ISSN 0105-8517

METHODS FOR COMPUTING LALR(k) LOOKAHEAD

by

Bent Bruun Kristensen*
and
Ole Lehrmann Madsen

DAIMI PB-101
July 1979
(revised April 1980)

* Aalborg University Center, Aalborg, Denmark

Computer Science Department
AARHUS UNIVERSITY
Ny Munkegade – DK 8000 Aarhus C – DENMARK
Telephone: 06 – 12 83 55

ABSTRACT

Methods for constructing LALR(k) parsers are discussed. Algorithms for computing LALR(k)-lookahead are presented together with the necessary theory to prove their correctness. Firstly a special algorithm for the LALR(1) case is presented. Secondly a general LALR(k)-algorithm with $k \geq 1$ is presented. Given an item and a state the algorithms compute their corresponding LALR-lookahead during a recursive traversal of the LR(0)-machine. Finally the LALR(k) algorithm is generalised to compute LALR(k)-lookahead for all items and states visited during the recursive traversal performed by the former algorithms.

Contents

1. Introduction.

The subject of this paper is a well known approach to the construction of LR(k) - parsers. In [DeRemer 69 and 71] DeRemer proposed to construct LR(k) -parsers indirectly by

- constructing the LR(0)- machine,
- trying to resolve parsing conflicts by adding lookahead to items in the LR(0)-machine, and
- if this fails, trying to solve parsing conflicts by splitting states in the LR(0) - machine.

For this purpose two subclasses of the LR(k) - grammars have been defined: the Simple LR(k)- grammars (SLR(k)) and the Lookahead LR(k) -grammars (LALR(k)).
A grammar is SLR (k) if parsing conflicts in its LR(0)- machine can be resolved by adding as lookahead to any item the set of terminal strings (of length k) that may follow the leftside of the production in the item in any sentential form. I. e. the lookahead of an item in a state T does not depend on T.
A grammar is LALR(k) if parsing conflicts in its LR(0)- machine can be resolved by adding only the necessary lookahead to the items. The SLR(k) -grammars are a proper subset of the LALR(k) - grammars. In the latter case lookahead for an item in a state depends on the state whereas this is not the case in the former situation. The LALR(k) - grammars are again a proper subset of the LR(k)-grammars.

Unfortunately DeRemer only gives a practical solution for the SLR(k) case. Other people ([LaLonde 71], [Anderson, Eve and Horning 73], [Johnson 74], [Pager 77a and 77b]) have later presented solutions for the other cases. The topic of this paper is to present and prove efficient algorithms for computing LALR(k) lookahead.

De Remer's approach is in widespread use and a number of parser generators based on SLR - and LALR grammars exist. The work reported here is based on the experience gained by implementing and using the BOBS-system, which is an LALR (1)- parser generator [Eriksen et- al 73].

The motivation for this paper is a general dissatisfaction with the published algorithms for computing LALR (k) - lookahead. They seem to be more complicated than necessary and their efficiency in practice is sometimes doubtful. Furthermore the correctness of these algorithms is seldom proved.
It is possible (and likely) that not all of the material in this paper is new. We have, however, felt the need for a consistent and coherent exposition.

The rest of this paper is organised as follows :

Chapter 2 is a summary of the basic terminology and results needed in the rest of the paper. Chapter 3 contains an informal description of an algorithm for computing LALR (k)- lookahead. In chapter 4 some properties of LALR (k) are proved. An algorithm for computing LALR (1) - lookahead is important in practice and such an algorithm is presented and proved correct in chapter 5. In chapter 6 a general LALR (k) algorithm is presented and proved correct. The algorithm and the one of chapter 5 computes lookahead for a single item in a state using a recursive procedure. An improved version of the LALR (k) - algorithm that computes lookahead for all items visited during the recursive calls is also presented. This will avoid recomputation of lookahead if different calls have overlapping recursive calls. In chapter 7 the algorithms are compared to other published algorithms.

Upper bounds for the complexity of the presented algorithms are given in the respective sections.
The algorithms follow the same scheme for solving a set of recursive equations. A general proof for the correctness of this scheme is given in appendix A. Appendix B shows an example of an LALR(1) computation.

2. Basic Terminology and Results

The reader is assumed to be familiar with the terminology and conventions from [Aho & Ullman 72a] concerning grammars and parsers. Especially the following concepts are used extensively : $FIRST_k$, EFF_k, $FOLLOW_k$, $\oplus_k$, LR-item, (canonical) collection of sets of LR(k)-items, GOTO, CORE, and KERNEL ([Aho & Ullman 77]).

In the following we shall repeat some definitions and theorems, sometimes in a modified form.

A context free grammar is always assumed to have the form $G = (N, \Sigma, P, S)$ where N is a finite set of nonterminal symbols, Σ is a finite set of terminal symbols, P is a finite set of productions, and S is the start symbol. All grammars are assumed to be free of "useless" symbols. They are also assumed to be extended with a new start symbol S' and the production $S' \to S \dashv^k$, where $\dashv$ is a symbol not in $(N \cup \Sigma)$.

We use the following conventions : small Greek letters such as α, β, γ are in $(N \cup \Sigma)^*$; small latin letters in the beginning of the alphabet such as a, b, c are in Σ; small Latin letters in the end of the alphabet such as v, x, y are in Σ^* ; capital Latin letters in the beginning of the alphabet such as A, B, C are in N; capital Latin letters in the end of the alphabet such as X, Y, Z are in $(N \cup \Sigma)$. The empty string is denoted by e.

If $A \to \alpha\beta$ is in P and $u \in \Sigma^{*k}$ then $[A \to \alpha . \beta, u]$ is an LR(k)-item.

If $[A \to \alpha . \beta, u] \in S$ and S is a canonical collection of LR(k)-items then $[A \to \alpha . \beta, u] \in$ KERNEL (S) iff $|\alpha| > 0$.

Recall that $EFF_k(\alpha)$ captures all members of $FIRST_k(\alpha)$ whose rightmost derivation does not use an e-production at the last step, when α begins with a nonterminal.

If $M, N \subseteq \Sigma^{*k}$ then $M \oplus_k N = FIRST_k(\{xy \mid x \in M, y \in N\})$.

If M is a set of subsets of Σ^{*k} then $\cup M$ means $\{x \mid x \in m, m \in M\}$.

Definition 2.1

Let G be a CFG, then the LR(k)-machine for G is $LRM_k^G = (M_k^G, IS_k^G, GOTO_k^G)$, where M_k^G is a set of (LR(k)-)states, one for each set of items in the canonical collection of LR(k)-items. We do not distinguish between a state and its corresponding set of items. IS_k^G is the initial state. $GOTO_k^G$ is the GOTO-function defined on $M_k^G \times (N \cup \Sigma) \to M_k^G$.

□

For a given grammar G we will in the following assume the existence of its LRM_k^G on this form. The superscript G is omitted when this causes no confusion. $GOTO_k$ is extended in the obvious way to $(M_k) \times (N \cup \Sigma)^* \to M_k$.

The number of items in an LRM_k is defined as $\#\text{ items} = \sum_{T \in M_k} |T|$, where $|T|$ is the number of items in T.

The notion of LALR can be summarized in the following definitions and theorems :

Definitions

Let G be a CFG , with LR(k)-states M_k, $k \geq 0$.

(2.2) Let $T \in M_k$, then
$LR_k([A \to \alpha.\beta], T) = \{ u \mid [A \to \alpha.\beta, u] \in T \}$.

(2.3) Let $[A \to \alpha.\beta, u]$ be a LR(k)-item and let $S \in M_k$, then
$CORE([A \to \alpha.\beta, u]) = [A \to \alpha.\beta]$, and
$CORE(S) = \{CORE(I) \mid I \in S\}$.
We shall not distinguish between the items $[A \to \alpha.\beta, e]$ and $[A \to \alpha.\beta]$.

(2.4) Let $T \in M_0$, then
$URCORE_k(T) = \{ S \in M_k \mid CORE(S) = T \}$.

(2.5) Let $T \in M_0$, then
$LALR_k([A \to \alpha.\beta], T) = \cup \{LR_k([A \to \alpha.\beta], S) \mid S \in URCORE_k(T)\}$.

