq A'$ or p_1 and p_2 are one from $DLRP(A)$ and one from $NLRP(A)$, the weak kind condition is fulfilled trivially.

If $A = A'$ and p_1, p_2 are both either from $DLRP(A)$ or $NLRP(A)$, then – according the (strong) kind condition – (6.2) holds. We can reformulate (6.2) as follows:

$$First_k(\beta\delta) \cap First_k(\gamma\delta') = \emptyset \tag{6.3}$$

for any δ, δ' such that $S \Rightarrow^* uA\delta$ and $S \Rightarrow^* u'A\delta'$ $(u, u' \in \Sigma^*)$.

As (6.3) holds for any δ and δ', it also holds for $\delta = \delta'$, what is the requirement from the second part of the weak kind condition. □

Lemma 6.5 Every weak 1-kind grammar is a 1-kind grammar.

Proof: Let us show that for $k = 1$ the weak kind condition implies the kind condition: First, in both conditions it should hold for every $A \in$ N that $NLRF(A) \cap DLRF(A) = \emptyset$.

Weak kind condition requires that for all symbol strings $\alpha, \beta, \gamma, \delta$, terminal string w, pairs of A-productions $A \to \alpha\beta$, $A \to \alpha\gamma \in NLRP$ or $A \to A\alpha\beta$, $A \to A\alpha\gamma \in DLRP$ such that $S \Rightarrow^* wA\delta$ and where the first symbol of β is different from the first symbol of γ it holds that

$$First(\beta\delta) \cap First(\gamma\delta) = \emptyset.$$

Let us discuss when δ influences the $First(\beta\delta)$ or $First(\gamma\delta)$. It may happen only when either $\beta \Rightarrow^* \varepsilon$ or $\gamma \Rightarrow^* \varepsilon$.

When both $\beta \Rightarrow^* \varepsilon$ and $\gamma \Rightarrow^* \varepsilon$, then

$$\emptyset \neq First(\delta) \subset First(\beta\delta) \cap First(\gamma\delta)$$

what does not conform the weak kind condition. (In the case that A is not reachable, δ and $Follow(A)$ are not defined, hence the productions fulfill the condition trivially.)

If $\beta \not\Rightarrow^* \varepsilon$ and $\gamma \not\Rightarrow^* \varepsilon$, δ has no influence in this case (and can therefore be replaced by $Follow(A)$). Under such condition the weak kind condition implies the kind condition.

The last case – one of β, γ can be rewritten to ε, the second one cannot; this situation is symmetric, let us therefore assume that $\beta \Rightarrow^* \varepsilon$ and that γ cannot be rewritten to ε – this implies that $First(\delta) \cap First(\gamma) = \emptyset$ (and $First(\delta) \cap First(\beta) = \emptyset$ for the symmetric problem). As

$$Follow(A) = \{a \in First(\delta) | \exists w \in \Sigma^* : \ S \Rightarrow^* wA\delta\},$$

we may write

$$First(\beta Follow(A)) \cap First(\gamma Follow(A)) = \emptyset$$

instead of

$$First(\beta\delta) \cap First(\gamma\delta) = \emptyset.$$

We then conclude that for $k = 1$ the kind and weak kind conditions are equivalent. □

Theorem 6.6 For $k \geq 2$ the grammar class LL(k) is incomparable with the class of k-kind grammars.

Proof: The grammar $S \to Sa \mid a$ is an example of a 1-kind grammar which is not LL(k) for any k (and hence also not for $k \geq 2$).

The grammar (adapted from [Chy84], Ex. 5.6.2)

$S \to 0A00B \mid 1A10B$
$A \to 1 \mid \varepsilon$
$B \to 1B \mid 0B \mid \varepsilon$

is an example of LL(2) grammar which is not k-kind for any k.

Different productions for nonterminals S and B starts with different terminals and they are therefore trivially distinguishable – i.e. they fulfill the LL(2) condition.

The LL(2) condition for the nonterminal A is also fulfilled:
if $S \Rightarrow^* 0A00B$, then $First_2(100) \cap First_2(\varepsilon 00) = \emptyset$ and if $S \Rightarrow^* 1A10B$, then $First_2(110) \cap First_2(\varepsilon 10) = \emptyset$.

The k-kind condition (and in this case also SLL(k) condition) is not fulfilled for any k: $10^{k-1} \in Follow_k(100B) \cap Follow_k(\varepsilon 10B)$. □

Theorem 6.7 Every reduced Ch(k) grammar is a weak k-kind grammar.

Proof: Reduced Ch(k) grammars do not have left-recursive productions (see Th. ,2.49). In other words, for every reduced Ch(k) grammar G $DLRP_G = \emptyset$ and therefore $DLRF_k(A) = \emptyset$ for each nonterminal A (as $DLRP(A) = \emptyset$). Hence the weak kind condition is therefore reduced to the definition of (reduced) Ch(k) grammar. □

Corollary 6.8 Every reduced LL(k) grammar is a weak k-kind grammar.

Proof: It directly follows from the facts that every LL(k) grammar is a Ch(k) grammar (Th. 2.48), and that every reduced Ch(k) grammar is a weak k-kind grammar (Lemma 6.7). □

Corollary 6.9 For $k \geq 2$ the class of (strong) k-kind grammars is a proper subclass of weak k-kind grammars.

Proof: Follows directly from the following facts:

1. The class of k-kind grammars is a subclass of weak k-kind grammars (Lemma 6.4).
2. The class of LL(k) grammars is a proper subclass of weak k-kind grammars.
3. For $k \geq 2$ there are grammars being LL(k) and being not k-kind (Th. 6.6). □

Note 6.10 The class of k-kind grammars is incomparable with SLR(k) grammars – as shown in [SSS90] even the class of SLL(k) is incomparable with SLR(k), i.e. k-kind grammars cannot be a subclass of SLR(k) grammars and as k-kind grammars do not allow indirect left recursion and SLR(k) grammars do, classes of k-kind grammars and SLR(k) grammars are incomparable.

Lemma 6.11 $\forall k \geq 1\ \exists G \in LC(1)$: G is not weak k-kind.

Proof: Let G has productions $S \to A$, $A \to a$, $A \to Cb$, $C \to c$, $C \to Ad$. This grammar is LC(1). As there is an indirect left recursion in G and weak k-kind grammars can have only productions from DLRP and NLRP, G is not weak k-kind for any k. □

Lemma 6.12 There is a 1-kind grammar that is not LC(k) for any k.

Proof: Grammar $S \to aAb$, $S \to aAc$, $A \to aA$, $A \to \varepsilon$ is 1-kind (and as there is no left recursion, it is also Ch(1)). The left corner of productions $S \to aAb$ and $S \to aAc$ is the terminal a. According to LC(k) condition $First(Ab) \cap First(Ac)$ should be ε but it is not so, as $a^k \in First(Ab) \cap First(Ac)$. □

Corollary 6.13 The grammar classes of k-kind grammars, weak k-kind grammars and Ch(k) grammars are incomparable with LC(k) grammars for any $k \geq 1$.

Lemma 6.14 The class of k-kind grammars is incomparable with the class of Ch(k) grammars.

Proof: The proof is divided into two parts: in the first one it is shown that the class of Ch(k) grammars is no subclass of k-kind grammars and in the second that the class of k-kind grammars is no subclass of Ch(k) grammars.

1. The class of k-kind grammars is incomparable with the class of LL(k) grammars (see Th. 6.6) and the class of LL(k) grammars is a proper subclass of Ch(k) grammars (see Th. 2.48). Hence there are grammars that are Ch(k) grammars and not k-kind grammars (for an example see the proof of Th. 6.6) and therefore **the class of Ch(k) grammars is no subclass of k-kind grammars**.

2. Theorem 2.49 states that any reduced Ch(k) grammar cannot be left recursive. On the other hand, the k-kind grammars (the reduced ones inclusive) can have direct left recursive productions. An example is a 1-kind grammar of a simple arithmetic expression from Ex. 3.1. This grammar is 1-kind as is not Ch(k) grammar for any k. Hence **the class of k-kind grammars is no subclass of Ch(k) grammars**. □

Note 6.15 The class of weak k-kind grammars is incomparable with the class of SLR(k) grammars.

As a subset of weak k-kind grammars, the SLL(k) grammars exceed SLR(k) grammars (see [SSS90]), weak k-kind grammars cannot be subclass of SLR(k) grammars. Similarly, as SLR(k) grammars allow indirect left recursion and weak k-kind grammars do not, weak k-kind grammars cannot be superclass of SLR(k) grammars.

6.2 Expressive Power

What languages are generated by kind grammars and weak kind grammars? Are they deterministic context-free languages, LL-languages, or is there yet another language class somewhere between these two?

Let us solve this problem by returning to 'play' with the parsing structures. They will be transformed (simplified) in such a way that the determined language will be kept the same.

As used parsing structure is a forest of production trees, we should start with the description of production trees:

Definition 6.16 (Tree representation of a set of strings):
Let Δ be a set of symbols (an alphabet). Let S be a set of strings over Δ such that no $s \in S$ is a prefix of another $s' \in S$. Let $T = (V, E)$ be a split tree. We say that T is a *tree representation of* S if there is a labeling $l : E \to \Delta$ such that concatenating labels along any path p from the root to a leaf (we call such path *complete*) gives a string s from S. (We then say that s is represented by p.) For any node it holds that edges starting in this node are labeled by different symbols. It should moreover hold that different complete paths in T represent different strings from S and that there is a bijection between the paths from root to leaf in T and the strings from S.

Definition 6.17 (Production tree):
Let A be a nonterminal of a (weak) kind grammar $G = (\mathrm{N}, \Sigma, P, S)$. Let Π be a set of symbols such that $\Pi \cap (\mathrm{N} \cup \Sigma) = \emptyset$. Let there is a bijection between Π and P.

Production tree for $NLRP(A)$ is a tree representation of right-hand sides of all productions from $NLRP(A)$ concatenated with the symbols uniquely identifying them.

Production tree for $DLRP(A)$ is a tree representation of tails of right-hand sides of all productions from $DLRP(A)$ concatenated with the symbols uniquely identifying them.

The addition of special symbols at the production ends is necessary, as there can be A-productions $p_1, p_2 \in P$ such that right-hand side of p_1 is a proper prefix of right-hand side of p_2 what can cause problems.

The identification of the productions is necessary whenever we need more complex result of the parsing than YES or NO.

Production trees are often used as a documentation tool during the maintenance of compilers.

We show that it is possible to reorganize and decompose production trees for a (weak) k-kind grammar so that the new structure will define the same language as the original one and at the same time the new structure will represent a SLL(k) (LL(k)) grammar.

There will be performed only two types of transformations:

1. left recursion removal, and

2. 'simplifying' of production trees – they will be 'cut' in such a way that at the end they will fork in their roots only.

At the beginning let us look how it works in the nature of production trees (for simplicity the trees without lookaheads are used). As an example the nonterminal E from the grammar of a simple arithmetic expression is used – see Fig. 6.1.

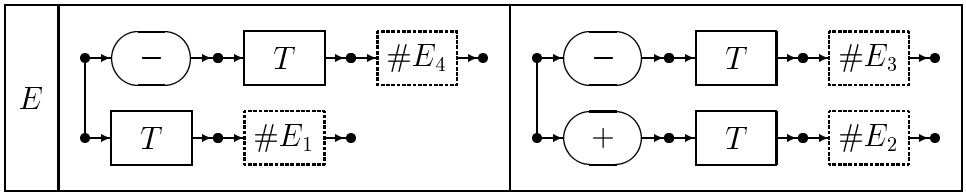


Figure 6.1: Nonterminal E of a simple arithmetic expression

For this nonterminal (E) there can be defined a new one (E'), which is inserted at the end of all E-productions – just before the leaves labeled with production identification – Fig. 6.2.

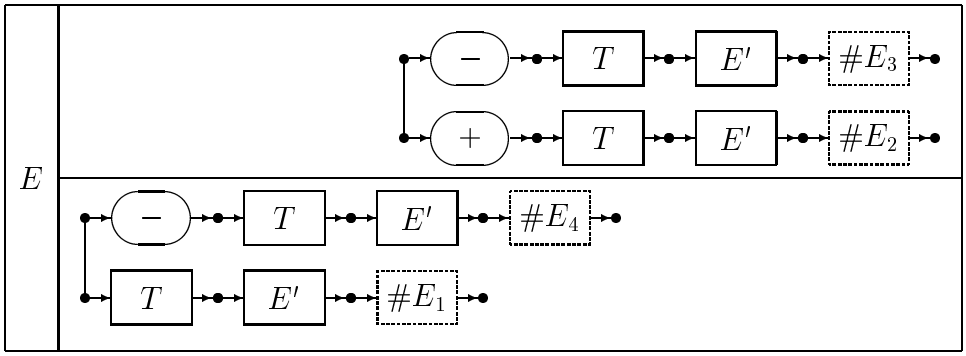


Figure 6.2: Nonterminal E of a simple arithmetic expression after addition (insertion) of a new nonterminal E' at the ends of the productions

Now the production tree of left recursive E-productions can be separated and transformed (relabeled) to be the production tree of right recursion of the nonterminal E' – now it is advantageous that the edge for the nonterminal E (the recurring nonterminal) was already removed from this production tree. For actual situation see Fig. 6.3.

After addition of the production $E' \to \varepsilon$ that allows to stop the recursion of E' the left recursion of E is successfully removed – Fig. 6.4.

The left recursion of other nonterminals can be removed similarly. Current grammar (and production tree forest) may violate rules for being an

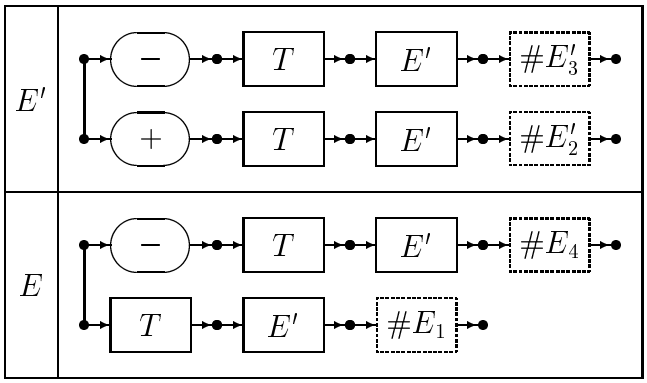


Figure 6.3: Nonterminal E of a simple arithmetic expression and a germ of a new nonterminal E'

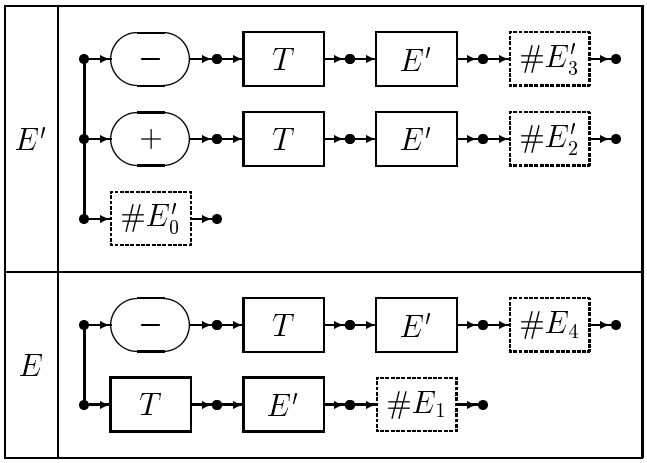


Figure 6.4: Left recursion removal – nonterminals E and E' of a simple arithmetic expression

LL-grammar only in one feature – it can contain productions with common prefixes of several productions. The simple arithmetic expression does not need such feature, so let us switch to our second example, to the simple command nonterminal and its production tree.

Such critical situations can be removed. The process is shown in Fig. 6.5–6.7. In principle all productions with common prefixes are substituted by groups of productions consisting of the one representing the common prefix and several ones created from the original ones by replacing the common prefix by the newly defined nonterminal. In the 'structural' view it is possible to talk about removing path forks by cutting them out of the original structure and making them separate production trees with 'forks' only at their

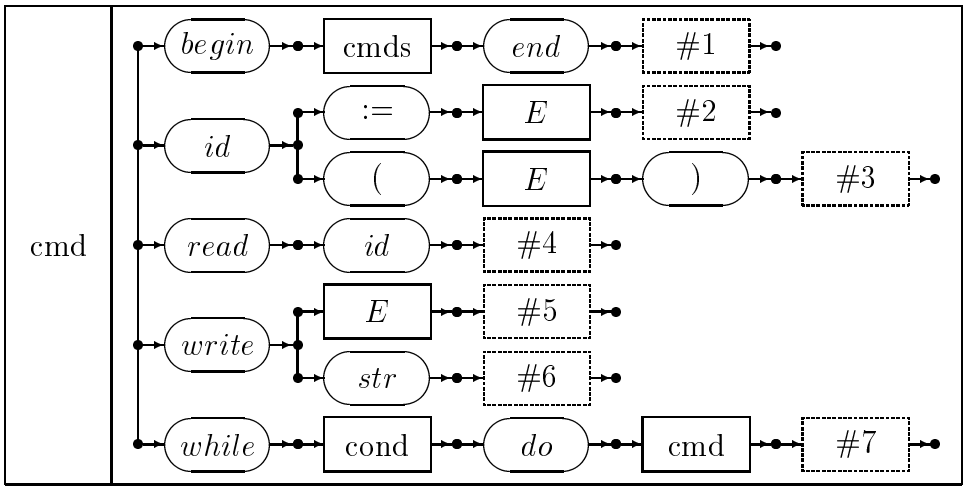


Figure 6.5: Production tree cutting through – original tree

roots – see the changes between Figs. 6.5 and 6.6 and between Figs. 6.6 and 6.7. These changes should be done in the order given by decreasing length of the common prefixes, respectively from leaves to roots in the terms of the production tree nature.

The structure shown in Fig. 6.7 already fulfills all the requirements for LL(1) grammars.

All the above described changes did not influence recognized language.

Left Recursion Removal.

Left recursion removal is mentioned in many books without detailed proofs (see e.g. [AU72, pp. 344–345] or [Har78, lemma 4.6.2, pp. 111–112]). As we want to show that it preserves the property "being a (weak) kind grammar" requiring to handle the grammars with ε-productions, we need to go into details of this topic.

Let us modify the common leftmost derivation to prefer a selected nonterminal:

Definition 6.18 (Semileftmost derivation preferring A):

A relation $x \Rightarrow_{G\ slmA} y$ of sentential forms generated by CFG $G = (\mathrm{N}, \Sigma, P, S)$ is *direct semileftmost derivation preferring* A, A is a selected nonterminal of G, if and only if $x = \alpha B\gamma$, $y = \alpha\beta\delta$ (i.e. $\alpha B\gamma \Rightarrow_{G\ slmA} \alpha\delta\gamma$) where either

$$(B = A) \wedge (A \text{ is not in } \alpha) \wedge (A \to \beta \in P)$$

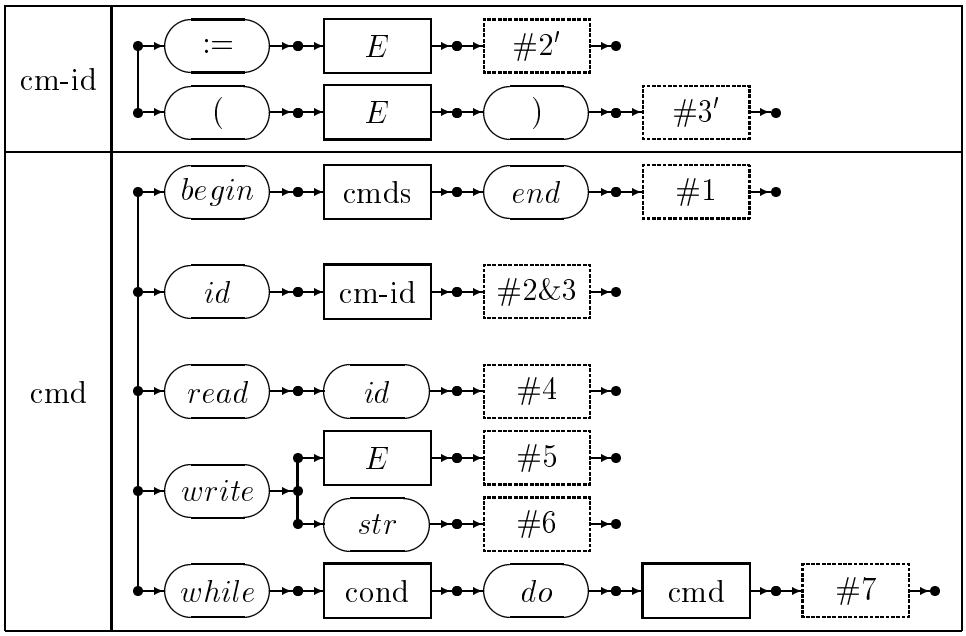


Figure 6.6: Production tree cutting through – partially cut through tree with separated fork

or

$$(B \neq A) \wedge (\alpha \in \Sigma^*) \wedge (A \text{ is not in } \alpha B \gamma) \wedge (B \to \beta \in P).$$

Informally: the leftmost occurence of A is rewritten, if any; otherwise the leftmost nonterminal is rewriten[1].

