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 of 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 of 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 [4] 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 [7] show how to generate LR-parsers directly from extended context-free grammars. Another less efficient algorithm to generate LL-parsers can be found in [5].

This paper is a summary of [2]; some less important lemmata and almost all proofs have been omitted; the interested reader is referred to [2] and [3] for details. The length of look-ahead has been restricted to 1; the general case does not lead to efficient algorithms since Lemma (1.4) that is heavily used to derive the algorithms presented in this thesis, is not valid if the look-ahead is greater than 1.

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 question 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 through 3, the fundamental notions of the theory are introduced.

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.

After having shortly considered the notion of being reduced in chapter 8, the follow-sets are defined in chapter 9 and we see that they are also the least fixed point of a system of set equations.

In chapters 10 through 13 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.

This algorithm needs a precomputation described in chapter 14 that 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(1) conditions guaranteeing that an LL-parser with look-ahead length 1 exists for a given grammar; and chapter 16 gives the structure of the ELL(1) - parser and describes how to generate it. At last we shall see that the time and the space the generated parser needs is linear in the length of the input the parser has to analyze.

1. *Operations on words and languages*

To reason concisely about first- and follow-sets and equations between them, we here introduce the first-mapping "F" and the first-operation "o"; that is the image of concatenation under the first-mapping (see theorem 1.5).

Let V denote a non-empty set (the alphabet) and V^* the set of words over V.

Definition (1.1): (First-mapping)

$F: V^* \to V^*$, $F(\varepsilon) := \varepsilon$, $F\,(a_1 \cdots a_l) := a_1$

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

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

For $L \subseteq V^*$ let $F(L) = \{\ F(w)\ \textit{where}\ w \in L\ \}$.

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

Definition (1.2):

For $L, M \subseteq V \cup \{\varepsilon\}$ let $L\ o\ M = F\,(L \cdot M)$.

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

Prop. (1.3):

For all L, M, $K \subseteq V \cup \{\varepsilon\}$ holds:

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

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

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

(d) $L \; o \; (M \; o \; K) = (L \; o \; M) \; o \; K$.

The operation "o" can be replaced by conventional operations of set theory:

Lemma (1.4):

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

This lemma implies $L \; o \; M \subseteq L \cup M$. Therefore, $L \; o \; M$ 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).

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 (1.5):

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

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

(b) $F\,(L \cdot M) = F(L) \; o \; F(M)$;

(c) $F\,(L^*) \quad = \{\varepsilon\} \cup F(L)$.

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 $(R_1 \mid \cdots \mid R_n) \in RE(V)$;

(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 $(V \cup M)^*$. If we write $R = R_1 \cdots R_n$ later on, this notation shall imply that all R_i are different from ε and cannot be written as product of some smaller regular 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 [5]).

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 an 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) $A \in V_N$: $\quad A \rightarrow R \qquad$ *iff* $\;R = p\,(A)$;

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

(d) $(Q) * \rightarrow R \qquad$ *iff* $\;R = \varepsilon$ *or* $R = Q\,(Q)*$.

(e) $R_1 \cdots R_n \rightarrow R$ *iff there is an* $i \in \{1, \ldots, n\}$ *such that*

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

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

Note that 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 "$-*->$" 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* $R -*-> w\}$.

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

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

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

(d) $L(\,(R_1 \mid \cdots \mid R_n)\,) = L(R_1) \cup \cdots \cup L(R_n)$;

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

(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 [6]). 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) $R = \varepsilon$: $ST(R) =$ ε

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

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

(d) $R = (R_1 \mid \cdots \mid R_n)$: $ST(R) =$ $(R_1 \mid \cdots \mid R_n)$

$ST(R_1) \ \cdots \ ST(R_n)$

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

$ST(R_1) \ \cdots \ ST(R_n)$

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

$ST(Q)$

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 $ST(p(A_1)), \ldots, ST(p(A_r))$.

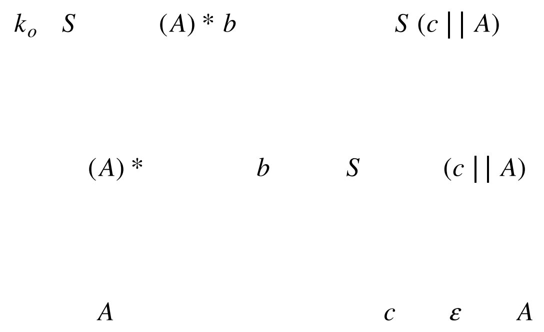
2) For all $A \in V_N$: Draw an edge from each node marked by A to the root of $ST(p(A))$.

