Charles University

Faculty of Mathematics and Physics

Principles of Kind Parsing – An Introduction

Technical report

Mgr. Michal Žemlička

Department of Software Engineering
Malostranské náměstí 25
Prague, Czech Republic

December 2002

This work has been partially supported by the Grant Agency of Czech Republic, grant No. 201/02/1456.

Abstract:
New kind of (semi)top-down parsing is presented in this work. It specifies a hierarchy of grammar classes that is a proper supercalss of SLL(k) grammars 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 parses 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: is easy to learn, parsers are easy to read and to modify, semantic actions can be performed almost everywhere inside phrases, parsers can be parse-time extensible (what was domain of tagged languages like XML).

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

Chapter 1

Introduction

This work describes new classes of grammars and parsers in two different views: the first is the usual formal view and the second 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 need. For example, huge information systems cannot be implemented without software confederation architecture [KŽ2000]. This architecture expects that services are interconnected by *configurable* components.

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

There exist extensible languages and tools for a long time. Most popular are probably SGML [ISO86] and newly XML [XML]. 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 (be discussed here).

These techniques may be (and often are) combined.

The idea of an extensible language/parser is not new – an example of an 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. 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.

Where 'f' and 'F' are nonterminals. (They belong, together with 'E' and 'T' to the ones that define arithmetic expressions. They are ordered E,T,F,f.) 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 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','loop';
terminal 'endloop','endloop';
terminal 'on','on';
terminal 'exit','exit';
rule '[loop1]::=loop1:(loop)';
rule '[loop2]::=loop2:[loop1][cmds](on)<cond>(exit)';
rule '[cmd]::=loop3:[loop2][cmds](endloop)';
```

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

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.

Chapter 2

Notation and Known Results

Terminology and notation are usually taken from [AU72, Chy84]. Some basic notation and definitions are repeated here:

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
$x \in M$	(item) x is an element of (set) M
$A \subset B$	(set) A is a subset of (set) B
$A \cup B$	union of (sets) A and B
$A \cap B$	intersection of (sets) A and B
$\emptyset$	empty set
$\mathcal{P}(A)$	power set of A (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 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 (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

$$(N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$$

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

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

Definition 2.3 (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 $\beta \rightarrow \delta$ 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 2.4 (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 2.5 *(Taken from [AU72] page 86)*:
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 $\beta \rightarrow \delta$ 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 $S \Rightarrow^* x$.

Definition 2.6 (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 $\alpha \in (N \cup \Sigma)^*$.

Definition 2.7 *(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 2.8 *(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 (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 $Q \times \Sigma \to \mathcal{P}(Q)$ 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 2.10 (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 2.11 (First):
(Taken from [AU72] page 300.)

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

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

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

Definition 2.12 (Follow):
(Taken from [AU72] page 343.)
Let $G = (N, \Sigma, P, S)$ be a CFG. We define $\text{Follow}_k^G(\beta)$, 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 $\text{First}_k^G(\gamma)\}$. We shall omit k and G whenever they are understood.

Definition 2.13 (LL(k) grammar):
(Taken from [AU72] page 336.)
Let $G = (N, \Sigma, P, S)$ 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 $\text{First}_k(x) = \text{First}_k(y)$, it follows that $\beta = \gamma$.

Definition 2.14 (Strong LL(k) grammar):
(Taken from [AU72] page 343.)
Let $G = (N, \Sigma, P, S)$ be a CFG such that holds: If $A \to \beta$ and $A \to \gamma$ are distinct A-productions, then $\text{First}_k(\beta\ \text{Follow}_k(A)) \cap \text{First}_k(\gamma\ \text{Follow}_k(A)) = \emptyset$. Such grammar is called *strong* $LL(k)$ grammar.

Note 2.15 Every LL(1) grammar is a strong LL(1) grammar. For $k > 1$ there are LL(k) grammars that are not strong LL(k) grammars.

Definition 2.16 (Pushdown automaton, configuration, and move):
(Taken from [AU72] pages 167–168; slightly modified.)
A *pushdown automaton* (PDA for short) is a 7-tuple

$$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F),$$

where

1. Q is a finite state of *state* symbols representing the possible states of the finite state control,
2. Σ is a finite *input alphabet*,
3. Γ is a finite alphabet of *pushdown list symbols*,
4. δ is a mapping $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$ to the finite subsets of $Q \times \Gamma^*$,
5. $q_0 \in Q$ is the *initial state* of the finite control,
6. $Z_0 \in \Gamma$ is the symbol that appears initially on the pushdown list (the *start symbol*), and
7. $F \subseteq Q$ is the set of *final states*.

A *configuration* of P is a triple $(q, w, \alpha) \in Q \times \Sigma^* \times \Gamma^*$, where

1. q represents the current state of the finite control,

2. w represents the unused portion of the input. The first symbol of w is under the input head. $w = \varepsilon$ indicates that all of the input tape has been read.

3. α represents the contents of the pushdown list. The leftmost of α is the topmost pushdown symbol. $\alpha = \varepsilon$ indicates that the pushdown list is empty.

A *move* by P will be represented by the binary relation $\vdash_P$ (or $\vdash$ whenever P is understood) on configurations. We write

$$(q, aw, Z\alpha) \vdash (q', w, \gamma\alpha) \tag{2.1}$$

if $\delta(q, a, Z)$ contains (q', γ) for any $q \in Q$, $a \in \Sigma \cup \{\varepsilon\}$, $w \in \Sigma^*$, $Z \in \Gamma$, and $\alpha \in \Gamma^*$.

If $a \neq \varepsilon$, Eq. (2.1) states that if P is in a configuration such that the finite control is in state q, the current input symbol is a, and the symbol on the top of pushdown list is Z, then P may go into a configuration in which the finite control is now in the state q', the input head has been shifted one symbol to the right, and the topmost symbol on the pushdown list has been *popped*.

If $a = \varepsilon$, then the move is called an *ε-move*. In an ε-move the current input symbol is not taken into consideration, and the input head is not moved. However, the state of the finite control can be changed, and the contents of the memory can be adjusted. Note that an ε-move can occur even if all of the input has been read.

No move is possible if the pushdown is empty.

We can define the relations $\vdash^i$, for $i \geq 0$, $\vdash^*$, and $\vdash^+$ in the customary fashion. Thus, $\vdash^*$ and $\vdash^+$ are, respectively, the reflexive-transitive and transitive closures of $\vdash$.

An *initial configuration* of P is one of the form (q_0, w, Z_0) for some $w \in \Sigma^*$. That is, the finite state control in the initial state, the input contains the string to be recognized, and the pushdown list contains only the symbol Z_0. A *final configuration* is one of the form (q, ε, α), where $q \in F$ and $\alpha \in \Gamma^*$.

We say that a string w is *accepted* by P if $(q_0, w, Z_0) \vdash^* (q, \varepsilon, \alpha)$ for some $q \in F$ and $\alpha \in \Gamma^*$. The *language defined by P*, denoted $L(P)$, is the set of strings accepted by P. $L(P)$ will be called a *pushdown automaton language*.

Definition 2.17 (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 2.18 (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 $\alpha A w = \gamma B x$. (That is, $\alpha = \gamma$, $A = B$, and $x = y$.)

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

Definition 2.19 (SLR(k) grammar):
(Taken from [SSS90] – adapted.)

Grammar $G = (\mathrm{N}, \Sigma, P, S)$ is *SLR(k)* if its parser 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.

Theorem 2.20 (Characterization of SLR(k) grammars):
(Taken from [SSS90] .)

The SLR(k) parser of grammar G is deterministic if and only if the following two statements are true for all states q in the LR(0) machine for the \$-augmented grammar for G.