(2.6) G is said to be LALR(k), $k \geq 0$, if for all $T \in M_0$, and for all distinct items $[A \to \alpha.\beta]$ and $[B \to \gamma.]$ in T we have

(*) $$EFF_k(\beta) \oplus_k LALR_k([A \to \alpha.\beta], T) \cap LALR_k([B \to \gamma.], T) = \emptyset .$$

(2.7) Let $T \in M_k$, $X \in (N \cup \Sigma)$ and $\alpha \in (N \cup \Sigma)^*$, then

$$PRED(T,\alpha) = \begin{cases} \{T\} & \text{if } \alpha = e \\ \cup \{PRED(S,\alpha') \mid GOTO_k(S,X) = T\} & \text{if } \alpha = \alpha'X . \end{cases}$$

(2.8) Let $T \in M_k$, then $SUCC(T) = \cup\{GOTO_k(T,X) \mid X \in N \cup \Sigma\}$.

□

(*) The $\oplus_k$-operator has higher precedence than the $\cap$-operator.

Theorems

(2.9) Let $T \in M_k$, then

$$LR_k([A \to \alpha . \beta], T) = \{ w \mid w \in FIRST_k(y) \land S' \Rightarrow^*_{rm} \gamma A y \Rightarrow \gamma \alpha \beta y \land GOTO_k(IS_k, \gamma\alpha) = T \}.$$

(2.10) Let $T \in M_0$, then

$$LALR_k([A \to \alpha . \beta], T) = \{ w \mid w \in FIRST_k(y) \land S' \Rightarrow^*_{rm} \gamma A y \Rightarrow \gamma \alpha \beta y \land GOTO_o(IS_o, \gamma\alpha) = T \}.$$

(2.11) Let $T \in M_k$, then

$$\forall S \in PRED(T, \alpha) : LR_k([A \to \alpha . \beta], T) = LR_k([A \to . \alpha\beta], S)$$

(2.12) Let $T \in M_k$ and let $[A \to . \alpha] \neq [S' \to . S \dashv^k]$, then

$$LR_k([A \to . \alpha], T) = \cup \{ FIRST_k(\psi) \oplus_k LR_k([B \to \varphi . A\psi], T) \mid [B \to \varphi . A\psi, u] \in T \}.$$

Proofs

2.9, 2.11 and 2.12 follow directly from the algorithms developed in section 5.2.3 in [Aho & Ullman 72a] for constructing the canonical collection of LR(k)-items. 2.10 may be proved using 2.5 and 2.9.

□

3 Informal Description of the LALR(k)-algorithm

The $LALR_k$-lookahead of an item $[A \to \alpha.]$ in a state T may informally be described as the set of terminal strings (of length k) that may appear on input if during parsing the reduction $A \to \alpha$ can be applied in state T.

We want to compute $LALR_k([A \to \alpha.], T)$ using the LR(0)-machine, LRM_0. We are thus interested in the set of states where the parsing may be resumed after the considered reduction.

Let S be a state containing the item $[A \to .\alpha]$ and $GOTO(S, \alpha) = T$. The parsing may be resumed in S after the reduction and will then continue with a read transition on A. PRED (T, α) is exactly the set of such states.

After the transition on A in S, the parse stack has the form : $v_1\ v_2 \ldots v_n$ S R, where $R = GOTO(S,A)$ is the top member of the stack. Any terminal string of length k that may be read starting from this parse stack is in $LALR_k([A \to \alpha.], T)$. These terminal strings may be characterised as follows :

(3.1) They are in $LALR_k([A \to .\alpha], T)$, and

(3.2) State S contains items of the form $[B_i \to \varphi_i\ .\ A\ \Psi_i]$, $i = 1, 2, \ldots, p$, $p > 0$, and R will thus contain the items $[B_i \to \varphi_i\ A.\ \Psi_i]$, $i = 1, 2, \ldots, p$. This implies that

$$\cup \{ FIRST_k(\Psi_i) \mid i = 1, 2, \ldots, p \}$$

can be read from R. Now if some $\Psi_i \Rightarrow^* w$ and $|w| < k$, then we may reduce by $B_i \to \varphi_i\ A\ \Psi_i$ before we have read a string of length k. This means that we must compute the terminal strings that may follow B_i after such a reduction. We know that S is still on the stack when the reduction $B_i \to \varphi_i\ A\ \Psi_i$ is applied, so $LALR_k([B_i \to \varphi_i . A\ \Psi_i], S)$ is the set of terminal strings that may legally follow B_i after the reduction.

In order to compute $LALR_k([B_i \to \varphi_i . A\Psi_i], S)$ we may recursively repeat the above process by considering $PRED(S, \varphi_i)$ etc. This gives rise to a recursive $LALR_k$-algorithm. Fig. 3.4 gives a picture of the situation.

There are a number of problems with formulating a recursive $LALR_k$-algorithm. First the LRM_0-machine is in general full of cycles which may cause difficulties in terminating the recursion. Second the predecessor "tree" obtained by tracing backwards is very little tree structured in general, as a lot of overlapping takes place. This may cause difficulties in avoiding recomputation of $LALR_k$-values that have been computed. It is a question of devising an algorithm that is linear in # items instead of being exponential.

Before we present an algorithm we shall formalize the above discussion by characterising $LALR_k$ in terms of LRM_0 in the same way as LR_k is characterised in terms of LRM_k (2.10, 2.11).

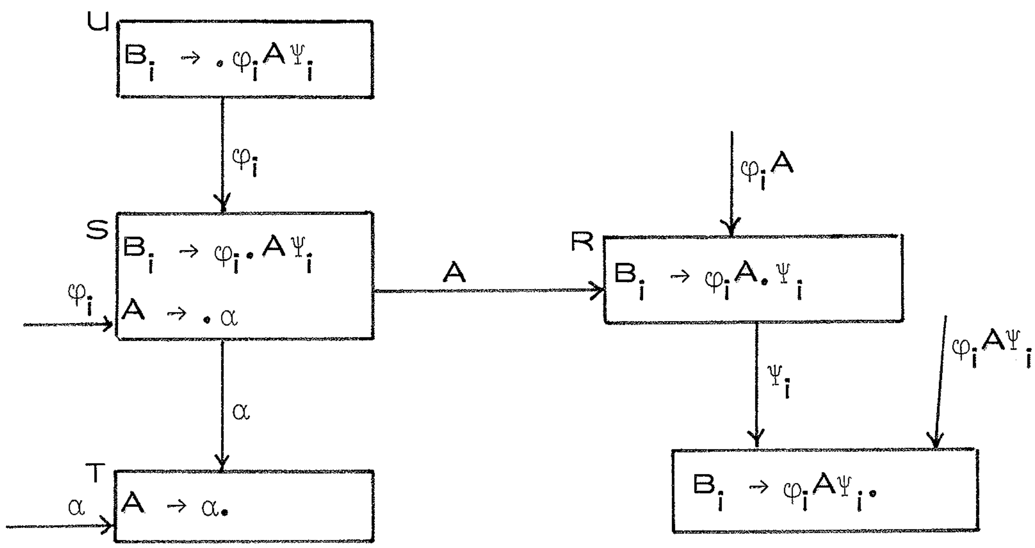


Fig. 3.4

4 Properties of LALR(k)

The first step in section 3 (3.1) was to trace backwards in LRM_0. We characterised $LALR_k([A \to \alpha . \beta], T)$ in terms of $LALR_k([A \to . \alpha\beta], S)$ for all S in PRED(T, α).

Lemma 4.1

Let $T \in M_0$, then
$$LALR_k([A \to \alpha . \beta], T) = \cup\{LALR_k([A \to . \alpha\beta], S) \mid S \in PRED(T, \alpha)\}$$

Proof :

Definition 2.5 and Theorem 2.11 may be used to show that

$$LALR_k([A \to \alpha . \beta], T) =$$
$$\cup\{LR_k([A \to . \alpha\beta], R) \mid R \in M_k \wedge CORE(GOTO_k(R, \alpha)) = T\}$$

and

$$\cup\{LALR_k([A \to . \alpha\beta], S) \mid S \in PRED(T, \alpha)\} =$$
$$\cup\{LR_k([A \to . \alpha\beta], R) \mid R \in M_k \wedge GOTO_0(CORE(R), \alpha) = T\}.$$

The lemma now follows from the fact that

$$CORE(GOTO_k(R, \alpha)) = GOTO_0(CORE(R), \alpha)$$

□

The next step (3.2) was to characterise $LALR_k([A \to .\alpha], T)$.

