An efficient

ELL (1) – Parser Generator

Reinhold Heckmann
Fachbereich 10
Universität des Saarlandes

Technischer Bericht Nr. A 85 / 04

An efficient ELL (1) - Parser Generator

Reinhold Heckmann

Universität des Saarlandes
6600 Saarbrücken
Bundesrepublik Deutschland

ABSTRACT

Extended context-free grammars also called regular right part grammars allow for compact and readable descriptions of the syntax of programming languages. Recursion in conventional context-free grammars can in most cases be replaced by iteration. Parser tables of predictive LL-parsers for extended grammars can be generated very efficiently if the length s of the look-ahead is 1. The generation time is proportional to the size of the parser table, that is the product of grammar size and number of terminal symbols.

0. Introduction

Extended context-free grammars are grammars where the non-terminal symbols produce regular expressions over terminal and non-terminal symbols instead of strings. They are introduced because they are more compact and more readable than conventional ones. In addition, they allow for the replacement of some left or right recursion (e.g. for parameter lists in most programming languages) by a "star expression" using the Kleene-star "*". Left factoring - often needed to obtain an LL-grammar - can also be done easily without introducing additional non-terminals when regular right parts are allowed.

Predictive LL-parsers for extended grammars can be generated very efficiently if the length s of the look-ahead is 1. Naturally, a simplified version of this algorithm can be used to generate LL-parsers also for conventional LL(1)-grammars more efficiently than the algorithm given in [1].

Heilbrunner [3] used another approach; he transformed extended grammars into conventional ones and generated the parser for the resulting grammar. But this parser finds the complete structure of the input under the new grammar, even though much of this structure may be unnecessary for reconstructing the parse for the original grammar.

Purdom and Brown [6] show how to generate LR-parsers directly from extended context-free grammars. Another less efficient algorithm to generate LL-parsers directly from extended grammars can be found in [4].

This paper is a summary of [2]; some less important lemmata and almost all proofs have been omitted here. The absence of proofs in this paper should not suggest that the stated theorems are obvious or easy to prove; some of the omitted proofs are very extensive.

In a predictive LL-parser, first- and follow-sets are needed to obtain the look-ahead entries in the parsing table. Thus in chapters 1 through 15 we investigate the computation of these sets.

The theory in chapters 1 through 9 and 15 treats the general case of an arbitrary look-ahead length; its main results are theorems on how the first- and follow-sets can be deduced from each reduced extended grammar, and criteria for the existence of parsers for a given grammar.

In chapter 1, the fundamental operations occurring often in the formulas later on, are introduced, and their main properties are listed.

In chapter 2 we recall the definition of regular expressions and chapter 3 introduces extended grammars and the regular derivation.

An important tool to investigate a grammar is a graph associated with it, representing the relations between the terminal and non-terminal symbols and the structure of the produced regular expressions. It is described in chapter 4; here, we also introduce some terms and abbreviations denoting diverse subsets of the vertex set of the graph and successors of a given vertex.

In chapter 5, the first-sets are defined and it is stated that they are least fixed points of a system of set equations.

The regular derivation of chapter 3 is not adequate to define the follow-sets, thus in chapter 6 we define another kind of derivation called K-derivation. Chapter 7 treats the relations between these two derivations.

In contrast to first-sets which can also be computed for non-reduced grammars, it is necessary that the grammar is reduced when the follow-sets are computed. Therefore we state in chapter 8 how to transform a non-reduced grammar into an equivalent reduced one.

In chapter 9, the follow-sets are defined and we see that they are also the least fixed point of a system of set equations.

In chapters 10 through 14 we give the efficient computation of first- and follow-sets in linear time relative to the product of the number of terminals and the size of the grammar.

In chapter 10, a less efficient algorithm, the general goal to improve it by eliminating the outmost iteration and the first steps in this direction are given.

Chapter 11 gives the overall-structure of the efficient algorithm, and chapters 12 through 14 its parts.

The real computation of first- and follow-sets in chapters 12 and 13 needs a precomputation postponed to chapter 14. It finds the non-terminals which derive the empty word and those which derive nothing at all at the same time; its complexity is proportional to the size of the grammar and it may also be used independently from the task of generating an LL-parser.

Chapter 15 treats the ELL(s) conditions guaranteeing that an LL-parser with look-ahead length s exists for a given grammar; and chapter 16 gives the structure of the ELL(1) - parser, describes how to generate it and compares it with the conventional LL-parser in [1]. In chapter 17, it is proved correct and we see that the time and the space the generated parser needs is linear in the length of the input the parser has to analyze.

Finally we give a table indicating which chapters of [2] correspond to the chapters here for readers who want to look there for proofs and further details:

Chapters here:	1 - 4	5 - 9	10	11
Chapters of [2]:	1 - 4	6 - 10	13	13, 15

Chapters here:	12	13	14	15	16	17
Chapters of [2]:	15	15	14	11, 12	16	16, 17

1. Operations on words and languages

To reason concisely about first- and follow-sets and equations between them, we introduce the first-mapping "F", the first-operation "o" and the first-closure "^{o}". "o" and "^{o}" are the image of concatenation and Kleene-closure "*" under the first-mapping (see theorem 1.8).

Let V denote a non-empty set (the alphabet) and V^{*} the set of words over V and let s be a fixed natural number, the length of look-ahead of the future ELL-parser.

The first-function maps a word onto the substring consisting of its first s symbols. It is simply denoted by "F" without explicitly specifying the parameter s.

Definition (1.1):

$$F: V^* \rightarrow V^*,\ F(\varepsilon) := \varepsilon,$$

$$F(a_1 \ldots a_l) := \begin{cases} a_1 \ldots a_l & \text{if } l \leq s \\ a_1 \ldots a_s & \text{if } l \geq s \end{cases}$$

for all $l \in \mathbb{N}$, $a_1, \ldots, a_l \in V$.

(ε denotes the empty word and $\mathbb{N}$ the set of natural numbers.)

The behavior of concatenation under the mapping F is described by the partially defined first-operation "o":

Definition (1.2):

Let $V^o = \{w \in V^*$ such that $|w| \leq s\}$
and $o: V^o \times V^o \rightarrow V^o$ be the operation defined by $u \circ v = F(uv)$.

(V^o, o) is a monoid with neutral element ε and F is a homomorphism from the monoid $(V^*, \cdot)$ to (V^o, o).

Now "F" and "o" can be obviously extended to languages:

Definition (1.3):

$F: P(V^*) \rightarrow P(V^*)$ (powerset of V^*),
for $L \subseteq V^*$: let $F(L) = \{ F(w)$ where $w \in L \}$.
$o: P(V^o) \times P(V^o) \rightarrow P(V^o)$,
for $L, M \subseteq V^o$, let $L \circ M = \{ u \circ v$ where $u \in L$ and $v \in M \}$.

We now give the main properties of the operation "o":

Prop. (1.4):

For all $L, M, K \subseteq V^o$ holds:

(a) $L \circ M \neq \emptyset$ iff $L \neq \emptyset$ and $M \neq \emptyset$;

(b) $\varepsilon \in L \circ M$ iff $\varepsilon \in L$ and $\varepsilon \in M$;

(c) $L \circ \{\varepsilon\} = \{\varepsilon\} \circ L = L$;

(d) $L \circ (M \circ K) = (L \circ M) \circ K$;

In the particular case s = 1, the operation "o" can be replaced by conventional operations of set theory:

Lemma (1.5):

$$\text{For } s = 1: \quad L \circ M = \begin{cases} \emptyset & \text{if } M = \emptyset \\ L & \text{if } M \neq \emptyset \text{ and } \varepsilon \text{ not in } L \\ (L - \{\varepsilon\}) \cup M & \text{if } M \neq \emptyset \text{ and } \varepsilon \text{ in } L \end{cases}$$

This lemma implies $L \circ M \subseteq L \cup M$ if $s = 1$. Therefore, $L \circ M$ contains only words which are already in L or in M, permitting the efficient computation of first- and follow-sets (see chapter 10 through 13). If $s > 1$ this property may not be true, e.g. $s = 2$ and $\{a\} \circ \{b\} = \{ab\}$.

In analogy to

$$L^* = \bigcup_{n=0}^{\infty} L^n$$

we want to obtain a closure operator in the monoid (V^o, o).

Definition (1.6):

For $L \subseteq V^o$: Let $P_0(L) = \{\varepsilon\}$ and $P_{n+1}(L) = P_n(L) \circ L$;

then let $L^o = \bigcup_{n=0}^{\infty} P_n(L)$.

The new definition of V^o is compatible with the old one ($V^o = \{w \in V^*$ such that $|w| \leq s\}$).

The closure can also be obtained by a finite union:

Theorem (1.7):

$$L^o = \bigcup_{n=0}^{s} P_n(L).$$

In particular for $s = 1$, $L^o = \{\varepsilon\} \cup L$ holds.

The main result of this chapter is the following theorem which characterizes the behavior of the regular operations union, concatenation and closure under the first-mapping:

Theorem (1.8):

For all $L, M \subseteq V^*$:

(a) $F(L \cup M) = F(L) \cup F(M)$;

(b) $F(L \cdot M) = F(L) \circ F(M)$;

(c) $F(L^*) = (F(L))^o$.

2. Regular expressions

In this chapter, we give a definition of regular expressions suitable for our purposes and some further terms concerning them.

Definition (2.1):

Let $M = \{'(', ')', '/', '*'\}$ be a set of metasymbols disjoint from V. The set RE(V) of regular expressions over the alphabet V is the least subset of $(V \cup M)^*$ such that

(a) $\varepsilon \in RE(V)$;

(b) $V \subseteq RE(V)$;

(c) $R, Q \in RE(V)$ implies $RQ \in RE(V)$;

(d) $R_1, \ldots, R_n \in RE(V)$ where $n > 1$ implies $(R_1 / \ldots / R_n) \in RE(V)$;

(e) $R \in RE(V)$ implies $(R)* \in RE(V)$.

Two regular expressions are called equal iff they are equal as strings over $V \cup M$. Thus (a/b) and (b/a) are different although they generate the same language $\{a, b\}$ (see chapter 3).