1. Whenever q contains a pair of distinct items $[A_1 \to \omega_1 \cdot]$, $[A_2 \to \omega_2 \cdot]$, then
$$Follow_k(A_1) \cap Follow_k(A_2) = \emptyset.$$
2. Whenever q contains a pair of items $[A \to \alpha \cdot a\beta]$, $[B \to \omega \cdot]$, where a is a terminal, then
$$First_k(a\beta Follow_k(A)) \cap Follow_k(B) = \emptyset.$$

Note 2.21 The construction of an SLR(k) parser and the proof of Th. 2.20 are available e.g. in [SSS90].

Definition 2.22

concatenation of two strings u, v – uv can be for better readability also written as $u \cdot v$. Similarly $M \cdot v =_{df} \{uv | u \in M\}$, $u \cdot M =_{df} \{uv | v \in M\}$, and $M \cdot N =_{df} \{uv | u \in M v \in N\}$.

Definition 2.23 (Regular right part grammar):
(Taken from [WiMa95] page 317; synchronized notation.)

A *regular right part grammar* (rrpg) is a tuple $G = (\mathrm{N}, \Sigma, P, S)$, where N, Σ and S are defined as usual and $p : \mathrm{N} \to RE$ is a mapping of the nonterminals into the set RE of regular expressions over $\mathrm{N} \cup \Sigma$.

Definition 2.24 (Regular derivation):
(Taken from [WiMa95] page 317.)

Suppose G is an rrpg. The relation $\Longrightarrow_{R,lm}$ on RE, 'derives regurarly diretcly leftmost' is defined by:

$$\begin{array}{llll}
(a) & w\ X\ \beta & \Longrightarrow_{R,lm} & w\ \alpha\ \beta \quad \text{with } \alpha = P(X) \\
(b) & w(r_1|\ldots|r_n)\beta & \Longrightarrow_{R,lm} & w r_i \beta \quad \text{for } 1 \le i \le n \\
(c) & w(r)^*\beta & \Longrightarrow_{R,lm} & w\beta \\
(d) & w(r)^*\beta & \Longrightarrow_{R,lm} & wr(r)^*\beta
\end{array}$$

Let $\Longrightarrow^{*}_{R,lm}$ be the reflexive, transitive closure of $\Longrightarrow_{R,lm}$. The language defined by G is then $L(G) = \{w \in \Sigma^* | S \Longrightarrow^{*}_{R,lm} w\}$.

Definition 2.25 (Regular subexpression):
(Taken from [WiMa95] page 318.)

r_i is a *direct regular subexpression* of $(r_1|\ldots|r_n)$ and $(r_1\ldots r_n), 1 \leq i \leq n$; r is a *direct regular subexpression* of $(r)^*$ and of r itself; r_1 is a *regular subexpression* of r_2 if $r_1 = r_2$ or if r_1 is a direct regular subexpression of r_2 or a regular subexpression of a direct regular subexpression of r_2

Definition 2.26 (rrpg item):
(Taken from [WiMa95] page 318; synchronized notation.)

A tuple $(X, \alpha, \beta, \gamma)$ is an *rrpg item* of an rrpg $G = (\mathrm{N}, \Sigma, P, S)$, if $X \in \mathrm{N}$, $\alpha, \beta, \gamma \in (\mathrm{N} \cup \Sigma \cup \{(,),^*,|,\varepsilon\})^*$, $P(X) = \beta\alpha\gamma$ and α is a regular subexpression of $\beta\alpha\gamma$. This item is written as $[X \to \beta.\alpha\gamma]$.

Definition 2.27 (LLR(k) grammar):
(Taken from [SSS90] page 110, synchronized notation.)

Call a grammar $G = (\mathrm{N}, \Sigma, P, S)$ an *LLR(k) grammar* if the following statements hold.

1. G is unambiguous.

2. The conditions

   $S \Rightarrow^* xAy_1 \Rightarrow^* xwy_1$,
   $S \Rightarrow^* xwy_2$, and
   $First_k(y_1) = First_k(y_2)$

   allways imply

   $S \Rightarrow^* xAy_2$.

Definition 2.28 (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 *predcessor* of y and y is an *successor* of x (according to the edge (x, y)).

Definition 2.29 (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 2.30 (cycle):

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

Definition 2.31 (tree, root, leaf):

Oriented graph where exist node from where there are paths to all other nodes is called *(split) tree*. The unique node is called *root*. The tree nodes with no successors are called *leaves*.

Definition 2.32 *(Taken from [AU72] page 363)*:

Let G be a CFG. We say that $S \Rightarrow^*_{\mathrm{lc}} wA\delta$ if $S \Rightarrow^*_{\mathrm{l}m} wA\delta$ 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 $S \Rightarrow^*_{\mathrm{l}m} wA\alpha$.

Definition 2.33 (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 $S \Rightarrow_{\mathrm{lc}}^{*} wA\delta$. Then for each lookahead string u and derivation $A \Rightarrow^{*} B\gamma$, there is at most one production $B \rightarrow \alpha$ 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 2.34 (PLR(k) grammar):
(Taken from [SSU79] page 347; synchronized notation.)

Let k be a non-negative integer. A grammar $G = (\mathrm{N}, \Sigma, P, S)$ is said to be $PLR(k)$, if G is $LR(k)$ and in the augmented grammar G' for each production $A \rightarrow X\beta$, $X\beta \neq \varepsilon$, the conditions

$$S' \Rightarrow_{rm}^{*} \alpha A z_1 \Rightarrow_{rm} \underline{\alpha X}\beta z_1 \Rightarrow_{rm}^{*} \underline{\alpha X} y_1 z_1,$$

$$S' \Rightarrow_{rm}^{*} \alpha' B z_2 \Rightarrow_{rm} \underline{\alpha'\alpha'' X}\gamma z_2 \Rightarrow_{rm}^{*} \underline{\alpha'\alpha'' X} y_2 z_2,$$

$$\alpha'\alpha'' = \alpha \text{ and } First_k(y_1 z_1) = First_k(y_2 z_2)$$

always imply that $\alpha A = \alpha' B$.

A language $L \subseteq T^{*}$ is said to be $PLR(k)$ if there is a $PLR(k)$ grammar such that $L = L(G)$.

Chapter 3

Principles of Kind Parsing

3.1 Introduction

The tour around 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 (Grammar of a simple arithmetic expression):

$$
\begin{aligned}
&\mathrm{N} = \{S, E, T, F\}\\
&\Sigma = \{id, num, (,), +, *\}\\
&P = \left\{\begin{array}{l} S \rightarrow E, \\ E \rightarrow E + T, E \rightarrow T, \\ T \rightarrow T * F, T \rightarrow F, \\ F \rightarrow id, F \rightarrow num, F \rightarrow (E) \end{array}\right\}\\
&S = S
\end{aligned}
$$

□

The grammar is directly left recursive due to productions $E \rightarrow E + T$, and $T \rightarrow T * F$, so it cannot be LL(k) for any k (compare Def. 2.13. 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 recursivity. It is a rather 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 factorization. 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 grammar.

Here is presented how to create for a given grammar a structure that can afterwards be used as a background for a set of recursively called procedures, a pushdown automaton, for a deductive system, or for a direct generation or parsing of a text.

3.2 Parser structure

Any set of productions can be viewed as a *forest of production trees* that can be described by a rather cumbersome definition. Focusing to 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 recursivity (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 contain only the productions with the same

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 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.

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: Short description of productions divided by defined nonterminals and by presence of left recursion

The same can be written also in another style. A path through the production tree can be assigned to each production. The edges along the path are labelled by the symbols from the productions. For left recursive production the leading edge labelled with 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).

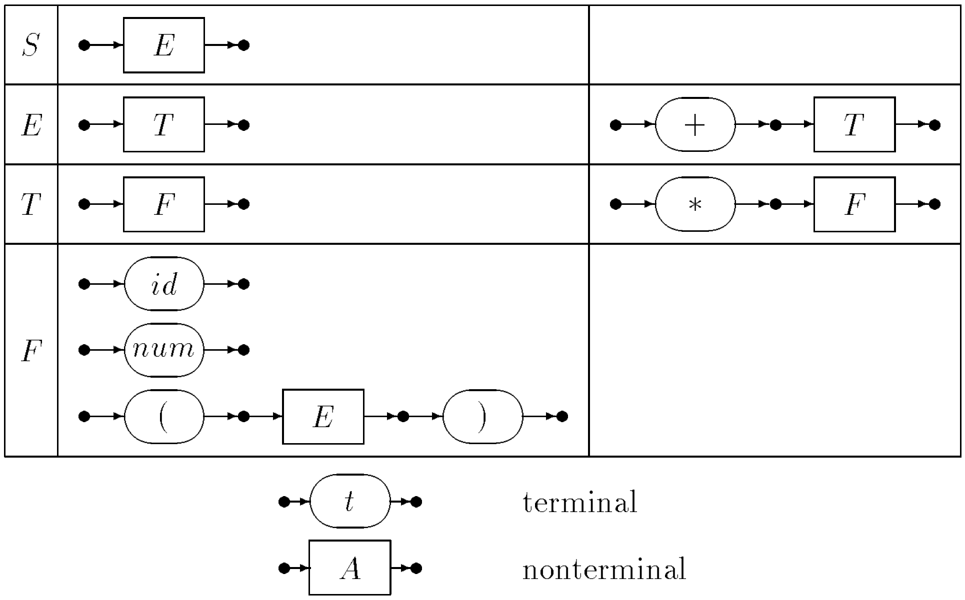

Figure 3.3: Productions written as paths in a directed acyclic graph

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 labelled with full productions but only with 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 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 $\rightarrow$ *begin* cmds *end*
cmd $\rightarrow$ *id* := E
cmd $\rightarrow$ *id* (E)
cmd $\rightarrow$ *read id*
cmd $\rightarrow$ *write* E
cmd $\rightarrow$ *write str*
cmd $\rightarrow$ *while* cond *do* cmd

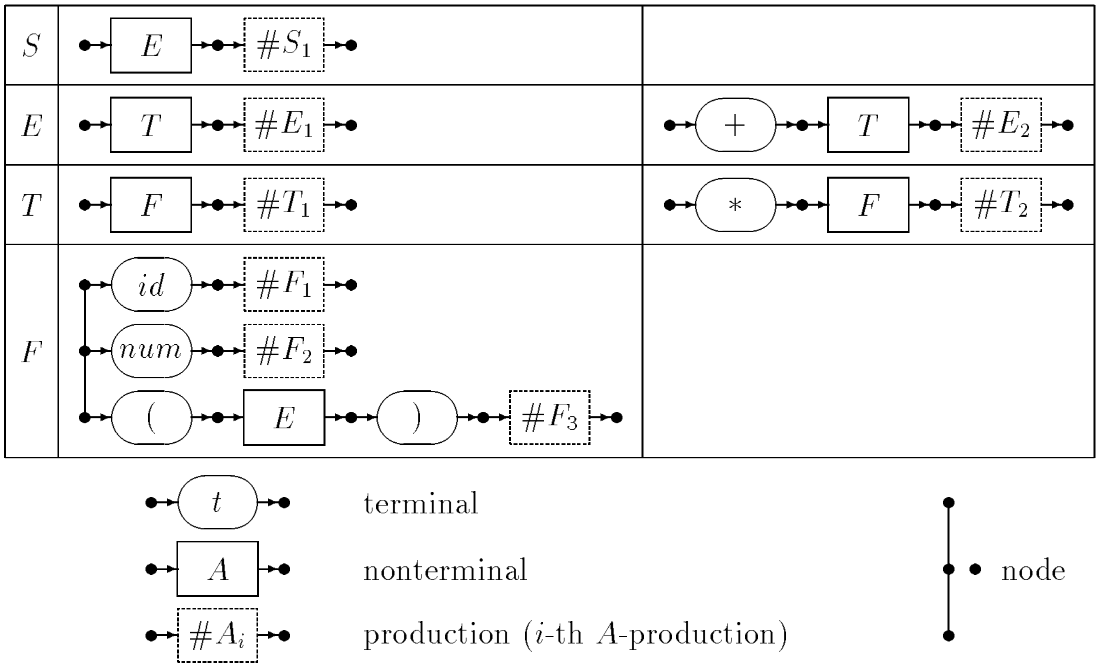

Figure 3.4: Productions written as paths in production trees

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

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" 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 all the automaton again and again and show every time the change of 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 make 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

[1]Dotted productions are described e.g. in [Nau94] or [AU72].

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

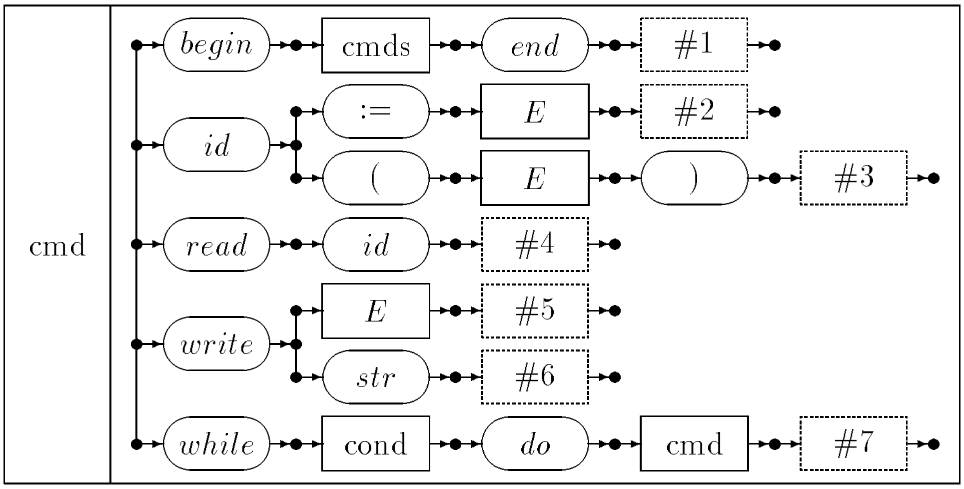

Figure 3.5: Simple command production tree

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) 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.

So a 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 representants 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		
$.3+2$	$[F \to .num]$	$[S \to E.]$ $[E \to T.]$ $[T \to F.]$
generation/reading of '3'		
$3.+2$	$[F \to num.]$	$[S \to E.]$ $[E \to T.]$ $[T \to F.]$
finishing of F (reduction by $\#F \to num$)		

input/output	state	pushdown
$3. + 2$	$[T \to F.]$	$[S \to E.]$ $[E \to T.]$
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'		
$3 + 2.$	$[F \to num.]$	$[S \to E.]$ $[E \to E + T.]$ $[T \to F.]$
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.		

For the needs of parsing it can be useful to have some hints which computational (parsing) path leads to the result if there are more ones. 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 to select proper edge along which the parsing should go). 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.