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: $S \to (A) * b$ and $A \to S\ (c \parallel A)$.



k_o has no predecessor, the roots of the trees $ST(p(A))$ for $A \in V_N$ 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 $M\colon K \to RE$ the labeling function such that $M(k)$ is the regular expression k is marked by.

K can be divided into several classes:

$K_{eps} = \{k \in K\colon M(k) = \varepsilon\}$	"ε - nodes"
$K_{term} = \{k \in K\colon M(k) \in V_T\} = K_T$	"terminal nodes"
$K_{nont} = \{k \in K\colon M(k) \in V_N\}$	"non-terminal nodes"
$K_{alt} = \{k \in K\colon M(k) = (R_1 \mid \cdots \mid R_n)\}$	"alternative nodes"
$K_{prod} = \{k \in K\colon M(k) = R_1 \cdots R_n,\ n > 1\}$	"product nodes"
$K_* = \{k \in K\colon M(k) = (R) * \}$	"star nodes"

The successors of a given node k are called $su_1(k), su_2(k), \cdots$ 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: $K \rightarrow P(V_T \cup \{\varepsilon\})$ ($powerset\ of\ V_T \cup \{\varepsilon\}$),

$let\ Fi(k) = F(L(M(k)))$ for $all\ k \in K$.

$$K \xrightarrow{\quad M \quad} RE \xrightarrow{\quad L \quad} P(V_T^*) \xrightarrow{\quad F \quad} P(V_T \cup \{\varepsilon\})$$

M is the marking function (Chapter 4), L the generated language (Def. 3.3), F the first-function on languages (Def. 1.1).

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 *epsilon* –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 $k \in K_{alt}$: $Fi(k) = Fi\ (su_1(k)) \cup \cdots \cup Fi\ (su_{N(k)}(k))$

The first-set of an alternative node is the union of the sets of its successors.

(d) For $k \in K_{prod}$: $Fi(k) = Fi\ (su_1(k))\ o \cdots o\ Fi\ (su_{N(k)}(k))$

The first-set of a product node is the first-product (see chapter 1) of the sets of its successors.

(e) For $k \in K_*$: $Fi(k) = \{\varepsilon\} \cup Fi\ (su(k))$

The first-set of a star node is the set of its successor augmented by ε.

The theorem is proved by (3.4) and (1.5).

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 fixed-point iteration.

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

$$Fo(k) = \bigcup_{S \, -*-> \, R \; M(k) \; Q} F(L(Q)).$$

An example will show the inadequacy of this definition:

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

Graph: k_o S $a\ (ab \mid bc)\ d$ k_1

k_2 a $(ab \mid bc)$ k_3 d k_4

k_5 ab bc k_6

k_7 a k_8 b k_9 b k_{10} c

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):

S	k_o	S
	k_1	
$a\ (ab \mid bc)\ d$		$a\ (ab \mid bc)\ d$
	$k_2\ k_3\ k_4$	
	$k_2\ k_5\ k_4$	
$a\ a\ b\ d$		$a\ b\ c\ d$
	$k_2\ k_7\ k_8\ k_4$	

The relation indicated by '   ' 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 $S \rightarrow a\ (ab \mid bc)\ d \rightarrow a\ a\ b\ d$ 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

$k \in K_{eps}$: $k >\!-\!- \varepsilon$ | $[epsilon] >\!-\!- \varepsilon$

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

$k \in K_{prod}$: $k >\!-\!- su_1(k) \cdots su_{N(k)}(k)$ | $[R_1 \cdots R_n] >\!-\!- [R_1] \cdots [R_n]$

$k \in K_{alt}$: $k >\!-\!- su_i(k)$ for *each* $i \in \{1, \ldots, N(k)\}$ | $[(R_1 \mid \cdots \mid R_n)] >\!-\!- [R_i]$

$k \in K_*$: $k >\!-\!- \varepsilon$ and $k >\!-\!- su(k)\ k$ | $[(R)^*] >\!-\!- \varepsilon$ *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 left-most 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

$M(epsilon) = epsilon$, $M(k_1 \cdots k_n) = M(k_1) \cdots M(k_n)$.