A positive closure ($\Rightarrow^+_{G\ slmA}$) is called *nontrivial semileftmost derivation preferring* A and reflexive and transitive closure ($\Rightarrow^*_{G\ slmA}$) *semileftmost derivation preferring* A.

The definition of semileftmost derivation is a bit more general than we need – in our case the preferred nonterminal occurs in any derivation step at most once.

Definition 6.19 (Direct left recursion removal from A):
Transformation of a context-free grammar $G_1 = (\mathrm{N}_1, \Sigma, P_1, S)$ to a context-free grammar $G_2 = (\mathrm{N}_2, \Sigma, P_2, S)$ such that $\mathrm{N}_2 = \mathrm{N}_1 \cup \{A'\}$, $A' \notin \mathrm{N}_1$, $A \in DLRP_{G_1}$, $P_1(A) = \{A \to \alpha_1, \ldots, A \to \alpha_n, A \to A\beta_1, \ldots, A \to A\beta_m\}$ be a set of all A-productions ($\forall i : \beta_i \not\Rightarrow^*_{G_1} \varepsilon$) of G_1 where no α_i starts with A. Let further $Q = \{A \to \alpha_1 A', \ldots, A \to \alpha_n A', A' \to \beta_1 A', \ldots, A' \to \beta_m A', A' \to \varepsilon\}$ and $P_2 = (P_1 \backslash P_1(A)) \cup Q$.

[1] It follows from the properties of derivations over context-free grammars that $C \Rightarrow^*_{G_2} w$ iff $C \Rightarrow^*_{G_2\ slmA} w$.

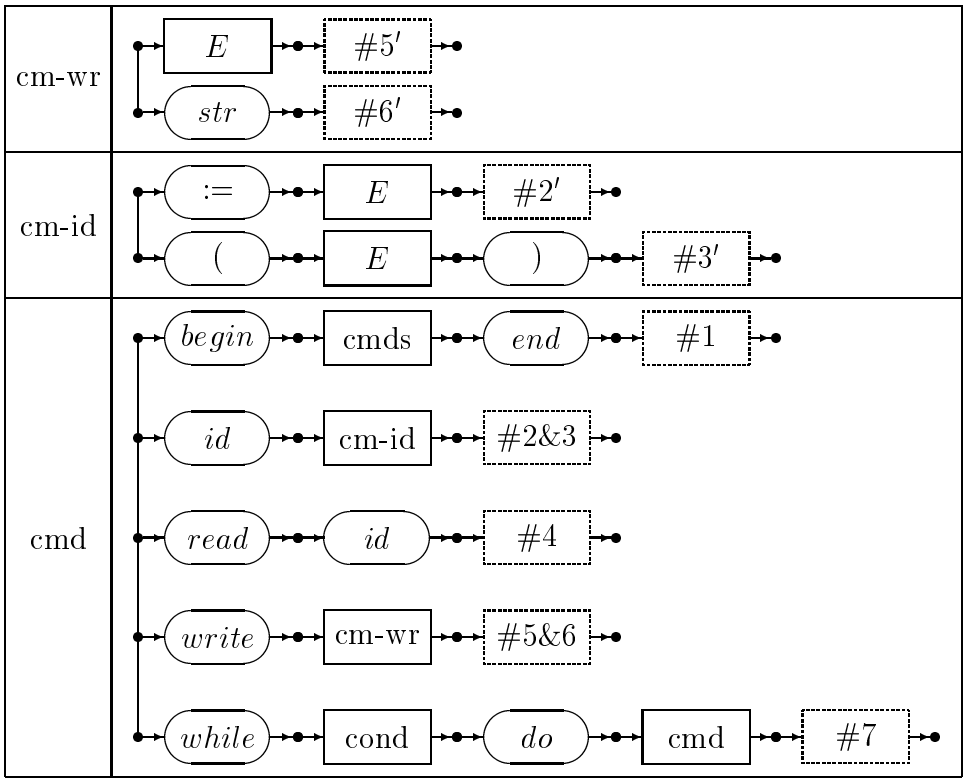


Figure 6.7: Production tree cutting through – a forest with forks only in the roots of separate trees

Definition 6.20

Let G_1 and G_2 be grammars from Def. 6.19. We call the derivation

$$A \Rightarrow_{G_1,lm} A\beta_{j_m} \Rightarrow^*_{G_1,lm} \alpha_i\beta_{j_1} \ldots \beta_{j_m}$$

A-derivation, and the derivation

$$A \Rightarrow_{G_2,lm} \alpha_i A' \Rightarrow_{G_2,slmA'} \alpha\beta_{i_1} A' \Rightarrow^*_{G_2,slmA'} \alpha_i\beta_{j_1} \ldots \beta_{j_m}$$

A'-derivation.

Lemma 6.21 Let G_1 and G_2 be grammars from Def. 6.19. Then to every A-derivation in G_1

$$A \Rightarrow_{G_1,lm} A\beta_{j_m} \Rightarrow^*_{G_1,lm} \alpha_i\beta_{j_1} \ldots \beta_{j_m}$$

there is a corresponding A' derivation in G_2

$$A \Rightarrow_{G_2,lm} \alpha_i A' \Rightarrow_{G_2,slmA'} \alpha\beta_{i_1} A' \Rightarrow^*_{G_2,slmA'} \alpha_i\beta_{j_1} \ldots \beta_{j_m}$$

and vice versa.

Proof: Obvious. □

Lemma 6.22 Let G_1 and G_2 be grammars from Def. 6.19. Then

$$\forall B \in (\mathrm{N}_1 \cap \mathrm{N}_2), w \in \Sigma^* : \ (B \Rightarrow^*_{G_1} w) \Leftrightarrow (B \Rightarrow^*_{G_2} w).$$

Proof:

1. $(B \Rightarrow^*_{G_1} w) \Rightarrow (B \Rightarrow^*_{G_2} w)$ using n A-derivations:

   (a) $n = 0$:
   Let for some $C \in (\mathrm{N}_1 \cap \mathrm{N}_2)$ and $w \in \Sigma^*$ $C \Rightarrow^*_{G_1} w$ using only the productions from $P_1 \cap P_2$. Then $C \Rightarrow^*_{G_2} w$ as the same productions can be used. Similarly, if $C \Rightarrow^*_{G_2} w$, then also $C \Rightarrow^*_{G_1} w$.

   (b) $n \to n+1$:
   Let us expect that it holds for any nonterminal D and word w for up to n A-derivations. Now we show that it must hold also for any nonterminal and word with up to $n+1$ A-derivations.

   Let $C \Rightarrow^*_{lm\ G_1} v\gamma A\delta$; $v \in \Sigma^*$ using only productions from $P_1 \cap P_2$ (hence also $C \Rightarrow^*_{G_2} vA\delta$), let $A \Rightarrow_{G_1,lm} A\beta_{j_m} \Rightarrow^*_{G_1,lm} \alpha_i\beta_{j_1} \ldots \beta_{j_m}$, and let $\gamma, \delta, \alpha_i, \beta_{j_1}, \ldots, \beta_{j_m}$ can be rewritten to $u_\gamma, u_\delta, u_{\alpha_i}, u_{\beta_{j_1}}, \ldots,$ $u_{\beta_{j_m}}$ using at most n A-derivations (they can be also similarly generated in G_2 – what is our induction hypothesis). Then $A \Rightarrow_{G_2,lm}$ $\alpha_i A' \Rightarrow_{G_2,slmA'} \alpha\beta_{i_1} A' \Rightarrow^*_{G_2,slmA'} \alpha_i\beta_{j_1} \ldots \beta_{j_m}$.

   It is, $\forall B \in (\mathrm{N}_1 \cap \mathrm{N}_2), w \in \Sigma^* : (B \Rightarrow^*_{G_1} w) \Rightarrow (B \Rightarrow^*_{G_2} w)$.

2. $(B \Rightarrow^*_{G_1} w) \Leftarrow (B \Rightarrow^*_{G_2} w)$ using n A'-derivations:

   (a) $n = 0$:
   As shown above, if a string can be generated from a nonterminal $C \in \mathrm{N}_1 \cap \mathrm{N}_2$ using only productions from $P_1 \cap P_2$ in G_2, the same string can be also generated in G_1.

   (b) $n \to n+1$: Let us assume that if a string w can be generated from a nonterminal $C \in \mathrm{N}_1 \cap \mathrm{N}_2$ using only productions from $P_1 \cap P_2$ in G_2 and n A'-derivations, it can be generated also in G_1. Let us show, that it holds also for strings that can be generated using $n+1$ A'-derivations.

   Let $C \Rightarrow^*_{lm\ G_2} \gamma A\delta$ using only productions from $P_1 \cap P_2$ (hence also $C \Rightarrow^*_{lm\ G_1} \gamma A\delta$), let $A \Rightarrow_{G_2,lm} \alpha_i A' \Rightarrow^*_{G_2,slmA'} \alpha_i\beta_{j_1} \ldots \beta_{j_m}$, and let $\gamma, \delta, \alpha_i, \beta_{j_1}, \ldots, \beta_{j_m}$ can be rewritten to $u_\gamma, u_\delta, u_{\alpha_i}, u_{\beta_{j_1}}, \ldots, u_{\beta_{j_m}}$

using at most n A'-derivations (they can be also similarly generated in G_1), then $C \Rightarrow^*_{G_2} u_\gamma u_{\alpha_i} u_{\beta_{j_1}} \dots u_{\beta_{j_m}} u_\delta$ using at most $n+1$ A'-derivations implies that $C \Rightarrow^*_{G_1} u_\gamma u_{\alpha_i} u_{\beta_{j_1}} \dots u_{\beta_{j_m}} u_\delta$.

It is, $\forall B \in (\mathrm{N}_1 \cap \mathrm{N}_2), w \in \Sigma^* : (B \Rightarrow^*_{G_2} w) \Rightarrow (B \Rightarrow^*_{G_1} w)$.

□

Corollary 6.23 Let G_1 and G_2 be grammars from Def. 6.19. Then $L(G_1) = L(G_2)$.

Proof: $S \in P_1 \cap P_2$, hence also (from lemma 6.22) $(B \Rightarrow^*_{G_1} w) \Leftrightarrow (B \Rightarrow^*_{G_1} w)$ for any $w \in \Sigma^*$ what means $L(G_1) = L(G_2)$. □

Corollary 6.24 Let G_1 and G_2 be grammars from Def. 6.19. Then

$$\forall B \in \mathrm{N}_1 \cap \mathrm{N}_2 : \ First^k_{G_1}(B) = First^k_{G_2}(B).$$

Proof: As any $B \in \mathrm{N}_1 \cap \mathrm{N}_2$ rewrites in both G_1 and G_2 to the same set of strings, it must hold also for the prefixes of length k. □

Corollary 6.25 Let G_1 and G_2 be grammars from Def. 6.19. Then

$$\forall \gamma \in (\mathrm{N}_1 \cap \mathrm{N}_2 \cup \Sigma)^* : \ First^k_{G_1}(\gamma) = First^k_{G_2}(\gamma).$$

Proof: Obvious. □

Lemma 6.26 Let G_1 and G_2 be grammars from Def. 6.19. Then

$$\forall B \in (\mathrm{N}_1 \cap \mathrm{N}_2) \backslash \{A\} : \ (S \Rightarrow^*_{G_1,lm} uB\gamma) \Leftrightarrow (S \Rightarrow^*_{G_2,slmA'} uB\gamma)$$

for any $u \in \Sigma^*$ and $\gamma \in (\mathrm{N}_1 \cap \mathrm{N}_2 \cup \Sigma)^*$.

Proof: $\forall B \in (\mathrm{N}_1 \cap \mathrm{N}_2) \setminus \{A\} : \ P_1(A) = P_2(A)$. As leftmost derivation in G_1 and semileftmost derivation in G_2 can be used, in G_1 all A-derivations can be performed and in G_2 all A'-derivation can be performed, also A generates in both grammars the same sentential forms. □

Corollary 6.27 Let G_1 and G_2 be grammars from Def. 6.19. Then

$$\forall B \in (\mathrm{N}_1 \cap \mathrm{N}_2) \setminus \{A\} : \ Follow^k_{G_1}(B) = Follow^k_{G_2}(B).$$

Proof: Obvious. □

Lemma 6.28 Let G_1 and G_2 be the grammars from Def. 6.19. Then

$$\forall B \in (\mathrm{N}_1 \cap \mathrm{N}_2) \setminus \{A\} : \ DLRF^k_{G_1}(B) = DLRF^k_{G_2}(B).$$

Proof: $\forall B \in (\mathrm{N}_1 \cap \mathrm{N}_2) \setminus \{A\} : \ DLRP_{G_2}(B) = DLRP_{G_1}(B)$. According Def. 5.9

$$DLRF^k_G(A) = \{a_1 \dots a_m \in First^k_G(\alpha \cdot Follow^k_G(A)) \mid A \to A\alpha \in P_G \wedge \alpha \neq \varepsilon\}.$$

According Corr. 6.27 $\forall B \in (\mathrm{N}_1 \cap \mathrm{N}_2) \setminus \{A\} : \ Follow^k_{G_2}(B) = Follow^k_{G_1}(B)$.

Hence $\forall B \in (\mathrm{N}_1 \cap \mathrm{N}_2) \setminus \{A\}$, $\forall \beta : B \to B\beta \in P_{G_2} : First^k_{G_2}(\gamma \cdot Follow^k_{G_2}(B)) = First^k_{G_1}(\gamma \cdot Follow^k_{G_1}(B))$. It is, $\forall B \in (\mathrm{N}_1 \cap \mathrm{N}_2) \setminus \{A\} : DLRF^k_{G_1}(B) = DLRF^k_{G_2}(B)$. □

Lemma 6.29 Let G_1 and G_2 be the grammars from Def. 6.19. Then

$$\forall B \in (\mathrm{N}_1 \cap \mathrm{N}_2) \setminus \{A\} : \ NLRF^k_{G_1}(B) = NLRF^k_{G_2}(B).$$

Proof: Lemma 6.26 says that $\forall B \in (\mathrm{N}_1 \cap \mathrm{N}_2) \setminus \{A\} : (S \Rightarrow^*_{G_1} uB\gamma) \Leftrightarrow (S \Rightarrow^*_{G_2} uB\gamma)$ for any $u \in \Sigma^*$ and $\gamma \in (\mathrm{N}_1 \cap \mathrm{N}_2 \cup \Sigma)^*$. It is, every such a nonterminal is for both grammars followed by the same strings.

We need to show that it holds also when we restrict ourselves to the cases when a nonterminal B is not actually used as left-recursive. If it occurs only within the productions that are both in G_1 and G_2, it follows that its $NLRF$'s must be the same. □

Lemma 6.30 Let G_1 and G_2 be the grammars from Def. 6.19. Then

$$NLRF^k_{G_2}(A) = NLRF^k_{G_2}(A') = Follow^k_{G_2}(A') = Follow^k_{G_2}(A).$$

Proof: A as well as A' are not left recursive in G_2. So it holds from the definition that $DLRP_{G_2}(A) = DLRP_{G_2}(A') = \emptyset$. Then $NLRF^k_{G_2}(A) = Follow^k_{G_2}(A)$ and $NLRF^k_{G_2}(A') = Follow^k_{G_2}(A')$.

It follows that $Follow^k_{G_2}(A) \subset Follow^k_{G_2}(A')$, A' occurs only at the end of all A-productions of G_2. A' occurs only at the end on A- and A' productions, so it must also hold that $Follow^k_{G_2}(A) \supset Follow^k_{G_2}(A')$. □

Lemma 6.31 Let G_1 and G_2 be grammars from Def. 6.19. Then

$$S \Rightarrow^*_{G_1} uA\gamma \Rightarrow^*_{G_1,lm} uA\beta_{i_1} \dots \beta_{i_r}\gamma \Rightarrow^*_{G_1,lm} u\alpha\beta_{j_1} \dots \beta_{j_s}\beta_{i_1} \dots \beta_{i_r}\gamma$$

$$\Updownarrow$$

$$S \Rightarrow^*_{G_2} uA\gamma \Rightarrow^*_{G_2} u\alpha\beta_{j_1} \dots \beta_{j_s}\beta_{i_1} \dots \beta_{i_r}\gamma$$

where $\beta_{i_s} \in \{\beta | A \to A\beta \in P_1\} \cup \{\varepsilon\}$.

Proof: From lemma 6.21 we know that for every A-derivation

$$S \Rightarrow^*_{G_1,lm} uA\beta_{i_1} \dots \beta_{i_r}\gamma \Rightarrow^*_{G_1,lm} uA\beta_{j_1} \dots \beta_{j_s}\beta_{i_1} \dots \beta_{i_r}\gamma \Rightarrow_{G_1,lm}$$

$$\Rightarrow_{G_1,lm} u\alpha\beta_{j_1} \dots \beta_{j_s}\beta_{i_1} \dots \beta_{i_r}\gamma$$

there is an A'-derivation

$$S \Rightarrow^*_{G_2,slmA'} uA\gamma \Rightarrow_{G_2,slmA'} u\alpha A'\gamma \Rightarrow^*_{G_2,slmA'} u\alpha\beta_{j_1} \dots \beta_{j_s}\beta_{i_1} \dots \beta_{i_r}\gamma$$

and vice versa – for every derivation in G_2

$$S \Rightarrow^*_{G_2,slmA'} uA\gamma \Rightarrow_{G_2,slmA'} u\alpha A'\gamma \Rightarrow^*_{G_2,slmA'} u\alpha\beta_{j_1} \dots \beta_{j_s}\beta_{i_1} \dots \beta_{i_r}\gamma$$

there is a derivation in G_1

$$S \Rightarrow^*_{G_1,lm} uA\beta_{i_1} \dots \beta_{i_r}\gamma \Rightarrow^*_{G_1,lm} uA\beta_{j_1} \dots \beta_{j_s}\beta_{i_1} \dots \beta_{i_r}\gamma \Rightarrow_{G_1,lm}$$

$$\Rightarrow_{G_1,lm} u\alpha\beta_{j_1} \dots \beta_{j_s}\beta_{i_1} \dots \beta_{i_r}\gamma$$

□

Corollary 6.32 Let G_1 and G_2 be grammars from Def. 6.19. Then

$$NLRF_{G_1}(A) = NLRF_{G_2}(A).$$

Proof: From Lemma 6.31 we know that

$$S \Rightarrow^*_{G_1} uA\beta_{i_1} \dots \beta_{i_r}\gamma \iff S \Rightarrow^*_{G_2} uA\gamma.$$

From the definition of it follows that

γ from $uA\gamma$ cannot start with any β such that $A \to A\beta$ in G_1. Hence

$$S \Rightarrow^*_{G_1} uA\gamma \iff S \Rightarrow^*_{G_2} uA\gamma$$

□

Lemma 6.33 Let G_1 and G_2 be the grammars from Def. 6.19. Then

$$First^k_{G_2}(A' \cdot Follow^k_{G_2}(A)) = Follow^k_{G_1}(A).$$

Proof: It follows directly from the construction of A-productions in $G-2$ and Corr. 6.32. □

We know that direct left recursion removal from A preserves $First$, $Follow$, $NLRF$, and $DLRF$ of the nonterminals different from A (and from A' that is not in G_1). Now we want to know whether the kind conditions are by the given operation also preserved.

Lemma 6.34 Let G_1 and G_2 be the grammars from Def. 6.19. Then

$$\forall B \in N_1 : NLRF^k_{G_1}(B) \cap DLRF^k_{G_1}(B) = \emptyset$$

$$\Downarrow$$

$$\forall B \in N_2 : NLRF^k_{G_2}(B) \cap DLRF^k_{G_2}(B) = \emptyset.$$

Proof: All C-productions ($C \in N_1 \setminus \{A\}$) are both in P_1 and P_2 and therefore $Follow^k_{G_2}(C) = Follow^k_{G_1}(C)$. We also know (from Lemmas 6.29 and 6.28) that $NLRF^k_{G_1}(C) = NLRF^k_{G_2}(C)$ and $DLRF^k_{G_1}(C) = DLRF^k_{G_2}(C)$. It means together that for such C

$$NLRF^k_{G_1}(C) \cap DLRF^k_{G_1}(C) = \emptyset$$

$$\Downarrow$$

$$NLRF^k_{G_2}(C) \cap DLRF^k_{G_2}(C) = \emptyset.$$

We also know that $DLRP_{G_2}(A) = DLRP_{G_2}(A') = \emptyset$. Hence

$$NLRF^k_{G_2}(A) \cap DLRF^k_{G_2}(A) = NLRF^k_{G_2}(A') \cap DLRF^k_{G_2}(A') = \emptyset.$$

□

The first part of k-kind condition (both strong and weak) is by the left recursion removal from a single nonterminal preserved. Now we need to check whether the second part of (strong, weak) k-kind condition is preserved – and therefore whether the fact that G_1 is strong or weak k-kind implies that G_2 is also strong/weak k-kind.

Lemma 6.35 Let G_1 and G_2 be the grammars from Def. 6.19. Then G_1 is k-kind implies that also G_2 is k-kind.