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:

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

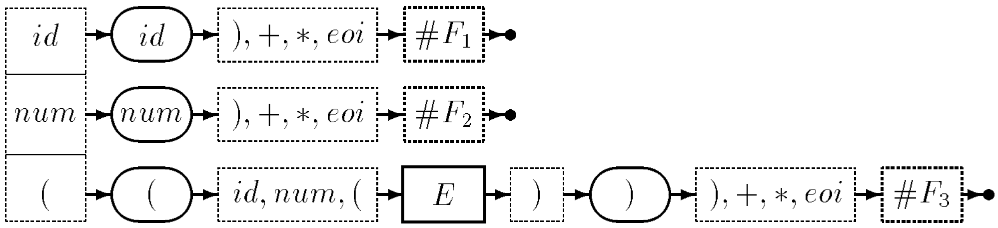

Figure 3.6: Nonterminal F with precomputed lookaheads

Chapter 4

Changes in a Parsing Structure

We control/implement the extensions via changes of production tree forests. Processing of extensible languages needs extensible/modifiable parsers. Let us show the ability of forests of production trees (and kind parsers that are 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 to extend parsers also in 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.

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 finishing of the parse is removed. By restriction some problems arise: it is possible, that if there are no restrictions 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

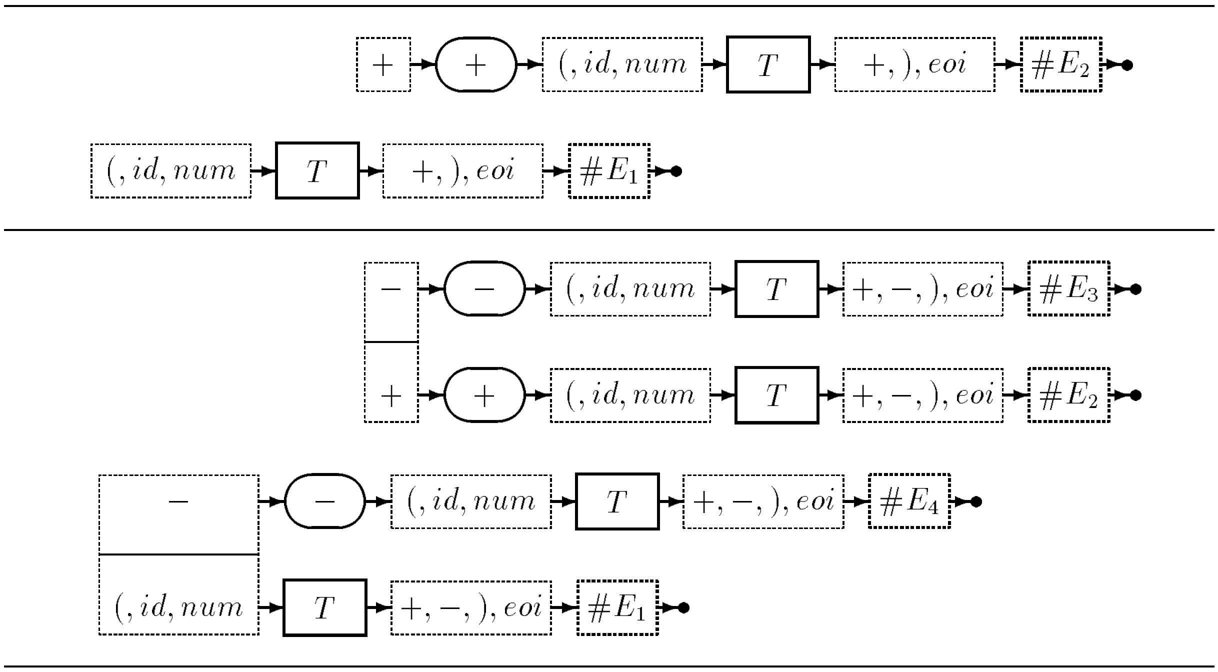

Figure 4.1: Original and extended production tree of the nonterminal E of a simple arithmetic expression (In both cases the tree in the bottom and left represents the structure for non-recursive productions, the tree on top and right represents the left recursive productions.)

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

Each production of an arbitrary reduced context-free grammar can be assigned to exactly one of following five groups:

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 tend 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]):

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

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

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

[1] DLRF = directly left recursive productions
[2] NLRF = non-left-recursive productions

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 is clear which grammar G is mentioned.

5.2 Kind grammars

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

Definition 5.7 (Follow without left recursion):
A set of terminal strings of the length k that may follow after given nonterminal A except its left recursion is called *follow of A without (its) left recursion (NLRF(A) – non-left-recursive follow)* if they can be derived immediately after given nonterminal everywhere except its left recursion. Formally:

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

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

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

Definition 5.8 (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).*

$$DLRF_G^k(A) = \{a_1 \dots a_m \in First_G^k(\alpha \cdot Follow_G^k(A)) \mid A \to A\alpha \in P_G, \wedge \alpha \neq \varepsilon\}$$

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 5.9 (k-kind grammar):

A context-free grammar G having only productions without left recursion or with direct left recursion is called *k-kind* if
$\forall A \in \mathrm{N},\ \forall \alpha, \beta, \gamma \in (\mathrm{N} \cup \Sigma)^*,\ X, Y \in \mathrm{N} \cup \Sigma$:

1. $\forall$ A$\to A\alpha$Xβ, A$\to A\alpha$Y$\gamma \in$ DLRP $\vee$ A$\to \alpha$Xβ, A$\to \alpha$Y$\gamma \in$ NLRP:
   X$\neq$Y $\Rightarrow First_k(\mathrm{X}\beta\cdot\mathrm{Follow}_k(\mathrm{A})) \cap First_k(\mathrm{Y}\gamma\cdot\mathrm{Follow}_k(\mathrm{A})) = \emptyset$
   and

2. $DLRF_k(\mathrm{A}) \cap NLRF_k(\mathrm{A}) = \emptyset$

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

Definition 5.10 (Kind grammar):

A context-free grammar G is called *kind* if it is k-kind for some $k > 0$. 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 language. Similar situations in real life can be described e.g. by a Czech word "přívětivý".

Example 5.11 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 were fulfilled for grammar G and for $k = 1$, therefore G is a 1-kind grammar. $\square$

Chapter 6

Power of Kind Grammars

6.1 Lower bound

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

Proof: Comparing definitions of strong LL(k) grammars and k-kind grammars shows that the strong LL(k) grammars are special case of k-kind grammars. If it is in the definition of k-kind grammars set $\alpha = \varepsilon$, and substituted β and γ from the definition of strong LL(k) grammars for Xβ and Yγ respectively, it is exactly what is stated in the definition of strong LL(k) grammars.
More formally:

By definition for any two distinct A-productions of a strong LL(k) grammar $A \to \beta$ and $A \to \gamma$ holds that $\mathrm{First}_k(\beta\ \mathrm{Follow}_k(A)) \cap \mathrm{First}_k(\gamma\ \mathrm{Follow}_k(A)) = \emptyset$ (see Def. 2.14).

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

Now we need to test, whether $\forall$ A$\to$ αXβ, A$\to$ αYγ $\in$ DLRP $\vee$ A$\to$ αXβ, A$\to$ αYγ $\in$ NLRP implies: X$\neq$Y $\Rightarrow$ $First_k(\mathrm{X}\beta\cdot\mathrm{Follow}_k(\mathrm{A})) \cap First_k(\mathrm{Y}\gamma\cdot\mathrm{Follow}_k(\mathrm{A})) = \emptyset$. Both productions are from $NLRP(A)$ ($DLRP(A)$ is empty as there are no left recursive productions in strong LL(k) grammars). It holds that $\mathrm{First}_k(\beta\ \mathrm{Follow}_k(A)) \cap \mathrm{First}_k(\gamma\ \mathrm{Follow}_k(A)) = \emptyset$, so the kind condition must be fulfilled. □

Theorem 6.2 The class of k-kind grammars is a proper superclass of SLL(k) grammars.

Proof: The class of k-kind grammars is a superclass of SLL(k) grammars – it was shown in Lemma 6.1. The grammar from Ex. 1.1 is k-kind and is not SLL(k) grammar because there are left recursive productions in it what is not allowed in SLL(k) grammars. □

6.2 Incomparability

Theorem 6.3 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

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

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 therefore are trivially distinguishable – i.e. fulfill the LL(2) condition.

LL(2) condition for the nonterminal A is also fulfilled:
$First_2(100) \cap First_2(\varepsilon 00) = \emptyset$ and $First_2(110) \cap First_2(\varepsilon 10) = \emptyset$.

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

Note 6.4 For $k = 1$ the classes of LL(k) and SLL(k) grammars are equal, therefore the class of LL(1) grammars is a proper subclass of 1-kind grammars.

Note 6.5 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.

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

Lemma 6.7 $\exists G\ \forall k \geq 1 :\ G \in KG(1) \wedge G \notin LC(k)$.

Proof of lemma 6.7: $G = (\{S, A\}, \{a, b, c\}, \{S \to bAb, S \to bAc, A \to aA, A \to \varepsilon\}, S)$ fulfills the above condition:

1. G is k-kind, as the kind condition is fulfilled ($First_1(b) \cap First_1(c) = \emptyset$, $First_1(a) \cap First_1(\varepsilon \cdot NLRF_1(A)) = \emptyset$).

2. G is not LC(k) for any k, as A can generate arbitrary long string of a's and therefore it is not decidable (using the lookahead of k) whether the production $S \to bAb$ or $S \to bAc$ should be used.

□

Proof of theorem 6.6: From lemma 6.7 $\forall k \geq 1 :\ KG(k) \not\subset LC(k)$. From theorem 6.3 $LL(k) \not\subset KG(k)$. As $\forall k > 0 :\ LL(k) \subset LC(k)$ (see [RL70]) it must hold that $LC(k) \not\subset KG(k)$. □

6.3 Expressive power

What languages are generated by 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.

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 only in the root.

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.

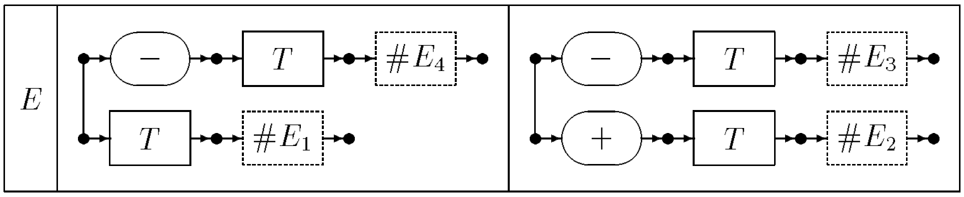

Figure 6.1: Nonterminal E of a simple arithmetic expression

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 labelled with production identification – Fig. 6.2.

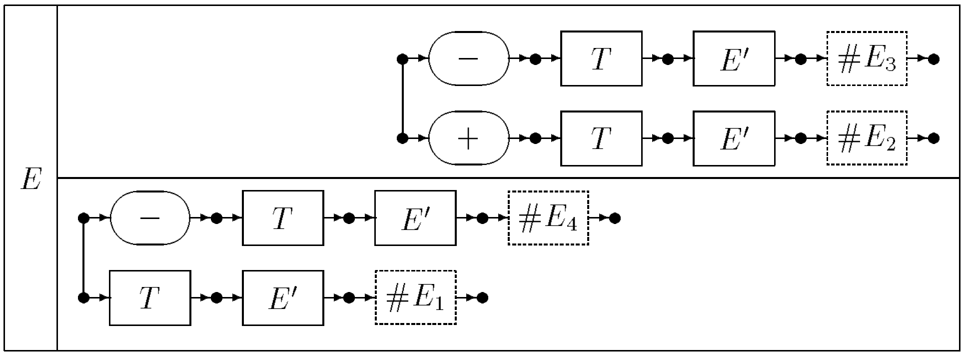

Figure 6.2: Nonterminal E of a simple arithmetic expression after addition (insertion) of a new nonterminal E' at the ends of the productions

Now the production tree of left recursive E-productions can be separated and transformed (relabelled) 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 $E' \rightarrow \varepsilon$ 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 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. The principle is that all productions with common prefixes are substituted by groups of productions consisting of one representing the common prefix and several ones created from

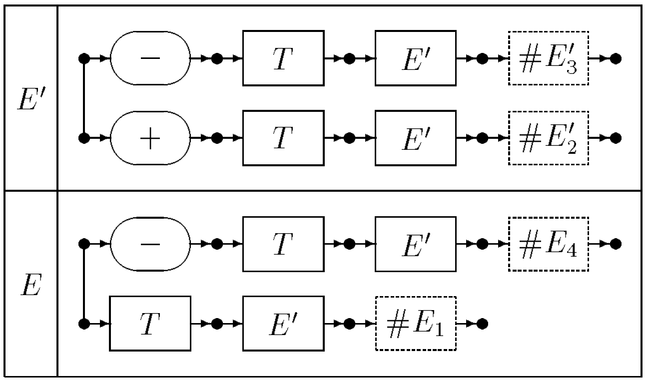

Figure 6.3: Nonterminal E of a simple arithmetic expression and a germ of a new nonterminal E'

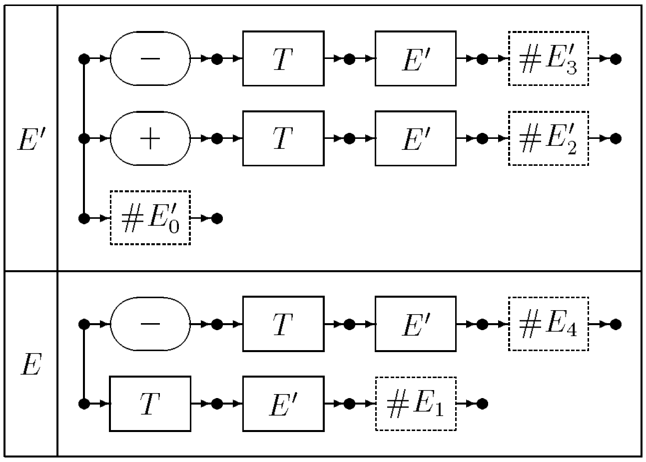

Figure 6.4: Left recursion removal – nonterminals E and E' of a simple arithmetic expression

the original ones by replacement of the common prefix by the newly defined nonterminal. In the 'structural' view it is possible to talk about removement of path forks by cutting them out of the original structure and making them separate production trees with 'forks' only at the beginning – 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 – shown result has nothing that exceeds them.

All the above described changes did not influence recognized language.

Algorithm 6.8 (Left recursion removal):

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

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

Method:

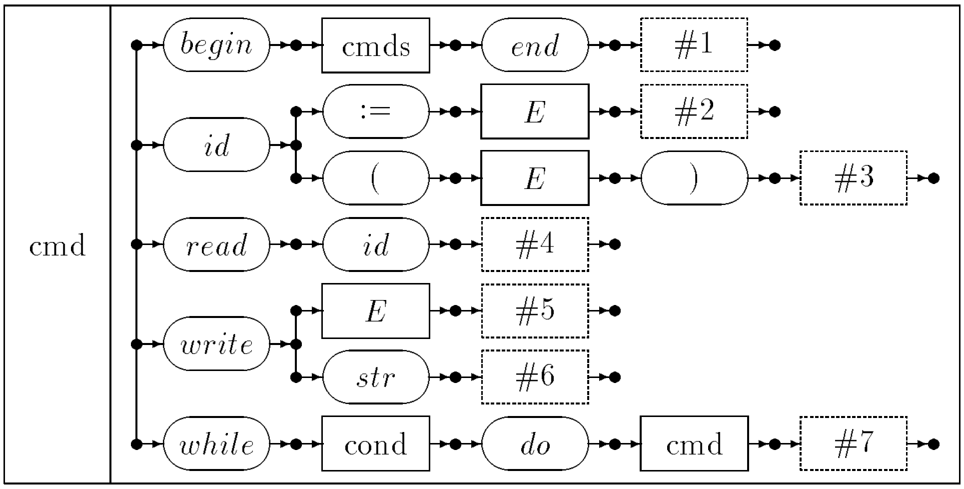

Figure 6.5: Production tree cutting through – original tree

1. Set $G_0 = (\mathrm{N}_0, \sigma_0, P_0, S)$ to G_{pr}, $i = 0$.
2. While $DLRP_{G_i} \neq \emptyset$ do:
   (a) Let $A \in \mathrm{N}_i :\ DLRP_{G_i}(A) \neq \emptyset$.
   (b) Set $\mathrm{N}_{i+1} = \mathrm{N}_i \cup \{A'\}$; $A' \notin \mathrm{N}_i$.
   (c) Set $P_{non-rec} = \{A \to \alpha A' | A \to \alpha \in NLRP_{G_i}(A)\}$.
   (d) Set $P_{left-rec} = \{A' \to \beta A' | A \to A\beta \in DLRP_{G_i}(A)\}$.
   (e) Set $P_{unchanged} = \{B \to \beta \in P_i | B \neq A\}$.
   (f) Set $P_{i+1} = P_{non-rec} \cup P_{left-rec} \cup P_{unchanged} \cup \{A' \to \varepsilon\}$.
   (g) Set $G_{i+1} = (N_{i+1}, \Sigma_i, P_{i+1}, S)$.
   (h) Set i to $i + 1$.

   endwhile
3. Let $G_{nr} = G_i$.

Lemma 6.9 Algorithm 6.8 is finite.

Proof: The step 2 (and substeps 2a–2h) are made for each nonterminal at most once. Steps 1 and 3 are made exactly once. Therefore the algorithm must stop after a finite number of steps. □

Lemma 6.10 The grammar G_{nr} produced by the Alg. 6.8 generates the same language as the original grammar G_{pr}.

Proof: The transformation from G_{pr} to G_{nr} made by Alg. 6.8 is done in finite number of steps (Lemma 6.9). Hence the validity of Lemma 6.10 can be derived from the fact that all the steps made by Alg. 6.8 preserve equality of generated languages by original grammar and by grammar derived from the original one in a single transformation (step 2 of Alg. 6.8).

The generated language preservation between G_i and G_{i+1} where G_{i+1} is derived from G_i in one iteration of Alg. 6.8 can be proven by an induction by the number of rewritings

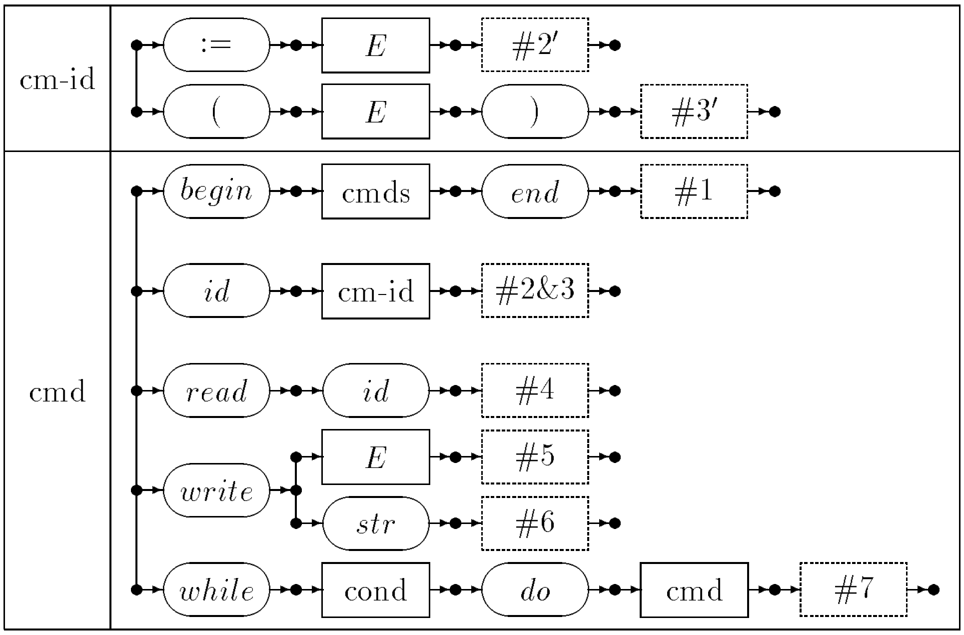

Figure 6.6: Production tree cutting through – partially cut through tree with separated fork

of the nonterminal for which the left recursion was in given iteration removed (let it be A).

Let the nonterminal A does not occur in the derivation of some word w in G_{i+1}. Then this word can be also derived in G_i and vice versa. In this case the language is preserved.

Let us expect show that if derivation of any word w the nonterminal A can be rewritten n times in derivations both in G_i and in G_{i+1}, then any word derivable in G_i using $n+1$ rewritings of A can be derived in G_{i+1} and vice versa:

Let there is a succession of derivations in G_i containing

$$A \Longrightarrow A\beta_{i_1} \Longrightarrow^* A\beta_{i_1} \dots \beta_{i_h} \Longrightarrow \alpha_j\beta_{i_1} \dots \beta_{i_h}.$$

Then there is in G_{i+1} a succession of derivations

$$A \Longrightarrow \alpha_j A' \Longrightarrow \alpha_j\beta_{i_1}A' \Longrightarrow^* \alpha_j\beta_{i_1} \dots \beta_{i_h}A' \Longrightarrow \alpha_j\beta_{i_1} \dots \beta_{i_h}.$$

And vice versa. In the derivations successions leading from $\alpha\beta_{i_1} \dots \beta_{i_h}$ there can be n other rewritings of A and the word still can be derived using both grammars (as follows from the induction hypothesis).

As the number of rewritings of A can be arbitrary large without any change of the set of derived words, $L(G_i) = L(G_{i+1})$.

If all the finite number of iterations of some algorithm preserve some property, the property is preserved by the algorithm as a whole. Hence $L(G_{nr}) = L(G_{pr})$. □

Lemma 6.11 The grammar produced by the Alg. 6.8 is k-kind.

Proof: Any k-kind grammar must fulfill the kind condition. It consists from two parts: one part concerns any two A-production being both from $DLRP(A)$ or being from

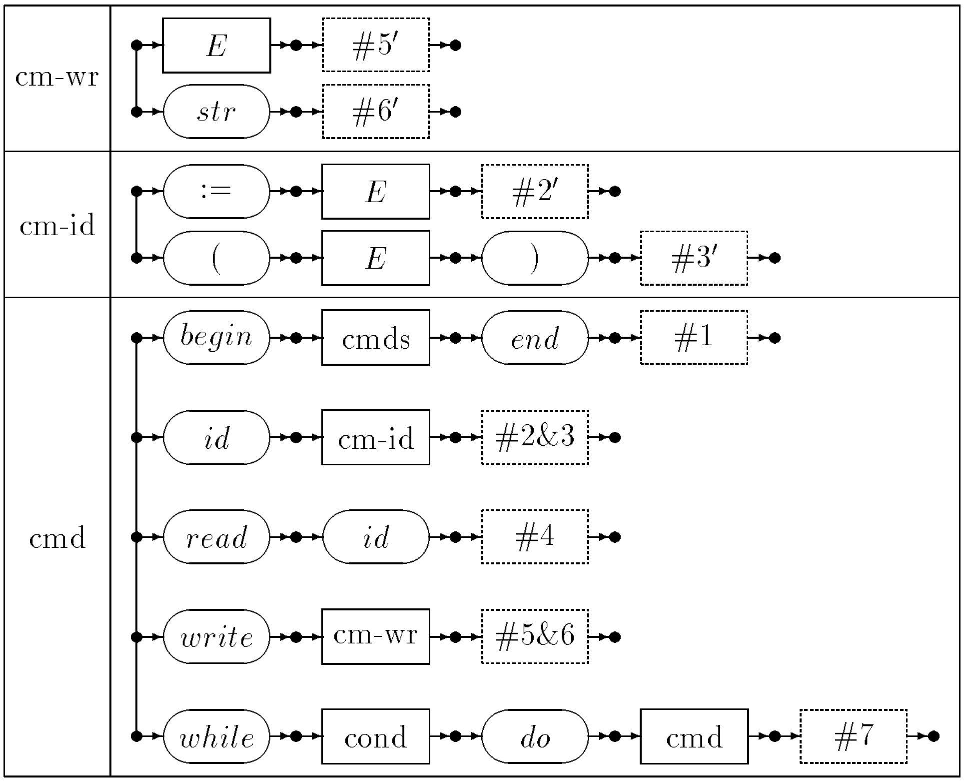

Figure 6.7: Production tree cutting through – a forest with forks only in the roots of separate trees

$NLRP(A)$. The second part ($\forall A : \ DLRF_k(A) \cap NLRF_k(A) = \emptyset$) need to be tested only for grammars with left recursive productions, otherwise is the second part of kind condition trivially fulfilled (as in the case of G_{nr}). The kind condition was fulfilled for G_{pr} (otherwise it would not be a k-kind grammar). Hence the kind condition is fulfilled for pairs of productions that are in G_{nr} and in G_{pr}.

As the Alg. 6.8 replaces all A-productions for any left recursive nonterminal A, it cannot happen that it should be tested the kind condition for a pair of productions where one production is in G_{pr} and the second one is not. So we only need to show that kind condition is fulfilled also on productions that are in G_{nr} and are not in G_{pr}.

For any two A-productions $A \rightarrow \alpha\gamma_1 A'$ and $A \rightarrow \alpha\gamma_2 A'$ from G_{nr} (where A' is the nonterminal added in the step 2f of Alg. 6.8 to handle the left recursion of A) we need to show that

$$First^k_{G_{nr}}(\gamma_1 A' \cdot Follow^k_{G_{nr}}(A)) \cap First^k_{G_{nr}}(\gamma_2 A' \cdot Follow^k_{G_{nr}}(A)) = \emptyset.$$

As A' rewrites to strings of β_i or to ε (see Alg. 6.8 step 2),

$$First^k_{G_{nr}}(A' \cdot Follow^k_{G_{nr}}(A)) = DLRF^k_{G_{pr}}(A) \cup NLRF^k_{G_{pr}}(A) = Follow^k_{G_{pr}}(A).$$

The two A-productions $A \rightarrow \alpha\gamma_1 A'$ and $A \rightarrow \alpha\gamma_2 A'$ from G_{nr} have their counterparts in G_{pr}: $A \rightarrow \alpha\gamma_1$ and $A \rightarrow \alpha\gamma_1$. The pair of productions from G_{pr} fulfill the kind condition (otherwise the original grammar cannot be k-kind) and

$$First^k_{G_{pr}}(\gamma_1 \cdot Follow^k_{G_{pr}}(A)) \cap First^k_{G_{pr}}(\gamma_1 \cdot Follow^k_{G_{pr}}(A)) = \emptyset.$$

As $\gamma_i \Longrightarrow^* w$ in both grammars, the kind condition must be for the discussed pair of A-productions from G_{nr} fulfilled.

Similarly to any two A'-productions $A' \to \beta\delta_1 A'$ and $A' \to \beta\delta_2 A'$ from G_{nr} there are two productions in G_{pr} $A \to A\beta\delta_1$ and $A \to A\beta\delta_2$ allowing to show that the kind condition is fulfilled for the given two A'-productions.

Now we show that also $A' \to \beta A'$ and $A' \to \varepsilon$ fulfill the kind condition. $First^k_{G_{nr}}(\beta A' \cdot Follow^k_{G_{nr}}(A')) = First^k_{G_{nr}}(\beta A' \cdot Follow^k_{G_{nr}}(A))$ as the nonterminal A' is at the end of all A-productions in G_{nr}.

Similarly $First^k_{G_{nr}}(\varepsilon \cdot Follow^k_{G_{nr}}(A')) = Follow^k_{G_{nr}}(A') = Follow^k_{G_{nr}}(A) = NLRF^k_{G_{nr}}(A) = NLRF^k_{G_{pr}}(A)$.

As for the $A' \to \beta A' \in P_{nr}$ there is $A \to A\beta \in P_{pr}$ $First^k_{G_{nr}}(\beta A' \cdot Follow^k_{G_{nr}}(A)) \subset DLRF^k_{G_{pr}}(A)$.

As G_{pr} is k-kind,

$$DLRF^k_{G_{pr}}(A) \cap NLRF^k_{G_{pr}}(A) = \emptyset$$

and therefore

$$First^k_{G_{nr}}(\beta A' \cdot Follow^k_{G_{nr}}(A')) \cap First^k_{G_{nr}}(\varepsilon \cdot Follow^k_{G_{nr}}(A')) = \emptyset;$$

i.e. kind condition is fulfilled also for $A' \to \beta A'$ and $A' \to \varepsilon$.

It was shown that kind condition is fulfilled also for G_{nr}. □

Algorithm 6.12 (Transformation of a non left-recursive k-kind grammar to a SLL(k)-grammar):

Input: A k-kind grammar G_{k-k} without left recursive productions.

Output: A SLL(k) grammar $G_{SLL(k)}$ such that $L(G_{SLL(k)}) = L(G_{k-k})$

Method:

1. Set $G_0 = (\mathrm{N}_0, \Sigma, P_0, S)$ to G_{k-k}, set i to 0.
2. `While` there are at least two distinct productions with a common prefix and the same nonterminal on their left-hand side `do`:
   (a) Let the group of A-productions with the longest common prefix α be: $A \to \alpha\beta_1, \ldots, A \to \alpha\beta_m$.
   (b) $\mathrm{N}_{i+1} = \mathrm{N}_i \cup \{A'\}$; $A' \notin \mathrm{N}_i$.
   (c) Set $P_{Unchanged}$ to $P_i \backslash \{A \to \alpha\beta_1, \ldots, A \to \alpha\beta_m\}$.
   (d) Set $P_{Replaced}$ to $\{A' \to \beta_i | A \to \alpha\beta_i \in P_i\}$.
   (e) Set P_{New} to $\{A \to \alpha A'\}$
   (f) Set P_{i+1} to $P_{Unchanged} \cup P_{Replaced} \cup P_{New}$.
   (g) Let $G_{i+1} = (\mathrm{N}_{i+1}, \Sigma, P_{i+1}, S)$.
   (h) Set i to $i+1$.

   `endwhile`
3. Set $G_{SLL(k)}$ to G_i.

Lemma 6.13 Algorithm 6.12 stops after finite number of steps for every grammar G_{k-k}.

Proof: The algorithm works in steps. Steps 1 and 3 are done exactly once, step 2 iterates. Therefore we need to prove that this step is done only finite number of iterations.

Each iteration transforms G_i to G_{i+1}. The difference between G_i and G_{i+1} is that one group of productions with common prefix is removed.

At the beginning there can be at most $|P_i|-1$ groups of productions with some common prefix.

As in each iteration the longest common prefix is removed, the 'replaced' productions have no common prefix, hence number of common prefixes is decreased.

After finite number of steps (at most $(|P_i|-1)+2$) the algorithm must end; the algorithm is finite. □

Lemma 6.14 Let G_2 be derived from G_1 by the Alg. 6.12 in one step (i.e. the substeps 2a–2h are processed just once). Then if G_1 is a k-kind grammar, G_2 is also a k-kind grammar and $L(G_1) = L(G_2)$.

Proof: The languages generated by both grammars must be the same because every time an A-production from the replaced group $A \rightarrow \alpha\beta_i \in G_1$ is used, the same string $(\alpha\beta_i)$ can be derived in G_2 only by applying $A \rightarrow \alpha A'$ and $A' \rightarrow \beta_i$ – and vice versa.