Due to rules (a) and (c) of the definition, RE(V) is a submonoid of $(V \cup M)^*$. A regular expression R is called reducible if it is the product of two non-empty expressions:

$$R = Q_1Q_2 \text{ where } Q_1, Q_2 \text{ in } RE(V) - \{\varepsilon\}.$$

It is called prime if it is different from ε and not reducible. Then we can prove that each non-empty expression can be written as product of uniquely defined prime expressions.

3. Extended grammar and regular derivation

In extended grammars, the non-terminal symbols produce regular expressions over terminals and non-terminals instead of strings of these symbols (see Def. 3.1). Since a grammar should generate a language, we must define what the language associated with a regular expression is. This is done by introducing a derivation on regular expressions (Def. 3.2). The language generated by an expression then is the set of terminal words derivable from it (Def. 3.3). Theorem (3.4) lists the main properties of the language mapping; it just transforms the syntactical operations "/", concatenation and "*" on regular expressions into the regular operations union, product and closure on languages.

Definition (3.1):

An extended context free grammar (sometimes called regular right part grammar) is a quadruple $G = (V_N, V_T, p, S)$.

V_N: a finite non-empty set (set of non-terminal symbols)

V_T: a finite non-empty set disjoint from V_N (set of terminal symbols)

p: $V_N \rightarrow RE$ where $RE := RE\ (V_N \cup V_T)$:
a mapping associating a regular expression with each non-terminal symbol

$S \in V_N$: the start symbol of the grammar.

For $A \in V_N$, p(A) is called the regular expression produced by A.

The productions may be written as a mapping because non-terminal symbols not occurring on the left side of at least one production can be omitted, and different productions with the same left side can be combined using alternation "/".

Usually, regular expressions are introduced together with the regular sets they denote. The elementary derivation steps induced by a production $A \rightarrow R$ are then declared as $A \rightarrow w$ where w is an element of the regular set denoted by R (see [4]).

The problem connected to this approach is that for any non-terminal A, there may be an infinite number of strings directly derivable from A, and that no LL-parser may be able to decide which one to select. Take $R = (a)* (b/c)$ as an example.

Therefore we prefer a different notion of derivation whose steps are finer and which can be simulated by a LL-parser (if the grammar satisfies some conditions).

Definition (3.2):

"$\rightarrow$" is a relation in RE called regular derivation (shorter R-derivation). It is recursively defined by:

(a) $\varepsilon \rightarrow R$ and $a \rightarrow R$ where $a \in V_T$ are always false.

(b) $A \in V_N$: $A \rightarrow R$ iff $R = p\ (A)$;

(c) $(R_1\ /\ \ldots\ /\ R_n) \rightarrow R$ iff $R = R_i$ for an $i \in \{1, \ldots, n\}$;

(d) $(Q)* \rightarrow R$ iff $R = \epsilon$ or $R = Q\ (Q)*$.

(e) Let $Q = R_1 \ldots R_n$, $n > 1$, where all the R_i are prime.

$Q \rightarrow R$ iff there is an $i \in \{1, \ldots, n\}$ such that

$R = R_1 \ldots R_{i-1}\ R'\ R_{i+1} \ldots R_n$ and

$R_1, \ldots, R_{i-1} \in V_T$ and $R_i \rightarrow R'$.

Note that the definition is recursive only in the last case, where $R_1 \ldots R_n \rightarrow R$ is traced back to $R_i \rightarrow R'$ which can be decided without further recursion since R_i is prime. The derivation is leftmost due to the condition "$R_1, \ldots, R_{i-1} \in V_T$".

Later, we shall see that R-derivation is suitable to define first-sets, but we need another kind of derivation to define follow-sets (see chapter 6).

Definition (3.3):

Let "$-*\rightarrow$" be the reflexive transitive closure of "$\rightarrow$". The language generated by a regular expression in RE is defined by

L: RE $\rightarrow P\ (V_T^*)$; for $R \in$ RE let $L(R) = \{w \in V_T^*$ such that $R -*\rightarrow w\}$.

The language described by the whole grammar is $L(S)$.

The following theorem states that the mapping "generated language" of this paper just transforms the syntactical operations on regular expressions into the regular operations on languages; therefore it is the same as the commonly defined one. Besides, we shall see that $L(R)$ for an expression R not containing non-terminal symbols is just the regular set denoted by R.

Theorem (3.4):

(a) $L(\epsilon) = \{\epsilon\}$;

(b) For $a \in V_T$: $L(a) = \{a\}$;

(c) For $A \in V_N$: $L(A) = L(\ p(A)\ )$;

(d) $L(\ (R_1\ /\ \ldots\ /\ R_n)\ ) = L(R_1) \cup \ldots \cup L(R_n)$;

(e) $L(R_1 \ldots R_n) = L(R_1) \cdot \ldots \cdot L(R_n)$;

(f) $L(\ (R)*\ ) = L(R)^*$.

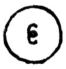
Now, we could define the first-set of an expression R as $F(L(R))$, but we prefer to define the first-sets for the nodes of the grammar graph introduced in the following chapter.

4. The grammar graph

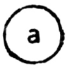
This chapter associates a graph with each grammar; it is the representation of the grammar in the parser generator, thus first- and follow-sets will be defined for its nodes (the idea is taken from [5]). At first, the structure of the grammar graph for a given fixed grammar G will be described informally; then an example will follow. Finally some notations are introduced concerning subsets of the set of nodes and successors of nodes.

Informal description of the grammar graph:
We associate a structure tree ST(R) with each R $\in$ RE.

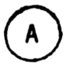
(a) $R = \varepsilon$: ST(R) =



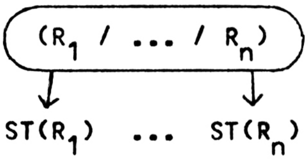
(b) $R = a \in V_T$: ST(R) =



(c) $R = A \in V_N$: ST(R) =

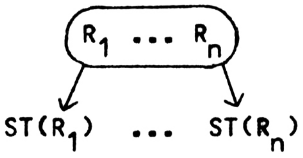


(d) $R = (R_1 / \ldots / R_n)$: ST(R) =

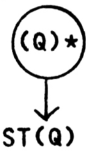


(e) $R = R_1 \ldots R_n$, $n > 1$: ST(R) =

where all R_i are prime



(f) $R = (Q)*$: ST(R) =



Vertices are labeled with regular expressions in cases (d), (e), and (f) only in the presentation of the algorithm. The implementation only requires labels standing for "/", "•", or "*".

If $V_N = \{A_1, \ldots, A_r\}$, the graph is created as follows:

1) Build the structure trees $ST(p(A_1)), \ldots, ST(p(A_r))$.

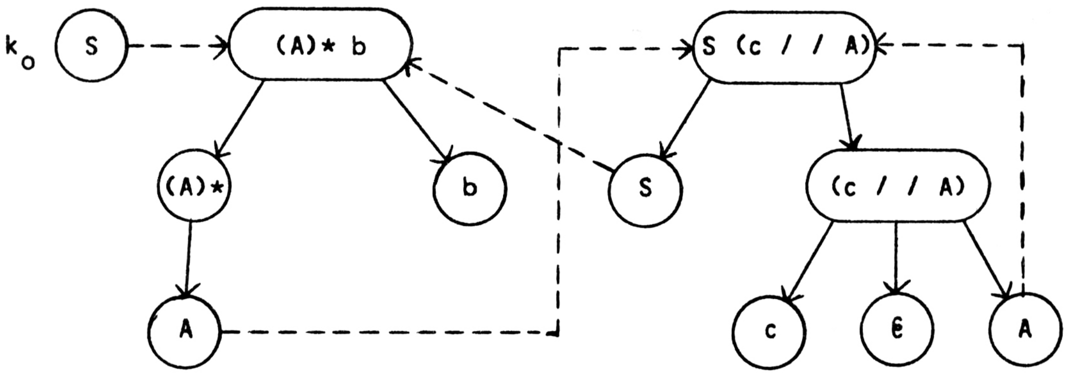
2) For all $A \in V_N$: Draw an edge from each node marked by A to the root

of ST(p(A)).

3) Add a special node k_o labeled with S and an edge from k_o to the root of ST(p(S)).

Example for a grammar graph:

V_N = {S, A}; V_T = {b, c};
Productions: S → (A)* b and A → S (c // A).



k_o has no predecessor, the roots of the trees ST(p(A)) for A ∈ V_N have as many predecessors as there are occurrences of A in the right sides of the productions. All the other nodes have exactly one predecessor.

Let K denote the set of the vertices in the grammar graph and M: K → RE the labeling function such that M(k) is the regular expression k is marked by.

The cardinality of K is in the order of the size of the grammar, more exactly $|K| \leq |G| + 1$ holds where |G| denotes the size of the extended context-free grammar G defined as the sum of the number of its productions and the numbers of terminals, non-terminals and metasymbols '(', ')', '/' and '*' occurring on the right hand sides of its productions.

K can be divided into several classes:

K_{eps} = {k ∈ K: M(k) = ε}	"ε - nodes"
K_{term} = {k ∈ K: M(k) ∈ V_T} = K_T	"terminal nodes"
K_{nont} = {k ∈ K: M(k) ∈ V_N}	"non-terminal nodes"
K_{alt} = {k ∈ K: M(k) = (R_1/.../R_n)}	"alternative nodes"
K_{prod} = {k ∈ K: M(k) = $R_1 \ldots R_n$, n>1, all R_i prime}	"product nodes"
K_{star} = {k ∈ K: M(k) = (R)*}	"star nodes"

The successors of a given node k are called $su_1(k)$, $su_2(k)$, ... from left to right, and their number is denoted by N(k). If there is only one successor (that is if k is a non-terminal or star node), the index '1' can be omitted.

5. The first-sets

We start with the definition of the first-sets for nodes in the grammar graph. After that, theorem (5.2) gives us the equations to compute the first-set of a node from the sets of its successors, and theorem (5.3) tells us that the first-sets are the least solution of the system of these equations relative to set inclusion.