Proof: According Lemma 6.34 if the first part of the kind condition is fulfilled for G_1, it is fulfilled also for G_2. We need to show that also the second part of kind condition holds. According Def. 5.11 it holds for any two productions being either both from $NLRP_{G_1}(C)$ ($C \to \alpha\beta$, $C \to \alpha\gamma$ such that α is the longest common C-prefix α) or both from $DLRP_{G_1}(C)$ ($C \to C\alpha\beta$, $C \to C\alpha\gamma$ and sharing the longest common C-prefix $C\alpha$) it holds that

$$First^k_{G_1}(\beta \cdot Follow^k_{G_1}(C)) \cap First^k_{G_1}(\gamma \cdot Follow^k_{G_1}(C)) = \emptyset.$$

Let us discuss the individual cases:

$C \in \mathrm{N}_1$**:** The two productions under discussion exist also in G_2. Then

$$Follow^k_{G_1}(C) = Follow^k_{G_2}(C)$$

and the condition is fulfilled.

$C = A$**:** From Lemma 6.33 we know that

$$First^k_{G_2}(A' \cdot Follow^k_{G_2}(A')) = Follow^k_{G_1}(A).$$

Let both $A \to \alpha\beta_1$ and $A \to \alpha\beta_2$ be from $NLRP_{G_1}$. Then their counterparts from G_2 are $A \to \alpha\beta_1 A'$ and $A \to \alpha\beta_2 A'$. Then it holds

$$First^k_{G_2}(\beta_1 A' \cdot Follow^k_{G_2}(A)) = First^k_{G_2}(\beta_1 \cdot First^k_{G_2}(A' \cdot Follow^k_{G_2}(A'))) =$$

$$= First^k_{G_2}(\beta_1 \cdot Follow^k_{G_1}(A)) = First^k_{G_1}(\beta_1 \cdot Follow^k_{G_1}(A)).$$

Similarly

$$First^k_{G_2}(\beta_2 A' \cdot Follow^k_{G_2}(A)) = First^k_{G_1}(\beta_2 \cdot Follow^k_{G_1}(A)).$$

G_1 is k-kind, hence

$$First^k_{G_1}(\beta_1 \cdot Follow^k_{G_1}(A)) \cap First^k_{G_1}(\beta_2 \cdot Follow^k_{G_1}(A)) = \emptyset.$$

Using the last equalities we get

$$First^k_{G_2}(\beta_1 A' \cdot Follow^k_{G_2}(A)) \cap First^k_{G_2}(\beta_2 A' \cdot Follow^k_{G_2}(A)) = \emptyset.$$

$C = A'$**:** Let both $A \to A\alpha\beta_1$ and $A \to A\alpha\beta_2$ be from $DLRP_{G_1}$. Then their counterparts from G_2 are $A' \to \alpha\beta_1 A'$ and $A' \to \alpha\beta_2 A'$.

Lemma 6.30 shows that $Follow^k_{G_2}(A') = Follow^k_{G_2}(A))$. Hence

$$First^k_{G_2}(\beta_1 A' \cdot Follow^k_{G_2}(A')) = First^k_{G_2}(\beta_1 A' \cdot Follow^k_{G_2}(A)) =$$

$$= First^k_{G_1}(\beta_1 \cdot Follow^k_{G_1}(A)).$$

Similarly

$$First^k_{G_2}(\beta_2 A' \cdot Follow^k_{G_2}(A')) = First^k_{G_1}(\beta_2 \cdot Follow^k_{G_1}(A)).$$

G_1 is k-kind, hence

$$First^k_{G_1}(\beta_1 \cdot Follow^k_{G_1}(A)) \cap First^k_{G_1}(\beta_2 \cdot Follow^k_{G_1}(A)) = \emptyset.$$

Using the last equalities we get

$$First^k_{G_2}(\beta_1 A' \cdot Follow^k_{G_2}(A')) \cap First^k_{G_2}(\beta_2 A' \cdot Follow^k_{G_2}(A')) = \emptyset.$$

□

Lemma 6.36 Let G_1 and G_2 be the grammars from Def. 6.19. Then G_1 is weak k-kind implies that G_2 is also weak k-kind.

Proof: According Lemma 6.34 if the first part of weak k-kind condition is fulfilled for G_1, it is fulfilled also for G_2. We need to show that also the second part of weak k-kind condition holds. According Def. 5.15 if there are $w \in \Sigma^*$ and $\delta \in (\mathrm{N}_1 \cup \Sigma)^*$ such that $S \Rightarrow_{G_1,lm} wC\delta$, and there are two productions being either both from $NLRP_{G_1}(C)$ ($C \to \alpha\beta$, $C \to \alpha\gamma$ such that α is the longest common C-prefix α) or both from $DLRP_{G_1}(C)$ ($C \to C\alpha\beta$, $C \to C\alpha\gamma$ and the longest common C-prefix is $C\alpha$), then

$$First^k_{G_1}(\beta \cdot \delta) \cap First^k_{G_1}(\gamma \cdot \delta) = \emptyset.$$

We need to show that it holds also in G_2. Let us discuss the individual cases expecting that $w \in \Sigma^*$ and $\delta \in (\mathrm{N}_2 \cup \Sigma)^*$ exist and we have two C-productions being either both from $NLRP_{G_2}(C)$ or both from $DLRP_{G_2}(C)$:

$C \in \mathrm{N}_1 \setminus \{A\}$**:** The two productions exist in both grammars, according Lemma 6.26 the strings that may follow such nonterminal are the same, hence the weak k-kind condition must be fulfilled in this case.

$C \in \{A, A'\}$: The two productions are either $A \to \delta\delta_i A'$ and $A \to \delta\delta_j A'$ or $A' \to \delta\delta_i A'$ and $A' \to \delta\delta_j A'$ (where δ is the longest common prefix of the productions). We need to show that

$$First^k_{G_2}(\delta_i A'\gamma) \cap First^k_{G_2}(\delta_j A'\gamma) = \emptyset.$$

We know that if G_1 is weak k-kind, then for any two productions $(A \to \delta\delta_i,\ A \to \delta\delta_j)$ or $(A \to A\delta\delta_i,\ A \to A\delta\delta_j)$ in G_1 such that

$$First^k_{G_1}(\delta_i\theta\gamma') \cap First^k_{G_1}(\delta_j\theta\gamma') = \emptyset$$

for some $\gamma' \in (\mathrm{N}_1 \cup \Sigma)^*$, $\theta = \beta_{s_1} \ldots \beta_{s_t}$, $\beta_{s_r} \in \{\beta | A \to A\beta \in P_1\} \cup \{\varepsilon\}$. Setting $\gamma = \theta\gamma'$ we see that the second part of the weak k-kind condition is fulfilled also in this case.

□

We have shown that direct left recursion removal from a single nonterminal of a (weak) k-kind grammar preserves the language generated by the grammar as well as the property of the grammar that it is (weak) k-kind. Now we generalize this property to the removal of direct left recursion from the whole grammar.

Algorithm 6.37 (Left recursion removal):

Input: (weak) k-kind grammar G_{pr} (potentially having directly left-recursive productions)

Output: (weak) k-kind grammar G_{nr} having no left recursive productions and $L(G_{nr}) = L(G_{pr})$.

Method:

1. Set G_0 to G_{pr}, set i to 0.
2. While $DLRP_{G_i} \neq \emptyset$ do:
   (a) Take any A such that $DLRP_{G_i}(A) \neq \emptyset$.
   (b) Remove direct left recursion from A in G_i giving G_{i+1}.
   (c) Set i to $i + 1$.

   endwhile
3. Set G_{nr} to G_i.

Lemma 6.38 Algorithm 6.37 is finite.

Proof: Steps 1 and 3 are performed exactly once. Steps 2a – 2c are performed for each nonterminal at most once (as in every loop the left recursion is removed for given nonterminal). At the beginning there is a finite number of left-recursive nonterminals, in every iteration is this number decreased, and the loop terminates when the number of left-recursive nonterminals reaches 0. Hence the algorithm must halt after finite number of steps. □

Lemma 6.39 For G_{pr} and for G_{nr} from Alg. 6.37 hold that $L(G_{nr}) = L(G_{pr})$.

Proof: According corollary 6.23 it holds that $L(G_{i+1}) = L(G_i)$ for any G_i and G_{i+1} occurring in the Alg. 6.37. As $G_{pr} = G_0$ and $G_{nr} = G_i$ for the highest i, we get $L(G_{pr}) = L(G_0) = L(G_1) = \ldots = L(G_{nr})$. □

Lemma 6.40 For G_{pr} and for G_{nr} from Alg. 6.37 hold that if G_{pr} is weak k-kind then G_{nr} is also weak k-kind.

Proof: As shown above, if G_i is a weak k-kind grammar, then also G_{i+1} is a weak k-kind grammar. Moreover, $G_0 = G_{pr}$ and $G_{nr} = G_i$ for highest i. Hence if G_{pr} is weak k-kind, then also G_{nr} is weak k-kind. □

Lemma 6.41 For G_{pr} and for G_{nr} from Alg. 6.37 hold that if G_{pr} is (strong) k-kind then G_{nr} is also (strong) k-kind.

Proof: As shown above, if G_i is a (strong) k-kind grammar, then also G_{i+1} is a (strong) k-kind grammar. Moreover, $G_0 = G_{pr}$ and $G_{nr} = G_i$ for highest i. Hence if G_{pr} is (strong) k-kind, then also G_{nr} is (strong) k-kind. □

Now we can for every weak k-kind grammar G create a $\mathrm{Ch}(k)$ grammar (weak k-kind grammar without left recursion) G' such that $L(G') = L(G)$. If G is a strong k-kind grammar, then G' is also a strong k-kind grammar.

Simple Left Factoring.

Definition 6.42 (Single common nonempty prefix removal):
Let $G_1 = (\mathrm{N}_1, \Sigma, P_1, S)$ and $G_2 = (\mathrm{N}_2, \Sigma, P_2, S)$ be CFG. We say that G_2 is a product of *single common nonempty prefix removal* from G_1 iff $Q = \{A \to \alpha\beta_1, \ldots, A \to \alpha\beta_m\}$ is a set of all the A-productions from G_1 having longest common prefix α ($\alpha \neq \varepsilon$), $\mathrm{N}_2 \backslash \mathrm{N}_1 = \{A'\}$, $\mathrm{N}_2 = \mathrm{N}_1 \cup \{A'\}$, $P_2 \cap P_1 = P_0$, $P_1 = P_0 \cup Q$, $P' = \{A' \to \beta_i | A \to \alpha\beta_i \in Q\}$, $P_2 = P_0 \cup P' \cup \{A \to \alpha A'\}$.

Definition 6.43
Let G_1 and G_2 be the grammars from Def. 6.42. We call the derivation $A \Rightarrow_{G_1} \alpha\beta_i$ *X-derivation* and the derivation $A \Rightarrow_{G_2} \alpha A' \Rightarrow_{G_2} \alpha\beta_i$ *Y-derivation.*

In the following text concerning the simple left factoring we write that some nonterminal is from $\mathrm{N}_1 \cap \mathrm{N}_2$ just to emphasize the fact that the nonterminal is in both G_1 and G_2 although $\mathrm{N}_1 \cap \mathrm{N}_2 = \mathrm{N}_1$.

Lemma 6.44 Let $G_1 = (\mathrm{N}_1, \Sigma, P_1, S)$, $G_2 = (\mathrm{N}_2, \Sigma, P_2, S)$ are CFG such that $P_1 = NLRP_{G_1}$ and $P_2 = NLRP_{G_2}$. Let G_2 is a product of single common nonempty prefix removal from G_1. Then $\forall B \in (\mathrm{N}_1 \cap \mathrm{N}_2), u \in \Sigma^* : (B \Rightarrow^*_{G_1} u) \Leftrightarrow (B \Rightarrow^*_{G_2} u)$.

Proof:

1. $(B \Rightarrow^*_{G_1} u) \Rightarrow (B \Rightarrow^*_{G_2} u)$ using n X-derivations:

   (a) $n = 0$:
   Let us assume that $u \in \Sigma^*$ is derived in G_1 from some nonterminal (C) using productions from $P_1 \cap P_2$ only. As G_2 has all the productions used in the derivations, it is possible to derive u from C in G_2 too. Similarly, if $C \Rightarrow^*_{G_2} u$, then $C \Rightarrow^*_{G_1} u$ as the same productions can be used.

   (b) $n \to n + 1$:
   Now let us assume that $D \Rightarrow^*_{G_1} \gamma A\delta$ using only the productions from $P_1 \cap P_2$. Then (as shown above) also $D \Rightarrow^*_{G_2} \gamma A\delta$. As $A \to \alpha\beta_i \in P_1$ we have $D \Rightarrow^*_{G_1} \gamma\alpha\beta\delta$. As $A \to \alpha\beta_i$ has been in G_2 replaced by $A \to \alpha A'$ and $A' \to \beta_i$, then in G_2: $A \Rightarrow_{G_2} \alpha A' \Rightarrow_{G_2} \alpha\beta_i$. Let $\gamma\alpha\beta\delta \Rightarrow^*_{G_1} u$ using n X-derivations. Then u can be derived in G_2 using n Y-derivations. It is, if $D \Rightarrow^*_{G_1} u$ using $n + 1$ X-derivations implies that $D \Rightarrow^*_{G_2} u$.

   Hence $D \Rightarrow^*_{G_1} u$ implies $D \Rightarrow^*_{G_2} u$ for any $D \in \mathrm{N}_1$ and $u \in \Sigma^*$.

2. $(B \Rightarrow^*_{G_1} u) \Leftarrow (B \Rightarrow^*_{G_2} u)$ using n X-derivations:

   (a) $n = 0$:
   As shown above (in the opposite case), it holds that if we use only the productions from $P_1 \cap P_2$, we can derive the same strings.

(b) $n \to n+1$:
As A' occurs only at right-hand-side of a single production of the whole grammar G_2, any application of any production $A' \to \beta_j$ must be preceded by an application of the production $A \to \alpha A'$ what makes together an Y-derivation. So we can induce in the number of Y-derivations.

Now let us expect that $D \Rightarrow^*_{G_2} \gamma A\delta$ using only the productions from $P_1 \cap P_2$. Then also $D \Rightarrow^*_{G_1} \gamma A\delta$.

Let $A \to \alpha A'$ and $A' \to \beta_i$ are used in G_2 ($A \Rightarrow_{G_2} \alpha A' \Rightarrow_{G_2} \alpha\beta_i$), hence $D \Rightarrow^*_{G_2} \gamma\alpha\beta_i\delta$. Then in G_1 can be applied $A \to \alpha\beta_i$ ($A \Rightarrow_{G_1} \alpha\beta_i$), hence also $D \Rightarrow^*_{G_1} \gamma\alpha\beta_i\delta$.

Let $\gamma\alpha\beta\delta \Rightarrow^* u$ can be derived using n Y-derivations. Then it can be derived in G_1 using n X-derivations. It is, if $D \Rightarrow^*_{G_2} u$ using $n+1$ Y-derivations implies that $D \Rightarrow^*_{G_1} u$.

Hence $D \Rightarrow^*_{G_2} u$ implies $D \Rightarrow^*_{G_1} u$ for any $D \in \mathrm{N}_1 \cap \mathrm{N}_2$ and $u \in \Sigma^*$.

□

Corollary 6.45 Let G_1, G_2 are CFG. Let G_2 is a product of single common nonempty prefix removal from G_1. Then $L(G_2) = L(G_1)$.

Proof: From lemma 6.44 $(A \Rightarrow^*_{G_1} u) \Leftrightarrow (A \Rightarrow^*_{G_2} u)$ for any $A \in \mathrm{N}_1 \cap \mathrm{N}_2$. As $S \in \mathrm{N}_1 \cap \mathrm{N}_2$, it must hold that $L(G_2) = L(G_1)$. □

Lemma 6.46 Let $G_1 = (\mathrm{N}_1, \Sigma, P_1, S)$ and $G_2 = (\mathrm{N}_2, \Sigma, P_2, S)$ be CFG such that G_2 is a product of single common nonempty prefix removal from G_1. Then for any $A \in \mathrm{N}_1 \cap \mathrm{N}_2$ holds that $First^k_{G_1}(A) = First^k_{G_2}(A)$.

Proof: $First^k_G(A)$ is defined as prefix of length k (if possible) of any u such that $A \Rightarrow^* u$. From lemma 6.44 $(A \Rightarrow^*_{G_1} u) \Leftrightarrow (A \Rightarrow^*_{G_2} u)$. Both these facts together imply that $First^k_{G_1}(A) = First^k_{G_2}(A)$ for any $A \in \mathrm{N}_1 \cap \mathrm{N}_2$. □

Corollary 6.47 Let $G_1 = (\mathrm{N}_1, \Sigma, P_1, S)$ and $G_2 = (\mathrm{N}_2, \Sigma, P_2, S)$ be CFG such that G_2 is a product of single common nonempty prefix removal from G_1. Then for any $\alpha \in ((\mathrm{N}_1 \cap \mathrm{N}_2) \cup \Sigma)^*$ holds that $First^k_{G_1}(\alpha) = First^k_{G_2}(\alpha)$.

Proof: α is a string of terminal and nonterminal symbols. Terminal symbols do not change and nonterminal symbols generate in both grammars the same strings – hence also prefixes of the terminal strings generated from the symbol string must be the same. □

Lemma 6.48 Let $G_1 = (\mathrm{N}_1, \Sigma, P_1, S)$ and $G_2 = (\mathrm{N}_2, \Sigma, P_2, S)$ be CFG such that G_2 is a product of single common nonempty prefix removal from G_1. Then for any $\beta \in ((\mathrm{N}_1 \cap \mathrm{N}_2) \cup \Sigma)^*$ holds that $Follow^k_{G_1}(\beta) = Follow^k_{G_2}(\beta)$.

Proof: Let $S \Rightarrow^*_{G_1} \alpha\beta\gamma$. Then also $S \Rightarrow^*_{G_2} \alpha\beta\gamma$ as any sentential form derivable in G_1 is also derivable in G_2. As $\gamma \in (\mathrm{N}_1) \cup \Sigma)^*$, $Follow^k_{G_1}(\beta) \subset Follow^k_{G_2}(\beta)$.

Let $S \Rightarrow^*_{G_2} \alpha\beta\gamma$. Then $S \Rightarrow^*_{G_1} \alpha\beta\delta$ such that $\gamma \Rightarrow^*_{G_2} \delta$ (by proper application of productions $A' \to \beta_i$).

According lemma 6.44 it holds that for eny v $\delta \Rightarrow^* v$ in both G_1 and G_2 or in none of them. Hence $Follow^k_{G_2}(\beta) \subset Follow^k_{G_1}(\beta)$.

Having both $A \subset B$ and $B \subset A$ we get $A = B$ (or $Follow^k_{G_1}(\beta) = Follow^k_{G_2}(\beta)$). □

Lemma 6.49 For single common nonempty prefix removal it holds that if G_1 is its input grammar and G_2 is its output grammar, $G_1 \in Ch(k) \Rightarrow G_2 \in Ch(k)$.

Proof: Let $G_1 = (\mathrm{N}_1, \Sigma, P_1, S)$ and $G_2 = (\mathrm{N}_2, \Sigma, P_2, S)$ such that $P_1 = NLRP_{G_1}$ and $P_2 = NLRP_{G_2}$. Let us show that the resulting grammar (G_2) also fulfills the condition stated in Def. 2.47 (for a terminal string w, a nonterminal A, and strings γ, α, δ_1, and δ_2 in $(\mathrm{N} \cup \Sigma)^*$ such that $A \to \alpha\delta_1$ and $A \to \alpha\delta_2$ have the longest common prefix α, the condition $S \Rightarrow^*_{lm} wA\gamma$ implies that $First^k_{G_2}(\delta_1\gamma) \cap First^k_{G_2}(\delta_2\gamma) = \emptyset$).

Let us discuss the individual cases:

$B \to \alpha\delta_1, B \to \alpha\delta_2 \in P_1 \cap P_2$ (both productions are not affected by the grammar transformation): The condition is fulfilled trivially since (according Lemma 6.46) $First^k_{G_1}(\sigma) = First^k_{G_2}(\sigma)$ is for any $\sigma \in (\mathrm{N}_1 \cup \Sigma)^*$ (as $\mathrm{N}_1 = \mathrm{N}_1 \cap \mathrm{N}_2$).

$A \to \alpha\delta_1 \in P_1 \cap P_2,\ A \to \alpha\delta_2 \in P_2 \backslash P_1$ (only one production is changed): In this case some common prefix β longer than α (having α as prefix) has been removed. Let $\beta = \alpha\beta'$ and $A' \in \mathrm{N}_2 \backslash \mathrm{N}_1$. Then $\delta_2 = \beta' A'$. As $A' \Rightarrow_{G_2} \beta_i$, and $First^k_{G_1}(\delta_1\gamma) \cap First^k_{G_1}(\beta'\beta_i\gamma) = \emptyset$ for any i, it must hold that

$$First^k_{G_2}(\delta_1\gamma) \cap First^k_{G_2}(\beta' A'\gamma) = \emptyset$$

what is the requirement on Ch(k) grammars.

$A \to \alpha\delta_1, A \to \alpha\delta_2 \in P_2 \backslash P_1$: As both the productions are from $P_2 \backslash P_1$, there are in P_1 corresponding productions $B \to \beta\alpha\delta_1$ and $B \to \beta\alpha\delta_2$. As G_1 is Ch(k), we have

$$First^k_{G_1}(\delta_1\gamma) \cap First^k_{G_1}(\delta_2\gamma) = \emptyset.$$

As $\delta_1, \delta_2, \gamma \in ((\mathrm{N}_1 \cap \mathrm{N}_2) \cup \Sigma)^*$, then (according Lemma 6.44)

$$First^k_{G_2}(\delta_1\gamma) \cap First^k_{G_2}(\delta_2\gamma) = \emptyset,$$

the Ch(k) condition is then for the two productions fulfilled.