Lemma 4.2

Let $T \in M_0$, $[A \to .\alpha] \in T$ and $A \neq S'$. Then
$LALR_k([A \to .\alpha], T) =$
$\cup\{FIRST_k(\Psi) \oplus_k LALR_k([B \to \varphi.A\Psi], T) \mid [B \to \varphi.A\Psi] \in T\}$.

Proof

By using 2.5 and 2.12 we obtain :

$LALR_k([A \to .\alpha], T) =$
$\cup\{FIRST_k(\Psi) \oplus_k LR_k([B \to \varphi.A\Psi], S) \mid S \in URCORE_k(T) \wedge$
$[B \to \varphi.A\Psi] \in T\} =$

$\cup\{FIRST_k(\Psi) \oplus_k M \mid [B \to \varphi.A\Psi] \in T \wedge$
$M = \cup\{LR_k([B \to \varphi.A\Psi], S) \mid S \in URCORE_k(T)\}\}$.

Finally we have that

$$M = LALR_k([B \to \varphi.A\Psi], T)$$

and this proves the lemma.

□

By combining lemma 4.1 and 4.2 we obtain the following theorem :

Theorem 4.3

Let $T \in M_0$, $[A \to \alpha.\beta] \in T$ and $A \neq S'$. Then
$LALR_k([A \to \alpha.\beta], T) =$
$\cup\{FIRST_k(\Psi) \oplus_k LALR_k([B \to \varphi.A\Psi], S) \mid$
$S \in PRED(T, \alpha) \wedge [B \to \varphi.A\Psi] \in S\}$.

□

5. The LALR(1) Case

In practice algorithms for computing LALR(1) are of much more interest than a general LALR(k)-algorithm for $k > 1$.
In this section we shall present an LALR(1) algorithm. Theorem 4.3 may in this case be simplified. $FIRST_1$ in the equations 4.3 is straightforward to compute directly on LRM_0. This is expressed in the following definition and lemma.

Definition 5.1

Let $T \in M_0$, then

$$TRANS(T) = \{ a \mid [B \to \varphi . a\Psi] \in T\} \cup$$
$$\cup\{TRANS(GOTO_0(T,A)) \mid [B \to \varphi . A\Psi] \in T \wedge A \Rightarrow^* e\}$$

□

Lemma 5.2

Let $T \in M_0$, then

$$TRANS(T) = \cup\{FIRST_1(\beta) \mid [A \to \alpha . \beta] \in KERNEL(T)\} - \{e\}$$

□

Theorem 4.3 may now be reformulated in the following theorem

Theorem 5.3

Let $T \in M_0$, then

$$LALR_1([A \to \alpha . \beta], T) = \cup\{ L(S,A) \mid S \in PRED(T,\alpha)\}$$

where

$$L(S,A) = TRANS(GOTO_0(S,A)) \cup$$
$$\cup\{LALR_1([B \to \varphi . A\Psi], S) \mid [B \to \varphi . A\Psi] \in S \wedge \Psi \Rightarrow^* e\}.$$

Proof

It is easy to show that

$$LALR_1([A \to . \alpha], S) =$$
$$\cup\{FIRST_1(\Psi) \mid [B \to \varphi A . \Psi] \in GOTO_0(S,A) \} - \{e\} \cup$$
$$\cup\{LALR_1([B \to \varphi . A\Psi], S) \mid [B \to \varphi . A\Psi] \in S \wedge \Psi \Rightarrow^* e\}.$$

The Theorem may now be proved using 5.2.

□

Let us now consider the identities in 5.3 as a set of equations defining a recursive function $LALR_1$ from Item × State to $\mathcal{P}(\Sigma^{*k})$. We may then solve these equations in order to compute LALR(1)-lookahead. The equations may have more than one solution. However, we are only interested in the smallest solution as expressed by the following theorem.

<u>Theorem 5.4</u>

Consider the function

$$F : \text{Item} \times \text{State} \rightarrow \mathcal{P}(\Sigma^{*k})$$

defined by the set of equations

$$F(A \rightarrow \alpha . \beta], T) = \begin{cases} \{e\} & \text{if } A = S' \wedge [A \rightarrow \alpha . \beta] \in T \\ \cup\{L(S,A) \mid S \in PRED(T,\alpha)\} & \text{if } A \neq S' \end{cases}$$

where

$$L(S,A) = TRANS(GOTO_0(S,A)) \cup$$
$$\cup\{F([B \rightarrow \varphi . A\Psi], S) \mid [B \rightarrow \varphi . A\Psi] \in S \wedge \Psi \Rightarrow^* e\}.$$

$LALR_1$ is the smallest solution to the equations in the sense that if G is another solution then $LALR_1(I,T) \subseteq G(I,T)$ for all I and T.

<u>Proof</u>

$LALR_1$ is clearly a solution (5.3).

Let $w \in LALR_1([A \rightarrow \alpha . \beta], T)$, then we have from (2.10) a derivation of the form :

$$S' \Rightarrow^* \gamma\, B_0\, y \Rightarrow \gamma\, \varphi_0\, B_1\, \Psi_0\, y \Rightarrow^* \gamma\, \varphi_0\, B_1\, z$$

$$\Rightarrow \gamma\, \varphi_0\, \varphi_1\, B_2\, \Psi_1\, z \Rightarrow^* \gamma\, \varphi_0\, \varphi_1\, B_2\, z \Rightarrow \dots$$

$$\Rightarrow \gamma\, \varphi_0\, \varphi_1 \cdots \varphi_n\, A\, \Psi_n\, z \Rightarrow^* \gamma\, \varphi_0\, \varphi_1 \cdots \varphi_n\, A\, z$$

$$\Rightarrow \gamma\, \varphi_0\, \varphi_1 \cdots \varphi_n\, \alpha\, \beta\, z$$

where $B_i \to \varphi_i\, B_{i+1}\, \Psi_i$, $i = 0, 1, \dots, n$, $n \geq 0$ are productions and $B_{n+1} = A$, $y \neq z$, hence $w \in FIRST_1(\Psi_0)$ and $GOTO_0(IS_0, \gamma\, \varphi_0\, \varphi_1 \cdots \varphi_n\, \alpha) = T$.

Let $S_i = GOTO_0(IS_0, \gamma\, \varphi_0\, \varphi_1 \cdots \varphi_i)$, $i = 0, 1, \dots, n$, then $[B_i \to \varphi_i \,.\, B_{i+1}\, \Psi_i] \in S_i$, and $S_i \in PRED(T, \varphi_{i+1} \cdots \varphi_n\, \alpha)$, $i = 0, 1, \dots, n$.

Hence

$$F([B_0 \to \varphi_0 .\, B_1\, \Psi_0], S_0) \subseteq F([B_1 \to \varphi_1 .\, B_2\, \Psi_1], S_1)$$
$$\vdots$$
$$\subseteq F([B_n \to \varphi_n .\, A\, \Psi_n], S_n)$$
$$\subseteq F([A \to \alpha . \beta], T)$$

As $w \in FIRST_1(\Psi_0)$ we have that $w \in F([B_0 \to \varphi_0 \,.\, B_1\, \Psi_0], S_0)$ and then $w \in F([A \to \alpha . \beta], T)$.

This proves the Theorem.

□

The algorithm may be realised by the function LALR-1 (5. 5), which has a local recursive procedure LALR. Procedure LALR is a straightforward transcription of the equations in 5. 3 (or 5. 4). Each level of recursion adds lookahead symbols to a global variable LA. The recursion is stopped either when no $\psi \Rightarrow^* e$ or when making a recursive call with a set of parameter values that have appeared in another recursive call. A global variable Done collects these parameter values. The intuition behind Done is that $LALR_1$ for parameter values in Done is either computed (and added to LA) or a recursive call to compute it is initiated. In the latter case we have that $LALR_1$ for the considered parameter values is circularly dependent on itself. A general algorithm following this scheme is presented and proved correct in Appendix A.

The function TRANS is implemented in a similar way. The global variable TM collects parameter values of TRANS.