The first-set of a node in the grammar graph is the set of the beginnings of the words in the language generated from the regular expression the node is marked by.

Definition (5.1):

$Fi: K \to P(V_T^0)$ (powerset of V_T^0),
let $Fi(k) = FLM(k)$ for all $k \in K$.

$$\begin{tikzcd} K \arrow[r, "M"] & RE \arrow[r, "L"] & P(V_T^*) \arrow[r, "F"] & P(V_T^0) \end{tikzcd}$$

M is the marking function (Chapter 4), L the generated language (Def. 3.3), F the first-function on languages (Def. 1.3).
The first-set of a node can be computed from the first-sets of its successors:

Theorem (5.2): (First-equations)

(a) For $k \in K_{eps} \cup K_{term}$: $Fi(k) = \{ M(k) \}$
The first-set of a ε-node is $\{\varepsilon\}$, and the first-set of a terminal node marked by 'a' is $\{a\}$.

(b) For $k \in K_{nont}$: $Fi(k) = Fi\ (su(k))$
The first-set of a non-terminal node is the same as the first-set of its successor.

(c) For $k \in K_{alt}$: $Fi(k) = Fi\ (su_1(k)) \cup \ldots \cup Fi\ (su_{N(k)}(k))$
The first-set of an alternative node is the union of the sets of its successors.

(d) For $k \in K_{prod}$: $Fi(k) = Fi\ (su_1(k)) \circ \ldots \circ Fi\ (su_{N(k)}(k))$
The first-set of a product node is the first-product (see chapter 1) of the sets of its successors.

(e) For $k \in K_{star}$: $Fi(k) = (Fi\ (su(k)))^{o}$
The first-set of a star node is the first-closure of the set of its successor.

The theorem is proved by (3.4) and (1.8).

Systems of set equations like the one of theorem (5.2) in general have more than one solution.

Theorem (5.3):

The first-sets are the least solution of the system of first-equations relative to set inclusion.

From (5.3) we see by applying a fixed point theorem that "Fi" can be computed by iterative applications of the first-equations starting from the configuration where the empty set is associated with each node. In chapter 10 we shall see that we can compute the first-sets more quickly without this iteration if the look-ahead length is 1.

6. The K-derivation

In the previous chapter we saw that the notion of R-derivation from chapter 3 was adequate to define first-sets. Now we try to define the follow-sets using R-derivation and show that this definition is inadequate. In the example showing this, sequences of words not consisting of symbols, but of nodes of the grammar graph are obtained. These sequences belonging to R-derivation paths become K-derivation paths by definition (6.3). In definition (6.4), the languages generated by K-derivation from words of nodes are defined, and the marking function and the first-set mapping are extended to words of nodes.

What is the formal definition of the follow-set of a vertex?

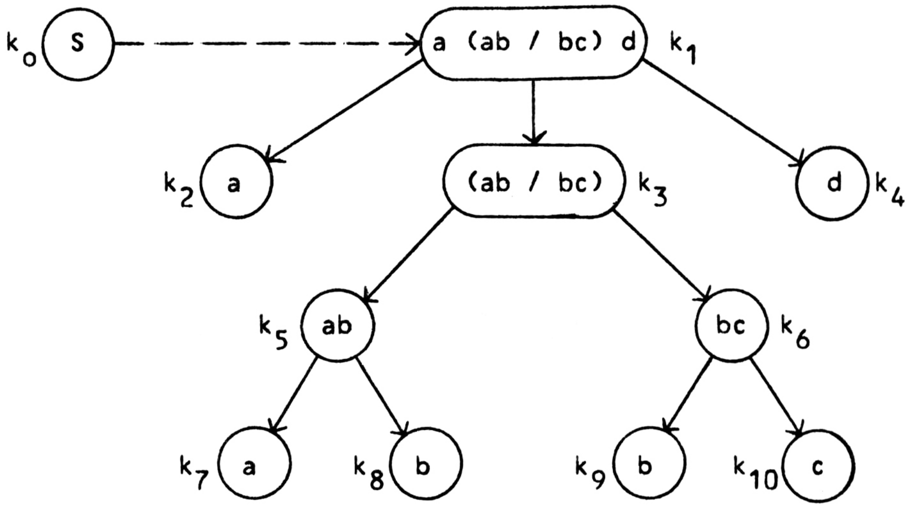
A first approach is

$$Fo(k) = \bigsqcup_{S \xrightarrow{*} R\ M(k)\ Q} FL(Q).$$

An example will show the inadequacy of this definition:

Example (6.1): $V_N = \{S\}$; $V_T = \{a, b, c, d\}$;   p(S) = a (ab / bc) d.

Graph:



What is $Fo(k_5)$?

The various derivations from S are S -*-> S, S -*-> a (ab/bc) d, S -*-> a $\underline{a\ b}$ d, S -*-> $\underline{a\ b}$ c d. Hence we obtain $Fo(k_5) = \{d, c\}$ by the definition above, although only the derivation S -*-> a a b d involves k_5 and thus we want to have $Fo(k_5) = \{d\}$. Additionally, these follow-sets are hard to compute because the substring 'ab' in the word 'abcd' results from k_2 and k_9 which are not local to k_5.

To make precise the correspondence between a derivation path and the vertices it involves, consider the following pictures:

Diagram (6.2):

$$\begin{tikzcd}
S \arrow[r, leftrightarrow] \arrow[d] & k_o \\
a\ (ab\ /\ bc)\ d \arrow[r, leftrightarrow] \arrow[dr, leftrightarrow] \arrow[dd] & k_1 \\
 & k_2\ k_3\ k_4 \\
a\ a\ b\ d \arrow[r, leftrightarrow] \arrow[dr, leftrightarrow] & k_2\ k_5\ k_4 \\
 & k_2\ k_7\ k_8\ k_4
\end{tikzcd}
\qquad
\begin{tikzcd}
S \arrow[r, leftrightarrow] \arrow[d] & k_o \\
a\ (ab\ /\ bc)\ d \arrow[r, leftrightarrow] \arrow[dr, leftrightarrow] \arrow[dd] & k_1 \\
 & k_2\ k_3\ k_4 \\
a\ b\ c\ d \arrow[r, leftrightarrow] \arrow[dr, leftrightarrow] & k_2\ k_6\ k_4 \\
 & k_2\ k_9\ k_{10}\ k_4
\end{tikzcd}$$

The relation indicated by '$\longleftrightarrow$' depicts the fact that the labels of the vertices on the right side written down consecutively result in the expression on the left side. Here, the order of the vertices corresponding to an expression is important, thus tuples or words of nodes are associated with each expression. The set of all such words is K^*.

The R-derivation path S → a (ab / bc) d → a a b d corresponds to a sequence of words of vertices, that is

$$k_o, \quad k_1, \quad k_2\, k_3\, k_4, \quad k_2\, k_5\, k_4, \quad k_2\, k_7\, k_8\, k_4.$$

This sequence becomes a derivation path from k_o to the terminal word $k_2\, k_7\, k_8\, k_4$ if a suitable derivation on K^* is introduced. Due to this K-derivation we will later obtain $Fo(k_5) = \{d\}$.

To make the following definition of K-derivation more transparent, we shall add to each case of the exact definition a supplement where a node marked by the regular expression R is denoted by [R]. These additions cannot replace the exact definition since there are in general many nodes having the same marking.

Definition (6.3): Let ">--" be a relation on K^*, the K-derivation.

Derivations from a single node:

$k \in K_{term}$: terminal nodes do not derive anything

$k \in K_{eps}$: k >-- ε [ε] >-- ε

$k \in K_{nont}$: k >-- su(k) [A] >-- [regular expression produced by A]

$k \in K_{prod}$: k >-- $su_1(k) \ldots su_{N(k)}(k)$

[$R_1 \ldots R_n$] >-- [R_1] ... [R_n]

$k \in K_{alt}$: k >-- $su_i(k)$ for each $i \in \{1, \ldots, N(k)\}$

[(R_1 / ... / R_n] >-- [R_i]

$k \in K_{star}$: k >-- ε and k >-- su(k) k

[$(R)^*$] >-- ε or >-- [R] [$(R)^*$]

Derivations from a word:

u >-- v iff there are $t \in K_T^*$, $k \in K$ and $x, w \in K^*$ such that

u = t k w, v = t x w and k >-- x.

Therefore u >-- v is possible only if u does not belong to K_T^*; at the transition from u to v, the leftmost non-terminal component of u is replaced by x.

Definition (6.4):

Let ">-*-" be the reflexive transitive closure of ">--" and

$L'(u) = \{t \in K_T^*$ such that u >-*- t$\}$ be the language generated from a word of nodes u.

Extension of M and Fi to K^*:

M: $K^* \rightarrow$ RE, let M be defined by

$M(\varepsilon) = \varepsilon$, $M(k_1 \ldots k_n) = M(k_1) \ldots M(k_n)$.

Fi: $K^* \rightarrow P(V_T^o)$, $Fi(u) = FLM(u)$ for $u \in K^*$.

Then for all u, v $\in K^*$: $L'(u\ v) = L'(u) \cdot L'(v)$, $M\ (u\ v) = M(u)\ M(v)$ and $Fi\ (u\ v) = Fi(u) \circ Fi(v)$ hold.

7. Relations between R- and K-derivation

R- and K-derivation are similar but they are defined on different domains; R-derivation on RE which is a monoid according to (2.1), but has also a more complex structure, K-derivation on K^* which is also a monoid, but not more. The marking function forms a monoid homomorphism from K^* into RE and it is a derivation homomorphism, too:

Theorem (7.1):

For all u, v $\in K^*$, u >-*- v implies M(u) -*-> M(v).

The theorem states that each K-derivation path can be transformed in a R-path by means of the marking function M. The inverse does not hold; take the example of chapter 6. Here $M(k_8)$ = b -*-> $M(k_9)$ = b holds, but k_8 >-*- k_9 does not.

The following theorem gives a restricted inversion of the claim of theorem (7.1):

Theorem (7.2):

For all u $\in K^*$ and R $\in$ RE, M(u) -*-> R implies the existence of a word v $\in K^*$ such that M(v) = R and u >-*- v.