□

Lemma 6.50 For single common nonempty prefix removal holds that if G_1 is its input grammar and G_2 is its output grammar, G_1 is k-kind implies that G_2 is also k-kind.

Proof: Let $G_1 = (\mathrm{N}_1, \Sigma, P_1, S)$ and $G_2 = (\mathrm{N}_2, \Sigma, P_2, S)$. Let $B \to \gamma\delta_i$ and $B \to \gamma\delta_j$ are two productions from P_2. Then one of the following situations must occur:

1. both productions are from $P_1 \cap P_2$;
2. $B = A$ and the newly defined nonterminal occurs in δ_i or in δ_j.
3. $B = A$ and the newly defined nonterminal occurs in both δ_i and δ_j.

Let us solve the individual cases:

Both productions are from $P_1 \cap P_2$: If both productions are from $P_1 \cap P_2$, the kind condition is fulfilled for them also in G_2 trivially.

One production changed, one unchanged: Let the unchanged production be $A \to \alpha\beta$ and the changed one $A \to \alpha\gamma A'$ where A' is the newly defined nonterminal. Let $A' \to \delta_1$, ..., $A' \to \delta_s$ be all the A'-productions.

Then there are in G_1 corresponding productions $A \to \alpha\beta$, $A \to \alpha\gamma\delta_1$, ..., $A \to \alpha\gamma\delta_s$. These productions fulfill the Ch(k) condition. It must therefore hold that

$$First^k_{G_1}(\beta \cdot Follow^k_{G_1}(A)) \cap First^k_{G_1}(\gamma\delta_i \cdot Follow^k_{G_1}(A)) = \emptyset; \quad i = 1..s.$$

It holds that $Follow^k_{G_2}(B) = Follow^k_{G_2}(A) = Follow^k_{G_1}(A)$. Hence

$$First^k_{G_2}(\beta \cdot Follow^k_{G_1}(A)) \cap First^k_{G_2}(\gamma A' \cdot Follow^k_{G_1}(A)) = \emptyset.$$

Both productions changed: Both productions can be changed only in the case when a common prefix is removed form both productions. Then B must be a new nonterminal. If B is a newly defined nonterminal, there must be in N_2 a nonterminal (let us denote it A) such that $A \to \alpha B$ and there must be in P_1 productions $A \to \alpha\beta_i$ and $A \to \alpha\beta_j$ such that $\beta_i = \gamma\delta_i$ and $\beta_j = \gamma\delta_j$.

As G_1 is a k-kind grammar, it must hold that $A\alpha\gamma\delta_i$ and $A\alpha\gamma\delta_j$ have a longest common prefix (let it be $\alpha\gamma\delta_{ij}$; then $\delta_i = \delta_{ij}\delta_i'$ and $\delta_j = \delta_{ij}\delta_j'$) and it must hold that

$$First^k_{G_1}(\delta_i' \cdot Follow^k_{G_1}(A)) \cap First^k_{G_1}(\delta_j' \cdot Follow^k_{G_1}(A)) = \emptyset.$$

But then also

$$First^k_{G_2}(\delta_i' \cdot Follow^k_{G_2}(A)) \cap First^k_{G_2}(\delta_j' \cdot Follow^k_{G_2}(A)) = \emptyset.$$

□

Algorithm 6.51 (Simple Left Factoring):

Input: (weak) k-kind grammar without left recursion G_{Ch} (Ch(k) grammar)

Output: (S)LL(k) grammar G_{LL} such that $L(G_{Ch}) = L(G_{LL})$.

Method:

1. Set G_0 to G_{Ch}, set i to 0.
2. While there is a group of A-productions (for some A) with longest common prefix do:
   (a) If there are more such groups, take the group where the common prefix is longest.
   (b) Apply on this group single common prefix removal from G_i giving G_{i+1}.
   (c) Set i to $i + 1$.

   endwhile
3. Set G_{LL} to G_i.

Lemma 6.52 Algorithm 6.51 is finite.

Proof: Steps 1 and 3 are performed exactly once. Steps 2a – 2c are performed for each group of productions sharing common nonempty prefix at most once (as it is always taken group with longest common prefix, the productions from this group are fully separated). The number of such groups is limited by the number of productions in the original grammar – as any group bound either two productions or groups of productions and every production or group of productions can have only one longest common prefix bounding it to another production or group of productions. It is, the number of groups of productions having nonempty common prefix is at the beginning finite positive number and in every loop of the cycle it is decreased by 1. The loop ends when this number reaches 0. Hence the loop must end and the algorithm must halt after finite number of steps. □

Lemma 6.53 For G_{Ch} and from G_{LL} from Alg. 6.51 it holds that $L(G_{Ch}) = L(G_{LL})$.

Proof: According corollary 6.45 it holds that $L(G_{i+1}) = L(G_i)$ for any G_i and G_{i+1} occurring in the Alg. 6.51. As $G_{Ch} = G_0$ and $G_{LL} = G_i$ for the highest i, we get $L(G_{Ch}) = L(G_0) = L(G_1) = \ldots = L(G_{LL})$. □

Lemma 6.54 For G_{Ch} and for G_{LL} from Alg. 6.37 it holds that if G_{Ch} is Ch(k) (i.e. weak k-kind without left recursion) then G_{LL} is LL(k) and therefore also Ch(k).

Proof: As shown above, if G_i is Ch(k) grammar then also G_{i+1} is a Ch(k) grammar. Moreover, $G_0 = G_{Ch}$ and $G_{LL} = G_i$ for highest i. Hence if G_{Ch} is Ch(k), then also G_{LL} is Ch(k).

As G_{LL} contains no group of productions having nonempty common prefix, it must be LL(k). As for every two A-productions $A \to \alpha$ and $A \to \beta$ from G_{LL} it holds that $S \Rightarrow^*_{G_{LL}} uA\delta$ for some u and δ implies that $First^k_{G_{LL}}(\alpha\delta) = First^k_{G_{LL}}(\beta\delta)$ what corresponds to the definition of LL(k) grammars. □

Lemma 6.55 For G_{Ch} and for G_{LL} from Alg. 6.37 it holds that if G_{Ch} is a (strong) k-kind grammar without left recursion, then G_{LL} is an SLL(k) grammar and therefore also a (strong) k-kind grammar without left recursion.

Proof: As shown above, if G_i is a (strong) k-kind grammar then also G_{i+1} is (strong) k-kind grammar. Moreover, $G_0 = G_{Ch}$ and $G_{LL} = G_i$ for highest i. Hence if G_{Ch} is a (strong) k-kind grammar without left recursion, then also G_{LL} is a (strong) k-kind without left recursion.

As G_{LL} has no groups of productions sharing nonempty common prefix, it must (from k-kind condition and this fact) hold that for any two A-productions $A \to \alpha$, $A \to \beta$ from G_{LL} that $First^k_{G_{LL}}(\alpha \cdot Follow^k_{G_{LL}}(A)) = First^k_{G_{LL}}(\beta \cdot Follow^k_{G_{LL}}(A))$ what corresponds to the definition of SLL(k) grammars. □

6.3 Conclusions

It has been proven that k-kind grammars generate LL(k) languages.

The resulting relations between grammar classes could be divided in two groups: first, presented on Fig. 6.8, summarizes inclusions, second, presented on Fig. 6.9, shows incomparability of grammar classes. Incomparability of grammar classes are displayed as a link.

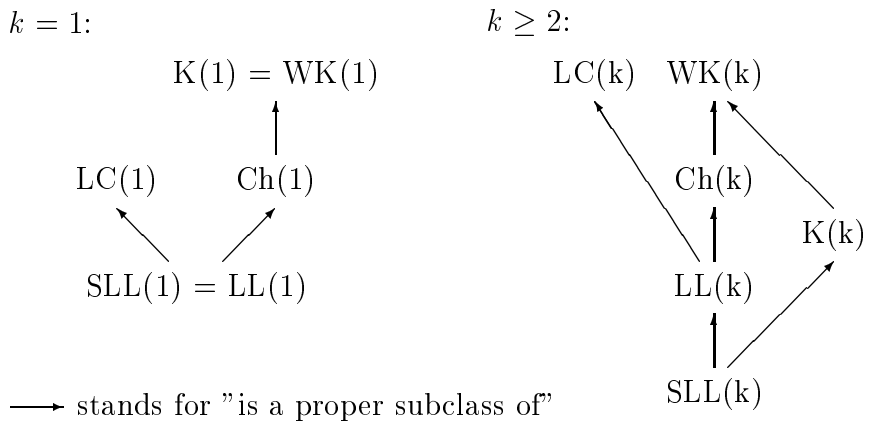


Figure 6.8: Inclusion of selected top-down or semi top-down parsable grammar classes

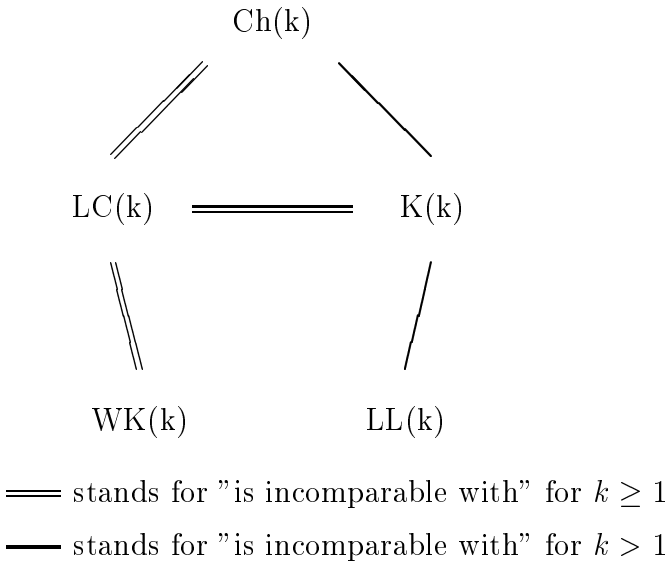


Figure 6.9: Incomparability of selected grammar classes

Chapter 7

Oracle Pushdown Automata

7.1 Motivation

The motivations of this chapter are:

1. to build a formal model of the concept of lookahead being in fact rather informal in the literature on parsing,

2. to show that the formalization is fruitful as it refines the intuition and enables to study parser effectiveness problems, and

3. to design a formal model as a basic of a study of very general parser implementations in software engineering and computational linguistics.

Standard model of pushdown automaton (PDA, see Def. 2.34) works with lookahead so that it encodes the lookahead into the states of the PDA (compare e.g. the Figs. 6.7 and 6.8 in [SSS90]). Such a model is advantageous for a theoretical analysis of automata features but it does not reflect the implementation of PDA in compilers – especially in the case when recursive descent parsing is used. It also does not match the intuition that lookahead analysis can be understood as a process of making decision which move is possible in a given configuration. The set of states of such a standard model automaton is too large.

We will discuss a more general model (*oracle pushdown automata*) inspired by oracle from [Tur39] and a practical model (*lookahead pushdown automata*) that suffices for the design and implementation of recursive descent parsers.

If the lookahead analysis is done in a configuration where different moves are possible for different lookaheads, the analysis can be restricted only to the shortest lookahead that must be really used to make the correct choice of

a transition to be used. In common programming languages there are only a few places where a lookahead longer than 1 is necessary. There are many places where the analysis of the lookahead is not necessary at all. In other situations inspection of less than k lookahead symbols suffices. It allows us to keep the size and of the analyzers within reasonable bounds even for long lookaheads and, at the same time, to improve effectiveness of the parsing.

The concept of the lookahead inspection can be generalized to understand it as an oracle in common sense, i.e. as a (black-box) activity making decisions. The oracle can be implemented e.g. as a finite automaton or as a tool of communication with people or with other automata or software components being peers in a service-oriented system. The concept of oracle can be then used as a theoretical background for the modeling of some service-oriented systems (i.e. systems having service-oriented architecture – SOA; [KŽ04]). The oracle is then one of the services.

Let us give an example showing the application of the concept: It can happen that during the parsing of a natural language the parser is unable to parse given text properly and it could be useful to let the user cooperate with the parser (to enter a proper result). If such cases are rare enough, it could be at present state of art an optimal solution. It could be moreover used for improvement (learning) of the parser via remembering the human decisions in a data store that can be used by the parser. It is also possible that parser can (when necessary) in the way known from service-oriented systems call specialized applications solving specific linguistic issues. These software engineering and computational linguistic issues will be topics of further study.

7.2 Oracle Pushdown Automata

To define oracle pushdown automaton we need to go deeper into the definition of usual pushdown automaton. The following definitions are derived from Defs. 2.32, 2.33, and 2.34.

Definition 7.1 (Oracle, authorized transition, authorized computation):

Let $\mathcal{M} = (Q, \Sigma, \Gamma, \delta)$ be a PDM. Let c be a configuration of $\mathcal{M}$. We say, that a transition t of $\mathcal{M}$ is *authorized* and can be performed, iff a predicate $O(\mathcal{M}, c, t)$ is fulfilled. We call the predicate *oracle*. An authorized transition is called *move* for short. Similarly, authorized transition relation between configurations c and c' is called *move relation*, we write $c \longmapsto c'$ if there is a move between c and c'.

An *authorized computation* is an element of the reflexive and transitive closure of the move relation. We denote a valid computation starting from c and leading to c' by $c \stackrel{*}{\longmapsto} c'$.

For any word $x \in \Sigma^*$ we introduce the relation on internal configurations of $\mathcal{M}$, denoted $\stackrel{x}{\Longmapsto}$ and defined by:

$$(q, \alpha) \stackrel{x}{\Longmapsto} (q', \alpha') \iff \exists y \in \Sigma^* : \ (xy, q, \alpha) \stackrel{*}{\longmapsto} (y, q', \alpha').$$

We clearly have $\stackrel{x}{\Longmapsto} \circ \stackrel{y}{\Longmapsto} = \stackrel{xy}{\Longmapsto}$.

A quintuple $\mathcal{M}' = (Q, \Sigma, \Gamma, \delta, O)$ is called *oracle pushdown machine* (OPDM for short).

We say that $\mathcal{M} = (Q, \Sigma, \Gamma, \delta)$ is a *basic pushdown machine* (or *basic PDM* for short) of $\mathcal{M}'$.

Definition 7.2 (Reachable (internal) configuration):
Let $\mathcal{M}$ be an OPDM. An internal configuration (q', α') of $\mathcal{M}$ is *reachable* from an internal configuration (q, α) of $\mathcal{M}$, or equivalently, (q, α) is *co-reachable* from (q', α') if there is some $u \in \Sigma^*$ such that $(q, \alpha) \stackrel{u}{\Longmapsto} (q', \alpha')$. Similarly, a configuration (q', v, α') of $\mathcal{M}$ is *reachable* from a configuration (q, uv, α) of $\mathcal{M}$, or equivalently, (q, uv, α) is *co-reachable* from (q', v, α') if $(q, uv, \alpha) \stackrel{*}{\longmapsto} (q', v, \alpha')$.

Definition 7.3 (Oracle Pushdown Automaton):
An *oracle pushdown automaton* (an OPDA for short) is an oracle pushdown machine $(Q, \Sigma, \Gamma, P, O)$, together with a distinguished *initial state* $q_0 \in Q$, a distinguished *initial pushdown symbol* $Z_0 \in \Gamma$ (forming together with a word $x \in \Sigma$ to be recognized an *initial configuration* $c_0 = (q_0, x, Z_0)$), and a set $F \subset Q$ of *accepting states*. An oracle pushdown automaton is therefore an 8-tuple $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, O, q_0, Z_0, F)$. A PDA $\mathcal{B} = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is said to be the *basic automaton* of $\mathcal{A}$.

For an OPDA $\mathcal{M}$, an (internal) configuration of $\mathcal{M}$ is *reachable* if it is reachable from the initial (internal) configuration of $\mathcal{M}$, and is *co-reachable* if it is co-reachable from an accepting (internal) configuration of $\mathcal{M}$.

A word $x \in \Sigma^*$ is *recognized* by an OPDA $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, O, q_0, Z_0, F)$ if there are $q \in F$ and $\alpha \in \Gamma^*$ such that $(q_0, x, Z_0) \stackrel{*}{\longmapsto} (q, \varepsilon, \alpha)$.

We denote the *language accepted* by an OPDA $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, O, q_0, Z_0, F)$ being the set of all words accepted by $\mathcal{A}$ by $L(\mathcal{A})$.

There is no transition starting with empty pushdown – hence when the pushdown is empty, the computation terminates. Unless an accepting configuration is reached, the computation terminates with an error.

Language accepted by an oracle pushdown automaton is called *oracle pushdown automaton language*.

Convention 7.4 It is possible to view an OPDA as an ordered pair $(\mathcal{A}, O)$ where $\mathcal{A}$ is its basic automaton and O is its oracle.

Definition 7.5 (Parsing situation):
Let $\mathcal{M} = (Q, \Sigma, \Gamma, \delta)$ be a PDM. A *parsing situation* of $\mathcal{M}$ is an ordered pair $(q, Z) \in Q \times \Gamma$ where q is the current state, and Z is the top-most stack symbol.

To support our intuition we can define an oracle function that returns for each configuration (or an internal configuration or a parsing situation or a state) a set of transitions authorized for given configuration.

Definition 7.6
Let $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, O)$ be an OPDM, $\mathcal{M}' = (Q, \Sigma, \Gamma, \delta)$ its basic PDM, and c their configuration. Let us define an *oracle function* o implementing the oracle O:

$$o(\mathcal{M}', c) = \{t \in \delta | O(\mathcal{M}', c, t)\}$$

Whenever it is clear which automaton is understood, it is possible to omit the first parameter and write just $o(c)$ (or even $O(c)$ whenever it is not necessary explicitly distinguish between functional and predicate form of the oracle).

In practice it is possible to implement the oracle by any computing mechanism – e.g. by invocation of an external application or even by a communication with real world.

Note 7.7 In practice we can use different types of oracle function. Let us discuss the differences in the oracles according the information used for their computation.

1. state + at most k symbols of input;
2. state + top-most pushdown symbol + at most k symbols of input;
3. state + pushdown + at most k symbols of input;
4. whole configuration (like in the Def. 7.1);
5. configuration + information from previous computations;
6. configuration + information from previous computations + external sources (users, other applications).

The first four types returns for the same input the same result – they are deterministic. The last two types use also other information source – they can behave according configuration non-deterministically.

The application of the individual types of oracles is e.g. in SLL(k)- or kind- parsing, lookahead automata (see below), LL(k)-parsing (for $k \geq 2$), unlimited lookahead (useful for C++ parsing; supported by e.g. COCO/R), parsing of constructs being not context-free (like $a^n b^n c^n$), and constructing complex parsers for natural language processing.

The oracle concept primarily models intuitive approach used by the parser implementations in compilers and linguistics. The oracle concept allows to optimize parsers and to extend their applicability.

Definition 7.8 (Deterministic oracle pushdown automaton):
We say that an OPDA $\mathcal{A}$ is *deterministic* (DOPDA for short), if it its oracle authorizes for each configuration at most one transition. Then we can write the values returned by the oracle function as singletons.

Convention 7.9 Let $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, O)$ be an OPDM. When the computation of the oracle O depends only on state and part of the pushdown of $\mathcal{M}$ and on a prefix of unread part of the input, we can write prefix of the unread part of input that is used by the oracle for its computation in brackets ('[',']') behind the symbol to be read, and similarly we can write in brackets the used prefix of the stack behind the symbol on top of the stack.

In order to simplify the description of the moves near the end of input string we assume that the input string is augmented by $\$^k$ – we use an augmented grammar (Def. 2.36).

Note 7.10 Oracle in practical use need not depend on the basic automaton and its current configuration only. It can depend on the interaction with "real world" (for example with users). It need not be deterministic. For this thesis we will restrict ourselves to the deterministic behavior of the oracles.

The k-lookahead oracle function is an oracle function having the property that it depends only on k symbols on input, state, and top of the pushdown.

In the case when the oracle uses only a restricted part of the configuration, it is possible to use as the input of the oracle only this restricted part of the configuration. (Example: SLL(k) parsing – the oracle decides using only the state, top of the stack, and the first k symbols from the unread part of the input.)

Definition 7.11 (k-lookahead configuration):
Let $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, O)$ be an OPDM. We can define to every configuration $c = (q, w, \alpha)$ of $\mathcal{M}$ a *k-lookahead configuration*

$$C_k(c) = (q, First_k(w), First_1(\alpha)).$$

Example 7.12 Let us have an OPDA which oracle makes its decision using only k-lookahead configuration

$$C_k = (q, a_1..a_k, Z).$$

A move of the OPDA from given configuration to the state q', reading the symbol a_1 and replacing the top-most pushdown symbol by the string α will be denoted $(q, a_1[a_2...a_k], Z) \to (q', \alpha)$. The case when no symbol is read is written as follows: $(q, [a_1...a_k], Z) \to (q', \alpha)$. □

Let us look at how different oracles can "work":

Trivial: The oracle 'authorizes' every transition. (Let $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, O)$. Then for any $c = (q, x, Z\alpha)$ be a configuration of $\mathcal{M}$ and $\forall t = (q, b, Z, q', \beta) \in \delta : O(\mathcal{M}, c, t)$.) It is, every transition of the PDM is an authorized transition of OPDM. Trivial oracle will be denoted Tr.