<u>Notation</u>

In the algorithm the construct

> <u>FOR</u> a ∈ M <u>WHERE</u> P(a) <u>DO</u> S <u>END FOR</u> means
>
> <u>FOR</u> a ∈ M <u>DO</u>
>   <u>IF</u> P(a) <u>THEN</u> S <u>ENDIF</u>
> <u>ENDFOR</u>

The construct ASSUME has no effect and is used to give names to components of structured variables.

□

Algorithm 5.5

```
FUNCTION LALR-1 (I : Item ; T : State) : SET OF Σ ;
VAR  LA   : SET OF Σ ;
     Done : SET OF Item × State ;
     TM   : SET OF State ;

PROCEDURE TRANS (T : State) ;
BEGIN  TM : = TM ∪ {T} ;
       FOR [B → φ. X Ψ] ∈ T DO
         IF X ∈ Σ THEN  LA : = LA ∪ {X}
         ELSE IF (X ⇒* e) ∧ (GOTO0(T,X) ∉ TM) THEN  TRANS(GOTO0(T,X))
         END IF
       END FOR
END TRANS ;

PROCEDURE LALR (I : Item ; T : State) ;
BEGIN  Done : = Done ∪ { (I, T) } ;
       ASSUME  I = [A → α. β] ;
       FOR  S ∈ PRED (T, α) DO
            TM : = ∅ ;  TRANS(GOTO0(S,A)) ;
            FOR [B → φ. AΨ ] ∈ S
                WHERE (Ψ ⇒* e)  ∧  ([B → φ. AΨ ],S) ∉ Done   DO
                         LALR ([B → φ. AΨ],S)
            END FOR
       END FOR
END LALR ;

BEGIN    Done : = LA : = TM : = ∅;

      ASSUME  I = [A → α. β] ;
      IF  A = S'  THEN  LA := {e}  ELSE  LALR(I, T)
      ENDIF ;
      LALR-1 : = LA
END LALR-1 ;
```

Notice that the statement TM : = ∅ in procedure LALR may in fact be removed.

□

5.1 Worst Case Analysis

Below we give some remarks on time and space requirements of algorithm 5.5. This is done by discussing upper bounds of the number of recursive calls of procedure LALR.

A trivial upper bound is the size of the domain for Done,which is the number of states times the number of items. However, the algorithm only makes a recursive call LALR(I, T) if the item I is in state T. This gives that # items is an upper bound on the number of recursive calls.

The body of the outermost FOR-loop of procedure LALR computes $LALR_1([A \rightarrow .\alpha], S)$. The algorithm may be improved by adding $([A \rightarrow .\alpha], S)$ to Done in the beginning of the statement controlled by the FOR-loop, and only execute this statement if $([A \rightarrow .\alpha], S)$ was not already in Done.

The algorithm can be further improved by avoiding recursive calls of the form $LALR([B \rightarrow .A\Psi], S)$. This gives an upper bound that is the number of all items in the KERNELs of all states.

For a grammar of the size of that of Pascal one may typically have about 300 states with 10 items per state and 2 items in the KERNEL of each state. However, the difference between the upper bounds and the "normal" in practice is quite large.

5.2 The BOBS-implementation

A slightly different version of algorithm 5.5 is used in the BOBS-system. Consider the following definition

Definition 5.6

For all $A \in N$ and $S \in M_0$ we define

$$LA(A,S) = TRANS(GOTO_0(S,A)) \cup$$
$$\cup \{LALR_1([B \rightarrow \varphi . A\Psi], S) \mid [B \rightarrow \varphi . A\Psi] \in S \wedge \Psi \Rightarrow^* e\}$$

□

It is easy to see that the following lemma is true

Lemma 5.7

For all items $[A \to . \alpha]$ and $S \in M_0$ we have

$LALR_1([A \to .\alpha], S) = LA(A, S)$

□

The procedure LALR of algorithm 5.5 may now be improved to the following algorithm :

Algorithm 5.8

```
PROCEDURE LALR  (I : Item ; T : State) ;
BEGIN ASSUME  I = [A → α . β] ;
   FOR S ∈ PRED(T, α)
      WHERE     (A, S) ∉ Done DO
           Done := Done ∪ {(A, S)} ;
           TRANS(GOTO0(S, A)) ;
           FOR [B → φ . AΨ] ∈ S  WHERE  Ψ ⇒* e  DO
                 LALR([B → φ . AΨ], S)
           END FOR
   END FOR
END LALR ;
```

□

The domain of the set Done is now $N \times$ State. The function LA is computed by the statement list controlled by the outermost for-loop.

The set Done is represented as a linked list of states, one for each nonterminal. This could be improved by using a bit-vector of length equal to the number of nonterminal transitions in the LR(0)-machine. In order to ease the computation of PRED we have for all items $[A \to \alpha . \beta]$ a list of states in which the item appears.

The LALR(1) algorithm has been used in the BOBS-system since 1973 and has proved its usability in practice. The system also includes an SLR(1) lookahead algorithm, but the difference in speed between SLR(1) and LALR(1) is so little that LALR(1) is used by default.

Finally we notice that the difference in speed between the algorithms 5.5 and 5.8 is minor.

6. The LALR(k) Case

Here we shall give an algorithm for computing LALR(k) that works for all $k \geq 0$. We cannot just make a recursive procedure using 4.3 directly but we have to make a transformation of 4.3 in the same way as we did in the LALR(1) case. In the LALR(1) case 5.3 expresses that all recursive dependencies of $LALR_1$ are of the form $LALR_1(I', T') \subseteq LALR_1(I, T)$. In the LALR(k) case we need to k-concatenate $FIRST_k(\Psi)$ with the result of a recursive call $LALR_k([B \to \varphi . A \Psi], S)$. This approach will not work if we use the same scheme as in algorithm 5.5. The reason is that a recursive call $LALR_k(I, T)$ will not necessarily compute $LALR_k(I, T)$ because of the way the recursion is stopped when there are cycles in the LR(0)-machine. We shall thus reformulate 4.3 into the following theorem :

Theorem 6.1

Let $T \in M_k$, $[A \to \alpha . \beta] \in T$ and $A \neq S'$. Then

$$LALR_k([A \to \alpha . \beta], T) = \bigcup \{ F_k(S,A) \cup FL_k(S,A) \cup L_k(S,A) \mid S \in PRED(T, \alpha) \}$$

where

$$F_k(S,A) = \{ w \mid w \in FIRST_k(\Psi) \wedge |w| = k \wedge [B \to \varphi . A \Psi] \in S \},$$

$$FL_k(S,A) = \bigcup \{ (FIRST_k(\Psi) - \{e\}) \oplus_k LALR_i([B \to \varphi . A\Psi], S) \mid [B \to \varphi . A\Psi] \in S \wedge i = k - |FIRST_k(\Psi) - \{e\}|_{min} \wedge 0 < i < k \},$$

$$L_k(S,A) = \bigcup \{ LALR_k([B \to \varphi . A\Psi], S) \mid [B \to \varphi . A\Psi] \in S \wedge \Psi \Rightarrow^* e \}$$

□

Notation

Let $M \subseteq \Sigma^{*k}$, then $|M|_{min}$ is the length of the shortest string in M ; if $M = \emptyset$ then $|M|_{min} = 0$.

We may now view 6.1 as a set of equations defining the recursive functions $LALR_k$, $LALR_{k-1}$, ... , $LALR_1$. The interesting property of 6.1 is that the recursive dependencies have been separated into two cases : either $LALR_k(I', T') \subseteq LALR_k(I, T)$ or $(FIRST_k(\Psi) - \{e\}) \oplus_k LALR_i(I', T') \subseteq LALR_k(I, T)$ with $0 < i < k$. Consequently $LALR_k$ is recursive in itself in the same way as $LALR_1$ is in itself. $LALR_k$ may use $LALR_i$, $i < k$ but the opposite is not the case.

If 6.1 is viewed as a set of equations, it is again the smallest solution that interests us, and a theorem similar to 5.4 could be formulated. Using 6.1 we may now define a recursive function LALR-k that has an item, a state, and k as a parameter. All inner recursive calls of LALR-k are with a decreased k-value so the recursion will stop. As LALR-k is a function, all inner calls will of course return the desired result which may be used for k-concatenation.

The function LALR-k has a local recursive procedure, LALR, which computes $LALR_k$ for fixed k in the same way as the procedure LALR of 5.5 did handle $LALR_1$. For the correctness of this we again refer to the appendix.