Fi: $K^* \rightarrow P(V_T \cup \{\varepsilon\})$, $Fi(u) = F(L(M(u)))$ 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 $Fi\ (u\ v) = Fi(u)\ o\ Fi(v)$ 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 $u \in K^*$, $L(M(u)) = M(L'(u))$ 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 $Fi(k) = F(M(L'(k)))$.

8. *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. Therefore we now define the notion of being reduced and give a criterion for it.

Definition (8.1):

A grammar is called reduced if for all $k \in K$ there are words $u, v \in K^*$ and $t \in K_T^*$ such that $k_o > - * - u\ k\ v > - * - t$.

Theorem (8.2):

A grammar is reduced iff all first-sets are non-empty and each $k \in K$ is reachable from k_o by a path in the grammar 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: $K \to P(V_T \cup \{\varepsilon\})$:

For each node $k \in K$ let $Fo(k) = \{\ x \in V_T \cup \{\varepsilon\} \mid$ there are $u, v \in K^*$ such that $k_o > - * - u\ k\ v$ and $x \in Fi(v)\ \}$,

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

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 $k \in K_{root} - \{k_o\}$: $Fo(k) = \bigcup_{k' \in prs(k)} Fo(k')$.

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.

For (c) through (e), let k be in K_{int} and $k' = pr(k)$.

(c) If $k' \in K_{alt}$: $Fo(k) = Fo(k')$.

The follow-set of a successor of an alternative node is the same as the set of the node itself.

(d) If $k' \in K_*$: $Fo(k) = Fi(k')\ o\ Fo(k')$.

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 $k' \in K_{prod}$ and $k = su_i(k')$:

$Fo(k) = Fi(su_{i+1}(k'))\ o \cdots o\ Fi(su_{N(k')}(k'))\ o\ Fo(k')$.

$Fi(rs(k))$

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.

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. But there is an algorithm explained in chapters 10 through 14 computing follow-sets without this iteration.

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, and in the following four chapters the improved algorithms are explained.

Simple algorithms using fixed-point iteration:

ALGORITHM (10.1):	ALGORITHM (10.2):
(First-set computation)	(Follow-set computation)
for all k in K do Fi(k) := ∅;	for all k in K do Fo(k) := ∅;
evaluate the equations	evaluate the equations
given in theorem (5.2)	given in theorem (9.2)
until no further changes occur	until no further changes occur

The time complexity of the set operations "$\cup$" and "o" is $O(|V_T|)$ since all sets occurring here are subsets of $V_T \cup \{\varepsilon\}$. Thus the complexity of one evaluation of the equation system 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 evaluations of the equations. 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 $O(|K| \cdot |V_T|)$ 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 "$\cup$" is the only remaining operation, and traversing the graph while observing dependences among the sets to be united.

By lemma (1.4), we obtain

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

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': $P(V_T \cup \{\varepsilon\}) \times P(V_T \cup \{\varepsilon\}) \rightarrow P(V_T \cup \{\varepsilon\})$ be the operation defined by

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

"o'" has the same properties as "o" (see Prop. 1.3) 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 theorem 5.2). 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 on special dependence graphs 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 $O(|V_T| \cdot |K|)$.

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. For the computation of the first-sets, we would then have to visit each node of the grammar graph to construct the dependence graph, then this graph (which would have as many nodes as the grammar graph) would have to be passed, and at each node the actions of the graph algorithm would have to be executed.

In the method proposed here, there are also two passes through K to compute the first-sets, and in addition one pass through K_{root} when the graph algorithm is applied. Thus the two possibilities are equal in the order of complexity but differ in the constants: the method proposed here executes about $44|K| + 29|K_{nont}| + 52|K_{root}|$ PASCAL statements, and the other possibility would execute about $85|K| + 26|K_{nont}| + 6|K_{root}|$.

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 $Ep(k) = (epsilon\ in\ Fi(k))$ be the precomputed predicate (see chapter 14). 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) := ∅;

evaluate the following equations

$$Fi'(k) = \begin{cases} \emptyset & \text{if } k \in K_{eps} \\ \{M(k)\} & \text{if } k \in K_{term} \\ Fi'(su(k)) & \text{if } k \in K_{nont} \cup K_{*} \\ Fi'(su_1(k)) \cup \cdots \cup Fi'(su_{n(k)}(k)) & \text{if } k \in K_{prod} \cup K_{alt} \end{cases}$$

until no further changes occur.