It implies that each R-derivation path starting from the marking of a word of nodes can be simulated by a K-path starting from this word. Theorem (7.1) and (7.2) are illustrated by diagram (6.2). Note that a K-derivation path is in general longer than the corresponding R-path.

Languages defined by K-derivation are essentially the same as those defined by R-derivation:

Theorem (7.3):

For all $u \in K^*$, $LM(u) = ML'(u)$ holds.
(The mapping "M" on the right hand side of the formula is extended to the powerset of K^*.)

Due to this theorem, the first-sets are also definable by means of the K-derivation as $Fi(k) = FML'(k)$.

8. Non-reduced grammars

Up to this point, the considered grammars did not have to be reduced, the theorems of the first-sets also hold for non-reduced grammars. But to compute the follow-sets, the grammar must be reduced (see chapter 9). Therefore we now treat the transformation of a non-reduced grammar into an equivalent reduced one with respect to the generated language.

A non-extended grammar is commonly said to be reduced if all non-terminal symbols are productive (i.e. derive terminal words) and reachable (i.e. occur in a sentential form derived from the start symbol). Here, these terms are defined for nodes in the grammar graph using K-derivation:

Definition (8.1):

A vertex k is called unproductive if $L'(k) = \emptyset$ (or equivalent $LM(k) = \emptyset$ or $Fi(k) = \emptyset$).

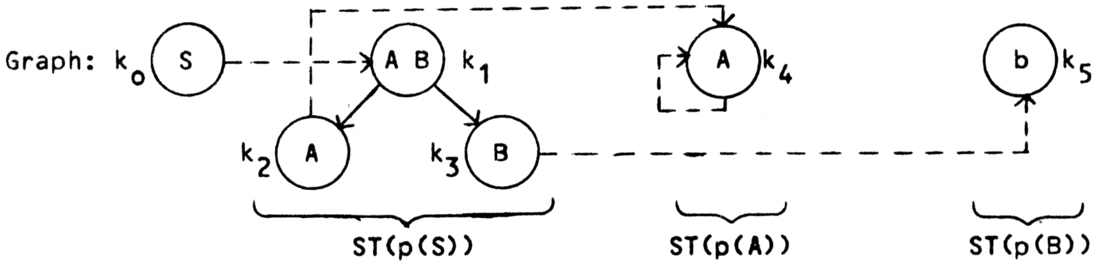
It is K-unreachable if there are no words $u, v \in K^*$ such that k_0 >-*- u k v.

It is G-unreachable if there is no path in the grammar graph from k_0 to k.

A grammar is reduced if there are no unproductive or K-unreachable nodes in its graph.

If a node is G-unreachable, then it is also K-unreachable. The inverse does not generally hold:

Example: Let V_T = {b}, V_N = {S, A, B}, p(S) = A B, p(A) = A, p(B) = b.



(k_o, k_1, k_3, k_5) is a graph path from k_o to k_5. The derivations of k_o are k_o >-- k_1 >-- k_2 k_3 >-- k_4 k_3 >-- k_4 k_3 >-- ..., thus there are no words u, v ∈ K^* such that k_o >-*- u k_5 v.

If there are no unproductive vertices in the graph, G- and K-unreachability is the same which allows for finding out the unreachable nodes by graph algorithms.

We now give an algorithm to transform a grammar into an equivalent reduced one. Starting from the original grammar, compute at first the predicate "Fi(k) ≠ ∅" for all nodes k of the grammar graph by the algorithm given in chapter 14. Then delete all subtrees of the structure trees in the graph which have unproductive roots, and finally eliminate all structure trees having unreachable roots.

The resulting graph represents a reduced grammar. At its nodes, it has the same languages and first-sets as the original graph.

9. The follow-sets

In this chapter, the follow-set of a node is defined by means of K-derivation; theorem (9.2) gives the equations to compute the follow-set of a vertex from the sets of its predecessors, and theorem (9.3) states that the follow-sets are the least solution of these equations.

Definition (9.1):

Follow-sets Fo: K → P(V_T^o):

For each node k ∈ K let Fo(k) = { x ∈ V_T^o | there are u, v ∈ K^* such that k_o >-*- u k v and x ∈ Fi(v) },

or shortly $\text{Fo}(k) = \bigsqcup_{k_o \text{ >-*- } u\, k\, v} \text{Fi}(v)$.

For the follow-rules, we need some further notations:

K_{root} = set of roots of the structure trees (including k_o),
K_{int} = $K - K_{root}$ = set of internal vertices in the trees,
$prs(k)$ for $k \in K_{root}$ = set of all predecessors of k,
$pr\ (k)$ for $k \in K_{int}$ = the unique predecessor of k.

Theorem (9.2): (Follow-equations)

If the considered grammar is reduced, then

(a) $Fo(k_o) = \{\varepsilon\}$.
The follow-set of the start node is $\{\varepsilon\}$.

(b) If $k \in K_{root} - \{k_o\}$: $Fo(k) = \bigsqcup_{k' \in prs(k)} Fo(k')$.
The follow-set of a root is the union of the sets of its predecessors. These are non-terminal nodes marked by the non-terminal that produces the regular expression the root belongs to.

(c) If $k \in K_{int}$ and $pr(k) \in K_{alt}$: $Fo(k) = Fo(pr(k))$.
The follow-set of a successor of an alternative node is the same as the set of the node itself.

(d) If $k \in K_{int}$ and $pr(k) \in K_{star}$: $Fo(k) = Fi(pr(k)) \circ Fo(pr(k))$.
The follow-set of the only successor k of a star node is obtained by the first-product "o" of the first-set and the follow-set of the star node.

(e) If $k \in K_{int}$ and $pr(k) \in K_{prod}$ and $k = su_i(pr(k))$:
$$Fo(k) = \underbrace{Fi(su_{i+1}(pr(k))) \circ \ldots \circ Fi(su_{N(pr(k))}pr(k))}_{Fi\ (rs(k))} \circ Fo(pr(k)).$$
The follow-set of a successor of a product node is the first-product of the first-set of the right siblings of the node - "Fi(rs(k))" - and the follow-set of the product node. Fi(rs(k)) is $\{\varepsilon\}$ for the rightmost successor of the product node.

Remark:

The follow-equations do not hold in general if the grammar is not reduced. Take the example in chapter 8. Here $Fo(k_3) = \{\varepsilon\}$ holds, but $Fo(k_5) = \emptyset$ since there are no words $u, v \in K^*$ such that k_o >-*- u k_5 v. This is a contradiction to equation (b).

Theorem (9.3):

The follow-sets are the least solution of the system of set-equations of theorem (9.2) relative to inclusion.

Theorem (9.3) implies that the follow-sets can also be computed by iterative applications of a system of set equations. There is an algorithm explained in chapters 10 through 14 without this iteration if the length of look-ahead is 1.

10. Simple algorithms to compute first- and follow-sets

From chapter 5 and chapter 9 we can derive simple algorithms to compute first- and follow-sets. In this chapter we present these algorithms and give the fundamental idea to improve them if the look-ahead length is 1. In the following four chapters the improved algorithms are explained.

In the algorithms and the comments belonging to them, we use the expressions "Fi(k)" and "Fo(k)" to denote the abstract first- and follow-set of the node k, and "FI[k]" and "FO[k]" for the variables containing the current values of first- and follow-sets during the algorithms. If the algorithms are correct, FI[k] = Fi(k) and FO[k] = Fo(k) must hold after their termination.

Now the simple algorithms follow in a notation similar to PASCAL:

Algorithm (10.1): (First-computation for all look-ahead values)

```
var S:      set of terminals (including $\varepsilon$);
    CHANGE: boolean;
begin for all k in K do FI[k] := $\emptyset$;       (* initialization *)
    repeat CHANGE := false;
       for all k in K do begin
          case k is in
               $K_{term}$:      S := {M(k)};
               $K_{eps}$ :      S := {$\varepsilon$};
               $K_{nont}$:      S := FI[su(k)];
               $K_{alt}$ :      S := FI[$su_1$(k)] U ... U FI[$su_{N(k)}$(k)];
               $K_{prod}$:      S := FI[$su_1$(k)] o ... o FI[$su_{N(k)}$(k)];
               $K_{star}$:      S := FI[su(k)]$^o$;
          end; (* case *)
          CHANGE := CHANGE or (S $\neq$ FI[k]);
          FI[k]  := S
       end (* for *)
    until not CHANGE
  end;
```

In the algorithm, the first-equations of theorem (5.2) are evaluated iteratively beginning from empty sets until no further changes occur. In the following we shall write such algorithms like this:

ALGORITHM (10.1):
(First-set computation)

```
for all k in K do FI[k] := $\emptyset$;
evaluate the equations
   given in theorem (5.2)
until no further changes occur
```

ALGORITHM (10.2):
(Follow-set computation)

```
for all k in K do FO[k] := $\emptyset$;
evaluate the equations
   given in theorem (9.2)
until no further changes occur
```

If the length of look-ahead is 1, we may replace the first-closure "$FI[su(k)]^{o}$" by "$\{\varepsilon\} \cup FI[su(k)]$" due to theorem (1.7). The time complexity of the set operations "U" and "o" then is $O(|V_T|)$ since all sets occurring here are subsets of $V_T \cup \{\varepsilon\}$. Thus the complexity of the for-loop containing the case-statement is $O(|K|\cdot|V_T|)$ and therefore the whole algorithms need time $O(c\cdot|K|\cdot|V_T|)$ where c is the number of executions of the outmost loop. Since there are grammars where the number c is about $|V_T|$ or even about $|K|$, the complexity of the algorithms can be significantly reduced to $O(|K|\cdot|V_T|) = O(|G|\cdot|V_T|)$ by removing the outmost iteration ("fixed point iteration").

This is our goal for the remaining of this chapter and the following ones. It is reached by eliminating the first-product "o" from the algorithms so that "U" is the only remaining operation, and traversing the graph while observing dependencies among the sets to be united.