We assume the existence of a function FIRST-k which is easy to implement (see e.g. [Aho & Ullman, 72a]).

Algorithm 6.2

```
FUNCTION LALR-k  (I : Item ; T : State ; k : Integer) : SET OF Σ*^k ;
VAR  LA : SET OF Σ*^k ;   Done : SET OF Item × State ;

   PROCEDURE  LALR (I : Item ; T : State) ;
   VAR  F : SET OF Σ*^k ;   i : Integer ;
   BEGIN  Done : = Done ∪ {(I, T)} ;
      ASSUME  I = [A → α . β] ;
      FOR  S ∈ PRED (T, α)  DO
         FOR [B → φ. A Ψ] ∈ S  DO
            F : = FIRST_k(Ψ) ;  LA : = LA ∪ {w ∈ F |  |w| = k} ;
            i  : = k - |F - {e} |_min ;
            IF  0 < i < k  THEN
               LA : = LA ∪(F-{e}) ⊕_k LALR-k([B → φ. AΨ], S, i)
            ENDIF ;
            IF  e ∈ F ∧ ([B → φ. A Ψ], S)  ∉  Done  THEN
               LALR([B → φ. A Ψ], S)
            ENDIF
         ENDFOR
      ENDFOR
   END LALR ;

BEGIN    LA : = Done : = ∅ ;
         ASSUME  I = [A → α. β] ;
         IF  A = S'  THEN LA : ={e} ELSE LALR(I, T)
         ENDIF ;
         LALR-k : = LA
END      LALR-k ;
```

6.1 Worst Case Analysis

The trivial upper bounds of the number of recursive calls of procedure LALR in algorithm 6.2 is clearly much larger than that of algorithm 5.5.

For each invocation of the function LALR-k we may use, as an upper bound on the number of recursive calls of the procedure LALR, the same bound as for procedure LALR of algorithm 5.5. This bound was the number of items in all states (# items). Each invocation of LALR may call recursively on LALR-k where k is at least decreased by one. This gives an upper bound on $O((\# \text{ items})^k)$ for the total number of recursive calls of procedure LALR for all invocations of LALR-k.

We may, however, save values of LALR-k that have been computed and in this way avoid recursive calls of LALR-k that have previously been computed.

For fixed i, a call of the form LALR-k(I,T,i) will thus only be performed at most $O(\# \text{ items})$ times. Each invocation of LALR-k has $O(\#\text{items})$ as a bound on the number of recursive calls of its local procedure procedure LALR. Thus $O((\# \text{ items})^2)$ will be a bound on the number of recursive calls of procedure LALR in all possible calls LALR-k(I,T,i). Considering all calls of LALR in all instances of LALR-k we thus obtain a bound on one call LALR-k(I,T,k) to be

$$O(\# \text{ items} + (k-1) \cdot (\# \text{ items})^2).$$

If we want to compute $\text{LALR}_k(I,T)$ for all I,T we will thus have a bound on

$$O(k \cdot (\# \text{ items})^2).$$

6.2 Improving the LALR(k)-algorithm

In the preceding section we have indicated that the efficiency of the $LALR_k$-algorithm may be improved by saving already computed $LALR_k$-information.

It is, however, possible to do better. In the following we shall describe an improvement of algorithm 6.2. Consider the definitions :

[6.3] Let $T \in M_k$, $I = [A \to \alpha.\beta]$, $I \in T$ and $A \neq S'$. Then
$D_k(I,T) = \{([B \to \varphi.A\psi], S) \mid S \in PRED(T,\alpha) \wedge [B \to \varphi.A\psi] \in S\}$, and
$Closure_k(I,T) = \{(I,T)\} \cup \bigcup\{Closure_k(J,S) \mid (J,S) \in D_k(I,T)\}$.

$D_k(I,T)$ is the set of pairs of items for which the double FOR-loop in 6.2 is executed. $Closure_k(I,T)$ is the set of pairs of (item, state) that are visited during the recursive activation of the initial call of LALR(I, T). The improved algorithm will compute $LALR_k$-lookahead for all elements in $Closure_k(I,T)$.

Consider the procedure LALR of algorithm 6.2. Symbol strings added to the set LA between the entry and exit of a call LALR(I, T) are clearly a subset of $LALR_k(I,T)$.

[6.4] Let the set of strings added to LA between the entry and exit of a call LALR(I, T) be called <u>Partial LALR</u>(I, T) (or <u>PLA</u>(I, T) for short). Thus $PLA(I,T) \subseteq LALR_k(I,T)$.

Thus PLA is a function that is determined by the performance of algorithm 6.2 (to assure that PLA is well defined, we assume that the double FOR-loop goes through the elements in $D_k(I,T)$ in a fixed order.

In the improved algorithm procedure LALR will save PLA(I, T) in a global variable Res(I, T) for all I and T visited during recursive calls of LALR. Res(I, T) will be marked such that it is possible to see whether Res(I, T) = LALR(I, T) or not. In the latter case the marking will also indicate the lookahead set that has to be added.

The reason that we only have inclusion in [6.4] is that we do not call on LALR for parameter values which are already in Done.

[6.5] If a call LALR(I, T) implies that LALR(J, S) is a candidate for a call (i.e. $(J,S) \in D_k(I,T)$) and $(J,S) \in$ Done, then $LALR_k(J,S) \subseteq LALR_k(I,T)$ but PLA(I, T) does not necessarily include $LALR_k(J,S)$.

A parameter set (J, S) may be in Done for one of the following two reasons

[6.6] A call LALR(J, S) has been initiated but not yet completed; i.e. an instance of LALR with parameters J, S is on the runtime stack.

[6.7] A call LALR(J, S) has been executed and completed.

In case [6.6] we have in addition to [6.5] that

[6.8a]
$$\begin{aligned} LALR_k(I,T) &\subseteq LALR_k(J,S), && \text{and thus} \\ LALR_k(I,T) &= LALR_k(J,S) && \text{and} \\ PLA(I,T) &\subseteq PLA(J,S) \end{aligned}$$

In case [6.7] we may have a dependency which is similar to that in [6.8a]. $LALR_k(J,S)$ may depend on one or more elements on the runtime stack. In this case we have that

there exists an (L, R) on the runtime stack such that :

[6.8b]
$$\begin{aligned} LALR_k(L,R) &= LALR_k(J,S), \\ LALR_k(I,T) &\subseteq LALR_k(L,R), \text{ and thus} \\ LALR_k(I,T) &= LALR_k(L,R) = LALR_k(J,S), \text{ and} \\ PLA(I,T) &\subseteq PLA(L,R), \\ PLA(J,S) &\subseteq PLA(L,R). \end{aligned}$$

In case [6.7] we may alternatively have that $LALR_k(J,S)$ does not depend on an element on the runtime stack. Then $PLA(J,S) = LALR_k(J,S)$.

In case [6.8a-b] we have that PLA(J,S) must be added to Res(I,T) in order to complete it to $LALR_k(I,T)$. In both cases we have that an element, say (M,U), on the runtime stack will include PLA(J,S). In case [6.8a] (M,U) = (J,S) and in case [6.8b] (M,U) = (L,R). We may then mark Res(I,T) in such a way that we can add PLA(M,U) to Res(I,T) when PLA(M,U) is computed.

The improved algorithm is outlined below :

[6.9] The set Done is separated into two sets Stack and FIN in order to distinguish between the situations [6.6] and [6.7]. At entry to a call LALR(I,T), (I,T) is added to Stack; at exit from the call, (I,T) is removed from Stack and added to FIN. (See also the algorithms in appendix A).

[6.10] Instead of saving lookahead elements in LA, these are saved in Res(I,T).

[6.11] Suppose that we in the body of a call LALR(I,T) are going to involve a recursive call LALR(J,S) (i.e. $(J,S) \in D_k(I,T)$). Instead of performing the call, we check for one of the following situations :

[6.12a] If $(J,S) \in$ FIN then we add Res(J,S) to Res(I,T) ;

[6.12b] If $(J,S) \in$ Stack then we add a special symbol #(J,S) to Res(I,T);

[6.12c] If $(J,S) \notin$ FIN $\cup$ Stack then we perform a recursive call LALR(J,S) and add Res(J,S) to Res(I,T). (Res(J,S) includes PLA(J,S)).