Classic: The oracle selects for each configuration at most one move based on state, top of the stack, and exactly the first k symbols from the unread part of the input. It corresponds to classic kinds of deterministic parsing (e.g. SLL(k), SLR(k)). For any OPDM $\mathcal{M}$ and its configurations c and c' it holds that $C_k(c) = C_k(c') \Longrightarrow o(c) = o(c')$. This oracle is sometimes denoted as k-lookahead oracle.

Lazy: The oracle uses only smallest part of the configuration necessary for the selection of applicable transitions.

Adaptive: The oracle can ask other applications or even users for help when meets unpredicted situation. It remembers the hint for next occurrences of the problem.

Definition 7.13 (Sibling automata):
We say that two oracle pushdown automata $\mathcal{A} = (P_\mathcal{A}, O_\mathcal{A})$, $\mathcal{B} = (P_\mathcal{B}, O_\mathcal{B})$ where $P_\mathcal{A} = P_\mathcal{B}$ are *siblings*. We shall also say that $\mathcal{A}$ is a sibling of $\mathcal{B}$ and $\mathcal{B}$ is a sibling of $\mathcal{A}$.

Lemma 7.14 Let OPDA $\mathcal{A} = (P, Tr)$. Then $\mathcal{A}$ and P go through the same configurations and recognize the same language.

Proof: In both cases the initial configurations are identical: (q_0, w, Z_0). Tr does not restrict the set of moves used by P. Hence A can in each configuration use the same set of moves as P do. Both automata must therefore go through the same configurations and recognize the same language. □

Definition 7.15 (Oracle refinement):
Let us have two sibling OPDA $\mathcal{A} = (P, O)$ and $\mathcal{A}' = (P, O')$. We say that O is a *refinement* of O' if for every configuration c of the basic automaton P $o(c) \subseteq o'(c)$ or, equivalently $O(\mathcal{A}, c, t) \Longrightarrow O'(\mathcal{A}', c, t)$. We also say that $\mathcal{A}$ *is a refinement of* $\mathcal{A}'$.

Theorem 7.16 If it is allowed to use oracles of arbitrary strength, it is possible to construct OPDA recognizing language that cannot be recognized by any PDA.

Proof: An example is an OPDA accepting a non-context-free language $\{a^n b^n c^n | n \geq 1\}$. It uses a strong oracle equipped with its own pushdown.

The basic automaton counts the a's and when it "matches" the b's, the oracle "count" the b's too and then "enforces" the basic automaton to count the same number of c's. □

7.3 Lookahead Pushdown Automata

This work is dedicated to kind parsing [ŽK99a, ŽK99b, KŽ03a] that is deterministic. Let us therefore focus on deterministic easily implementable oracles using only state, limited prefix of unread part of input, and top of stack symbol for its computations (what makes the oracle deterministic) and that return at most one applicable move (what makes the resulting automaton deterministic). We then refine oracle pushdown automata to lookahead pushdown automata.

Definition 7.17 (Lookahead Pushdown Automaton):
We say that an oracle pushdown automaton is an *exactly k lookahead pushdown automaton* ($!k$-LPDA) if its oracle is deterministic and selects for a given state, top stack symbol, and the k-symbol prefix of unread input at most one move. Similarly, we say that an oracle pushdown automaton is an *at most k lookahead pushdown automaton* ($\leq k$-LPDA) if its oracle deterministically selects only one move only according state and first at most k symbols from the beginning of unread part of the input. We say that an at most k lookahead pushdown automaton is *minimal* if the oracle uses only the smallest prefix of unread part of the input necessary for the selection of the unique move.

Note 7.18 Every $!k$-LPDA is a $\leq k$-LPDA.

Convention 7.19 We write LPDA if $!k$-LPDA, $\leq k$-LPDA, or both are meant.

It can be useful to extend the term *move* for LPDA so that the same moves for different lookaheads can be distinguished as different moves.

The task of transition selection using lookaheads differs from other tasks performed by Moore machines [Moo56]: here it is not necessary to read whole input to check whether it belongs to given language; the issue is to select the variant of computation having chance to finish the computation successfully.

It is possible to have different requirements on the optimality of an oracle. It is possible to "list" all the possible lookaheads and convert this list to a tree (see the case **a)** in Fig. 7.1). It is possible to minimize the tree so that only the shortest lookaheads necessary for proper transition selection are used – i.e. the corresponding $\leq k$-LPDA is minimized (case **b)** in Fig. 7.1). It is possible to renounce from the requirement to use tree-structured automata (that simplifies run-time extensibility of the parser) and use DAG-structured automata (case **c)** in Fig. 7.1). The oracle can be optimized for its size what can lead to minimum-state solutions like the case **d)** in Fig. 7.1.

The example below demonstrates that we can minimize an automaton representing the oracle in several ways: we can minimize number of its states, the number of symbols of lookahead that it uses for its computation, simplicity of its extension, etc. Sometimes it can be useful to recognize an inconsistency (or error) on input sooner, sometimes it can be better to recognize it closer to the inconsistent (erroneous) place. Hence we cannot always use automata minimization (see Def. 2.23) as known from the literature. The selection of "optimal" type of an oracle for a LPDA depends on the specification what implementation is optimal.

Example 7.20 (Different approaches to lookahead handling):

Let us consider situation when we need to select between several moves starting from given state using lookahead. The task can be solved by an acyclic DFA. Let the moves are:

$t_1 = (q, [adk], Z) \rightarrow (q_1, Z)$,
$t_2 = (q, [ael], Z) \rightarrow (q_2, Z)$,
$t_3 = (q, [bgm], Z) \rightarrow (q_3, Z)$,
$t_4 = (q, [cek], Z) \rightarrow (q_1, Z)$,
$t_5 = (q, [cfn], Z) \rightarrow (q_4, Z)$.

At least 4 basic solutions shown in Fig. 7.1 can be used:

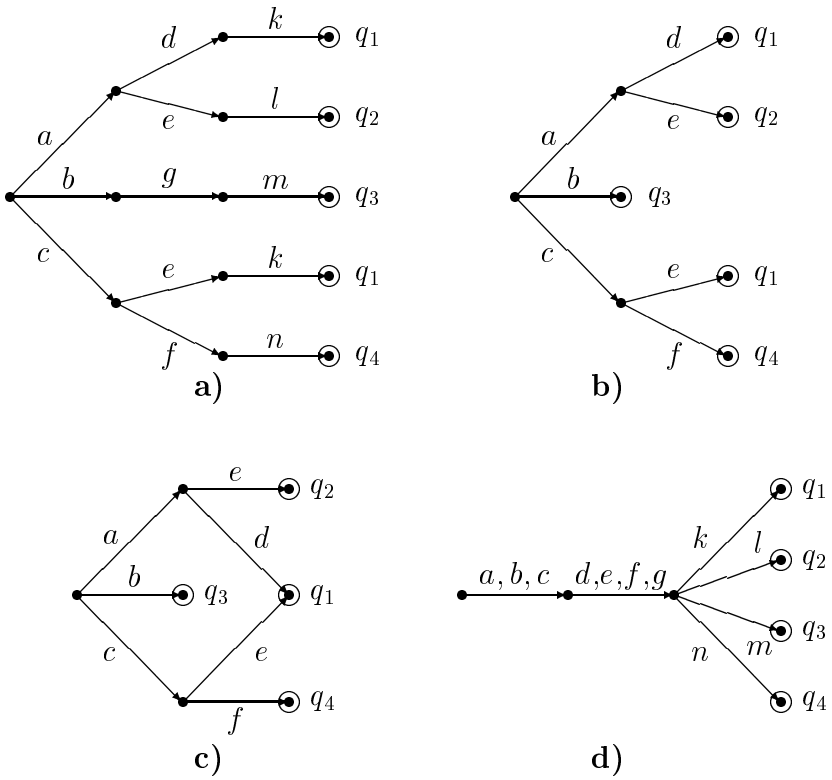


Figure 7.1: Different automata for the same lookahead.

1. Tree-like automaton checking all possible lookaheads of the length k. It uses lookahead of 3 symbols, has 14 states, and makes up to 6 comparisons.

2. Tree-like automaton checking lookaheads of minimal possible lengths. It uses lookahead of 1 or 2 symbols, has 8 states, and makes up to 5 comparisons.

3. DAG-like automaton checking lookaheads of minimal possible lengths. It uses lookahead of 1 or 2 symbols, has 7 states, and makes up to 5 comparisons.

4. Automaton performing only minimal checking. It uses the lookahead of 3 symbols, has 7 states, and makes up to 11 comparisons. It is possible to check only the 3rd symbol what simplifies the test (then it makes up to 4 comparisons).

All the mentioned transitions are ε-transitions and therefore it is possible to label the final states of the oracle automata by the corresponding states where the basic automaton should move instead of labeling them by different transitions. □

Definition 7.21 (Lookahead of given transition):
For every move t of an $\leq k$-LPDA we define

$$lookahead(t) = \begin{cases} a_1 a_2 \dots a_m & \text{for } t = (q, a_1[a_2 \dots a_m], Z) \to (q', \alpha) \\ a_1 \dots a_m & \text{for } t = (q, [a_1 \dots a_m], Z) \to (q', \alpha) \end{cases}$$

where $m \leq k$.

Lemma 7.22 For each $\leq k$-LPDA A there is a sibling $!k$-LPDA A' such that A' is a refinement of A and $L(A') = L(A)$.

Proof: Extending the lookaheads to the length k using the strings from $\bigcup_{0 \geq i \geq j} \Sigma^i \$^{j-i}$ we get the required automaton. □

In practice lookaheads are typically used for elimination of failing computations or for reduction of their number. We will therefore restrict ourselves only to this type of LPDA – to natural LPDA.

Definition 7.23 (Natural LPDA):
Let $A = (P, O)$ be an LPDA. We say that A is *natural* iff $L(A) = L(P)$.

In a natural LPDA the oracle (lookahead) is usually constructed so that A is deterministic.

Theorem 7.24 To every natural $\leq k$-LPDA A there is a sibling minimal $\leq k$-LPDA A' such that A is a refinement of A' and $L(A') = L(A)$.

Proof is on the page 79.

Definition 7.25 (Acceptable configuration):
Let $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a natural (L)PDA. We say that a configuration $c = (q, w, Z)$ is *acceptable by* P, if $\exists c' = (q', \varepsilon, \alpha)$ where $q' \in F$ such that $c \overset{*}{\vdash} c'$. (We can also say that c is acceptable, if it is co-reachable.)

It is often not reasonable to expect that the oracle can restrict its LPDA to visit only acceptable configurations. Hence we define promising configurations as configurations that the oracle cannot recognize as not co-reachable.

Definition 7.26 (Promising configuration):
Let M be a natural LPDA. We say that a configuration c of M is k-*promising*, if it is not possible to recognize it as not accessible using lookahead of just k symbols.

We say that a configuration is *promising* if it is k-promising for some $k \geq 1$.

$PC_k(P)$ denote the set of all k-promising configurations of P.

Definition 7.27
All transitions starting in given parsing situation $s = (q, Z)$ are called (q, Z)-*transitions* (or s-*transitions*).

Definition 7.28
$Clean_\Sigma$ is a homomorphism defined on $(\Sigma \cup \{\circ, \$\})^*$, $\Sigma \cap \{\circ, \$\} = \emptyset$ (where $\circ$ stands for uninitialized input symbol and \$ for sentinel) such that

$$Clean_\Sigma(a) =_{df} \begin{cases} a & \forall a \in \Sigma \\ \varepsilon & \forall a \notin \Sigma \end{cases}$$

Definition 7.29 (Possible lookahead strings):
Let $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a PDA. Let $s = (q, Z)$ be an arbitrary parsing situation of P. The set of all possible lookahead strings of length at most k of parsing situation s is

$$\begin{aligned} LAS_{P,\leq k}(s) = &\left\{a_1...a_j \in \textstyle\bigcup_{0 \leq i < j \leq k} \Sigma^i \$^{j-i} \mid (q, Clean(a_1...a_j), Z) \in PC_k(P)\right\} \\ \cup &\left\{a_1...a_j \in \textstyle\bigcup_{0 \leq j \leq k} \Sigma^j \mid \exists w \in \Sigma^* : (q, a_1...a_j w, Z) \in PC_k(P)\right\} \end{aligned}$$

The set of all posible lookahead strings of length k of parsing situtation s is

$$\begin{aligned} LAS_{P,!k}(s) = &\left\{a_1...a_k \in \textstyle\bigcup_{0 \leq i < k} \Sigma^i \$^{k-i} \mid (q, Clean(a_1...a_k), Z) \in PC_k(P)\right\} \\ \cup &\left\{a_1...a_k \in \Sigma^k \mid \exists w \in \Sigma^* : (q, a_1...a_k w, Z) \in PC_k(P)\right\} \end{aligned}$$

It is easier to imagine different relations on pictures. We therefore show the lookahead handling on lookahead trees.

Definition 7.30 (Lookahead tree, navigation tree):
Tree representation T (see Def. 6.16) of a set of lookahead strings for a situation s is called *lookahead tree for* s.

If the nodes of a lookahead tree T for s are moreover labeled by sets of transitions of P that are possible for a given situation s and for the part of lookahead corresponding to the path from root to given node, we call it *navigation tree for* s.

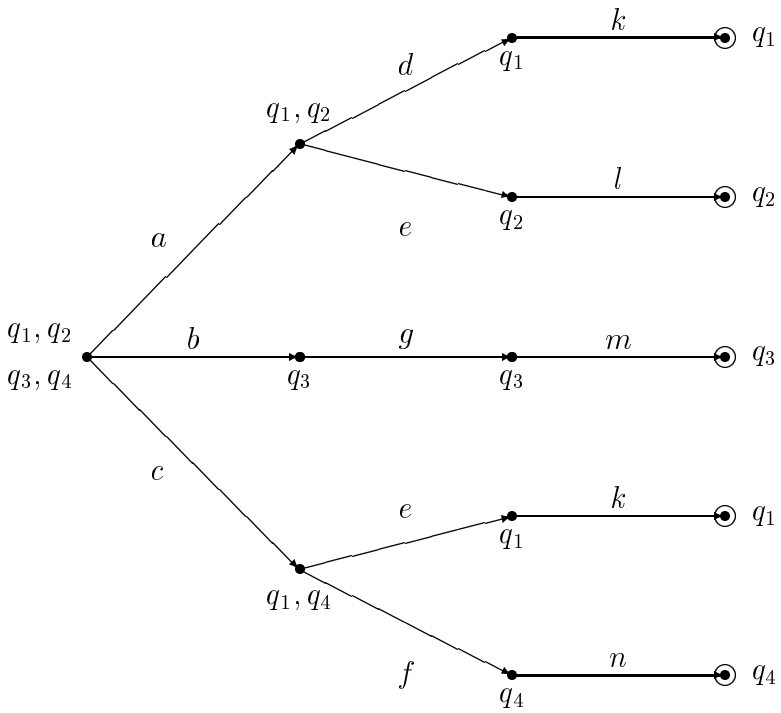


Figure 7.2: Example of a navigation tree

Definition 7.31 (Depth-minimal oracle, depth-minimal lookahead): An oracle is said to be *depth-minimal*, if it uses only the shortest prefix of unread part of the input string necessary to uniquely select an admissible move. Lookahead of a depth-minimal oracle (or navigation tree representing it) is said to be also depth-minimal.

To lower the number of tests necessary to select usable transition we minimize navigation tree in the sense that comparisons performed when there is only one transition left are eliminated.

Algorithm 7.32 (Minimization of a Navigation Tree):
Input: Navigation tree T;
Output: Navigation tree T modified to be depth-minimal;
Method:
```
BEGIN
  WHILE there is a non-leaf node n labeled with a singleton DO
    Delete the subtree rooted in n except n itself;
END;
```

Lemma 7.33 Algorithm 7.32 halts within finite number of steps.

Proof: The graph is at the start of the algorithm finite. In each iteration of the loop at least one node is deleted. Hence the algorithm must halt within finite number of steps. □

Lemma 7.34 In every step of the Alg. 7.32 T is a navigation tree for some fixed situation.

Proof: At the start of the algorithm T is a navigation tree for some situation s.

The change of T in every step consists of deletion of some part (a subtree without its root) of T. It follows from the loop condition that the root of deleted "subtree" (let us call that node n) is labeled by a single move m.

Then all nodes on all paths from n to all leaves of the subtree are also labeled only by m. Removal of these nodes does not influence the fact that T is a navigation tree for s. □

It can be moreover noted that if T_i and T_j are two different modifications of T occurring in the run of Alg. 7.32 and if T_i occured sooner than T_j, then T_j is a navigation tree (or lookahead) refinement of T_i.

Lemma 7.35 The navigation tree returned by Alg. 7.32 is prefix-minimal.

Proof: Let us for contradiction expect that it is possible to further optimize the given navigation tree. Then there must be at least one non-leaf node labeled by a singleton. But if such node exists, Alg. 7.32 enters its WHILE-loop and removes the "subtree" [contradiction]. □

Algorithm 7.36 (Minimization of Lookaheads):

Input:	Any natural $\leq k$-LPDA A
Output:	Depth-minimal natural $\leq k$-LPDA A' such that A and A' are siblings and $L(A') = L(A)$
Method:	On each parsing situtation of A apply Alg. 7.32.

Proof of Th. 7.24: Alg. 7.36 returns for any $\leq k$-LPDA A minimized $\leq k$-LPDA A' such that $L(A') = L(A)$. □

Theorem 7.37 To every $!k$-LPDA A there is a DPDA P such that $L(A) = L(P)$.

Proof of this theorem is on page 82.

The following algorithm and the lemmas concerning it seem to be known but we fail to find any formal descriptions and proofs of it in the literature and therefore we decided to include them here.

Algorithm 7.38 (Construction of an equivalent DPDA to given $!k$-LPDA):

Input: $!k$-LPDA $A = (Q_A, \Sigma, \Gamma, \delta_A, q_0, Z_0, F_A)$.
Output: DPDA $P = (Q_P, \Sigma, \Gamma, \delta_P, (q_0, \circ^k), Z_0, F_P)$.
Method:

$Q_P = Q_A \times (\Sigma \cup \{\circ, \$\})^k$ {$\circ$ stands for uninitialized symbol}

$F_P = \{(q, \$^k) | q \in F_A\}$

$\delta_{RdChr} = \{((q, a_1...a_k), a_{k+1}, Z) \to ((q', a_2...a_{k+1}), \alpha)$
$|(q, a_1..a_k, Z) \to (q', \alpha) \in \delta_A\}$

$\delta_{RdEnd} = \{((q, a_1...a_k), \varepsilon, Z) \to ((q', a_2...a_k\$), \alpha) | (q, a_1..a_k, Z) \to (q', \alpha) \in \delta_A\}$

$\delta_{Move} = \{((q, a_1...a_k), \varepsilon, Z) \to ((q', a_1...a_k), \alpha) | (q, a_1..a_k, Z) \to (q', \alpha) \in \delta_A\}$

$\delta_{IniC} = \{((q_0, \circ^{k-j} a_1...a_j), a_{j+1}, Z_0) \to ((q_0, \circ^{k-j-1} a_1...a_{j+1}), Z_0)\}$

$\delta_{IniE} = \{((q_0, \circ^{k-j} a_1...a_j), \varepsilon, Z_0) \to ((q_0, \circ^{k-j-1} a_1...a_j\$), Z_0)\}$

$\delta_P = \delta_{IniC} \cup \delta_{IniE} \cup \delta_{Move} \cup \delta_{RdChr} \cup \delta_{RdEnd}$

Note that δ_{RdChr} stands for moves reading a character from input far enough from the input end, δ_{RdEnd} stands for moves reading a character from input close to the input end, δ_{Move} represents ε-moves, δ_{IniC} represents initialization moves when the input is long enough, and δ_{IniE} represent initialization moves when the input end is reached (when the input is shorter than k symbols).

Definition 7.39

For Q_P and Q_A from Alg. 7.38 let us define a mapping $State : Q_P \to Q_A$ as

$$State((q, a_1 \dots a_k)) = q.$$

Let us moreover define a function $Input$ mapping configurations of P into Σ^* so that

$$Input((q, a_1 \dots a_k), w, \alpha) = Clean(a_1 \dots a_k) \cdot w.$$

Lemma 7.40 Let $!k$-LPDA A from Th. 7.37 go during the computation over w through configurations $c_0, \dots, c_m$. Then the DPDA P goes during its computation over w through configurations $c'_{-k}, \dots, c'_m$. It holds that if $c'_i = ((q, a_1 \dots a_k), w, \alpha)$, then $c_i = (State((q, a_1 \dots a_k)), Input(c'_i), \alpha)$ for $0 \leq i \leq m$. It moreover holds that $c'_i = ((q_0, \circ^{-i} a_1 \dots a_{k+i}, u_{k+i}, Z_0)$ for $-k \leq i \leq 0$ and where $a_1 \dots a_{k+i} u_{k+i} = w$.