If the kind condition is fulfilled for G_1, it must be fulfilled also for the group of A-productions $A \rightarrow \alpha\beta_i$ (i.e. $First^k_{G_1}(\beta_i \cdot Follow^k_{G_1}(A)) \cap First^k_{G_1}(\beta_j \cdot Follow^k_{G_1}(A)) = \varepsilon$ for any $i \neq j$).

As $First^k_{G_1}(\beta_i \cdot Follow^k_{G_1}(A)) = First^k_{G_2}(\beta_i \cdot Follow^k_{G_2}(A'))$, the kind condition is fulfilled for G_2 too and G_2 is also a k-kind grammar. □

Lemma 6.15 Algorithm 6.12 does not change the generated language and k-kind property.

Proof: The Alg. 6.12 works in steps. Each step preserves the both properties, hence also the algorithm as a whole must preserve these properties too. □

Lemma 6.16 The grammar $G_{SLL(k)}$ generated by the Alg. 6.12 is a strong LL(k) one.

Proof: Grammar produced by the Alg. 6.12 is a k-kind one without left recursive productions and there are no common prefixes. Hence only the first part of kind condition should be fulfilled (for two distinct productions $A \rightarrow \alpha\beta_1$ and $A \rightarrow \alpha\beta_2$: $First_k(\beta_1 \cdot Follow(A)) \cap First_k(\beta_2 \cdot Follow(A)) = \emptyset$ – the rest is fulfilled trivially. As there are no common prefixes in the grammar, $\alpha = \varepsilon$.

We have a grammar where for any two distinct productions $A \rightarrow \beta_1$ and $A \rightarrow \beta_2$: $First_k(\beta_1 \cdot Follow(A)) \cap First_k(\beta_2 \cdot Follow(A)) = \emptyset$ what is exactly that what is in the definition of strong LL(k) grammars (see Def. 2.14). □

Chapter 7

Lookahead Automata

To prepare grounds for kind parsers based on pushdown automata there is presented a new type of automata: lookahead automata. For this purpose two types of them are useful: lookahead finite automata (for lexical analysis), and lookahead pushdown automata (for parsing).

7.1 Lookahead finite automata

Definition 7.1 (Nondeterministic lookahead finite automaton):
A *nondeterministic lookahead finite automaton* is a 6-tuple $M = (k, Q, \Sigma, \delta, q_0, F)$, where

1. $k > 0$ is a finite *lookahead* depth (head width),
2. Q is a finite set of *states*,
3. Σ is a finite set of permissible *input symbols*,
4. δ is a mapping $Q \times \bigcup_{i=0}^{k} \Sigma^i \rightarrow \mathcal{P}(Q \times \{0, .., k\})$ which dictates the behavior of the finite state control (the number between 0 and k specifies the number of input symbols to be read; sometimes it is called *shift*); δ is sometimes called *state transition function*,
5. $q_0 \in Q$ is the *initial state* of the final state control, and
6. $F \subseteq Q$ is the set of *final states.*

Lemma 7.2 For any nondeterministic finite automaton A there is a nondeterministic lookahead finite automaton M recognizing the same language.

Proof: Let $A = (Q, \Sigma, \delta, q_0, F)$. Setting $M = (1, Q, \Sigma, \delta_M, q_0, F)$ where

$$\delta_M = \left\{ (q, a) \rightarrow \{(q_1, 1), \ldots, (q_n, 1)\} \;\middle|\; \begin{array}{l} q, q_1, \ldots, q_n \in Q, a \in \Sigma, \\ (q, a) \rightarrow \{q_1, \ldots, q_n\} \in \delta \end{array} \right\}$$

allows to fully simulate A. □

Definition 7.3 (Deterministic lookahead finite automaton):

Let $M = (k, Q, \Sigma, \delta, q_0, F)$ be a nondeterministic lookahead finite automaton. We say that M is *deterministic* if $\delta(q, u)$ has no more than one member for any $q \in Q$ and $u \in \bigcup_{i=1}^{k} \Sigma^k$. If $\delta(q, u)$ always has exactly one member, we say that M is *completely specified.*

Lemma 7.4 For any deterministic finite automaton A there is a deterministic lookahead finite automaton M recognizing the same language.

Proof: Let $A = (Q, \Sigma, \delta, q_0, F)$. Setting $M = (1, Q, \Sigma, \delta_M, q_0, F)$ where

$$\delta_M = \{(q_1, a) \to (q_2, 1) \mid q_1, q_2 \in Q, a \in \Sigma, (q_1, a) \to q_2 \in \delta\}$$

allows to fully simulate A. □

Lemma 7.5 For any deterministic lookahead finite automaton $\mathcal{A}$ there is a deterministic finite automaton $\mathcal{A}'$ recognizing the same language.

Proof basics: It is needed to simulate the lookahead in an additional part of a state of the finite control. That means that the automaton should first set up this "buffer", and then it can simulate the steps of the original automaton one by one.

Proof: Let $\mathcal{A} = (k, Q, \Sigma, \delta, q_0, F)$ and $\mathcal{A}' = (Q', \Sigma', \delta', q_0', F')$. If both the automata should accept the same language, it must hold $\Sigma' = \Sigma$. Setting

$$\begin{aligned}
&Q' = Q \times (\Sigma \cup \{\$, \circ\})^k,\\
&q_0' = (q, \underbrace{\circ \ldots \circ}_{k}),\\
&F' = \{(q, \underbrace{\$ \ldots \$}_{k}) \mid q \in F\},\\
&\delta' = \{((q_0, a_1 \ldots a_i \underbrace{\circ \ldots \circ}_{k-i}), a_{i+1}) \to (q_0, a_1 \ldots a_{i+1} \underbrace{\circ \ldots \circ}_{k-i-1}) \mid\\
&\qquad (q_0, a_1 \ldots a_j) \in \delta, 0 \leq i \leq j, j = min(k, n)\}\\
&\quad \cup \{((q, a_1 \ldots a_k), \varepsilon) \to (q', a_1 \ldots a_k) \mid (q, a_1 \ldots a_k) \to (q', 0) \in \delta\}\\
&\quad \cup \{((q, a_1 \ldots a_k), a_{k+1}) \to (q', a_2 \ldots a_{k+1}) \mid\\
&\qquad (q, a_1 \ldots a_k) \to (q', 1) \in \delta, a_{k+1} \in (\Sigma \cup \$)\}
\end{aligned}$$

where $\circ$ represents an uninitialised lookahead element of $\dashv$ and \$ represents an item not belonging to the alphabet, allow to simulate the original lookahead automaton by the 'usual' one. □

Note 7.6 Lemma 7.5 can be extended to nondeterministic versions of lookahead finite automata and finite automata.

The lookahead finite automaton is very useful for a construction of a run-time extensible lexical analyzer. It is possible to separate scanning of different lexemes. Therefore the lexical analyzer can be changed after scan of any terminal – no information needed for the next one is hidden in an actual state. Pure extension can be made at any time because of preservation of already existing states.

7.2 Lookahead pushdown automaton

Definition 7.7 (Lookahead pushdown automaton):
A *lookahead pushdown automaton* (LPDA for short) is a 8-tuple

$$P = (k, Q, \Sigma, \Gamma, \delta, q_0, Z_0, F),$$

where

1. $k > 0$ is a number of input symbols visible at once (*lookahead* depth),
2. Q is a finite set of *state* symbols representing the possible states of the finite state control,
3. Σ is a finite *input alphabet*,
4. Γ is a finite alphabet of *pushdown list symbols*,
5. δ is a mapping $Q \times \bigcup_{i=0}^{k} \Sigma^i \times \Gamma$ to the finite subsets of $Q \times \Gamma^* \times \{0..k\}$.
6. $q_0 \in Q$ is the *initial state* of the finite control,
7. $Z_0 \in \Gamma$ is the symbol that appears initially on the pushdown list (the *start symbol*), and
8. $F \subseteq Q$ is the set of *final states.*

A *configuration* of P is a triple $(q, w, \alpha) \in Q \times \Sigma^* \times \Gamma^*$, where

1. q represents the current state of the finite control,
2. w represents the unused portion of the input. The first k symbols of w are under the input head (if $|w| < k$ then all the w is seen). If $w = \varepsilon$, then it is assumed that all of the input tape has been read.
3. α represents the contents of the pushdown list. The leftmost of α is the topmost pushdown symbol. If $\alpha = \varepsilon$, then the pushdown list is assumed to be empty.

A *move* by P will be represented by the binary relation $\vdash_P$ (or $\vdash$ whenever P is understood) on configurations. We write

$$(q, a_1 \dots a_i a_{i+1} \dots a_j w, Z\alpha) \vdash (q', a_{i+1} \dots a_j w, \gamma\alpha) \tag{7.1}$$

if $\delta(q, a, Z)$ contains (q', γ) for any $q \in Q$, $a_1 \dots a_j \in \bigcup_{i=0}^{k} \Sigma^i$, $w \in \Sigma^*$, $Z \in \Gamma$, $\alpha \in \Gamma^*$, and $0 \leq i \leq j \leq k$ ($j < k \Rightarrow w = \varepsilon$).

If $a_1 \dots a_i \neq \varepsilon$, Eq. (7.1) states that if P is in a configuration such that the finite control is in state q, the current input symbol are $a_1 \dots a_i$, and the symbol on the top of pushdown list is Z, then P may go into a configuration in which the finite control is now in the state q', the input head has been shifted i symbols to the right, and the topmost symbol on the pushdown list has been *popped.*

If $a = \varepsilon$, then the move is called an ε-*move.* In an ε-move the input head is not moved. However, the state of the finite control can be changed, and the contents of the memory can be adjusted. Note that an ε-move can occur even if all of the input has been read.

No move is possible if the pushdown is empty.

We can define the relations $\vdash^i$, for $i \geq 0$, $\vdash^*$, and $\vdash^+$ in the customary fashion. Thus, $\vdash^*$ and $\vdash^+$ are, respectively, the reflexive-transitive and transitive closures of $\vdash$.

An *initial configuration* of P is one of the form (q_0, w, Z_0) for some $w \in \Sigma^*$. That is, the finite state control in the initial state, the input contains the string to be recognized, and the pushdown list contains only the symbol Z_0. A *final configuration* is one of the form (q, ε, α), where $q \in F$ and $\alpha \in \Gamma^*$.