For alternative nodes, the number n(k) is the number of all successors of k, and for product nodes, 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) := ∅;

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):

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

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. The sets $g(k)$ and $h(k)$ can be obtained by considering the invariant equations of algorithm (12.1) and (12.2).

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 $|K_{root}| = |V_N| + 1$.

To obtain the combined equation belonging to a root r, take equation (12.3) for r and substitute $f(k')$ in it by the right hand side of the equation for k' if k' is not a root. Repeat that substitution process until an equation is obtained having only f-sets of roots on the right hand side. Thus we obtain

Equation (12.4):

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

where $G(r) \subseteq V_T \cup \{\varepsilon\}$ is the union of the g-sets of all nodes reached during the substitution process, 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 first- resp. follow-dependence graph defined in the following chapter. The two dependence graphs can be computed by algorithms 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) 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).

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 (12.4).

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 (12.4)

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

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 (12.4) 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: $G'(C) = \bigcup_{r \in C} G(r)$.

From system (12.4) 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

$$(13.1) \quad f(C) = G'(C) \cup \bigcup_{(C, C') \in E'} f(C')$$

Although system (13.1) looks like (12.4), there is an important difference: digraph D' contains no cycles! Therefore we can solve system (13.1) 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 [8]). 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 (13.2):

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 $|E'| \le |E|$; 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 $|E| \le |K_{nont}|$ holds and $|K_{root}| + |E| \le |K_{root}| + |K_{nont}| \le 2|K|$. 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. 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 first-sets are empty, 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

true if $k \in K_{*} \cup K_{eps}$

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

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.

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. *ELL(1) conditions and unambiguity*

After having discussed in the previous chapters how the first- and follow-sets can be efficiently computed, we now consider the $ELL(1)$ conditions before introducing a predictive $ELL(1)$ parser in the next chapter.

If there is an arbitrary look-ahead length k, we must distinguish between weak and strong $ELL(k)$ conditions. In addition, we can define $ELL(k)$ either by R- or by K-derivation. Hence, many different $ELL(k)$ classes result.

In this thesis, we restrict ourselves to the $KELL(1)$ property; for the other kinds of $ELL(k)$-conditions and the relations between them, we refer to [2].

Definition (15.1): ($KELL(1)$)

A grammar is KELL(1) if for all $u, v_1, v_2, w \in K^*$ and $k \in K$

$$\begin{array}{ll} & \scriptstyle >\!\!-\; u\ v_1\ w \\ k_o >-*-\ u\ k\ w & \\ & \scriptstyle >\!\!-\; u\ v_2\ w \end{array} \qquad \text{and } v_1 \neq v_2 \text{ implies}$$

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

There is a local criterion for KELL(1) using first- and follow-sets:

Theorem (15.2): A reduced grammar has the KELL(1) property iff

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

(b) for all $k \in K_*$, $Fi(su(k))$ 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 $su_1(k), su_2(k), \ldots$ 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 (15.3):

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

$$\begin{array}{ll} & \scriptstyle >\!\!-\; u_1 >-*-\ t_1 \in K_T^* \\ k_o >-*-\ u & \\ & \scriptstyle >\!\!-\; u_2 >-*-\ t_2 \in K_T^* \end{array} \qquad \text{where } u_1 \neq u_2 \text{ implies } M(t_1) \neq M(t_2).$$

KELL(1) 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.

16. *The ELL(1) - parser*

In this chapter, we first describe how the ELL(1) parser works, then we give an algorithm to generate it and list its possible actions. At last the generated parser is stated to be correct and its time and space complexity are 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. Then a legal configuration of the parser looks like $(u.\ v\#,\ t.\ xk_\#)$ 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.

The internal situation of the parser is represented by the 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 $k_o > - * - t\ x$. $k_\#$ 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 such that the leftmost node of this word is at the top of the 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.

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. The table is here represented by a two-dimensional array

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

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

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.

Now we explain how the contents of the parser table can be derived from the grammar graph, enumerate the possible parser actions and describe how the parser executes them.