Proof: The computation of P can be divided into two parts: in the first part the DPDA collects the information on the lookahead, in the second it makes the computation.

In the initialization phase of its computation DPDA P goes through the configurations corresponding to the initial configuration of the LPDA A.

At the beginning A is in the configuration $c_0 = (q_0, w, Z_0)$. p starts in the configuration $c'_{-k} = ((q_0, \circ^k), w, Z_0)$. It holds that

$$(State((q_0, \circ^k)), Input(c_{-k}), \alpha) = (q_0, w, Z_0) = c_0.$$

In individual steps (moves) of its initialization P performs these moves according $\delta_{IniC} \cup \delta_{IniE}$ when moves from δ_{IniC} are made first (if there is at least one symbol on the input), then the moves from δ_{IniE} are done (conditionally, only if the input is shorter than k symbols).

In every of the initial configurations $c'_{-k}..c'_{-1}$ of the DPDA P it is possible to perform exactly one move. $c'_i = ((q_0, \circ^{-i} a_1 \dots a_{k+i}), u_{k+i}, Z_0)$ for some i, $-k \leq i < 0$. It further holds that $u_i = a_{i+1} u_{i+1}$. Then there exists only one move $((q_0, \circ^{-i} a_1 \dots a_{k+i}), a_{k+i+1}, Z_0) \rightarrow ((q_0, \circ^{-i-1} a_1 \dots a_{k+i+1}), Z_0)$. This move ends in configuration $c'_{i+1} = ((q_0, \circ^{-i-1} a_1 \dots a_{k+i+1}, u_{k+i+1}, Z_0)$. For all these configurations it holds that

$$\forall i : \ -k \leq i \leq i : \ (State((q, \circ^{-i} a_1 \dots a_{k+i})), Input(c'_i), \alpha) = (q_0, w, Z_0) = c_0.$$

The main computation of the DPDA P is made in steps corresponding to the steps of LPDA A.

Let A be in configuration $c_i = (q, a_1 \dots a_{k+1} v, Z\alpha)$, $i \geq 0$ and P is in configuration $c'_i = ((q, a_1 \dots a_k), a_{k+1} v, Z\alpha)$. Then one of following situations (according to A) must occur:

1. reading a symbol, lookahead contains only symbols from Σ;
2. reading a symbol, lookahead contains also end of input;
3. ε-move;
4. no move is applicable.

Let us discuss individual cases:

1. Let it is for A applicable move $(q, a_1[a_2 \dots a_k], Z) \rightarrow (q', \beta)$ leading to the configuration $c_{i+1} = (q', a_2 ... a_{k+1} v, \beta\alpha)$. For P it is then applicable just one move $((q, a_1 ... a_k), a_{k+1}, Z) \rightarrow ((q', a_2 ... a_{k+1}), \beta)$ going to configuration $c'_{i+1} = ((q', a_2 ... a_{k+1}), v, \beta\alpha)$.

   It holds that $c_{i+1} = (State((q', a_2 \dots a_{k+1})), Input(c'_{i+1}), \beta\alpha)$.

2. Let it is for A applicable move $(q, a_1[a_2 \dots a_j\$^{k-j}], Z) \to (q', \beta)$ going to configuration $c_{i+1} = (q', a_2 ... a_j, \beta\alpha)$. For P it is then applicable just one move $((q, a_1 ... a_j\$^{k-j}), \varepsilon, Z) \to ((q', a_2 ... a_j\$^{k-j+1}), \beta)$ going to configuration $c'_{i+1} = ((q', a_2 ... a_j\$^{k-j+1}), \varepsilon, \beta\alpha)$.

   It holds that $c_{i+1} = (State((q', a_2 \dots a_j\$^{k-j+1})), Input(c'_{i+1}), \beta\alpha)$.

3. Let it is for A applicable move $(q, [a_1 \dots a_k], Z) \to (q', \beta)$ going to configuration $c_{i+1} = (q', a_1 ... a_{k+1} v, \beta\alpha)$. For P it is then applicable just one move $((q, a_1 ... a_k), \varepsilon, Z) \to ((q', a_1 ... a_k), \beta)$ going to configuration $c'_{i+1} = ((q', a_1 ... a_k), a_{k+1} v, \beta\alpha)$.

   It holds that $c_{i+1} = (State((q', a_1 \dots a_k)), Input(c'_{i+1}), \beta\alpha)$.

4. If there is no move possible for A, it follows from the definition of P that there is no move possible also for P. The only exception is the initial configuration of A (and therefore also configurations of the initial part of computation of P) when P may go through all configurations corresponding to the initial configuration of A.

□

Proof of Th. 7.37: Suitable automaton is, for example, the one created by Alg. 7.38. It holds from the Lemma 7.40 that the LPDA and DPDA go through corresponding configurations and therefore they recognize the same language. □

Theorem 7.41 For every DPDA P there is a LPDA A such that $L(A) = L(P)$.

Proof: It suffices to take any LPDA with a lookahead of 1 that for ε-transitions performs given move for any lookahead. □

7.4 Conclusions

We introduced two models corresponding to the work of real recursive descent parsers more precisely than the pushdown automata do. The first, more general, model – oracle pushdown automaton – adds to classical pushdown automaton an opportunity to restrict during the computation the set of applicable moves using other computations or using outer intervention (by user).

Partitioning the problem into parsing and restriction of usable moves allows us to keep the parser well arranged. It simplifies development and

maintenance of so designed parsers. The opportunity to influence the computation from outside is advantageous for the situations when the recognized language is not fully specified (for example, it is impossible to make exact grammar or implement the parser for natural languages, as the language is continuously changing).

Computational power of oracle automata depends on the power of the oracle. The use of simple oracle (lookahead of k symbols inclusive) does not change recognized class of context-free languages. It only brings the model to the intuition closer than the existing model does or it can make the computation more effective. The use of strong oracle (e.g. having unlimited lookahead or having its own stack) allows us to recognize languages that are not context-free.

The second model (lookahead pushdown automata) is a special case of oracle pushdown automata when a finite automaton is used as an oracle. The finite automaton using lookahead of at most or exactly k symbols determines whose moves are applicable. We expect that this model will be used for static and extensible parsing of deterministic languages. In such cases the automata implementing the oracle always select exactly one of the moves or terminate the computation from the reason of improper input.

Chapter 8

Parsers

Some structures for kind parsing are already introduced and described in Chap. 3 together with the principles of their work. This chapter is dedicated to the classical description of kind parsers – i.e. as (lookahead) pushdown automata.

Construction of production trees was already described in Chap. 3. Now they will be transformed into the 'language' of lookahead pushdown automata.

An overview of activities performed during parsing using a production tree forest is in Tab. 8.1.

8.1 Kind (Pushdown) Automaton

Divided (dotted) productions are introduced e.g. in [AU72, NL94]. There can be defined an equivalence on them:

Definition 8.1 (Kind equivalence of dotted productions):

Let the *kind equivalence of dotted productions* be for any context free grammar $G = (\mathrm{N}, \Sigma, P, S)$ as follows:

$$(A \to \alpha.\beta =_G B \to \gamma.\delta) \Longleftrightarrow_{df} (A = B) \wedge (\alpha = \gamma) \quad \left| \begin{array}{l} A, B \in \mathrm{N} \\ \alpha, \beta, \gamma, \delta \in (\mathrm{N} \cup \Sigma)^* \\ A \to \alpha\beta, B \to \gamma\delta \in P. \end{array} \right.$$

If $p : A \to \alpha.\beta$ is a dotted production from G, we denote the equivalence class containing p by $[A \to \alpha.\beta]_{=_G}$ If it is clear that given equivalence is mentioned, only by $[A \to \alpha.\beta]$.

We use for parsing of kind grammars a lookahead pushdown automaton called *k-kind automaton*. For the lookahead description we use the notation from Conv. 7.9.

transition along an edge labeled by a terminal	change the state from the one belonging to the starting point of the edge to the one belonging to the ending point of the edge and read given symbol from the input tape
transition along an edge labeled by a nonterminal	change the state from the one belonging to the starting point of the edge to the one representing the root of the production tree of the non-recursive productions of the nonterminal; state representing end of the original edge is put onto pushdown; input tape is left intact
transition along an edge labeled by a production	lookaheads into left recursion of the defined nonterminal and behind it are checked; if the actual lookahead matches left recursion (belong to DLRF) the state is changed to the one representing root of the left recursive production tree, if the current input lookahead matches the lookahead behind the nonterminal (NLRF), the processing of the nonterminal ends and the state is changed to the one fetched from the pushdown; if the actual lookahead does not match any of both the lookaheads and the stack is empty, the computation ends and the input is accepted; otherwise the computation also ends but the input is rejected

Table 8.1: Parsing using production trees

Algorithm 8.2 generates for given grammar G a kind (pushdown) automaton M such that $L(M) = L(G)$.

Algorithm 8.2 (Creation of a k-kind automaton from k-kind grammar):
Input: k-kind grammar $G = (\mathrm{N}_G, \Sigma_G, P_G, S_G)$
Output: lookahead pushdown automaton M (k-kind automaton)
Method:

1. $Q = \{[A \to \alpha.\beta]_{=_G} \mid A \to \alpha.\beta \in P_G\}$
2. $\Sigma_M = \Sigma_G$

3. $\Gamma = Q \cup \{\circ\}, \quad \circ \notin Q$
4. $q_0 = [S_G \to .\alpha]_{=_G}, \quad S_G \to \alpha \in P_G$
5. $Z_0 = \circ$
6. $F = \{[S_G \to \alpha.]_{=_G} \mid S_G \to \alpha \in P_G\}$
7. $\delta_{Scan} = \{([A \to \alpha.a\beta]_{=_G}, a[b_2 \ldots b_k], q) \to ([A \to \alpha a.\beta]_{=_G}, q, 1) |$
   $A \to \alpha a\beta \in P_G \wedge ab_2 \ldots b_k \in First_G^k(a\beta \cdot Follow_G^k(A)) \wedge q \in \Gamma\}$
8. $\delta_{Descent} = \{([A \to \alpha.B\beta]_{=_G}, [b_1 \ldots b_k], q) \to ([B \to .\gamma]_{=_G}, [A \to \alpha B.\beta]_{=_G}, q, 0) |$
   $A \to \alpha a\beta, B \to \gamma \in P_G \wedge b_1 \ldots b_k \in First_G^k(B\beta \cdot Follow_G^k(A)) \wedge q \in \Gamma\}$
9. $\delta_{Recurse} = \{([B \to \alpha.]_{=_G}, [a_1 \ldots a_k], q) \to ([B \to B.\beta]_{=_G}, q, 0) |$
   $B \to \alpha, B \to B\beta \in P_G \wedge a_1 \ldots a_k \in DLRF_G^k(B) \wedge q \in \Gamma\}$
10. $\delta_{Finish} = \{([B \to \gamma.]_{=_G}, [a_1 \ldots a_k], [A \to \alpha B.\beta]_{=_G}) \to ([A \to \alpha B.\beta]_{=_G}, \varepsilon, 0) |$
    $B \to \gamma, A \to \alpha B\beta \in P_G \wedge a_1 \ldots a_k \in NLRF_G^k(B)\}$
11. $\delta = \delta_{Scan} \cup \delta_{Descent} \cup \delta_{Recurse} \cup \delta_{Finish}$

Algorithm 8.2 just translates construction and behavior of a production-tree-based parser into the world of (lookahead) pushdown automata.

8.2 Extension

It was stated in Chap. 3 how the structure of a production tree forest can be extended. It is always true that existing nodes and edges are preserved. Even if the generator or analyzer has stored some reference to any edge or node, it cannot get in troubles this way. Its state will always be correct and without change – even in the case of an extension of its structure. This feature is preserved also in the transformations: (lookahead) pushdown automata built following above stated methods handling production trees (based on them) can be arbitrarily (within the principles) extended without any loss of run-time collected information. They are only added new states and transitions (respective items and derivations) that reflect the new edges and nodes in the production tree forest.

An extension can be done at any phase of computation. For the compiler domain it can be useful to restrict extensions of compiled language only

to places where there are subroutines, data types, or variables defined – and possibly only to the highest level (i.e. where the global definitions may occur).

If it will be allowed to extend the syntax anywhere (and the technology allows it), there can arise some semantic and practical problems: the code becomes hard-to-understand and also somehow ambiguous. It is similar situation to the declaration of variables: even if it is allowed to do it everywhere (e.g. in C/C++) it is not used in practice. Sometimes it can be more useful to design a more restrictive language just to force programmers to keep the code well-readable and easy to understand.

8.3 Reduction

Reduction of the automaton is potentially possible everywhere but with caution to keep resulting grammar k-kind (keeping the grammar reduced can be done automatically) but it is also needed to check whether the reduction would not block successful finishing of the computation. For proper function of the parser – in any of its possible forms – it is necessary to preserve all edges and nodes currently in use. It is also needed to sustain the 'path' from these places to the leaves of corresponding production trees (or their encodings). That is, also other productions ensuring the path all used nonterminal edges of production trees (or their encodings) that should be visited in the finishing of the computation are blocked this way.

This condition can be weakened if it is allowed to extend and reduce the parser concurrently or in the time between the reduction and entering the critical point of the computation (but this is intuitively very risky).

For practical use it will be useful to check preserving of positions in use and after any set of extensions and reductions at one position of computation test whether the some path to successful finishing of the computation (reading of the text) exists.

For off-line changes it is possible to follow the same method – only there is no need to check for 'used' production tree edges and nodes in the structure of the parser.

Chapter 9

Grammars for Practical Use

In practice the problem is usually not just to state whether the text belongs to the given language, but to parse the text and use the parse for translation or for collection of some important use. Therefore we introduce grammars that fit this purpose.

9.1 Semantic Actions

Parsers in their theoretical nature return only results 'yes' or 'no' answering the question whether the input belongs to the given language.

For practical use it is better when the output is in the form of syntax tree or of the form of strings of symbols representing semantic actions that should be done (however in proper order).

Such parsers return for an input of the form $a_1 a_2 \ldots a_n$ an output of the form $\sigma_0 a_1 \sigma_1 a_2 \ldots a_n \sigma_n$ where σ_i are strings of semantic symbols[1]. These semantic actions may be used as invocation of activities of the application in which is the parser built in.

Therefore there will also be grammars having – as an addition to the terminal and nonterminal symbols – also semantic symbols mentioned.

Example 9.1 (Simple arithmetic expression and semantic actions):

If there are also semantic actions used in the definition of a grammar of a simple arithmetic expression, there would be got a tool describing not only syntax but also (in some sense) the semantics of described language.

For easier recognition of a symbol type the different symbol types will be written in different brackets:

[1] Presence of terminals in the output string is important especially in a situation when terminal symbols represent some values (e.g. strings, identifiers, numbers). Sometimes their values may be used.

()	terminal
[]	nonterminal
{}	semantic action

The resulting grammar with its semantic actions is here:

$[S] \to [E]\{\text{print}\}$
$[E] \to [T]$
$[E] \to [E](`+')[T]\{\text{addition}\}$
$[T] \to [F]$
$[T] \to [T](`*')[F]\{\text{multiplication}\}$
$[F] \to (id)\{\text{fetch}\}$
$[F] \to (num)\{\text{fetch}\}$
$[F] \to (`(')[E](`)')$

{addition}	fetch (pop) two topmost values form the semantic pushdown, add them, and push the result back onto semantic pushdown
{multiplication}	fetch (pop) two topmost values form the semantic pushdown, multiply them, and push the result back onto semantic pushdown
{print}	fetch (pop) the topmost value from semantic pushdown and print it out
{fetch}	fetch a terminal value from lexical analyzer and push it onto the semantic pushdown

Table 9.1: Meaning of semantic actions of a simple arithmetic expression

Meaning of individual semantic actions is described in Tab. 9.1. It is expected that there is a pushdown for value storing and that the lexical analyzer is able to give actual values of the terminals *num* and *id* – i.e. their numerical values, and identifier strings, respective values represented by the identifiers. □

The idea of easy-to-read languages is very advantageous for the processing of such grammars: if it is possible to write semantic actions at arbitrary places within the productions (with the exception of beginning of left-recursive productions), the LR(k) grammars would be restricted to LL(k) ones because providing of semantic actions should be decided in time and not at the end of the productions, as LR-grammars allow (see [PQ96]).

9.2 Translational Grammars

A similar role as semantic actions have for the parser is played by the output (terminal) symbols. Parsers with output alphabet are usually called transducers ([AU72, MČJR99]).

Pure transducers can be used as simple message format convertors, e.g. as simple front-end gates in software confederations.

9.3 Semantic Translational Grammars

Definition 9.2 (Semantic translational grammars):

A *semantic translational context-free grammar* (STCFG) is a 6-tuple $G = (\mathrm{N}, \Sigma, \Phi, \Psi, P, S)$ where

- N is a nonempty finite set of *nonterminal symbols*,
- Σ is a nonempty finite set of *(input) terminal symbols* that can also be called *input alphabet*,
- Φ is a finite set of *semantic actions*,
- Ψ is a finite set of *output terminal symbols* that can be also called *output alphabet*,
- P is a finite subset of $\mathrm{N} \times (\mathrm{N} \cup \Sigma \cup \Phi \cup \Psi)^*$ called set of *productions*,
- $S \in \mathrm{N}$ is a distinguished symbol in N called *starting symbol*, and

N, Σ, Φ, and Ψ are pairwise disjunct.

If $\Psi = \emptyset$ then grammar is called *semantic context-free grammar* (SCFG). If $\Phi = \emptyset$ then grammar is called *translational context-free grammar* (TCFG).

Definition 9.3

Let G be a STCFG. Context-free grammar G' is called *a simplification of* G if for all symbols a_i from Φ and Ψ new nonterminals A_i and new productions $A_i \to \varepsilon$ are added and all production containing a_i are substituted by productions containing at the same position A_i.

Definition 9.4 (Classification of STCFGs):

A STCFG G is called deterministic, LR(k), LL(k), (weak) k-kind if and only if a simplification of the grammar G is deterministic, LR(k), LL(k), (weak) k-kind respectively.

Translational grammars are discussed e.g. in [MČJR99], semantic grammars can be found e.g. in [ŽK99a].

Both translational and semantic grammars are used to build compilers (see e.g. [Krá82, MČJR99]). In the construction of LR-based parsers it is rather common to perform semantic actions when reducing by some production. This situation can be in this formalism captured by the addition of respective semantic actions at the ends of productions. Similarly behaves also the compiler from [Žem94].

Distinguishing between output symbols and semantic actions may be useful: the first alphabet can contain symbols generated into output and the second one commands changing the internal structure (or state) of the application (e.g. variable declarations) or the parser itself (e.g. its oracle configuration).

Example 9.5 (Infix to postfix transformation with expression evaluation):

Let us have the following translational grammar with semantic actions allowing to translate simple arithmetic expression from infix to postfix notation and to evaluate the expression.

Nonterminals	$\{S, E, T, F\}$
(Input) terminals	$\{num, +, -, *, (.)\}$
Output terminals	$\{\underline{num}, \underline{+}, \underline{-}, \underline{*}, \underline{=}\}$
Semantic actions	{fetch, put, push, pop, add, sub, mul}
Productions	$S \to E \underline{=}$ pop put $\underline{num}$ $E \to E + T \underline{+}$ add $E \to E - T \underline{-}$ sub $E \to T$ $T \to T * F \underline{*}$ mul $T \to F$ $F \to num$ fetch push put $\underline{num}$ $F \to (E)$
Starting symbol	S

The grammar expects that lexical analyzer stores values of num into a special queue. Numeric values are taken from the buffer by the semantic action "fetch". There are also a semantic stack serving as a semantic store, and a special register – accumulator. There is also a writing unit that uses a special buffer for numeric values of output terminals $\underline{num}$. This buffer can be filled using a semantic action "put".

Semantic actions overview	
Name	**Description**
fetch	fetches a value from lexical analyzer into accumulator
put	puts a value from accumulator into writing unit
push	pushes a value from accumulator onto semantic stack
pop	pops a value from semantic stack into accumulator
add	pops two topmost values from semantic stack, adds them, and the result is pushed back onto semantic stack
sub	pops two topmost values from semantic stack, subtracts them (the topmost from the second topmost), and the result is pushed back onto semantic stack
mul	pops two topmost values from semantic stack, multiplies them, and the result is pushed back onto semantic stack

□

9.4 Grammar Extensions

Definition 9.6 (Grammar extension):
A 3-tuple $E = (\mathrm{N}', \Sigma', P')$ is called *grammar extension* of a grammar $G = (\mathrm{N}, \Sigma, P, S)$ if:

1. $(\mathrm{N} \cup \mathrm{N}') \cap (\Sigma \cup \Sigma') = \emptyset$, and
2. $P' \subset (\mathrm{N} \cup \mathrm{N}') \times (\mathrm{N} \cup \mathrm{N}' \cup \Sigma \cup \Sigma')^*$.

Grammar $G' = G \cdot E$ is called *extended to G by E* if $G' = (\mathrm{N} \cup \mathrm{N}', \Sigma \cup \Sigma', P \cup P', S)$.

Definition 9.7
Grammar extension is called *class preserving*, if the extended grammar $G \cdot E$ belong to the same class of grammars as the original grammar G.