We say that a string w is *accepted* by P if $(q_0, w, Z_0) \vdash^* (q, \varepsilon, \alpha)$ for some $q \in F$ and $\alpha \in \Gamma^*$. The *language defined by* P, denoted $L(P)$, is the set of strings accepted by P. $L(P)$ will be called a *lookahead pushdown automaton language.*

Lemma 7.8 For any pushdown automaton P there is a lookahead pushdown automaton P' accepting the same language.

Proof: Just set $k = 1$, Σ must be for both the automata the same, Q, F, q_0, Γ and Z_0 can be also the same for both P and P'. δ' can be a natural extension of δ. Then P' can exactly simulate the work of P. $\square$

Lemma 7.9 For any lookahead pushdown automaton P there is a pushdown automaton P' accepting the same language.

Proof idea:
The lookahead of the original lookahead pushdown automaton can be simulated in the states of a classic pushdown automaton. It is only needed to care about initialization of the lookahead and resolving short inputs.

Proof: Let $M = (k, Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ and $M' = (Q', \Sigma', \Gamma', \delta', q_0', Z_0', F')$. If both automata have to recognize the same language, then $\Sigma' = \Sigma$.

Other items can be put as follows ($\circ$ stays for uninitialized part of state representing one lookahead item, \$ stays for one item from input tape not from Σ, and m is the length of the longest string of pushdown alphabet put on pushdown at a time):

$$
\begin{aligned}
&Q' = Q \times \textstyle\bigcup_{i=0}^{k} \Sigma^i \\
&q_0' = (q_0, \varepsilon) \\
&\Gamma' = \Gamma \\
&Z_0' = Z_0 \\
&\delta_{Load} = \{((q, a_1 \dots a_i), a_{i+1}, Z) \to ((q, a_1 \dots a_{i+1}), Z) \\
&\qquad \mid 0 \leq i < \min(k, n), a_j \in \Sigma, q \in Q, Z \in \Gamma\} \\
&\delta_{ShortRead} = \{((q_1, a_1 \dots a_i), a_{i+1}, Z) \to ((q_2, a_j \dots a_{i+1}), \gamma) \\
&\qquad \mid q_1, q_2 \in Q,\ a_m \in \Sigma,\ 0 < j \leq i < k,\ Z \in \Gamma,\ \gamma \in \Gamma^*, \\
&\qquad (q_1, a_1 \dots a_i, Z) \to (q_2, \gamma, j) \in \delta\} \\
&\delta_{LongRead} = \{((q_1, a_1 \dots a_k), a_{k+1}, Z) \to ((q_2, a_j \dots a_{k+1}), \gamma) \\
&\qquad \mid q_1, q_2 \in Q, a_i \in \Sigma, 0 \leq i \leq k+1, 0 < j \leq k, Z \in \Gamma, \gamma \in \Gamma^*, \\
&\qquad (q_1, a_1 \dots a_k, Z) \to (q_2, \gamma, j)\} \\
&\delta_{Stay} = \{((q_1, a_1 \dots a_i), \varepsilon, Z) \to ((q_2, a_1 \dots a_i), \gamma) \\
&\qquad \mid q_1, q_2 \in Q, a_1 \dots a_i \in \Sigma^i, Z \in \Gamma, \gamma \in \Gamma^*, \\
&\qquad (q_1, a_1 \dots a_i, Z) \to (q_2, \gamma, 0) \in \delta\} \\
&\delta' = \delta_{Load} \cup \delta_{ShortRead} \cup \delta_{LongRead} \delta_{Stay}
\end{aligned}
$$

The above defined automaton simulates the original one. □

Partitioning to δ_{Load}, $\delta_{ShortRead}$, $\delta_{LongRead}$ and δ_{Stay} corresponds to different situations: δ_{Load} fills up the simulated lookahead, $\delta_{ShortRead}$ resolves reads of original lookahead pushdown automaton when close to the end of input tape, $\delta_{LongRead}$ simulates usual reads, and δ_{Stay} resolves ε-steps of the original automaton.

Corollary 7.10 The computational power of lookahead pushdown automaton and pushdown automaton is the same.

Note 7.11 As stated above both kinds pushdown automata have the same computational power – i.e. one system can be simulated by the second and vice versa.

The reason for the introduction of a modification of pushdown automaton is that real computational devices (and their programs written in languages like C) are closer to the new model and also that the new model has fewer states and is easier to maintain and modify.

The ability to use dynamic lookahead is very useful in cases like SQL that needs at some point lookahead larger than 10 but the rest of the language is not so complicated. If the parser will be written using standard parser generators it would be extremely large. Using dynamic lookahead the size of the generated parser can be kept within practical bounds.

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 or as deduction systems.

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 during parsing using production tree forest is in Tab. 8.1.

transition along an edge labelled 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 labelled 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 labelled 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 do not match any of both the lookaheads and the stack is empty, the computation ends and the input us accepted; otherwise the computation also ends but the input is rejected

Table 8.1: Parsing using production trees

8.1 Kind (pushdown) automaton

Divided (dotted) productions are introduced e.g. in [AU72, Nau94]. It 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:

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

Algorithm 8.2 generates to 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. $Q = \{[A \rightarrow \alpha.\beta]_{=_G} \mid A \rightarrow \alpha.\beta \in P_G\}$
2. $\Sigma_M = \Sigma_G$
3. $\Gamma = Q \cup \{\circ\}, \quad \circ \notin Q$
4. $q_0 = [S_G \rightarrow .\alpha]_{=_G}, \quad S_G \rightarrow \alpha \in P_G$
5. $Z_0 = \circ$
6. $F = \{[S_G \rightarrow \alpha.]_{=_G} \mid S_G \rightarrow \alpha \in P_G\}$
7. $\delta_{Scan} = \{([A \rightarrow \alpha.a\beta]_{=_G}, ab_2 \ldots b_k, q) \rightarrow ([A \rightarrow \alpha a.\beta]_{=_G}, q, 1) |$
   $A \rightarrow \alpha a\beta \in P_G \wedge ab_2 \ldots b_k \in First^k_G(a\beta \cdot Follow^k_G(A)) \wedge q \in \Gamma\}$
8. $\delta_{Descent} = \{([A \rightarrow \alpha.B\beta]_{=_G}, b_1 \ldots b_k, q) \rightarrow ([B \rightarrow .\gamma]_{=_G}, [A \rightarrow \alpha B.\beta]_{=_G}, q, 0)|$
   $A \rightarrow \alpha a\beta, B \rightarrow \gamma \in P_G \wedge b_1 \ldots b_k \in First^k_G(B\beta \cdot Follow^k_G(A)) \wedge q \in \Gamma\}$
9. $\delta_{Recurse} = \{([B \rightarrow \alpha.]_{=_G}, a_1 \ldots a_k, q) \rightarrow ([B \rightarrow B.\beta]_{=_G}, q, 0)|$
   $B \rightarrow \alpha, B \rightarrow B\beta \in P_G \wedge a_1 \ldots a_k \in DLRF^k_G(B) \wedge q \in \Gamma\}$
10. $\delta_{Finish} = \{([B \rightarrow \gamma.]_{=_G}, a_1 \ldots a_k, [A \rightarrow \alpha a.\beta]_{=_G}) \rightarrow ([A \rightarrow \alpha B.\beta]_{=_G}, \varepsilon, 0)|$
    $B \rightarrow \gamma, A \rightarrow \alpha B\beta \in P_G \wedge a_1 \ldots a_k \in NLRF^k_G(B)\}$
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 Deductive system

Principles of production-tree-based parsing can be described also in the notation of deductive systems. Creation of such system is described in Alg. 8.3.

Algorithm 8.3 (Construction of a deductive system):
Input: Grammar $G = (\mathrm{N}, \Sigma, P, S)$, input string $a_1 \ldots a_n$
Output: Deductive system D for given language and input
Method:

1. Create a production tree forest for the grammar G.

2. For every node create an item $[A \to \alpha.\beta, i, j], 0 \le i \le j$. $\mathcal{I}$ stands for a set of all these items.

3. It should be created a derivation for every edge of the production tree – always according to the type of the label of the current edge:

   (a) terminal – $[A \to \alpha.a\beta, i, j], [a_{j+1}, j, j+1], \ldots, [a_{j+k}, j+k-1, j+k] \vdash [A \to \alpha a.\beta, i, j+1]$ for all $0 \le i \le j$, for all $A \to \alpha\beta \in P$, and for all $a_{j+1} \ldots a_{j+k} \in First^k(\beta \cdot Follow^k(A))$. Set of such deductions is denoted $\mathcal{D}^{Shift}$.

   (b) nonterminal – $[A \to \alpha.B\beta, i, j], [a_{j+1}, j, j+1], \ldots, [a_{j+k}, j+k-1, j+k] \vdash [B \to .\gamma, j, j]$, $0 \le i \le j \wedge A \to \alpha B\beta$, $B \to \gamma \in P_G$, $a_{j+1} \ldots a_{j+k} \in First^k(\gamma \cdot Follow^k(B))$. Set of such deductions is denoted $\mathcal{D}^{Expand}$.

   (c) production –

      i. $[A \to \alpha., i, j], [a_{j+1}, j, j+1], \ldots, [a_{j+k}, j+k-1, j+k] \vdash [A \to A.\beta, j, j]$, $0 \le i \le j$, $A \to \alpha, A \to A\beta \in P$, $a_{j+1} \ldots a_{j+k} \in DLRF^k(A)$. Set of such deductions is denoted $\mathcal{D}^{Recurse}$.

      ii. $[A \to \alpha., i, j], [a_{j+1}, j, j+1], \ldots, [a_{j+k}, j+k-1, j+k], [B \to \beta.A\gamma, h, i] \vdash [B \to \beta A.\gamma, h, j]$, $0 \le h \le i \le j$, $A \to \alpha, B \to \beta A\gamma \in P$, $a_{j+1} \ldots a_{j+k} \in NLRF^k(A)$. Set of such deductions is denoted $\mathcal{D}^{Reduce}$.

   (d) $\mathcal{D}^{Init} = \{[S \to .\alpha, 0, 0] \mid S \to \alpha \in P\}$

   Set of all deductions $\mathcal{D} = \mathcal{D}^{Shift} \cup \mathcal{D}^{Expand} \cup \mathcal{D}^{Reduce} \cup \mathcal{D}^{Recurse} \cup \mathcal{D}^{Init}$.

4. For every symbol from the input string a_i it will be created a hypothesis $[a_i, i-1, i]$ $\forall i \in 1..n$. Set of all hypotheses will be denoted $\mathcal{H}$.

Deduction system is a triple $\langle \mathcal{I}, \mathcal{H}, \mathcal{D} \rangle$[1].

8.3 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

[1]To be correct let us add that original set of hypotheses is completed by the symbol of end of input, eoi, similarly as by the corresponding lookaheads.