Let $La(k) = Fi(k)\ o\ Fo(k)$ be the look-ahead set of each node $k \in K$, and let $T\ (i, ch) = (error)$ if i is the number of a node k and ch is not in La(k).

Let the actual configuration of the parser be $(u.\,av,\ t.\,kx)$, thus k is the actual node (the node on the top of the stack). Let i be 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 terminal node:If k is marked by 'a', the parser may go into the configuration $(ua.\,v,\ tk.\,x)$. To do so, the next symbol must be read from the input, and the actual node k must be popped from the stack. This action is called 'shift' in analogy to other parsers.

T (i, M(k)) = (shift)

shift = begin SCAN; TOP := TOP - 1 end.

Case k is ε node:According to the definition of K-derivation (see Def. 6.3), the next configuration is $(u.\,av,\ t.\,x)$, i.e. the actual node is popped from the stack, but no input symbol is read.

T (i, ch) = (epsilon-shift) for all $ch \in La(k)$

epsilon-shift = begin TOP := TOP - 1 end.

Case k is non − terminal node:A non-terminal node derives its successor ([A] >-- [p(A)]), thus the appropriate action is to replace the actual node by its successor.

T (i, ch) = (expand, j) for all $ch \in La(k)$

expand (j) = begin ST [TOP] := j end.

Case k is alternative node:The actual node must be replaced by one of its successors according to the lookahead. This action is already called 'expand', but we give it a new name since differences could arise if the parser is extended by semantic actions.

$T\ (i,\ ch) = (select,\ j_r)$ for $ch \in La\ (su_r(k))$

select (j) = begin ST [TOP] := J end.

*Case k is * node*:For star nodes, there are two possible derivations: $[(R)^*] > -- \varepsilon$ or $> -- [R]\ [(R)^*]$. Thus there are two possible next configurations:

(u.av, t.x) or (u.av, t.su(k) k x).

In the first case, the actual node must be popped from the stack (like epsilon-shift), and in the second case, the successor must be pushed onto the stack.

T (i, ch) = (star end) for all $ch \in Fo(k)$

T (i, ch) = (star expansion, j) for all $ch \in Fi(su(k))$

star end = begin TOP := TOP - 1 end

star expansion (j) = begin TOP := TOP + 1; ST [TOP] := J end.

Case k is product node:Then k >-- $su_1(k) \cdots su_{N(k)}(k)$, thus the actual node 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 i, its successors have the following numbers $i + 1, \ldots, i + N(k)$. This is possible since all successors of product nodes are never again product nodes due to the convention of chapter 2. The number of successors $N(k)$ then is the parameter of this action called product expansion.

T (i, ch) = (product expansion, N(k)) for all $ch \in La(k)$

product expansion (N) =

```
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.
```

Case $k = k_{\#}$:T (|K|, #) = (accept)

The action "accept" can be considered as a special kind of "shift" involving # and $k_{\#}$.

Since we have generated the parser table according to the definition of K-derivation, it is not hard to prove that the generated ELL(1) parser works correctly.

Theorem (16.1):

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 $L(G)$.

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 (16.2):

(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 (16.3):

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

Take the grammar family

G_n: axiom = A_o , A_i $--> A_{i+1}\ A_{i+1}$ for all $i \in \{0, \cdots, n-1\}$, $A_n --> \varepsilon$

as an example for the non-polynomial dependence of c upon |K|.

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. Heckmann, R.: An efficient ELL(1) - Parser Generator,
   Technischer Bericht Nr. A 85 / 04, Fachbereich 10, Universität Saarbrücken (1985).
4. Heilbrunner, S.: On the definition of ELR(k) and ELL(k) grammars, Acta Informatica 11, 169 - 176 (1979).
5. Lewi, J., de Vlaminck, K., Huens, J., Steegmans, E.: A programming methodology in compiler construction, North-Holland (1982).
6. Möncke, U., Wilhelm, R.: Iterative algorithms on grammar graphs, Proceedings of the 8th conference on graphtheoretic concepts in computer science, 177 - 194, Hanser (1982).
7. Purdom, P. W., Brown, C. A.: Parsing extended LR(k) grammars,
   Acta Informatica 15, 115 - 127 (1981).
8. Tarjan, R.: Depth-first search and linear graph algorithms,
   SIAM J. Comput., Vol. 1, 146 - 160 (1972).