Having executed the double FOR-loop we have the following two situations :

[6. 13] No symbol #(L, R) is in Res(I, T). Then Res(I, T) = $LALR_k$(I, T).

[6. 14] One or more symbols #(L, R) are in Res(I, T).
Let $\#(L_1, S_1)$, $\#(L_2, S_2)$, ..., $\#(L_n, S_n)$ be the special symbols in Res(I, T), and let $\#(L_i, S_i)$ be below $\#(L_{i+1}, S_{i+1})$ (i = 1, 2, .., n-1) on the runtime stack.
Then $PLA(L_i, S_i) \subseteq PLA(L_1, S_1)$, i = 2, 3, ..., n.
Consequently we need only keep $\#(L_1, S_1)$ in Res(I, T).
Furthermore if #(I, T) is equal to $\#(L_1, S_1)$ then we may even delete $\#(L_1, S_1)$. If we eliminate as many special symbols as possible then we end up with one of the following two situations :

[6. 15a] The symbol #(I, T) has been removed from Res(I, T). Then Res(I, T) = $LALR_k$(I, T). All occurrences of #(I, T) in sets Res(M, U) where (M, U) ∈ FIN are then expanded by Res(I, T).

[6. 15b] One symbol #(L, R) ≠ #(I, T) is in Res(I, T). All occurrences of #(I, T) in sets Res(M, U) where (M, U) ∈ FIN are in this case replaced by #(L, R). (Note that PLA(L, R) includes PLA(I, T)).

The expansions described in [6. 15a-b] may be performed by keeping track of the sets

[6. 16] R(#(I, T)) = {(M, U) ∈ FIN | #(I, T) ∈ Res(M, U)}

In [6. 15b] we have that Res(I, T) = {#(L, R)} ∪ $\mathfrak{m}$, where $\mathfrak{m} \subseteq \Sigma^{*k}$. It is sufficient just to save only #(L, R) in Res(I, T), and then let the call LALR(I, T) return {#(L, R)} ∪ $\mathfrak{m}$. Thus in [6. 12c], instead of adding Res(J, S) to Res(I, T), the set returned by LALR(J, S) should be added to Res(I, T). We will then have that for all (I, T) ∈ FIN, either Res(I, T) = $LALR_k$(I, T) or Res(I, T) = {#(L, R)} where (L, R) ∈ Stack and $LALR_k$(I, T) = $LALR_k$(L, R) and PLA(I, T) ⊆ PLA(L, R).

We have earlier argued that in a call of LALR-k(I, T, i) an upper bound on the number of calls on its local procedure LALR is $O(\#\ \text{items})$. If we call LALR-k for all I, T we get a total upper bound on LALR which is $O((\#\ \text{items})^2)$.

In the improved algorithm we save $LALR_k$ for all (J, S) visited during the call. Consequently a call LALR-k(I, T, i) will not call LALR on (J, S) if LALR(J, S) has been performed in another activation of LALR-k with the same i. An upper bound for calling LALR-k for all I, T will thus be $O(\#\ \text{items})$.

If we consider an upper bound that includes calls of procedure LALR in all recursive activations of LALR-k, then we may compute LALR-k(I, T) for all I, T with a bound

$$O(k \cdot (\#\ \text{items})).$$

In [Kristensen & Madsen 80] a general version of the above described algorithm is given in all details together with a correctness proof. A more detailed complexity analysis is also given. This analysis considers the number of times the statements in the double FOR-loop is executed. Furthermore the overhead involved in saving and expanding the #(L, R) symbols is considered. The results may be summarised as follows :

Consider the following definition

[6.17] $S_k(I,T) = \{(L,R) \mid (L,R) \in Closure_k(I,T) \wedge (I,T) \in Closure_k(L,R)\}$

The statements in the double FOR-loop is executed at most $|D_k(I,T)|$ times for an activation LALR(I, T).

The expansion described in [6.15] will perform at most $|S_k(I,T)|$ replacements for an activation LALR(I, T).

Let $n_k = \sum_{(I,T) \in \mathfrak{M}_k} |D_k(I,T)|$ and $m_k = \sum_{(I,T) \in \mathfrak{M}_k} |S_k(I,T)|$,

where $\mathfrak{M}_k = \{(I,T) \mid T \in M_k \wedge I \in T\}$. Then $LALR_k(I,T)$ may be computed for all (I,T) by executing at most

$$O(k \cdot (n_k + m_k))$$

primitive operations (like : =), $\cup$-operations on lookahead sets, and $FIRST_k$ computations. Both n_k and m_k are less than $\#\,\text{items}^2$.

In [Kristensen & Madsen 80] it is furthermore shown that ny using the UNION-FIND algorithm in [Aho, Hopcroft & Ullman 76] to implement the saving and expansion of the $\#(L,R)$-symbols the bound m_k may be replaced by

$$n_k \cdot G(n_k)$$

where $G(n) \in [1,5]$ if $n \in [1, 2^{65536}]$.

6.3 Computing $FIRST_k$ on the LR(0)-Machine

In algorithm 6.2 we have assumed the existence of a $FIRST_k$ algorithm for computing the lookahead strings. It is interesting to investigate the possibilities for computing the lookahead strings directly on the LRM_0. It might be simpler since a lookahead set is a union of sets of strings that can be read from certain states. Such approaches are used by DeRemer and LaLonde to compute SLR(k) and LALR(k) sets respectively. The procedure TRANS in algorithm 5.5 is an algorithm for the case where k = 1. The only complication is to handle e-productions.

Here we shall describe an algorithm for the general case with $k \geq 1$. Whether or not this algorithm is more efficient than a standard algorithm for computing the $FIRST_k$ function is open.

Consider the situation in algorithm 6.2 where in state S we are to compute $LALR_k$ for the item $[A \rightarrow .\alpha]$. Let the items $[B_i \rightarrow \varphi_i . A \Psi_i]$, $i = 1, 2, \ldots, n$, $n > 0$, be all items in S with a dot before A.

In order to compute $LALR_k$ we need the sets $FIRST_k(\Psi_i)$, $i = 1, 2, \ldots, n$ and we assumed the existence of a function $FIRST_k$.

The set $\cup\{FIRST_k(\Psi_i) \mid i = 1, 2, \ldots, n\}$ may be computed by simulating all possible steps that the parse algorithm may take starting in the state $GOTO_k(S, A)$ with an empty parse stack. Let w be the string read during a path in the simulated parsing. A path is continued until either

[6.9a] $|w| = k$, or

[6.9b] the parser is about to reduce with one of the productions $B_i \rightarrow \varphi_i A \Psi_i$ and the depth of the parse stack is $|\Psi_i|$.

It is not sufficient to compute $\cup\{FIRST_k(\Psi_i) \mid i = 1, 2, \ldots, n\}$, as a w with $|w| < k$ needs to be k-concatenated with $LALR_k([B \rightarrow \varphi_i \cdot A \Psi_i], S)$ if $\Psi_i \Rightarrow^* w$. We must be able to distinguish between a w coming from a Ψ_i and a v from a Ψ_j with $i \neq j$. For this purpose we introduce a set of new symbols

$$[6.10] \quad \wp = \{ \#_{[A \rightarrow \alpha . \beta]} \mid A \rightarrow \alpha\beta \in P \}.$$

which will be used as markers, added to the end of strings w for which $|w| < k$.

We will then compute the set

$$[6.11] \quad F_{\#}(S,A) = \cup\{FIRST_k(\Psi \#_{[B \rightarrow \varphi . A\Psi]}) \mid [B \rightarrow \varphi . A\Psi] \in S \}$$

The above sketched algorithm must then be modified. In case [6.9b] the string w must be replaced by $w \#_{[B_i \rightarrow \varphi_i . A \Psi_i]}$.

In algorithm 6.2 the lines

$\underline{\text{FOR}}$ $[B \to \varphi . A \Psi] \in S$ $\underline{\text{DO}}$
$F := FIRST_k(\Psi)$; $LA := LA \cup \{w \mid |w| = k\}$;

are replaced by :

$FI := F_{\#}(S,A)$; $LA := LA \cup (FI \cap \Sigma^{*k})$;
$\underline{\text{FOR}}$ $[B \to \varphi . A \Psi] \in S$ $\underline{\text{DO}}$
$F := \{ w \mid w\#_{[B \to \varphi . A \Psi]} \in FI \}$.