of its structure. This feature is preserved also in the transformations: both (lookahead) pushdown automata and deduction systems built following above stated methods handling production trees (based on them) can be arbitrary (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.

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.

8.4 Reduction

Reduction of the automaton is potentially possible everywhere but with caution to keep resulting grammar k-kind (the keeping 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 – is needed 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 returns only results 'yes' or 'no' answering the question whether the input belongs to the given language.

For practical use 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 $a_1 a_2 \dots a_n$ an output of the form $\sigma_0 a_1 \sigma_1 a_2 \dots a_n \sigma_n$ 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 type of a symbol the symbols will be written in different brackets:

()	terminal
[]	nonterminal
{}	semantic action

The resulting grammar with its semantic actions is here:

$[S] \to [E]\{\text{print}\}$
$[E] \to [T]$
$[E] \to [E](\text{'+'})[T]\{\text{addition}\}$
$[T] \to [F]$

[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.

$[T] \to [T]('*')[F]\{\text{multiplication}\}$
$[F] \to (id)\{\text{fetch}\}$
$[F] \to (num)\{\text{fetch}\}$
$[F] \to ('(')[E](')')$

Meaning of individual semantic actions is described in Tab. 9.1. It is expected that

{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 stack and print it out
{fetch}	fetch a terminal value from lexical analyzer and push it onto the semantic stack

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

there is a pushdown for value storing and that the lexical analyzer is able to give real values of the terminals *num* and *id* – i.e. their numerical values, and identifier strings, resp. 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 and of the productions, as LR-grammars allow (see [Par96]).

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, MČJR]).

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 *simplified to G* if for all symbols a_i from Φ and Ψ new nonterminals A_i and new productions $A_i \rightarrow \varepsilon$ 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), k-kind if and only if a simplified grammar to the grammar G is deterministic, LR(k), LL(k), k-kind respective.

Translational grammars are discussed e.g. in the works of Melichar [MČJR], semantic grammars can be found e.g. in [ŽK99a].

Both translational and semantic grammars are used to build compilers (see e.g. [Kra82, MČJR]). 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).

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	$\left\{\begin{array}{l} S \rightarrow E \underline{=} \text{ pop put } \underline{num} \\ E \rightarrow E + T \underline{+} \text{ add} \\ E \rightarrow E - T \underline{-} \text{ sub} \\ E \rightarrow T \\ T \rightarrow T * F \underline{*} \text{ mul} \\ T \rightarrow F \\ F \rightarrow num \text{ fetch push put } \underline{num} \\ F \rightarrow (E) \end{array}\right\}$
Starting symbol	S

The grammar expects that lexical analyzer stores values of num into a special buffer. 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 for 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):
3-tuple $E = (\mathrm{N}', \Sigma', P')$ is called *grammar extension* of 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 when not applied but also during generation/parse time. The parse-time extension can be caused by an external enity (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).

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 $\Sigma = \{a, b, c, d, e\}$, $\mathrm{N} = \{S, A, B\}$, starting symbol S, productions

$S \rightarrow AeB$
$A \rightarrow aAb \mid c\sigma \mid d\tau$
$B \rightarrow aBb$

and semantic actions $\Phi = \{\sigma, \tau\}$ where σ adds the production $B \rightarrow c$ and τ adds $B \rightarrow d$.

$L(G) = \{a^n cb^n ea^m cb^m | m, n \geq 0\} \cup \{a^n db^n da^m cb^m | m, n \geq 0\}$ □

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 seems to be non-LL one.

This language can be generated by a SLR(1) grammar:

$S \rightarrow CeC \mid DeD$
$C \rightarrow aCb \mid c$
$D \rightarrow aDb \mid d$

Conjecture 9.12 Language given by proper extensible grammar or extensible semantic grammar can exceed language class that can contain all languages specified (generated by the grammars) during the parsing process.

Chapter 10

Modifications

The basic idea of kind parsing (processing LL languages) can be extended in several ways. Two of them are described here.

10.1 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. 10.1.

Example 10.1 (Dangling-else problem in pascal):
Construction from Fig. 10.1 can be parsed two ways that are shown as Figs. 10.2 and 10.3. □

`IF` *condition1* `THEN` `IF` *condition2* `THEN` *command1* `ELSE` *command2*

Figure 10.1: Two pascal IF-commands with one ELSE

`IF` *condition1* `THEN`
   `IF` *condition2* `THEN` *command1*
`ELSE` *command2*

Figure 10.2: Two pascal IF-commands with one ELSE attached to the outer IF

`IF` *condition1* `THEN`
   `IF` *condition2* `THEN` *command1* `else` *command2*

Figure 10.3: Two pascal IF-commands with one ELSE attached to the closer IF

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].

10.2 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 is 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.

Chapter 11

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 worser 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.

11.1 Implementation details

Kind constructor work 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) should be described now.

11.1.1 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).

Structure of the subroutine is a large and complex selection (switch in C) 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.

11.1.2 Language description file

The whole description of a language for which the parser should be constructed 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. 11.1.

Name	Necessary	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 11.1: Overview of selected sections from a language description file

Example 11.1 (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 11.2 (Generated parser code):
The following lines were generated for the grammar from Ex. 11.1 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. 11.2, 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 12

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 mentioned for further integration. Therefore they 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* [KŽ2000, KŽ2001, KŽ2002].

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 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 12.1 (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 12.2 The Ex. 12.1 was processed by KindTran [Žem02b], i.e. both langauge 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 12.3 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.

Chapter 13

Applications

13.1 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. 11.

13.2 Algorithmic development environment

There was a try to implement a development tool for handling 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 complex expressions with an extensible syntax and once for import textual format of algorithm and data type descriptions.

Chapter 14

Related Results

There are several types of related works:

- extensible parsers
- extensible compilers
- recursive descent parsers
- extensible development tools

14.1 Extensible compilers

Most of extensible compilers that were 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

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].

14.2 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 [BB01].

Other techniques are used e.g. in Lua [IFF]: 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 suggest 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.

14.3 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 imperative programming languages) and other applications. This topic is discussed e.g. in [Gri71, Kra82, Dro85, Ruß].

The main result of [Kra82] and [Dro85] is that the recursive descent parsing can be used for LR grammars.

14.4 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 to it some parameters and data and use or at least show its results. Most integrated development environments are able to add in their menues invocation of third party tools 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

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/`.

Chapter 15

Conclusion

Kind grammars are able to define the syntax of many programming languages. Such languages are "almost" LL but they require a limited form of left recursion. Kind grammars are able to describe it as they allow to generate parsers that are legible and understandable. It facilitates "hand tuning" that is necessary e.g. for the production quality compilers.