By lemma (1.5), we obtain

$$L \circ M = \begin{cases} \emptyset & \text{if } M = \emptyset \\ L & \text{if } M \neq \emptyset \text{ and } \varepsilon \text{ not in } L \\ (L - \{\varepsilon\}) \cup M & \text{if } M \neq \emptyset \text{ and } \varepsilon \text{ in } L \end{cases}$$

Substitution of "o" in the algorithms by the right hand side of this law would yield equations only containing "$- \{\varepsilon\}$" and union as operations, but at each node the choice which operation is to do would depend on the current values of the operands of the former occurrences of "o". But to remove fixed-point iteration we need set equations merely containing union and merely depending on the actual node but not on the current values of first- or follow-sets. We call such equations invariant (towards fixed-point iteration).

As a first step in this direction we introduce an easier operation "o'":

Definition (10.3):

Let o': $P(V_T^o) \times P(V_T^o) \rightarrow P(V_T^o)$ be the operation defined by

$$L \text{ o' } M = \begin{cases} L & \text{if } \varepsilon \text{ not in } L \\ (L - \{\varepsilon\}) \cup M & \text{if } \varepsilon \text{ in } L. \end{cases}$$

"o'" has the same properties as "o" (see Prop. 1.4) except of (a). We can now simplify the algorithms:

Theorem (10.4):

Let the considered grammar be reduced. If in the algorithms (10.1) and (10.2) the operation "o" is replaced by "o'", they remain correct.

Now the problem to obtain invariant follow-equations is solved since the decision which case of definition (10.3) is to be chosen only depends on the left operand of "o'", and from the follow-equations we see that in the follow-set computation these left operands are always first-sets. If we assume that the first-set computation is finished when the follow-set computation is begun, we can decide for each node of the grammar graph before the fixed-point iteration starts which case of definition (10.3) is valid for this node. The resulting algorithm (12.2) is given at the beginning of the next but one chapter; it will there be further improved by eliminating the fixed point-iteration.

But we have not yet invariant first-equations since there are first-sets being left operands of "o'" (see Alg. 10.1). They become invariant if we precompute before first-computation starts which first-sets contain ε since this is just the information needed in Def. (10.3). This precomputation is postponed until chapter 14; in the next but one chapter we give the first- and follow-algorithms achieved so far and improve them further. Chapter 11 gives the general idea of this improvement.

11. Ideas to improve the algorithms

The previous chapter ended with algorithms to compute first- and follow-sets using fixed-point iteration of invariant equations. For each node there is a first- and a follow-equation containing union as the only operation, and not depending on the current value of the sets during the iteration. First- and follow-sets are the least solutions of these systems of equations which are listed at the beginning of the following chapter.

In this chapter and in the next two chapters, we explain how to solve these equations without fixed-point iteration. Remember that the grammar graph consists of structure trees linked by edges from some leaves to roots of trees. Therefore we can reduce the number of equations by combining all equations in a tree to one equation for the root containing only first- and follow-sets of other roots on the right hand side.

This is done in the next chapter, and in the following one, we solve these systems of root equations by a graph algorithm and thus obtain the first- and follow-sets at the roots of the trees in the grammar graph. Then it is easy to compute the sets at the other nodes. The resulting algorithm for the first- and follow-set computation will be as follows:

1) Compute which first-sets contain ε (chapter 14);
2) Generate the first-equations for the roots (chapter 12);
   Compute the first-sets at the roots (chapter 13);
   Compute the other first-sets (chapter 13);
3) Generate the follow-equations for the roots (chapter 12);
   Compute the follow-sets at the roots (chapter 13);
   Compute the other follow-sets (chapter 13);

The complexity of part (1) is $O(|K|)$, all the other computation steps need time $O(|V_T|\cdot|K|) = O(|V_T|\cdot|G|)$.

It would also be possible to apply the graph algorithm of chapter 13 used to compute the sets at the roots, directly to all equations without having reduced their number. But this would be less efficient than the method proposed here since the graph algorithm is rather complex.

12. Reducing the number of equations

In this chapter we start with first- and follow-algorithms using fixed-point iteration of invariant equations which were found in chapter 10. Then a common form for all equations is obtained, and the equations in the structure trees are combined to one equation for the root.

Let $Ep(k) = (\varepsilon \text{ in } Fi(k))$ be the precomputed predicate (see chapter 14) and EP[k] the corresponding program variable. Knowing Ep(k), the sets $Fi'(k) = Fi(k) - \{\varepsilon\}$ are still to compute.

Algorithm (12.1):

```
(first-computation using fixed-point iteration of invariant equations;
compare with algorithm (10.1))
  for all k in K do FI'[k] := $\emptyset$;
  evaluate the following equations
```

$$FI'[k] = \begin{cases} \emptyset & \text{if } k \in K_{eps} \\ \{M(k)\} & \text{if } k \in K_{term} \\ FI'[su(k)] & \text{if } k \in K_{nont} \cup K_{star} \\ FI'[su_1(k)] \cup \ldots \cup FI'[su_{n(k)}(k)] & \text{if } k \in K_{prod} \cup K_{alt} \end{cases}$$

```
  until no further changes occur.
```

For alternative nodes, the number n(k) is the number of all successors of k, and for product nodes k, n(k) is a number between 1 and the number of all successors depending on the Ep-values of the successors of k according to the definition of "o'".

Algorithm (12.2):

```
(follow-computation using fixed-point iteration of invariant equations;
compare with theorem (9.2))
  For all k in K do FO[k] := $\emptyset$;
  for each successor k of a product node compute the first-set of its
  right siblings Fi(rs(k)) and derive Fi'(rs(k)) = Fi(rs(k)) - {$\varepsilon$} from
  it;
  evaluate the following equations
     (a) $FO[k_o] = \{\varepsilon\}$,
     (b) If k is root of a structure tree: $FO[k] = \bigcup_{k' \in prs(k)} FO[k']$,
     (c) If k is successor of an alternative node k':
                                              $FO[k] = FO[k']$,
```

(d) If k is successor of a star node k':
$$FO[k] = FI'[k'] \cup FO[k'],$$

(e) If k is successor of a product node k':

(e1) if ε in Fi(rs(k)): $FO[k] = Fi'(rs(k)) \cup FO[k'],$

(e2) if ε not in Fi(rs(k)): $FO[k] = Fi'(rs(k))$

until no further changes occur.

Now we look for a common form of first- and follow-equations. In all equations, the set of a node results from some constant sets - e.g. {M(k)} in the first-equations or the first-sets in the follow-equations - and the sets of some successors or predecessors of the node. If we write "f" for "Fi'" resp. "Fo", we obtain as common form for all equations

Equations (12.3):

$$f(k) = g(k) \cup \bigsqcup_{k' \in h(k)} f(k')$$

where g(k) denotes the constant set and h(k) is the set of those successors/predecessors of k whose Fi'/follow-sets are united together with the constant set to build the Fi'/follow-set of k. From the invariant equations of algorithm (12.1) and (12.2), we obtain the sets g(k) and h(k):

(a) From the Fi'-equations:

$$g(k) = \begin{cases} \{M(k)\} & \text{if k is terminal node} \\ \emptyset & \text{otherwise} \end{cases}$$

and

$$h(k) = \begin{cases} \emptyset & \text{if k is terminal or } \varepsilon\text{-node} \\ \{su(k)\} & \text{if k is non-terminal or star node} \\ \{su_1(k), \ldots, su_{n(k)}(k)\} & \text{if k is alternative or product node} \end{cases}$$

(b) From the follow-equations:

$$g(k) = \begin{cases} \{\varepsilon\} & \text{in case (a)} \\ \emptyset & \text{in case (b) and (c)} \\ Fi'(pr(k)) & \text{in case (d)} \\ Fi'(rs(k)) & \text{in case (e)} \end{cases}$$

and

$$h(k) = \begin{cases} \emptyset & \text{in case (a)} \\ prs(k) & \text{in case (b)} \\ \{pr(k)\} & \text{in case (c), (d) and (e1)} \\ \emptyset & \text{in case (e2)} \end{cases}$$

Now the equations for the individual nodes in the trees are combined to one equation for the root of the tree. Thus we shall reduce the number of set equations from $|K|$ to $|K_{root}| = |V_N| + 1$.

In the following, let T(r) be the tree associated with the root r, that is the structure tree of the regular expression on the right hand side of the production belonging to r.

Reducing the number of Fi'-equations:

To combine the first-equations of T(r), take equation (12.3) for the root r and substitute f(k') in it by the right hand side of the equation for k' if k' is not a non-terminal node. Repeat that until an equation is obtained having only first-sets of non-terminal nodes which are leaves of T = T(r) on the right side. Formally let T' be the subtree of T which contains all those nodes that influence the first-set of the root; those are the nodes $k \in T$ that are reachable from r by a path $(r = k_1, k_2, \ldots, k_l = k)$ where $k_{i+1} \in h(k_i)$ for all $1 \le i < l$. Then we obtain the equation

$$Fi'(r) = \bigsqcup_{k \in T'} g(k) \cup \bigsqcup_{k' \in K_{nont} \text{ and leaf of } T'} Fi'(k')$$

by the described combination process.

Since Fi'(k) = Fi'(su(k)) holds for non-terminal nodes k, we can substitute Fi'(k') where k' is a non-terminal leaf of T' by Fi'(r') where r' = su(k') is a root of another structure tree. Then we obtain

Equation (12.4):

$$Fi'(r) = G(r) \cup \bigsqcup_{r' \in H(r)} Fi'(r')$$

where $G(r) = \bigsqcup_{k \in T'} g(k) = \{ M(k) \mid k \text{ is terminal node in } T' \}$ and

$H(r) = \{ r' \in K_{root} \mid \text{a predecessor of } r' \text{ is leaf of } T' \}$.

Equation (12.4) claims how the first-set of a root depends on the first-sets of the other roots. Since we want to solve the resulting system of root equations by a graph algorithm in the next chapter, we represent it as a digraph called Fi'-graph whose vertices are the elements of K_{root} and which contains an edge (r, r') iff r' is in H(r). Each vertex r of the graph is marked by the set G(r).