In general it is not possible to simulate all steps of the parser in order to compute the set $F_{\#}(S,A)$. If the grammar contains e-productions or some nonterminal, B, is circular ($B \Rightarrow^{+} B$), then there may be strings that have an infinite number of right-parses. In such cases the parser would enter a loop where it only performs reductions. This is the same problem as that of simulating a nondeterministic bottom-up parser (c.f. [Aho & Ullman 72 a] p. 303).

Let a configuration of the parser be a pair consisting of the parse stack and the string read so far. Let (L,w) , (M,v) be configurations. Let the relation $\to$ be defined by $(L,w) \to (M,v)$ iff the parser in one step may go from (L,w) to (M,v). I.e. either by a read-transition or by a reduce transition.

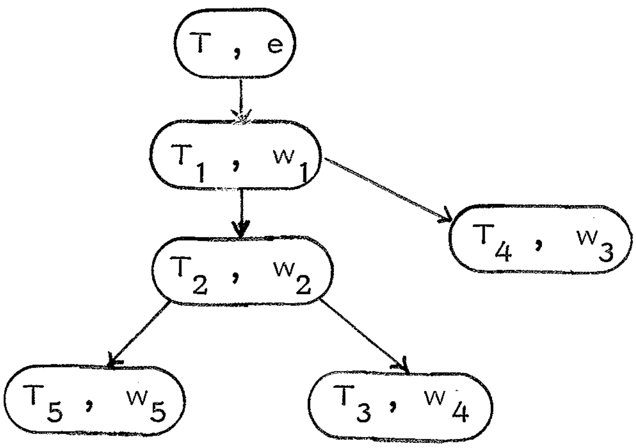
(1) if the grammar is circular we may reach a configuration (L,w) such that $(L,w) \to^{+} (L,w)$ is possible.

(2) if the grammar contains e-productions then we may reach a configuration (L,w) such that for all $n > 0$ there exist configurations (L_i,w) $(i = 1, 2, \ldots, n)$ such that $(L,w) \to^{+} (L_1,w) \to^{+} \ldots \to^{+} (L_n,w)$.

If one keeps track of all configurations being reached by the parser then it is easy to check for the above situations. In the second situation there is an $i > 0$ and a $j > i$ such that $(L,w) \rightarrow^+ (L_i, w) \rightarrow^+ (L_j, w)$ and $top(L_i) = top(L_j)$.

Below we describe a data structure that in a simple way keeps track of the configurations entered during the simulated parse and which makes testing for the two conditions simple.

The configurations may be collected in a tree where each node is a pair consisting of a state and a string :

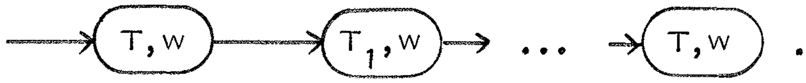


A node identifies a unique configuration in the following way :
The stack is the list of states in the nodes on the path from the root to the node. The string is the string at the node.

During the simulated parsing, the tree will be expanded with a node each time a transition is carried out.

If during parsing an attempt is made to add a node that is already in the tree then we have circularity.

If a branch in the tree has the following form then we have that the stack may grow infinitely



7 Comparison

Our LALR(1) algorithm was originally inspired by the one presented in [LaLonde 71]. This algorithm is presented in general terms leaving several questions unanswered and only a few arguments for its correctness are given. LaLonde's algorithm is very recursive and repetitive, but as stated in [LaLonde 71] this is nevertheless essential for computing localised lookahead. For this reason it is necessary to limit the amount of recursion as the algorithm otherwise might easily turn out to be inefficient. Lalondes algorithm involves a lot of overhead in keeping track of the so-called "mainline predecessor paths" and "side paths". The use of mainline predecessor paths and side paths seems to be an unnecessary complication. This complication seems to arise because of an attempt to compute $FIRST_k$ as an integral part of the LALR-algorithm. In our algorithm this corresponds to calling LALR-k recursively whenever the simulated parser (section 6.3) performs a reduction. However, there is a great difference between reductions that imply situation [6.9b] and those which do not. This clearly gives problems with handling the so-called side paths.

The reformulations of Theorem 4.3 into Theorems 5.3 and 6.1 are essential for avoiding this complication.

In [Andersen, Eve and Horning 73] two methods for constructing LALR(1)-parsers are treated. The first method involves the construction of LR(1)-items but a merging operation is used to combine states that only differ in their lookahead. A second method is in the form of a recursive expression of LALR(1)-lookahead. This expression is similar to Theorem 4.3 for $k = 1$. An outline of an actual algorithm is given without any proof of correctness. The outlined algorithm seems close to the LALR(1) algorithm of section 5, but on the other hand it is mentioned in [Andersen, Eve and Horning 73] that their technique is the one being used by [LaLonde 71].

In [Altman 73] a new type of grammar, the comprehensive LR(k)-grammars, are introduced. These grammars are situated inbetween SLR(k) and LALR(k) grammars and they are introduced in order to reduce the amount of overhead involved in LaLonde's algorithm.

The first LALR(k) definition in [Anderson, Eve & Horning 73] also appears in [Aho & Ullman 72a] and [De Remer 74]. Informally stated : all states in the LR(k)-machine that only differ in lookahead (having a common CORE) are merged into one state. If no conflicts arise, the grammar is said to be LALR(k). (This is the LALR(k) definition (2.6) used in this paper). Techniques based on an initial construction of a full LR(k)-machine, followed by a state merging are unrealistic for practical grammars, even in the case of k = 1. Especially space requirements are too expensive. Such approaches are thoroughly discussed in [Aho & Ullman 72b].

The Lane Tracing Algorithm of [Pager 77] is essentially another implementation of the methods of LaLonde, Anderson, Eve and Horning and our method. He gives a detailed description of an LALR(1)-algorithm. For a given item and state he computes a set of lanes. A lane is a list of pairs, (item, state) that appear in the recursive calls of algorithm 5.5. In order to reduce the number of different lanes that have to be treated, he makes optimisations that correspond to the way recursion is stopped in Algorithm 5.5. The upper bounds for his algorithm and algorithm 5.5 seem to be the same. However, his algorithm is very complicated and impenetrable and no correctness proof is given. It is claimed that similar principles can be used for a general LALR(k) algorithm but no details are given.

The methods described in [Johnson 74] and [Aho & Ullman 77] represent approaches that are quite different from the ones described above. Lookahead symbols are characterised as being generated either spontaneously or as propagating. Consider Theorem 5.3 and let w be in $LALR_1([A \to \alpha.\beta],T)$. If w is in $TRANS(GOTO_0(S,A))$ for some S, then w is generated spontaneously whereas if w comes from some

$LALR_1([B \to \varphi . A\psi], S)$ then w is said to propagate from the item $[B \to \varphi . A\psi]$ in S. The method of Aho & Ullman then consists of (1) computing all spontaneously generated symbols and (2) then keep on propagating symbols until no more propagation is necessary. This algorithm computes LALR(1) lookahead for all items in the LR(0)-machine. This is, however, usually not necessary as it is only necessary to compute the lookahead for conflicting items. According to Aho & Ullman the algorithm has been designed for speed and may take up too much space to be practical. A variant of this method is used in YACC [Johnson 74] where the use of space has been turned into time requirements.

Aho & Ullman do not give a correctness proof of their algorithm.

Since the submission of the original version of this paper, another efficient LALR(1) algorithm has been published in [De Remer & Pennello 79]. This algorithm resembles the one described in section 6.2, restricted to the k = 1 case. The main difference is that in [DeRemer & Pennello 79] the problem is transformed such that standard algorithms for directed graphs may be used. A brief comparison of the algorithms is included in [Kristensen & Madsen 80].

Acknowledgement

During the preparation of this paper we have received many helpfull comments from Peter Kornerup, Erik Meineche Schmidt, Peter Mosses and Thomas J. Pennello.

8. References

Aho, A.V., Hopcroft, J.E. and Ullman, J.D. [1976]
"The Design and Analysis of Computer Algorithms"
Addison-Wesley Publ. Comp., Mass., 1976.

Aho, A.V. and Ullman, J.D. [1972a, 1973]
"The Theory of Parsing, Translation and Compiling"
Vol. I & II, Prentice-Hall, Englewood Cliffs, N.J., 1973.