Lemma 9.8 For any grammar G and its grammar extension E holds

$$L(G) \subset L(G \cdot E)$$

Proof: For every word from $L(G)$ there exist a derivation in G:

$$S \Rightarrow_G \alpha_1 \Rightarrow_G \ldots \Rightarrow_G \alpha_m \Rightarrow_G w$$

Let there were used productions $p_{i_1} \ldots p_{i_m}$. As $p_{i_j} \in P_G$ and $P_G \subset P_{G \cdot E}$ it must also hold that $p_{i_j} \in P_{G \cdot E}$ and therefore also the same derivation is applicable using $G \cdot E$, i.e. $w \in L(G) \Rightarrow w \in L(G \cdot E)$. □

9.4.1 Parse-time extensions

Grammars and parsers can be extended not only off-line but also during generation/parse time. The parse-time extension can be caused by an external entity (and let us call it *asynchronous*), or it can be done as semantic action (and is called *synchronous*). Synchronous grammar extensions can be part of the grammar (i.e. predefined; specified by the author of the grammar/parser) or can be specified within the just parsed text (specified by the grammar/parser user – by the author of the parsed text). Asynchronous extensions may be used e.g. for maintenance of some software products.

As shown above, the extensions made off-line just switch (enrich) the language from $L(G)$ to $L(G \cdot E)$. The on-line (parse-time) extensions can potentially change even the grammar class of the recognized language.

Definition 9.9

Semantic grammar is called *extensible semantic grammar* if at least some of the semantic actions are grammar extensions. If all semantic actions in an extensible semantic grammar are grammar extensions, the grammar is called *proper extensible grammar*.

Example 9.10 (Proper extensible kind grammar):

Let grammar G has $\Sigma = \{a, b, c, d, e\}$, N $= \{S, A, B\}$, starting symbol S, productions

$S \to AeB$
$A \to aAb \mid c\sigma \mid d\tau$
$B \to aBb$

and semantic actions $\Phi = \{\sigma, \tau\}$ where σ adds the production $B \to c$ and τ adds $B \to d$.

$L(G) = \{a^n cb^n ea^m cb^m | m, n \geq 0\} \cup \{a^n db^n ea^m db^m | m, n \geq 0\}$ □

Note 9.11 The language $L(G)$ from the above example is parsable by run-time extensible deterministic top-down parsers (LL or kind ones) but the language itself is a non-LL one.

This language can be generated by a SLR(1) grammar G_{SLR}:

$S \to CeC \mid DeD$
$C \to aCb \mid c$
$D \to aDb \mid d$

Idea of the proof: For LL(k) grammars it is not possible to check whether left and right brackets always match and at the same time check whether there are only c's or only d's in just parsed word, if their first occurrence is in "deep enough" brackets.

Proof: Let us – for contradiction – expect, that there is some k and LL(k) grammar G' such that $L(G') = L(G)$. Every LL(k) language has an LL($k+1$) grammar in Greibach normal form [AU72, Ex. 5.1.24, page 362]. Let us therefore expect, that there exist a LL(k) grammar G in Greibach normal form such that $L(G) = L$.

Let $p = |N_{G'}|$. $L(G)$ contains also words like $a^{kp+k+i}cb^{kp+k+i}ea^mcb^m$ (for some $i, m \geq 0$). Let $n = kp + k + i$.

In parsing of languages specified by LL(k) grammars the productions to be applied are unambiguously selected using the nonterminal to be expanded and using k terminals of lookahead. Hence the first at least $n-k$ productions are selected using the lookahead of a^k only according to the nonterminal to be expanded. The first nonterminal is the starting symbol S.

As there are too many a's to be read and as there must be the same number of b's in a sequence of the input string, it must happen that at least one nonterminal must be expanded more than once. According to "pumping lemma" (see e.g. [AU72], pages 192–195) there must be a sequence of productions $A \Rightarrow^* a^{jm}Ab^{jm}$ for some nonterminal A.

The nonterminal A can be rewritten to

1. a^jcb^l,
2. a^jdb^l, or
3. any of a^jcb^l and a^jdb^l

for some $j, l \geq 0$. If any of the first two points is valid, the grammar cannot cover the given language as it cannot generate both $a^ncb^nea^mcb^m$ and $a^ndb^nea^mdb^m$. Hence both $A \Rightarrow^* a^jcb^l$ and $A \Rightarrow^* a^jdb^l$ hold.

At the same time $\alpha_i \Rightarrow^* b^iea^mcb^m$ or $\alpha_i \Rightarrow^* b^iea^mdb^m$. Expansions of A and α_i are independent. As α_i is derived sooner than c or d is in the lookahead, it cannot be influenced by first occurrence of c or d.

There are only following cases possible:

1. $\alpha_i \Rightarrow^* b^iea^mcb^m$,
2. $\alpha_i \Rightarrow^* b^iea^mdb^m$, and
3. $\alpha_i \Rightarrow^* b^iea^mcb^m$ or $\alpha_i \Rightarrow^* b^iea^mdb^m$.

As α_i is derived sooner than first c or d is visible in the lookahead, points 1 and 2 (with properly selected variant for A) restrict the language too much, whereas point 3 (with the variant 3 for A) extends the language too much. All other choices for A and α_i exceed the given language. There is therefore no LL grammar for the given language L. □

Conjecture 9.12 Language given by a proper extensible grammar or by an extensible semantic grammar can exceed language class that can contain all languages specified (generated by the grammars) during the parsing process.

9.5 Modifications

Some languages used in practice can exceed the abilities of kind parsing. It is therefore useful to extend the technique of static kind parsing not only in the semantic way, but also in the syntactic one: we can define solutions for some conflicts or even to allow nondeterminism.

9.5.1 Extended Kind Parser

Some programming languages are close to the LL languages but they are, in fact, not LL languages. There is a well-known example of such a language – the language `Pascal` with its 'dangling else' problem: there are constructions '`IF` *condition* `THEN` *command*' and '`IF` *condition* `THEN` *command* `ELSE` *command*' leading to grammatically ambiguous constructions, like in Ex. 9.13.

Example 9.13 (Dangling-else problem in Pascal):
Construction from Fig. 9.1 can be parsed two ways that are shown as Figs. 9.2 and 9.3. □

`IF` *condition1* `THEN IF` *condition2* `THEN` *command1* `ELSE` *command2*

Figure 9.1: Two pascal IF-commands with one ELSE

Solution: Similarly to the solution taken by N. Wirth in the Pascal language it can be adapted solution of such conflicts using attachment of the 'else' to the nearest unmatched 'if'. This approach was used also in [Žem94].

```
IF condition1 THEN
   IF condition2 THEN command1
ELSE command2
```

Figure 9.2: Two pascal IF-commands with one ELSE attached to the outer IF

```
IF condition1 THEN
   IF condition2 THEN command1 ELSE command2
```

Figure 9.3: Two pascal IF-commands with one ELSE attached to the closer IF

9.5.2 Generalized Kind Parser

The idea of production trees is applicable also for general context-free grammars. The only change is that more paths in the tree can be followed at a time (and, of course, they can be entered using the same lookahead).

This kind of parsing can work only with grammars having productions without left recursion and with direct left recursion. (Combinations of hidden and indirect left recursion are not allowed.) As these constructs are not necessary to cover all context-free languages, generalized kind parsers (their first implementation is in [Svo02]) are yet another parsing technique for the class of context-free languages.

9.6 Conclusions

The basic method of kind parsing can be extended in many ways: it can be applied non-deterministically, it can be extended by some additional solutions of conflicts, there can be added new features like output generation (changing the goal from parsing to transducing), or by extensibility of the parsers. These extension concepts can be applied orthogonally. It shows that kind parsing is a good background for further fruitful research and development.

Chapter 10

Kind Parser Constructor

For every subclass of deterministic context-free grammars it is common to create a program (constructor) that is able for any grammar from the given subclass to make a set of source files implementing a parser for given grammar. This rule is preserved also for the class of 1-kind grammars that are expected to be used most frequently – an example is [Žem02a] that generates to given grammar description a set of C-language source and header files implementing a parser for given language. Its code itself is, however, easy to read and easy to extend.

Semantic actions are also supported: if declared, they are generated into the parser code and there are also generated source and header files with skeletons of semantic action implementing functions.

The source code generation for longer lookaheads than 1 is not straightforward: every time the longer lookahead would be used, the corresponding place in the source code would be worse readable – at least there is no program construction for switching following more values concurrently (like case/switch command for single values). Such code generator is an issue for further work – check the homepage of [Žem02a] for actually available features.

10.1 Implementation Details

The work of kind constructor can be divided into two phases:

1. Creation of a production tree forest for given grammar,
2. generation of header and source files in C.

The first phase was already described in Chap. 3, the second (i.e. the encoding of the structure in a programming language) is described now.

10.1.1 Generation of a nonterminal parsing subroutine

The generation of a kind parser can be divided into separable parts – it is possible (and usual) to generate for each nonterminal a separate parsing subroutine (function in C language).

Structure of the subroutine is a large and complex selection (switch in C language) followed by an optional while loop that usually has in its body another complex selection. The loop is generated only in the case when the nonterminal has at least one left-recursive production.

10.1.2 Language description file

The whole description of a language the parser should be constructed for is located in one file. This file is divided into parts – *sections*. Each of the sections is dedicated to the one specific part of the language description. An overview is available in Tab. 10.1.

Name	Mandatory	Description
Language	yes	Settings of some language and application parameters (names of special terminals, starting symbol and the language itself, lookahead depth) and setting of some bounds (limits for number of terminals, nonterminals, productions, semantic actions, output terminals, keywords, symbol tree nodes, nodes and edges of production trees, and lookahead nodes).
Terminals	yes	List of names of all terminal symbols (special ones inclusive).
Nonterminals	yes	List of names of all nonterminal symbols (starting symbol inclusive).
OutTerms	no	List of names of all output terminal symbols.
Actions	no	List of names of all semantic actions.
Productions	yes	List of all productions.
Keywords	no	Definition of keyword shapes.
Symbols	no	Definition of symbol shapes (i.e. of terminal symbols containing only non-alphanumeric characters – e.g. "+" or ":=").

Table 10.1: Overview of selected sections from a language description file

Example 10.1 (Language description file for a language of simple arithmetic expressions):

```
[Language]
        Name = DefaultLang
        Terminals = 20
        Nonterminals = 10
        SymbNodes = 10
        edges = 200
        productions = 10
        views = 200
        lookahead = 1
        nodes = 200
        actions = 6
        StartingSymbol = S
        IdName = id
        NumName = num
        EofName = eoi
[Terminals]
        id
        num
        plus
        star
        lpar
        rpar
        eoi
        minus
[Nonterminals]
        S
        E
        T
        F
[Actions]
        getidval
        getnum
        add
        subtract
        multiply
        print
[Productions]
        S::=[E]{print}
```

```
        E::=[T]
        E::=[E](plus)[T]{add}
        E::=[E](minus)[T]{subtract}
        T::=[F]
        T::=[T](star)[F]{multiply}
        F::=(id){getidval}
        F::=(num){getnum}
        F::=(lpar)[E](rpar)
[Symbols]
        star = *
        lpar = (
        rpar = )
        plus = +
        minus = -
```

□

Example 10.2 (Generated parser code):

The following lines were generated for the grammar from Ex. 10.1 by the kind constructor [Žem02a]. The output was edited to be more legible: Tab characters were replaced by a proper number of spaces.

```
/*****************************************************************************/
/*                                                                           */
/* _parser.c: Syntactic analyzer (parser)                                    */
/* Generated by KindCons v0.10 written by Michal Zemlicka 2001               */
/*                                                                           */
/*****************************************************************************/

#include        "_parser.h"
#include        "_lexer.h"
#include        "_actions.h"

/* forward definition of all nonterminals */

ErrCode Nont_S(void);
ErrCode Nont_E(void);
ErrCode Nont_T(void);
ErrCode Nont_F(void);

/***********************************************/
/* Nont_S: parsing subroutine for nonterminal 'S' */
/***********************************************/

ErrCode Nont_S(void) {
```

```
        ErrCode         e;
        TermCode        Terminal;

        if  ( ( e = Nont_E() ) != ecOk )
                return  e;
        if  ( ( e = Action_print() ) != ecOk )
                return  e;
        return  ecOk;
}

/************************************************/
/* Nont_E: parsing subroutine for nonterminal 'E' */
/************************************************/

ErrCode Nont_E(void) {
        ErrCode         e;
        TermCode        Terminal;

        if  ( ( e = Nont_T() ) != ecOk )
                return  e;
        if  ( ( e = LookTerm(&Terminal) ) != ecOk )
                return  e;
        while  ( ( Terminal == Term_plus )||( Terminal == Term_minus ) ) {
                switch  ( Terminal ) {
                        case Term_plus:
                                if  ( ( e = MatchTerm(Term_plus) ) != ecOk )
                                        return  e;
                                if  ( ( e = Nont_T() ) != ecOk )
                                        return  e;
                                if  ( ( e = Action_add() ) != ecOk )
                                        return  e;
                                break;
                        case Term_minus:
                                if  ( ( e = MatchTerm(Term_minus) ) != ecOk )
                                        return  e;
                                if  ( ( e = Nont_T() ) != ecOk )
                                        return  e;
                                if  ( ( e = Action_subtract() ) != ecOk )
                                        return  e;
                                break;
                        default:
                                return  ecSyntax;
                }
                if  ( ( e = LookTerm(&Terminal) ) != ecOk )
                        return  e;
        }
        return  ecOk;
}
```

```
/***********************************************/
/* Nont_T: parsing subroutine for nonterminal 'T' */
/***********************************************/

ErrCode Nont_T(void) {
        ErrCode         e;
        TermCode        Terminal;

        if  ( ( e = Nont_F() ) != ecOk )
                return  e;
        if  ( ( e = LookTerm(&Terminal) ) != ecOk )
                return  e;
        while  ( Terminal == Term_star ) {
                if  ( ( e = MatchTerm(Term_star) ) != ecOk )
                        return  e;
                if  ( ( e = Nont_F() ) != ecOk )
                        return  e;
                if  ( ( e = Action_multiply() ) != ecOk )
                        return  e;
                if  ( ( e = LookTerm(&Terminal) ) != ecOk )
                        return  e;
        }
        return  ecOk;
}

/***********************************************/
/* Nont_F: parsing subroutine for nonterminal 'F' */
/***********************************************/

ErrCode Nont_F(void) {
        ErrCode         e;
        TermCode        Terminal;

        if  ( ( e = LookTerm(&Terminal) ) != ecOk )
                return  e;
        switch  ( Terminal ) {
                case Term_id:
                        if  ( ( e = MatchTerm(Term_id) ) != ecOk )
                                return  e;
                        if  ( ( e = Action_getidval() ) != ecOk )
                                return  e;
                        break;
                case Term_num:
                        if  ( ( e = MatchTerm(Term_num) ) != ecOk )
                                return  e;
                        if  ( ( e = Action_getnum() ) != ecOk )
                                return  e;
                        break;
                case Term_lpar:
```

```
                        if  ( ( e = MatchTerm(Term_lpar) ) != ecOk )
                                return  e;
                        if  ( ( e = Nont_E() ) != ecOk )
                                return  e;
                        if  ( ( e = MatchTerm(Term_rpar) ) != ecOk )
                                return  e;
                        break;
                default:
                        return  ecSyntax;
        }
        return  ecOk;
}

/*************************/
/* Parse: parse given file */
/*************************/

ErrCode Parse(char *InName) {
        ErrCode e;      /* status */

        if  ( ( e = LexOpen(InName) ) != ecOk )
                return  e;
        e = Nont_S();    /* call starting symbol subroutine */
        LexClose();
        return  e;
}
```

□

As mentioned in Ex. 10.2, the generated code is close to the hand-written ones. The generated code is legible and understandable and allows modifications by hand. It gives an opportunity to reengineer the original grammar.

The parser is designed as recursive descent one.

Chapter 11

Kind Transducer

One of important software engineering tasks is an integration of several standalone applications. It is usually done by message passing. The problem is that the applications (components of the new whole) can be independently developed and originally not meant for further integration. Therefore the applications often use different communication languages (message formats and semantics). Hence there is a need for translation of messages (or message sets) between different communication languages.

This architecture (building large systems from standalone applications) is known as *software confederation* [KŽ00, KŽ01, KŽ02].

The task of translation between different communication languages is in such systems dedicated into separate standalone applications – into *front-end gates*. If the communication languages are XML dialects, these front-end gates can be XSLT [W3 99b] scripts. If the communication languages are not XML-based, it is better to use usual transducers.

Kind transducer is based on semantic translational kind grammars.

Example 11.1 (Transformation of simple arithmetic expressions from infix to postfix notation with evaluation):

Language description file:

```
[Language]
        Name = DefaultLang
        Terminals = 20
        Nonterminals = 10
        SymbNodes = 10
        edges = 200
        productions = 10
```

```
        views = 200
        lookahead = 1
        nodes = 200
        actions = 6
        StartingSymbol = S
        IdName = id
        NumName = num
        EofName = eoi
        SemInsts = 50
        StackSize = 50
        OutTerms = 4
[Terminals]
        id
        num
        plus
        star
        lpar
        rpar
        eoi
        minus
[Nonterminals]
        S
        E
        T
        F
[OutTerms]
        OutPlus
        OutMinus
        OutStar
        OutEqual
[Actions]
        getidval
        getnum
        add
        subtract
        multiply
        print
[Productions]
        S::=[E]<OutEqual>{print}
        E::=[T]
        E::=[E](plus)[T]{add}<OutPlus>
```

```
            E::=[E](minus)[T]{subtract}<OutMinus>
            T::=[F]
            T::=[T](star)[F]{multiply}<OutStar>
            F::=(num){getnum}
            F::=(lpar)[E](rpar)
    [Symbols]
            star = *
            lpar = (
            rpar = )
            plus = +
            minus = -
    [OutShapes]
            OutPlus = +
            OutMinus = -
            OutStar = *
            OutEqual = =
    [Instructions]
            #getnum
            fetch    r0
            push    r0
            put      r0
            return
            #add
            pop      r0
            add      s0,r0
            return
            #subtract
            pop      r0
            sub      s0,r0
            return
            #multiply
            pop      r0
            mul      s0,r0
            return
            #print
            pop      r0
            put      r0
            return
```

Input file: `4*(3+2)-7*(3-2)`

Output file: `4 3 2 + * 7 3 2 - * - = 13`

□

Note 11.2 The Ex. 11.1 was processed by KindTran [Žem02b], i.e. both language description and source file were used as an input of this application and result is exactly taken from the file produced by this application.

Note 11.3 Power of applications based on semantic translational grammars is dependent on abilities of the application's semantic engines. It is even possible to exceed boundary of the original grammar class (i.e. it is possible with some powerful semantics to handle more complex languages than indicated by simplified context-free grammar to the STCFG one.

Note 11.4 The implementation of production tree nodes may be dependent on the use: For interpretation (transduction and extensible compiling) it is better to use node structure close to ternary search trees [BS98] whereas for parser construction it is better to use structures similar to the ones from [RM85].

Chapter 12

Applications

12.1 Extensible Compiler

Ideas of kind parsing were already tested in a run-time extensible compiler for LL(1) languages – [Žem94]. Its language abilities match the class of extended 1-kind parser. Example 1.1 is one of the examples from the installation disk of the compiler.

Although it is not 'production quality' compiler (it is just a prototype) it has shown, that one compiler for a whole class of languages can be written and that some kind of run-time extensibility of parsers and compilers may be implemented – sooner than XML and its dialects have occurred.

The ability of kind parsing to generate easy-to-read parsers is shown by kind constructor [Žem02a] more discussed in Chap. 10.

12.2 Algorithmic Development Environment

There was an attempt to implement a development tool for handling both algorithm and data types in their most general form ([ŽBB+98]). It also supported the definition and use of domain specific syntax. To be able to do this, the parser from [Žem94] was used: once for parsing of complex expressions with an extensible syntax and once for import of textual form of algorithm and data type descriptions.

Chapter 13

Related Results

There are several types of related works:

- extensible parsers
- extensible compilers
- recursive descent parsers
- extensible development tools
- dynamic parsing
- semantic predicates

13.1 Extensible Compilers

Most of extensible compilers that we have found are based on run-time library hooks – i.e. usually the authors try to extend only semantics and let the syntax intact – like extensions to the language ML (see e.g. [TS97]).

An example of other type of an extensible compiler is described by Bosch [Bos96, Bos97]. He suggests to decompose compiler and all its parts to small units that can be reused for construction of compiler modifications. The decomposition of parsers is briefly described later.

MAGIC

MAGIC is an extensible compiler that supports interface extensions. There is a special interface for dynamic linking of extension implementing modules. More information about this project is available e.g. from [Eng99].

13.2 Extensible Parsers

Nowadays, most widely used run-time extensible parsers are the ones for XML. They are based on fixed markers (tags) that are of three types: opening, closing, and empty. Any XML-language is a deterministic context-free language [BB00].

Other techniques are used e.g. in Lua [IdFF]: there are special points in the parser where user defined functions (fallbacks) may be inserted in extension queues. Therefore extensions are possible only at predefined places.

Bosch suggests in [Bos96, Bos97] to decompose parser into smaller units (grammar modules) that can cooperate in parsing the text as a whole. A module can delegate parsing to some other module. Modules share input and parse graph structure. Whole parser is a composition of such modules.