The Fi'-graph can be constructed by going through T'(r) for each root $r \in K_{root}$; by looking at the EP values we can decide which successors of the actual node belong to T'. The algorithm reaches each node at most once; the union in it (to compute G(r)) has complexity $O(|V_T|)$ since $G(r) \subseteq V_T^o = \{\varepsilon\} \cup V_T$ holds. Therefore the total complexity is $O(|K|+|K_T|\cdot|V_T|) = O(|V_T|\cdot|K|)$.

Reducing the number of follow-equations:

The follow-equations for roots $r \in K_{root} - \{k_o\}$ look like

$$Fo(r) = \bigsqcup_{k' \in prs(r)} Fo(k') \qquad (1)$$

where the nodes k' are non-terminal nodes and leaves of some trees. For all nodes except the roots, the follow-equations look like

$$Fo(k) = g(k) \quad (2) \qquad \text{or } Fo(k) = g(k) \cup Fo(pr(k)) \quad (3).$$

If we substitute Fo(k') in (1) by the right hand side of the equation for Fo(k'), we can do so until we reach an Fo(k) where k is a node whose equation looks like (2) or k is the root of the tree. Thus we obtain

Equation (12.5):

$$Fo(r) = G(r) \cup \bigsqcup_{r' \in H(r)} Fo(r')$$

where $G(r) \subseteq V_T^o$ is the union of all sets g(k) obtained by the substitutions and $H(r) \subseteq K_{root}$ is the set of all roots we reach. The resulting system of equations is represented as a digraph called Fo-graph; it is defined like the Fi'-graph and can be computed by an algorithm given in [2] whose complexity is $O(|V_T|\cdot|K|)$.

13. Solving the root equations

In the previous chapter we obtained the systems of set equations (12.4) and (12.5) implying that the first- resp. follow-set of a root r of a structure tree in the grammar graph can be computed from the union of a constant set G(r) associated with the root and the first- resp. follow-sets of some other roots building the set H(r). If we write "f" for "Fi'" and "Fo", these systems look like

$$(13.1)\qquad f(r) = G(r) \cup \bigcup_{r' \in H(r)} f(r') \text{ for all roots } r \in K_{root}$$

where $G(r) \subseteq V_T^o = \{\varepsilon\} \cup V_T$ is a set of terminal symbols and $H(r) \subseteq K_{root}$ is a set of roots.

Since there is no other difference between the first-system and the follow-system except in the sets G(r) and H(r), the same algorithm can be used to solve both systems. Thus we refer in the following to one system of set equations (13.1).

It is represented as a labeled digraph D = (V, E, G). Its vertex set V is the set K_{root} of roots of the structure trees in the grammar graph, its edge set is $E = \{(r, r') \mid r' \in H(r)\} \subseteq V^2$, and the vertices $r \in V$ are labeled with the sets G(r).

The equations of system (13.1)

$$f(r) = G(r) \cup \bigcup_{(r, r') \in E} f(r')$$

are in general mutually recursive, thus they cannot be solved by a single pass through V. By partitioning the digraph D into strongly connected components we can find an algorithm to solve system (13.1) efficiently.

Two vertices $r_1, r_2 \in V$ are called strongly connected iff there are paths in D from r_1 to r_2 and from r_2 to r_1. It is easy to prove that this is an equivalence relation. Its classes are called strongly connected components of the graph D.

By compressing the components in one vertex, we obtain the digraph D' = (V', E', G') where
V' is the set of strongly connected components of D,
$E' = \{(C, C') \in V'^2 \mid C \neq C'$ and there are $r \in C, r' \in C'$ such that

$(r, r') \in E\}$ is the set of all edges between different components and the label of a component is the union of the labels of all vertices in it: $G'(C) = \bigsqcup_{r \in C} G(r)$.

From system (13.1) we can infer that strongly connected nodes have the same f-set. If we write $f(C)$ for the common f-set of the nodes in the component C, we obtain as set equations

$$(13.2) \qquad f(C) = G'(C) \cup \bigsqcup_{(C,\, C') \in E'} f(C')$$

Although system (13.2) looks like (13.1), there is an important difference: digraph D' contains no cycles! Therefore we can solve system (13.2) directly without fixed-point iteration if the components are considered in a suitable sequence. There is an algorithm to compute the strongly connected components of any digraph (V, E) in time $O(|V|+|E|)$ (see [7]). In addition, the algorithm produces them suitably, since it does not output a component C before all end vertices of edges beginning in C were output in other components. The resulting algorithm is as follows:

Algorithm (13.3):

```
Repeat
    (a) call the graph algorithm to compute the next strongly connected component $C$;
    (b) compute $G'(C)$ by uniting all sets $G(r)$ of vertices $r$ in $C$;
    (c) compute $f(C)$ by equation (13.2) (the sets $f(C')$ are already known);
    (d) let $f(r) = f(C)$ for all vertices $r$ in $C$
until all components are found.
```

The complexity of the graph algorithm is $O(|V|+|E|) = O(|K_{root}|+|E|)$, the total number of unions in (b) is at most $|V| = |K_{root}|$ and in (c) at most $|E'| \leq |E|$; step (d) can be omitted if pointers are used. To each edge (r, r') in E, there is a path in the grammar graph from the root r to the root r'. This path must involve a non-terminal leaf, therefore $|E| \leq |K_{nont}|$ holds and $|K_{root}|+|E| \leq |K_{root}|+|K_{nont}| \leq 2|K|$. Each union takes time $O(|V_T|)$, thus the whole algorithm has complexity $O(|V_T| \cdot |K|)$.

When the first- resp. follow-sets at the roots are known, they are still to be computed for the other nodes in the grammar graph. The first-sets can be obtained by going bottom-up through each structure tree and the follow-sets by going top-down. Both computations need time $O(|V_T|\cdot|K|)$, too.

14. The computation of "Ep"

In this chapter, an algorithm will be given to compute which first-sets contain the empty word ϵ, at first informally, later formally. This information is needed for the efficient computation of first-sets given in the previous chapters. By a slight extension of the algorithm, we can also compute which nodes are unproductive, that is a part of the test whether the grammar is reduced.

The basis of the algorithm is the evaluation of the Ep-equations given below by fixed-point iteration. If the nodes are considered in a suitable sequence, only one iteration step is needed to obtain "Ep". Therefore the complexity of the algorithm will be $O(|K|)$.

For the description of the algorithm let Ep(k) = (ϵ in Fi(k)) be the value it must compute, and EP[k] be the actual value of a boolean variable associated with k. At the beginning, let EP[k] = 'false' for all $k \in K$, and at the end, EP = Ep must hold if the algorithm is correct.

The algorithm we start from is as follows:

Algorithm (14.1): (Computation of "Ep" using fixed-point iteration)

```
For all k in K do EP[k] := false;
evaluate the following equations
```

$$EP[k] = \begin{cases} \text{true} & \text{if } k \in K_{star} \cup K_{eps} \\ \text{false} & \text{if } k \in K_{term} \\ EP[su(k)] & \text{if } k \in K_{nont} \\ EP[su_1(k))] \text{ or } \dots \text{ or } EP[su_{N(k)}(k)] & \text{if } k \in K_{alt} \\ EP[su_1(k))] \text{ and } \dots \text{ and } EP[su_{N(k)}(k)] & \text{if } k \in K_{prod} \end{cases}$$

```
until no further changes occur.
```

Fixed-point theory implies that the EP value of a vertex can only change from 'false' to 'true'; if it is ever 'true' for a vertex it will never change later on. In addition, the value of a node can only change if the value of one of its successors has changed. Inversely, if the value of a node has changed we must test whether the values of its predecessors must be changed. Internal nodes have exactly one predecessor, therefore the test can immediately be executed. At roots, we must enter the predecessors in a worklist to test them one by one. This worklist is initialized with all nodes whose first-sets are known to contain ε, namely the ε and the star nodes.

In a main loop a node is taken from the worklist as actual node. Then an inner loop is entered, and the EP value of the actual node is changed to 'true'. If it is a root, its predecessors are entered into the worklist and the inner loop is terminated. Otherwise, we must test whether the EP value of the only predecessor remains 'false' or has already been 'true' (then the inner loop will be terminated) or is to be changed from 'false' to 'true' (then the predecessor becomes the actual node and the inner loop will be repeated). If the inner loop terminates, the main loop is repeated and a new vertex is taken from the worklist. The main loop and also the whole algorithm are terminated if the worklist is empty.

To prove the algorithm correct and calculate its complexity, note that

1) if a node is ever taken from the worklist, it will never again be added to it,

2) if a vertex k becomes actual node, Ep(k) is 'true', but EP[k] has still been 'false' and is changed to 'true',

3) each node having 'true' as Ep value will become actual node at some time.

Now we consider the decision in the inner loop more precisely. Let k be the actual node which is an internal node in a tree, and let k' be its predecessor. Then k' is either an alternative or a star or a product node. EP[k] has just changed to 'true'.

Case k' is alternative node:

Then Ep(k') = 'true' holds. If EP[k'] is already 'true', the inner loop is terminated; if EP[k'] is still 'false', k' becomes actual node and the inner loop is repeated.

Case k' is star node:

Then k' either is still in the worklist, then EP[k'] is 'false', or it is already taken from it, EP[k'] is 'true' and the inner loop is to be terminated. If EP[k'] is still 'false', we terminate it, too, because k' is then still in the worklist.

Case k' is product node:

Since EP[k] was 'false' till now, EP[k'] is always 'false' in this case. It has to be changed to 'true' iff the EP values of all successors of k' are 'true'. To avoid the large-scale investigation of the successors of k', we associate a number NF[k'] denoting the quantity of successors having 'false' as EP value with each product node k'. At the beginning, all EP values are 'false', therefore NF[k'] must equal the number of all successors N[k']. When a product successor k is actual node, the number NF[k'] is decremented by one; if it is still greater than 0, the inner loop is terminated, otherwise k' becomes actual node and the inner loop is repeated.