Other advantage of kind parsing is its ability to be extended or modified also during run-time. That is, that the parser can be extended by language extensions found in the parsed text or by asynchronous actions without the loss of already collected information. It seems to be the only one parsing technique allowing to parse arbitrary LL(k) language that's extensions can be achieved from just parsed text.

XML parsers have a similar ability but they expect tagged input, i.e. they just parse the linear form of parsing tree what is significantly easier than parsing of an untagged text.

For 1-kind grammars there can be created parsers with clear and easy-to-read structure. Such parsers are easy to maintain and modify.

The process of the creation of a kind parser from the given kind grammar is very easy and therefore it can successfully be used also for teaching purposes.

Future plans can be divided into two parts: theoretical ones and practical ones. In the first case it seems to be possible that the power of the principles of kind parsing can be enhanced. An open problem is what is the largest grammar class allowing parse-time extension of corresponding parsers. A very interesting problem seems to be the power of language and/or grammar extension: it is needed to proof whether all expected run-time extensions to LL or kind grammars stay in LR, and what is the impact of the run-time extensions gained from the just parsed text. In both cases only grammar extending semantic actions are expected, as if the semantic actions were more complex, they alone (without the grammar extension) can escape from given grammar/language class.

There are also plans for the practice: to develop the kind parsing based tools (KindCons, KindTran) from the current prototype stage to the production quality and to finish a new version of the run-time extensible compiler.

Bibliography

[ASU86] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullmann: *Compilers Principles, Techniques, and Tools*, Addison-Wesley, 1986. ISBN 0-201-10194-7.

[AU72] Alfred V. Aho, Jeffrey D. Ullman: *The Theory of Parsing, Translation, and Compiling, Vol. I: Parsing*, Prentice Hall, 1972. ISBN 0-13-914556-7.

[AU73] Alfred V. Aho, Jeffrey D. Ullman: *The Theory of Parsing, Translation, and Compiling, Vol. II: Compiling*, Prentice Hall, 1973. ISBN 0-13-914564-8.

[bison] `http://www.gnu.org/software/bison/bison.html`

[BB01] Jean Berstel, Luc Boasson: *Formal Properties of XML Grammars and Languages.* 2001.

[Bos96] Jan Bosch: Tool support for language extensibility. In: Lars Bendix, Kurt Normark, Kasper Osterbye (Eds.): *NWERP'96 Nordic Workshop on Programming Environment Research*, pp. 3–17, Aalborg, Denmark, 1996.

[Bos97] Jan Bosch: Delegating Compiler Objects: Modularity and Reusability in Language Engineering. In: *Nordic Journal of Computing*, Vol. 4, pp. 66–92, 1997.

[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. ISBN 80-214-0491-4.

[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 DÚ SPZV], June 1990

[DeR] F. L. DeRemer: Simple LR(k) grammars. In: *Communications of the ACM*, **14**, 1971, pp. 453–460.

[Dro85] Januš Drózd: *Syntaktická analýza téměř shora dolů* (In Czech: Semitopdown parsing). [Master Thesis], Praha, MFF UK 1985.

[Dro90] Januš Drózd: *Syntaktická analýza rekurzivním sestupem pro LR(k) gramatiky*, (In Czech: Recursive descent parsing for LR(k) grammars). [Kandidátská dizertační práce], Praha, MFF UK 1990.

[Ear68] J. Earley: *An efficient context-free parsing algorithm.* [Thesis], Carnegie-Mellon University, Pittsburgh, Pa, 1968.

[Eng99] Dawson R. Engler: Interface Compilation: Steps Toward Compiling Program Interfaces as Languages. In: *IEEE Transactions on Software Engineering*, 25(3), May/June 1999, pp. 387–400.

[Gri71] David Gries: *Compiler construction for digital computers.* John Wiley and Sons, 1971.

[IFF] Roberto Ierusalischy, Luiz Henrique de Figueiredo, 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.

[Knu65] Donald E. Knuth: On the translation of languages from left to right. *Inform. Control* 8, 1965. pp. 607-639.

[Kra82] Jaroslav Král: Syntaktická analýza a syntaxí řízený překlad. (In Czech: Parsing and syntax driven translation). [Technical report ÚVT ČVUT], ÚVT ČVUT, Praha, 1982.

[KŽ2000] Jaroslav Král, Michal Žemlička: Autonomous Components. In: Václav Hlaváč, Keith G. Jeffery, Jiří Wiedermann (Eds.): *SOFSEM 2000: Theory and Practice of Informatics.* Milovy, Springer V., 2000. pp. 375–383.

[KŽ2001] Jaroslav Král, Michal Žemlička: Electronic Government and Software Confederations. In: A M. Tjoa, R. R. Wagner (Eds.): *Twelfth International Workshop on Database and Experts System Application.* IEEE Computer Society, Los Alamitos, CA, USA, 2001. pp. 383–387. ISBN 0-7695-1230-5. ISSN 1529-4188.

[KŽ2002] Jaroslav Král, Michal Žemlička: Systemic Issues of Software Confederations. In: Robert Trappl (Ed.): *Cybernetics and Systems 2002*, volume 1. Austrian Society for Cybernetic Studies, Vienna, 2002. pp. 141–146. ISBN 3-85206-160-1.

[Lea66] B. M. Leavenworth: Syntax Macros and Extended Translation. In: *Comm. of ACM* 9, No. 11 (November 1966), pp. 790–793.

[MČJR] Bořivoj Melichar, Milan Češka, Karel Ježek, Karel Richta: *Konstrukce překladačů* (in Czech: Compiler construction). ČVUT, 1999. ISBN 80-01-02028-2.

[Nau94] Sven Naumann, 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, 1994, ISBN 3-519-02139-0.

[Par96] Terrence J. Parr, Russel W. Quong: LL and LR Translators Need $k > 1$ Lookahead. In: *ACM SIGPLAN Notices*, Volume 31, No. 2 February 1996, pp. 27–34.

[Plá88] Martin Plátek: Testovatelné podmínky pro bezpečnou skeletální množinu a jejich zobecnění (in Czech: Testable conditions for safe skelatal set and their generalization). In: *Sborník krátkých referátů SOFSEM'88*, ÚVT UJEP a JČMF, Bratislava, 1998

[Plá92] Martin Plátek: *Syntactic Error Recovery with Formal Guarantees I.* [Technical report MFF UK], Praha, MFF UK, April 1992.

[RL70] D. J. Rosenkrantz, P. M. Lewis II: Deterministic left corner parsing. In: *IEEE Conf. Record 11th Annual Symposium on Switching and Automata Theory*, 1970, pp. 139–152.

[Ruß] Arnd Rußmann: *Dynamic Left Parsing.*

[Sik93] Nicolaas Sikkel: *Parsing Schemata*, Proefschrift Enschede, 1993, ISBN 90-9006688-8

[SSS90] Seppo Sippu, Eljas Soisalon-Soininen: *Parsing Theory. Volume II: LR(k) and LL(k) Parsing.* Springer Verlag. EATCS 20. ISBN 3-540-51732-4.

[SSU79] Eljas Soisalon-Soininen, 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 Theses], MFF UK, Praha, January 2002.

[TS97] Walid Taha, Tim Sheard: Multi-stage programming with explicit annotations. In: *Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation.* pp. 203–217. ACM Press, New York, NY, USA, 1997. ISSN 0362-1340

[TN91] Masaru Tomita, See-Kiong Ng: The Generalized LR Parsing Algorithm. In: Masaru Tomita (Ed.): *Generalized LR Parsing*, Kluwer Academic Publishers, Norwell, Massachusetts, USA, 1991. pp. 1–16. ISBN 0-7923-9201-9.

[Töp89] Pavel Töpfer: An Error Recovery Method for Top-Down Syntactical Analysis. [Technical report MFF UK], MFF UK, Praha, February 1989.

[WiMa95] Reinhard Wilhelm, Dieter Maurer: *Compiler Design.* Addison-Wesley, 1995, ISBN 0-201-42290-5

[XML] XML, 1999, `http://www.xml.com/xml/pub/` Extensible Markup Language. A proposal of W3C consortium.

[XSLT] `http://www.w3.org/TR/xslt` The home page of the XSLT proposal.

[Žem94] Michal Žemlička: *Překladač rozšiřitelného jazyka.* (In Czech: Compiler for an extensible language). [Master Thesis], MFF UK, Praha, 1994.

[Žem95] Michal Žemlička: Extensible LL(1) parser. [Poster], SOFSEM'95, Milovy, 1995.

[Žem96] Michal Žemlička: *Syntaktická analýza rozšiřitelných jazyků*. (In Czech: Parsing of extensible languages). [Technical report KSI MFF UK] Praha, KSI MFF UK, prosinec 1996.

[Žem02a] Michal Žemlička: *KindCons – Kind Constructor*. [Application], KSI MFF UK, May 2002.
Home page: `http://www.ms.mff.cuni.cz/~zemlicka/KindCons/`.

[Žem02b] Michal Žemlička: *KindTran – Kind Transducer*. [Application], KSI MFF UK, June 2002.
Home page: `http://www.ms.mff.cuni.cz/~zemlicka/KindTran/`.

[ŽBB+98] Michal Žemlička, Václav Brůha, Milan Brunclík, Petr Crha, Jan Cuřín, Svatopluk Dědic, Luděk Marek, Roman Ondruška, Daniel Průša: *Projekt Algoritmy* (in Czech: The Algorithms project). [Software project]. MFF UK, Prague, 1998.

[ŽK99a] Michal Žemlička, Jaroslav Král: Run-Time Extensible (Semi-)Top-Down Parsing. In: Václav Matoušek, Pavel Mautner, Jana Ocelíková, Petr Sojka (Eds.): *Proceedings of the Workshop on Text, Speech and Dialogue*, LNAI/LNCS 1692, pp. 121–126, Springer, 1999.

[ŽK99b] Michal Žemlička, Jaroslav Král: Run-Time Extensible Deterministic Top-Down Parsing. In: *Grammars* **2**: pp. 283–293, 1999. ISSN 1386-7393.

[ŽK2000] Michal Žemlička, Jaroslav Král: Semitopdown Parsing in Information Systems. In: Proceedings of SCI/ISAS 2000 conference, Orlando, Florida, 2000.

[ZO2001] Matthias Zenger, Martin Odersky: *Implementing Extensible Languages*. Presented at ECOOP Workshop on Multiparadigm Programming with Object-Oriented Languages, Budapest, Hungary, June 2001.