Cardelli, Matthes, and Abadi describe in [CMA94] their extensible parser that is part of some database environment. It is able to parse LL(1) grammars and can be extended only off-line, i.e. the extended parser must be generated by some tool and then it can be applied – it does not allow language extension within parsed files.

Kolbly in [Kol02] suggests to use Earley parsing algorithm that can be used trivially and covers the whole class of context free languages. This generality is paid by slowing down of the parsing part of the system. Kolbly moreover states that parsing is in the comparison with other parts of the application very fast, therefore even significant slowdown of parsing made by using Earley algorithm instead of usual deterministic algorithms slows down the whole system only by a few percent.

Very deep research of extensible parsing has been made by Wegbreit [Weg70]. He allows arbitrary complex semantic actions and extensions (based on Turing machine). Such parsers are able to parse languages that exceed context-free languages.

Some newly published papers [For02, Gri04, Gri06] propose to use parsing expression grammars (PEGs) in combination with *packrat parsing* [For02]. The notation is similar to grammars with regular right hand sides but it is enhanced by some new constructs that makes PEGs better suited for precedence specification, etc. The backtracking can be speeded up by memoizing the partial results – what is the case of packrat parsing.

13.3 Recursive Descent Parsers

Recursive descent parsers are known for a very long time. They are used in (probably) most compilers (at least of the hand-written ones for impera-

tive programming languages) and other applications. This topic is discussed e.g. in [Gri71, Krá82, Dró85, Ruß97].

The main result of [Krá82] and [Dró85] is that the recursive descent parsing can be used for LR grammars.

13.4 Extensible Development Tools

Almost all software development environments are in some sense extensible: it is usually possible to integrate some additional tool and to pass some parameters and data to it and use or at least show its results. Most of integrated development environments are able to add invocation of third party tools in their menus or link additional libraries. So it is usually possible to extend semantic abilities of the compiled programming language.

Also make tools are usually able to call any command line tool – even the third party ones. So it is possible to divide a compilation process into separate parts done by separate tools. Examples of extensions done by this way are domain specific preprocessors.

MOZART

MOZART is an example of an extensible development tool. It is based on plug-in libraries that can extend the basic skills of the environment and tools. Its homepage is `http://mozart-dev.sourceforge.net/`.

13.5 Dynamic Parsing

Dynamic parsing presented by Rußmann ([Ruß98, Ruß97]) is based on production 'flagging': each production has a flag that indicates whether the production is currently available or not. These flags can are set and reset during the parsing process. Rußmann has also proved ([Ruß98]) that dyn-LL(1) grammars cover arbitrary deterministic context-free language and therefore they have the same generative power as LR(1) grammars.

13.6 Semantic Predicates

Ganapathi in [Gan89] used semantic predicates to select applicable productions during the parsing process. Application of such predicates are discussed also by Parr and Quong in [PQ94].

We suppose that our model of oracle is better usable in practice. We conjecture that the concept of oracle is more general than the use of "semantic" predicates.

13.7 Languages and compilers supporting user-defined operators

Many programming languages allow users to define their own operators. This kind of extensibility is significantly simpler than the extensibility presented in this work allowing to define whole constructs. The addition of user-defined operator can be based on principles known from identifier/keyword recognition. Fixed operator priority can be resolved in compile time, dynamic operator priority needs additional simple run-time support.

13.8 Parser Generators

There are many parser generators. The probably best-known lexer/parser generators are yacc [Joh77] and bison [bison]. They are based on table-driven LALR(1) parsing method. They allow creation of more general parsers than kind constructor does but the resulting generators cannot be so easily tuned by hand. Newer versions of bison supports also generalized LR parsing.

As the representatnts of other parser generators let us remind ANTLR [PQ95] based on syntactioc predicates [PQ94] and COCO [RM85] supporting productions with regular right hand side.

Chapter 14

Conclusion

The goal of this work was to develop such a variant of parsing that would be easily extensible[1], usable in practice[2], and able to process also grammars having limited left recursion, for example standard grammar of arithmetical expressions used in programming languages. This aim has been achieved by introduction of kind grammars and by the development of the principles of kind parsing applicable on kind grammars.

Kind grammars have like LL grammars [RS69] two variants – strong kind grammars and weak kind grammars. Every strong kind grammar is a weak kind grammar. The class of strong kind grammars is a proper superclass of strong LL grammars. The class of weak kind grammars is moreover a proper superclass of Ch grammars [NSS79] (and therefore also of LL grammars). It has been shown that the class of kind grammars is incomparable with the classes of LC grammars [RL70] and SLR grammars [DeR71].

One of the possible implementations of kind parsing is derived from lookahead pushdown automata. This type of automata allows easy extension. It – in combination with lookahead minimization – keeps the parser size of real-life (programming) languages reasonably small.

Kind parsers built on lookahead pushdown automata have the structure of recursive descent parsers. Parsers designed this way are very close to hand-written parsers. Kind parsers built as lookahead automata share with hand-written recursive descent parsers also such features like good under-

[1] A parser is *easily extensible* if its extensions enforce only local changes of the parser's structure and if it is possible to continue immediately after the parser extension without any necessity to restart the parsing.

[2] A parser is *usable in practice* if it is fast enough, compact (small), easily modifiable, and if it properly supports semantic analysis or other parts of applications using the parser (support of semantic actions located quite arbitrarily within the productions, the support of left recursion, support of groups of productions having the same nonterminal on the left-hand side and having common prefix on the right-hand side of the productions).

standability and opportunity to modify the code by hand. The ability of being simply modified is very important for later changes, as well as for possible tuning of the parser during its development (e.g. addition of error messages).

The usability of kind parsing has been tested by the implementation of the development tools (run-time extensible compiler, kind constructor, and kind transducer) and by investigation of their properties.

Kind parsing would be applicable in many areas:

- In compiler construction – kind parsing enables the development of fast and clearly structured parsers; it allows both creation of syntax tree and invocation of semantic actions on quite arbitrary place of a production (the only exception is the beginning of left-recursive productions where it is not allowed by any one-pass parser).
- In the construction of extensible development tools.
- In education – the construction of kind parsers is very simple.
- In conversion of existing documents into XML.
- In the implementation of front-end gates in service-oriented systems.
- Kind grammars specified languages can be used as a companion of XML dialects.

Bibliography

[ABB97] Jean-Michel Auterbert, Jean Berstel, and Luc Boasson. Context-free languages and pushdown automata. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, volume 1 – Word, Language, Grammar, chapter 3, pages 111–174. Springer, Berlin, 1997.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*, volume 10194 of *World Students Series Edition.* Addison-Wesley, 1986.

[AU72] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation and Compiling*, volume I.: Parsing. Prentice-Hall, Englewood Cliffs, N.J., 1972.

[AU73] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation and Compiling*, volume II.: Compiling. Prentice-Hall, Englewood Cliffs, N.J., 1973.

[BB00] Jean Berstel and Luc Boasson. Formal properties of xml grammars and languages. *The Computing Research Repository*, cs.DM/0011011, 2000.

[bison] `http://www.gnu.org/software/bison/bison.html`

[Bos96] Jan Bosch. Tool support for language extensibility. In Lars Bendix, Kurt Normark, and Kasper Osterbye, editors, *NWERP'96 Nordic Workshop on Programming Environment Research*, pages 3–17, Aalborg, Denmark, 1996.

[Bos97] Jan Bosch. Delegating compiler objects: Modularity and reusability in language engineering. *Nordic Journal of Computing*, 4:66–92, 1997.

[BS98] Jon Bentley and Bob Sedgewick. Ternary search trees. *Dr. Dobb's Journal*, April 1998.

[CMA94] Luca Cardelli, Florian Matthes, Martín Abadi: *Extensible Syntax with Lexical Scoping*, Dec. SRC Research Report 121, February 1994.

[ČBH93] Milan Češka, Miroslav Beneš, Tomáš Hruška: *Překladače.* (In Czech: Compilers), [Scriptum]. VUT Brno. 1993.

[Chy84] Michal Chytil: *Automaty a gramatiky* (in Czech: Automata and grammars). SNTL, Praha, 1984.

[Dem90] Jiří Demner. Mechanismy rozšiřitelného jazyka (in Czech: Mechanisms of an extensible language). Technical Report I-1-4/06, DÚ SPZV, Prague, Czech Republic, June 1990.

[DeR71] Frank DeRemer. Simple LR(k) grammars. *Commun. ACM*, 14(7):453–460, 1971.

[Dró85] Januš Drózd. Syntaktická analýza téměř shora dolů (in Czech: Semitopdown parsing). Master's thesis, Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic, 1985.

[Dró90] Januš Drózd. *Syntaktická analýza rekurzivním sestupem pro LR(k) gramatiky (in Czech: Recursive descent parsing for LR(k) grammars).* CSc. dissertation, Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic, 1990.

[Ear70] Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, 1970.

[Eng99] Dawson R. Engler. Interface compilation: Steps toward compiling program interfaces as languages. *IEEE Transactions on Software Engineering*, 25(3):387–400, May/June 1999.

[Erl05] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design.* Prentice Hall PTR, 2005.

[For02] Bryan Ford. Packrat parsing: simple, powerful, lazy, linear time, functional pearl. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 36–47, New York, NY, USA, 2002. ACM Press.

[Gan89] Mahadevan Ganapathi. Semantic predicates in parser generation. *Computer Language*, 14(1):25–33, 1989.

[Gri71] David Gries. *Compiler construction for digital computers.* John Wiley and Sons, 1971.

[Gri04] Robert Grimm. Practical packrat parsing. Technical Report TR2004-854, Dept. of Computer Science, New York University, 2004. Available at http://www.cs.nyu.edu/rgrimm/papers/tr2004-854.pdf.

[Gri06] Robert Grimm: *Better Extensibility through Modular Syntax.* To appear in *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation.* ACM Press, 2006.

[Har78] Michael A. Harrison. *Introduction to Formal Language Theory.* Addison-Wesley, Reading, Massachusetts, 1978.

[IdFF] Roberto Ierusalischy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. Lua – an extensible extension language.

[ISO86] International Organization for Standardization: *ISO 8879:1986 Information processing – Text and office systems – Standard Generalized Markup Language (SGML)*, 1986.

[Jan03] Jan Janeček. Rozšiřitelný lexikální analyzátor (in Czech: Extensible lexical analyzer). Master's thesis, Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic, 2003.

[JK75] Stanislaw Jarzabek and Tomasz Krawczyk. LL-regular grammars. *Information Processing Letters*, 4(2):31–37, November 1975.

[Joh77] Stephen C. Johnson. Yacc: Yet another compiler-compiler, 1977.

[KKR65] Herbert Kanner, P. Kosinski, and Charles L. Robinson. The structure of yet another algol compiler. *Commun. ACM*, 8(7):427–438, 1965.

[Knu65] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.

[Kol02] Donovan Michael Kolbly. *Extensible Language Implementation.* PhD dissertation, The University of Texas at Austin, December 2002.

[Krá82] Jaroslav Král. Syntaktická analýza a syntaxí řízený překlad (in Czech: Parsing and syntax driven translation). Technical report, ÚVT ČVUT, Prague, Czech Republic, 1982.

[KŽ00] Jaroslav Král and Michal Žemlička. Autonomous components. In Václav Hlaváč, Keith G. Jeffery, and Jiří Wiedermann, editors, *SOFSEM'2000: Theory and Practice of Informatics*, volume 1963 of *Lecture Notes in Computer Science*, pages 375–383, Berlin, 2000. Springer.

[KŽ01] Jaroslav Král and Michal Žemlička. Electronic government and software confederations. In A M. Tjoa and R. R. Wagner, editors, *Twelfth International Workshop on Database and Experts System Application*, pages 125–130, Los Alamitos, CA, USA, 2001. IEEE Computer Society. ISBN 0-7695-1230-5, ISSN 1529-4188.

[KŽ02] Jaroslav Král and Michal Žemlička. Systemic issues of software confederations. In R. Trappl, editor, *Cybernetics and Systems 2002*, volume 1, pages 141–146, Vienna, Austria, 2002. Austrian Society for Cybernetic Studies.

[KŽ03a] Jaroslav Král and Michal Žemlička. Semi-top-down syntax analysis. In Carlos Martín-Vide and Victor Mitrana, editors, *Grammars and Automata for String Processing*, volume 9 of *Topics in Computer Mathematics*, pages 77–90. Taylor and Francis, 2003.

[KŽ03b] Jaroslav Král and Michal Žemlička. Software confederations – an architecture for global systems and global management. In Sherif Kamel, editor, *Managing Globally with Information Technology*, pages 57–81, Hershey, PA, USA, 2003. Idea Group Publishing.

[KŽ04] Jaroslav Král and Michal Žemlička. Architecture and modeling of service-oriented systems. In Peter Vojtáš, Mária Bieliková, Bernadette Charon-Bost, and Ondrej Sýkora, editors, *SOFSEM 2005 Communications*, pages 71–80, Bratislava, Slovakia, 2004. Slovak Society for Computer Science.

[Lea66] B. M. Leavenworth. Syntax macros and extended translation. *Communications of the ACM*, 9(11):790–793, November 1966.

[Med00] Alexander Meduna: *Automata and Languages: Theory and Applications*. Springer, London, 2000.

[MČJR99] Bořivoj Melichar, Milan Češka, Karel Ježek, Karel Richta: *Konstrukce překladačů* (in Czech: Compiler construction). ČVUT, 1999. ISBN 80-01-02028-2.

[Moo56] E. F. Moore. Gedanken experiments on sequential machines. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, Princeton, 1956. Princeton University Press.

[Nij82] Anton Nijholt. On the relationship between LL(k) and LR(k) grammars. *Information Processing Letters*, 15:97–101, October 1982.

[Nij83] Anton Nijholt. *Deterministic Top-Down and Bottom-Up Parsing: Historical Notes and Bibliographies.* Mathematisch Centrum, Amsterdam, 1983.

[NL94] Sven Naumann and Hagen Langer. *Parsing: Eine Einführung in die maschinelle Analyse natürlicher Sprache (in German: An introduction into machine analysis of natural language).* B. G. Teubner, Stuttgart, Germany, 1994.

[NSS79] Anton Nijholt and Eljas Soisalon-Soininen. Ch(k) grammars: A characterization of LL(k) languages. In Jiří Becvář, editor, *MFCS*, volume 74 of *Lecture Notes in Computer Science*, pages 390–397. Springer, 1979.

[Par93] Terrence John Parr. *Obtaining Practical Variants of LL(k) and LR(k) for $k > 1$ by Splitting the Atomic k-Tuple.* PhD dissertation, Purdue University, August 1993.

[Plá92] Martin Plátek. Syntactic error recovery with formal guarantees I. Technical Report 100, Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic, April 1992.

[Plá98] Martin Plátek. Testovatelné podmínky pro bezpečnou skeletální množinu a jejich zobecnění (in Czech: Testable conditions for safe skeletal set and their generalization). In *Sborník krátkých referátů SOFSEM'88*, Bratislava, Slovakia, 1998. ÚVT UJEP and JČMF.

[PQ94] Terrence John Parr and Russell W. Quong. Adding semantic and syntactic predicates to $LL(k)$: $pred - LL(k)$. In Peter Fritzson, editor, *Compiler Construction 1994*, volume 786, pages 263–277. Springer, 1994.

[PQ95] Terence John Parr and Russell W. Quong. Antlr: A predicated-*ll(k)* parser generator. *Softw., Pract. Exper.*, 25(7):789–810, 1995.

[PQ96] Terrence John Parr and Russell W. Quong. LL and LR translators need $k > 1$ lookahead. *ACM SIGPLAN Notices*, 31(2):27–34, February 1996.

[RL70] Daniel J. Rosenkrantz and Philip M. Lewis II. Deterministic left corner parsing. In *Conference Record of 1970 Eleventh Annual Symposium on Switching and Automata Theory*, pages 139–152. IEEE, 1970.

[RM85] Peter Rechenberg and Hanspeter Mössenböck. *Ein Compiler-Generator für Mikrocomputer (in German: Compiler generator for microcomputers).* Hanser-Verlag, 1985.

[RS69] Daniel J. Rosenkrantz and Richard Edwin Stearns. Properties of deterministic top down grammars. In *STOC '69: Proceedings of the first annual ACM symposium on Theory of computing*, pages 165–180, New York, NY, USA, 1969. ACM Press.

[Ruß97] Arnd Rußmann. Dynamic LL(k) parsing. *Acta Informatica*, 34:267–289, April 1997.

[Ruß98] Arnd Rußmann. *Dynamische Gramatiken (in German: Dynamic grammars).* PhD dissertation, Johann Wolfgang Goethe-Universität, Frankfurt am Main, 1998.

[Sik93] Nicolaas Sikkel. *Parsing Schemata*, Proefschrift Enschede, 1993.

[SSS88] Seppo Sippu and Eljas Soisalon-Soininen. *Parsing Theory. Volume I: Languages and Parsing*, volume 15 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, Berlin, 1988.

[SSS90] Seppo Sippu and Eljas Soisalon-Soininen. *Parsing Theory. Volume II: LR(k) and LL(k) Parsing*, volume 20 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, Berlin, 1990.

[SSU70] Eljas Soisalon-Soininen and Esko Ukkonen. A method for transforming grammars into $LL(k)$ form. *Acta Informatica*, 12:339–369, 1970.

[Svo02] Petr Svoboda. Implementace zobecněného přívětivého analyzátoru (in Czech: Implementation of generalized kind parser). Master's thesis, Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic, January 2002.

[TN91] Masaru Tomita and See-Kiong Ng. The generalized LR parsing algorithm. In Masaru Tomita, editor, *Generalized LR Parsing*, Norwell, Massachusetts, USA, 1991. Kluwer Academic Publishers.

[Töp89] Pavel Töpfer. An error recovery method for top-down syntactical analysis. Technical Report 47, Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic, February 1989.

[TS97] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 203–217, New York, NY, USA, 1997. ACM Press.

[Tur39] Alan M. Turing: Systems of logic based on ordinals. *Proceedings of the London Mathematical Society*, ser. 2, vol. 45, pp. 161–228. (reprinted in Martin Davis (Ed.): The undecidable. Raven Press, Hewlett, New York, 1965.

[W3 99a] XML, 1999, `http://www.xml.com/xml/pub/` Extensible Markup Language. A proposal of W3C consortium.

[W3 99b] W3 Consortium: XSL Transformations (XSLT), 1999. http://www.w3.org/TR/xslt.

[Weg70] Ben Wegbreit, *Studies in Extensible Programming Languages*, ESD-TR-70-297, Harvard University, Cambridge, Massachusetts, May 1970. Reprinted as: Ben Wegbreit, *Studies in Extensible Programming Languages* [Outstanding Dissertations in the Computer Sciences], New York: Garland Publishing, Inc., 1980.

[WM95] Reinhard Wilhelm, Dieter Maurer: *Compiler Design.* Addison-Wesley, 1995.

[Yan95] Wuu Yang. On the look-ahead problem in lexical analysis. *Acta Informatica*, 32(5):459–476, 1995.

[Yan96] Wuu Yang. Mealy machines are a better model of lexical analyzers. *Comput. Lang.*, 22(1):27–38, 1996.

[Žem94] Michal Žemlička. Překladač rozšiřitelného jazyka (in Czech: Compiler for an extensible language). Master's thesis, Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic, May 1994.

[ŽBB+98] Michal Žemlička, Václav Brůha, Milan Brunclík, Petr Crha, Jan Cuřín, Svatopluk Dědic, Luděk Marek, Roman Ondruška, and Daniel Průša. Projekt Algoritmy (in Czech: The Algorithms project, 1998. Software project.

[Žem95] Michal Žemlička. Extensible LL(1) parser. Presented at SOFSEM'95 conference., 1995. http://kocour.ms.mff.cuni.cz/~zemlicka/poster.ps.

[Žem96] Michal Žemlička. Syntaktická analýza rozšiřitelných jazyků (in Czech: Parsing of extensible languages). Technical report, Charles University, Faculty of Mathematics and Physics, Department of Software Engineering, Prague, Czech Republic, December 1996.

[Žem02a] Michal Žemlička. Kindcons – kind constructor, 2002. http://www.ms.mff.cuni.cz/~zemlicka/KindCons/.

[Žem02b] Michal Žemlička. Kindtran – kind transducer, June 2002. http://www.ms.mff.cuni.cz/~zemlicka/KindTran/.

[ŽK99a] Michal Žemlička and Jaroslav Král. Run-time extensible (semi-)top-down parser. In Václav Matoušek, Pavel Mautner, Jana Ocelíková, and Petr Sojka, editors, *Proceedings of the 2nd International Workshop on Text, Speech and Dialogue (TSD-99)*, volume 1692 of *LNAI*, pages 121–126, Berlin, September 13–17 1999. Springer.

[ŽK99b] Michal Žemlička and Jaroslav Král. Run-time extensible deterministic top-down parsing. *Grammars*, 2(3):283–293, 1999.

[ŽK00] Michal Žemlička and Jaroslav Král. Semitopdown parsing in information systems. In *Proceedings of SCI/ISAS 2000 Conference*, Orlando, Florida, USA, 2000.

[ZO01] Matthias Zenger and Martin Odersky: *Implementing Extensible Languages*. Presented at ECOOP Workshop on Multiparadigm Programming with Object-Oriented Languages, Budapest, Hungary, June 2001.

List of Figures

List of Tables

Index