Now the final algorithm follows in a notation similar to PASCAL:

Algorithm (14.2):

```
begin
  for all k in K do begin           (* This part can already be done   *)
    EP[k] := false;                 (* during the construction of the *)
    if k is product node            (* graph in the machine.           *)
      then NF[k] := N(k)            (*                                 *)
  end; (* for *)                    (* T denotes the worklist. It is   *)
  T := $K_{eps}$ U $K_{star}$;                (* filled with ε and star nodes.   *)
  while T ≠ ∅ do begin              (* Beginning of the main loop.     *)
    k := arbitrary element of T; T := T - {k};
    repeat EP[k] := true;           (* Beginning of the inner loop.    *)
      if k in $K_{root}$ then begin       (* 'stop' indicates whether the    *)
        stop := true;               (* inner loop is to be terminated.*)
        T := T U prs(k)  (*)        (* The predecessors of the root k *)
      end                           (* are added to the worklist.      *)
      else begin k' := pr(k);     (* k is internal node,            *)
        case k' is in             (* k' its predecessor.            *)
          $K_{alt}$ : stop := EP[k'];
          $K_{star}$: stop := true;
          $K_{prod}$: begin NF[k'] := NF[k'] - 1; stop := NF[k'] > 0 end
        end; (* case *)
        if not stop then k := k'         (* k' becomes actual node. *)
      end (* else *)                     (*                         *)
    until stop                           (* End of the inner loop.  *)
  end (* while *)                        (* End of the main  loop.  *)
end;
```

Since each node becomes actual node at most once, the inner loop is repeated at most $|K|$ times. All operations in it are of complexity $O(1)$ except the assignment (*). But each node is also added to T at most once, therefore the total complexity of all executions of (*) is $O(|K|)$, too.

By a slight extension of the algorithm we can also compute the 'non-empty' predicate $Ne(k) = (Fi(k) \neq \emptyset)$ needed for the test whether the grammar is reduced. "Ne" is the least fixed point of some Ne-equations equal to the Ep-equations except Ne(k) = 'true' for terminal nodes. In addition, $Ep \leq Ne$ holds since $\varepsilon \in Fi(k)$ implies $Fi(k) \neq \emptyset$. Therefore we can compute "Ep" together with a part of "Ne". If the worklist is empty, it is refilled by the set of terminal nodes and the algorithm is started again to compute the remainder of "Ne".

15. The ELL(s) conditions

After having discussed in the previous chapters how the first- and follow-sets can be efficiently computed if the length of look-ahead is 1, we now consider the ELL(s) conditions for general look-ahead length s before introducing a predictive ELL(1) parser in the next chapter.

As in conventional grammars, we must distinguish between weak and strong ELL(s) conditions. In addition, we can define ELL(s) either by R- or by K-derivation. Hence, many different ELL(s) classes result.

In this thesis, we restrict ourselves to the KELL(s) and the strong KELL(s) property; for "very strong" KELL(s) conditions and RELL(s) and strong RELL(s) properties and the relations between them, we refer to [2].

Definition (15.1): (KELL(s) and SKELL(s))

(a) A grammar is KELL(s) if for all u, v_1, v_2, $w \in K^*$ and $k \in K$

$$k_o \;{>}\text{-}{*}\text{-}\; u\,k\,w \begin{array}{l} {>}\text{--}\; u\,v_1\,w \\ \\ {>}\text{--}\; u\,v_2\,w \end{array} \qquad \text{and } v_1 \neq v_2 \text{ implies}$$

$Fi(v_1\,w)$ and $Fi(v_2\,w)$ are disjoint.

(b) A grammar is strongly KELL(s) or SKELL(s) if for all u_1, u_2, v_1, v_2, w_1, $w_2 \in K^*$ and $k \in K$

$$k_o \begin{array}{l} {>}\text{-}{*}\text{-}\; u_1\,k\,w_1 \;{>}\text{--}\; u_1\,v_1\,w_1 \\ \\ {>}\text{-}{*}\text{-}\; u_2\,k\,w_2 \;{>}\text{--}\; u_2\,v_2\,w_2 \end{array} \qquad \text{and } v_1 \neq v_2 \text{ implies}$$

$Fi(v_1\,w_1)$ and $Fi(v_2\,w_2)$ are disjoint.

Theorem (15.2):

Each SKELL(s) grammar is KELL(s). The inverse implication is not generally true if $s > 1$, but KELL(1) and SKELL(1) are equivalent.

The example showing the inverse is not true for $s = 2$ is essentially the same as the one showing LL(2) ≠ SLL(2) for conventional grammars.

There is a local criterion for SKELL(s) using first- and follow-sets:

Theorem (15.3): A reduced grammar has the SKELL(s) property iff

(a) for all $k \in K_{alt}$, $i, j \in \{1, \ldots, N(k)\}$ where $i \neq j$, $Fi(su_i(k)) \circ Fo(k)$ and $Fi(su_j(k)) \circ Fo(k)$ are disjoint and

(b) for all $k \in K_{star}$, $Fi(su(k)\,k) \circ Fo(k)$ and $Fo(k)$ are disjoint.

If $s = 1$, (b) can be replaced by:

(b') For all $k \in K_{star}$, $Fi(su(k))$ and $Fo(k)$ are disjoint and $Fi(su(k))$ does not contain ε.

Alternative and star nodes are the only nodes that are able to derive more than one word of nodes (see definition 6.3). An alternative node k may derive each of its successors $su_1(k)$, $su_2(k)$,... and the successor

$su_i(k)$ is to be chosen if the look-ahead is in the set $Fi(su_i(k))$ o $Fo(k)$. A star node k may either derive ϵ or su(k) k. The derivation to ϵ is to be selected if the look-ahead is in the follow-set of k, and the other if it is in the first-set Fi(su(k)) of the successor of k.

Finally we introduce some kind of unambiguity. Other kinds can be found in [2].

Definition (15.4):

A grammar is called "strongly K-unambiguous" or SKUA iff

$$k_o \mathrel{>\!\!-\!\ast\!-} u \begin{array}{l} \mathrel{>\!\!--} u_1 \mathrel{>\!\!-\!\ast\!-} t_1 \in K_T^* \\ \mathrel{>\!\!--} u_2 \mathrel{>\!\!-\!\ast\!-} t_2 \in K_T^* \end{array} \quad \text{where } u_1 \neq u_2 \text{ implies } M(t_1) \neq M(t_2).$$

KELL(s) implies SKUA, and this unambiguity implies that there is exactly one K-derivation path from k_o to a terminal word of nodes marked by a given word in the language of the grammar. This path is the one found by the ELL(1)-parser of the following chapter.

Summary of chapter 15: (for reduced grammars)

$$\begin{array}{l} \text{criterion (15.3 a and b)} \iff \text{SKELL(s)} \implies \text{KELL(s)} \\ \text{criterion (15.3 a and b')} \iff \text{SKELL(1)} \iff \text{KELL(1)} \end{array} \implies \text{SKUA}$$

16. The ELL(1) - parser

In this chapter, we first describe by means of an example how the ELL(1) parser works, then list its possible actions. It follows an algorithm to generate the parser. Then we give an example for a parser table and finally compare our parser with the predictive LL-parser in [1]. In chapter 17, the generated parser is proved correct and its time and space complexity is considered.

The parser finds a K-derivation path from k_o to a terminal word of nodes whose marking is the word to be analyzed by the parser (if such a path exists).

Let w be this word. We shall write u.v# to denote the parsing situation where w = u v, u is already consumed by the parser and the first symbol of v# is the look-ahead symbol; # is a pseudo terminal symbol indicating the end of the input.

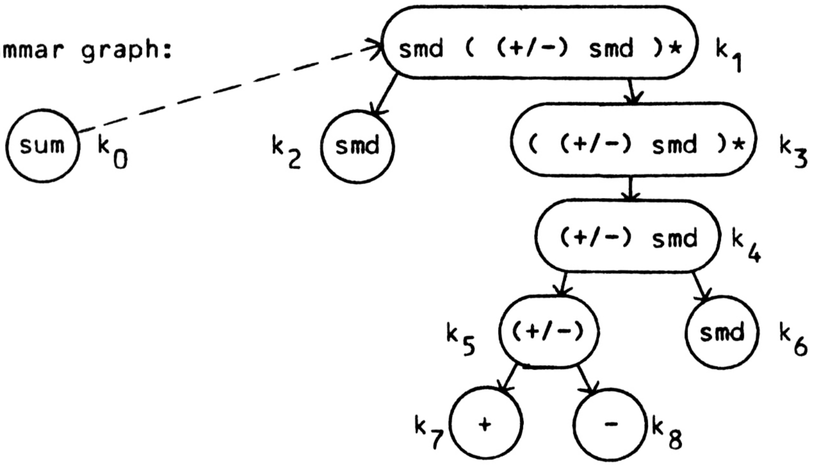
This situation can be represented by an item $t.xk_{\#}$ where t is a terminal word of nodes in K_T^* and x is a word in K^* such that M(t) = u and k_0 >-*- t x. $k_{\#}$ is a pseudo node not belonging to K considered as terminal node marked by #, thus its first-set is {#}. The parser does not contain the word t, but it has stored the word $xk_{\#}$ in its stack. A similar parser for conventional grammars is the predictive LL(1)-parser given in [1].

Since we introduced # and $k_{\#}$, we must replace ε in the follow-sets by #. In the first-sets, ε remains ε.

An example will show the working of the parser:

Example (16.1): Let V_N = {sum}, V_T = {smd, +, -} ("smd" = "summand") and p(sum) = smd ((+/-) smd)*. The parser shall analyze the input string 'smd + smd'.

Grammar graph:



In the example, we shall write a node marked by R as [R] (but note that there are two different nodes [smd] in our example!).

input		stack			executed action
	top = actual node			tail	
	ST [TOP]	...	ST[1]	ST[0]	
.smd + smd #			[sum]	[#]	expansion
.smd + smd #			[smd ((+/-) smd)*]	[#]	product expans.
.smd + smd #		[smd]	[((+/-) smd)*]	[#]	shift
smd.+ smd #			[((+/-) smd)*]	[#]	star expansion
smd.+ smd #		[(+/-) smd]	[((+/-) smd)*]	[#]	product expans.
smd.+ smd #	[(+/-)]	[smd]	[((+/-) smd)*]	[#]	select
smd.+ smd #	[+]	[smd]	[((+/-) smd)*]	[#]	shift
smd +.smd #		[smd]	[((+/-) smd)*]	[#]	shift
smd + smd.#			[((+/-) smd)*]	[#]	ε-shift
smd + smd.#				[#]	accept
smd + smd #.					

The nodes stored in the stack will be represented by integers; thus the stack ST is an array of integers. Let TOP be the index of the top of stack.

The parser action table associates an action code and at most one integer parameter with each pair of node number and look-ahead symbol. We now enumerate the possible parser actions and describe how the parser executes them.

Expansion and select both with parameter J:

The actual node is changed to J.

```
begin ST [TOP] := J end;
```

Shift without parameter:

The scanner reads a new look-ahead symbol and the actual node is popped from the stack.

```
begin SCAN; TOP := TOP - 1 end;
```

ε-shift without parameter:

No symbol is read and the actual node is popped.

```
begin TOP := TOP - 1 end;
```

Star_expansion with parameter J:

The node J is pushed onto the stack.

```
begin TOP := TOP + 1; ST [TOP] := J end;
```

Product expansion with parameter N:

The actual node is a product node which shall be replaced by the list of its successors. To avoid the necessity of more than one parameter, the numbers are given to the nodes in such a manner that if a product node k has number j, its successors have the following numbers j + 1, ..., j + N(k). This is possible since all successors of product nodes have prime markings and thus are never again product nodes. The number of successors N(k) then is the parameter of product expansion.

```
var I, J: integer;
begin J := ST [TOP];
  for I := 0 to N - 1 do ST [TOP + I] := J + N - I;
  TOP := TOP + N - 1
end;
```

There still are two further actions, namely "accept" and "error", both without parameters. The action "accept" can be considered as a special kind of "shift" involving # and $k_{\#}$.

Now an algorithm follows to generate the parser action table from the grammar graph and the first- and follow-sets. The table is here represented by a two-dimensional array

T: array [0 .. |K|, {#} U V_T] of (action, parameter).

Naturally, it is also possible to implement the table by nested case-statements and the action codes by procedure calls.

Algorithm (16.2):

Initialize the table by 'error' at each place.

Let at first T(|K|, #) = (accept). (Let |K| be the number of $k_{\#}$.)

Then pass through the grammar graph. Let k be the actual node and i its number; in addition, let j be the number of su(k) for non-terminal and star nodes, and j_r the number of $su_r(k)$ for alternative nodes.

Case k is non-terminal node:

T(i, ch) = (expansion, j) for all ch $\in$ Fi(k) o Fo(k).

Case k is terminal node: T(i, M(k)) = (shift).

Case k is ε node:

T(i, ch) = (ε-shift) for all ch $\in$ Fi(k) o Fo(k) = Fo(k).

Case k is product node:

T(i, ch) = (product expansion, N(k)) for all ch $\in$ Fi(k) o Fo(k).

Case_k_is_alternative_node:

For $r := 1$ to $N(k)$ do $T(i, ch) = (\text{select}, j_r)$ for all $ch \in Fi(su_r(k))$ o $Fo(k)$. If a conflict arises at this, the grammar is not ELL(1).

Case_k_is_star_node:

Let $T(i, ch) = (\text{star expansion}, j)$ for all $ch \in Fi(su(k))$ and $T(i, ch) = (\varepsilon\text{-shift})$ for all $ch \in Fo(k)$. If a conflict arises at this or if ε is in $Fi(su(k))$, the grammar is not ELL(1).

The parser driver works as follows:

```
ST[0] := |K|;           (* The stack is initialized by $k_o$ $k_{\#}$;          *)
TOP := 1; ST[1] := 0;   (* |K| is the number of $k_{\#}$, 0 the number of $k_o$ *)
repeat execute the action of T (ST [TOP], look-ahead symbol)
until 'accept' or 'error' are called.
```

As an example we give now the parser table belonging to the grammar of example (16.1).

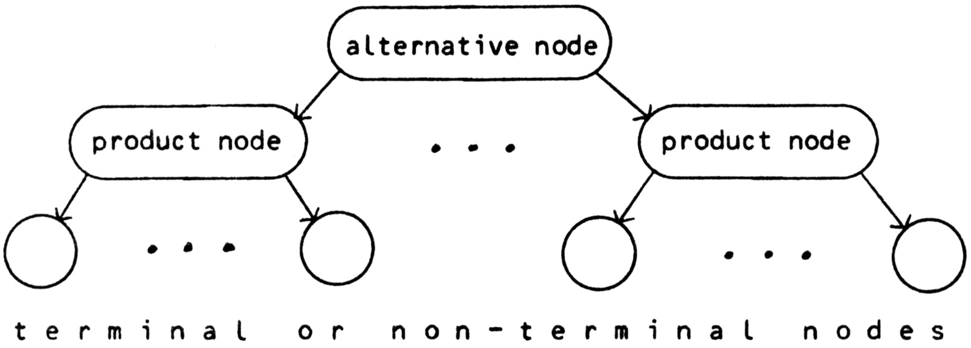
At first some information about the nodes in the grammar graph:

number	marking	class	first-set	follow-set
0	sum	non-terminal	{smd}	{#}
1	smd ((+/-) smd)*	product	{smd}	{#}
2	smd	terminal	{smd}	{+, -, #}
3	((+/-) smd)*	star	{+, -, ε}	{#}
4	(+/-) smd	product	{+, -}	{+, -, #}
5	(+/-)	alternative	{+, -}	{smd}
6	smd	terminal	{smd}	{+, -, #}
7	+	terminal	{+}	{smd}
8	-	terminal	{-}	{smd}
9	#	pseudo	{#}	

Parser table:

	smd	+	-	#
0	Expansion 1	Error	Error	Error
1	Product exp. 2	Error	Error	Error
2	Shift	Error	Error	Error
3	Error	Star exp. 4	Star exp. 4	ε-shift
4	Error	Product exp. 2	Product exp. 2	Error
5	Error	Select 7	Select 8	Error
6	Shift	Error	Error	Error
7	Error	Shift	Error	Error
8	Error	Error	Shift	Error
9	Error	Error	Error	Accept

If a conventional grammar were directly written as extended grammar, all productions from a non-terminal would be combined into one whose structure tree would generally look as follows:



The parser of Aho and Ullman in [1] has only two actions: 'shift' and 'expand'. While their action 'shift' is like ours, their action 'expand' starts from a non-terminal, pops it from the stack, selects the correct production by look-ahead and then pushes the produced string onto the stack. This would be done by our parser in three actions: 'expansion', 'select' and 'product expansion'. This splitting is inevitable because alternative expressions may be involved in other alternatives.

Since only strings of terminals and non-terminals are pushed on the stack of Aho and Ullman, their parser table entries need only be simple symbols while our entries are numbers denoting nodes standing for regular expressions. Thus our table directly produced from an extended grammar, is greater than the table of Aho and Ullman. On the other hand extended grammars are usually easier to construct and comprehend.

17. The correctness and complexity of the ELL(1) parser

At first, we prove that the generated ELL(1)-parser works correct, and then we consider the time and space complexity of this parser.

Theorem (17.1):

Let G be a reduced KELL(1) grammar. Then the parser generated from G is correct, i.e. it always terminates and accepts a word if and only if it is in L(G).

Proof: If an input word w belongs to L(G), there is a unique K-derivation path from k_o to a word of nodes u such that M(u) = w since w $\in$ L(G) = L(S) = LM(k_o) = ML'(k_o) (see chapter 7). The parser then executes as many actions as the length of this derivation path and accepts w at the end.

If w is not in L(G), it can be written as w# = w_1 t w_2 where w_1 is a prefix of a word in L(G) and t is a terminal (or #) such that w_1 t is not a valid prefix. The parser reads w_1 and then has a node k on the top of its stack whose look-ahead set Fi(k) o Fo(k) does not contain the next input symbol t. The analysis terminates since 'error' is called in this situation.

Now we shall see that the time and the space the parser needs to analyze an input is linear in the length of this input. At first, we define these terms exactly:

Definition (17.2):

(a) Let the time for the product expansion of a node k be N(k) and the time for any other action be 1. The time the parser needs to analyze an input is the sum of the times of the actions it must execute until 'accept' or 'error' are called.

(b) The space it needs is the maximal number of stack components occupied during the parsing.

Theorem (17.3):

The parser generated for the grammar G needs time O(c•n) and space O(|G|•n) to analyze an input word of n symbols where c depends only on G but is not in any polynomial relation to |G|.

Acknowledgements:

I would like to thank Reinhard Wilhelm who made the proposal to investigate the LL-parsing of extended grammars; I also wish to thank Dieter Maurer for many valuable discussions; in particular, he stimulated me to remove the fixed-point iteration from the computation of first- and follow-sets. Franz Geiselbrechtinger then read a preliminary version of this text, corrected it and told me how to make it clearer and more understandable.

References:

1. Aho, A., Ullman, J.: Principles of Compiler Design, Addison-Wesley, (1979), Chapter 5.5

2. Heckmann, R.: ELL(k) - Parsing and Efficient ELL(1) - Parser Generation, Diplomarbeit, Universität Saarbrücken (1984).

3. Heilbrunner, S.: On the definition of ELR(k) and ELL(k) grammars, Acta Informatica 11, 169 - 176 (1979).

4. Lewi, J., de Vlaminck, K., Huens, J., Steegmans, E.: A programming methodology in compiler construction, North-Holland (1982).

5. Möncke, U., Wilhelm, R.: Iterative algorithms on grammar graphs, Proceedings of the 8th conference on graphtheoretic concepts in computer science, 177 - 194, Hanser (1982).

6. Purdom, P. W., Brown, C. A.: Parsing extended LR(k) grammars, Acta Informatica 15, 115 - 127 (1981).

7. Tarjan, R.: Depth-first search and linear graph algorithms, SIAM J. Comput., Vol. 1, 146 - 160 (1972).