Aho, A.V. and Ullman, J.D. [1977]
"Principles of Compiler Design"
Addison-Wesley Publ. Comp., Mass., 1977.

Aho, A.V. and Ullman, J.D. [1972b]
"Optimization of LR(k) Parsers"
Journal of Computer and System Sciences 6, 573-602 (1972).

Altman, V.E. [1973]
"A Language Implementation System"
M A C TR-126, M.I.T., Cambridge, Mass. 1973.

Anderson, T., Eve, J., and Horning, J.J. [1973]
"Efficient LR(1) Parsers"
ACTA Informatica 2, 12-39 (1973).

DeRemer, F.L. [1969]
"Practical Translators for LR(k) Languages"
Ph.D.Diss., M.I.T., Cambridge, Mass., 1969.

DeRemer, F.L. [1971]
"Simple LR(k) Grammars"
Comm. ACM 14:7, 453-460, 1971.

DeRemer, F.L. [1974]
"Notes for a Course on Programming Linguistics"
Information Sciences, University of California,
Santa Cruz, California 95060, 1974.

DeRemer, F.L. and Pennello, T.J. [1979]
"Efficient Computation of LALR(1) Look-ahead Sets"
SIGPLAN Notices, 14 : 8, 176-187 (1979).

Eriksen, S.H., Jensen, B.B., Kristensen, B.B. and Madsen, O.L. [1973]
"The BOBS-System"
Computer Science Department, Aarhus University, 1973.
(revised version DAIMI PB-71, 1979).

Johnson, S.C. [1974]
"YACC - yet another Compiler Compiler"
CSTR 32, Bell Laboratories, Murray Hill, N.J., 1974.

Kristensen, B.B. and Madsen, O.L. [1980]
"A General Algorithm for Solving a Set of Recursive Equations (Exemplified by LR-theory)"
Computer Science Department, Aarhus University,
DAIMI PB-110, February 1980.

Lalonde, W.R. [1971]
"An Efficient LALR Parser Generator"
Computer Systems Research Group Technical Report 2,
University of Toronto, Canada, 1971.

Pager, D. [1977a]
"The Lane-Tracing Algorithm for Constructing LR(k) Parsers and Ways of Enchancing its Efficiency"
Inf. Sci. 12, 19-42 (1977).

Pager, D. [1977b]
"A Practical General Method for Constructing LR(k) Parsers"
ACTA Informatica 7, 249-268 (1977).

Appendix A - A correctness proof for a general algorithm.

The presented algorithms follow the same scheme for solving a set of recursive equations. Here we give a general proof for the correctness of this scheme.

Definition A1

Let F be a function from a set D into powersets of R and let F be defined by the following set of recursive equations :

$$\forall\ a \in D : \ F(a) = G(a) \ \cup \ \cup\{F(b) \mid b \in P_a\}$$

(*) where G is a function $D \rightarrow P(R)$ and for all $a \in D$, P_a is a powerset of D.

□

The following algorithm (A2) computes the smallest solution (F_0) to the above equations provided that the function G is correctly implemented and that the domain of D is finite.

The solution is smallest in the sense that if F_1 is another solution then $F_0(d) \subseteq F_1(d)$ for all $d \in D$.

(*) P(R) is the set of powersets of R.

Algorithm A2

```
FUNCTION F(a : D) : SET OF R ;
VAR  Res : SET OF R ;
     Done : SET OF D ;

     PROCEDURE  F1 (a : D) ;
     BEGIN
               Done : = Done ∪ {a} ;
               Res  : = Res  ∪ G (a) ;
               FOR b ∈ P_a WHERE b ∉ Done DO
                    F1 (b)
               ENDFOR ;
     END F1 ;

BEGIN
     Res : = Done : = ∅ ;
     F1 (a) ;
     F : = Res ;
END F ;
```

□

The functions computed by algorithm 5.5 and 6.2 do not directly have the same form as F but a simple transformation will do.

An upper bound on the number of activations of the local procedure F1 in algorithm A2 is

$$O(|D|)$$

for one activation of F.

By considering the FOR-loop in F1 to be of the kind

$$\underline{\text{FOR}}\ b \in P_a - \text{Done}\ \underline{\text{DO}}$$

then the algorithm F is linear in $|D|$.

To prove that algorithm A2 correctly computes the minimal solution to the equations in A1, we may split the set Done into Stack and Fin without affecting the result.

The elements in Stack correspond to recursive invocations of F1 that are on the runtime stack whereas the elements in FIN correspond to invocations of F1 that are processed.

The algorithm is equipped with assertions, and these are enclosed by ". As an abbreviation we define

$$FSF = \cup\{F_0(b) \mid b \in \text{Stack} \cup \text{FIN}\}$$
$$FS = \cup\{F_0(b) \mid b \in \text{Stack}\}$$

Res_P is the value of Res at assertion P.

Consider {P1} S {P2}. If x is a variable then x' in P2 is the value of x before the execution of S.

Algorithm A2 then appears as :

```
FUNCTION  F(a : D) : SET OF R;
VAR   Res : SET OF R ;
      FIN , Stack : SET OF D ;
      PROCEDURE F1 (a : D) ;
      BEGIN "{P} ≡ { FSF = Res  ∪ FS }"
            Stack : = Stack ∪{a} ; Res : = Res ∪ G(a) ;
            "{Q} ≡ {FSF = Res ∪ FS = Res_P  ∪ G(a) ∪ FS ∪
                                              ∪{F_0(c) | c ∈ P_a ∩ FIN } }"
            FOR  b ∈ P_a  WHERE ¬ b ∈ (Stack ∪ FIN) DO
              "{Q}"
              F1(b)
              "{R} ≡ {FSF = Res  ∪ FS = Res'  ∪ FS ∪ F_0(b)
                       ∧ b ∈ FIN ∧ Stack = Stack'}"
            END FOR ;
            "{Q ∧ ∀ b ∈ P_a  :  b ∈ Stack ∪ FIN }"
            Stack : = Stack - {a} ; FIN : = FIN ∪{a}
            "{S} ≡ {FSF = Res_p  ∪ FS ∪ F_0(a)}"
       END F1 ;

BEGIN   Res : = FIN : = Stack : = ∅
       "{FSF = Res ∪ FS ∧ Res = FIN = Stack = ∅}"
       F1(a)
       "{F_0(a) = Res  =  ∪{F_0(b) | b ∈ FIN} }"
       F : = Res
END  F ;
```

Lemma A3

$\{FSF = Res \cup FS\}$

$F1(a)$

$\{FSF = Res \cup FS = Res' \cup FS \cup F_0(a)\}$

$\{a \in FIN\} \wedge \{Stack = Stack'\}$

Proof

(a) We assume all inner calls of F1 to be correct and next verify that the body of F1 is correct.

(b) If P holds then Q will hold. Note that $\cup\{F_0(c) \mid c \in P_a \cap FIN\} \subseteq FSF$.

(c) R is true after the inner call.

(d) We must prove that R implies Q.
$Res' \cup FS$ in R is identical to the value of $Res \cup FS$ before the call F1(b) i.e.

$$
\begin{aligned}
R \supset \quad FSF &= Res \cup FS \\
&= Res_P \cup G(a) \cup FS \cup \\
&\qquad \cup\{F_0(c) \mid c \in P_a \cap FIN'\} \cup F_0(b) \\
&= Res_P \cup G(a) \cup FS \cup \\
&\qquad \cup\{F_0(c) \mid c \in P_a \cap FIN\}
\end{aligned}
$$

$\supset Q$

(e) $Q \wedge P_a \subseteq (Stack \cup FIN) \supset$

$$
\begin{aligned}
FSF = Res \cup FS = Res_P \cup FS \cup G(a) \cup \\
\cup\{F_0(c) \mid c \in P_a\}
\end{aligned}
$$

$= Res_P \cup FS \cup F_0(a).$

(f) Taking a off the stack and adding it to FIN does not change the above assertion, so we see that S holds.

(g) As D is finite, the FOR-loop and the recursion will stop as FIN is increased in each call.

□

Theorem A4

Algorithm A2 correctly computes the minimal solution to the equations in A1.

Proof :

Follows from lemma A1 and the pre/post assertion of the initial call of F1 in F.

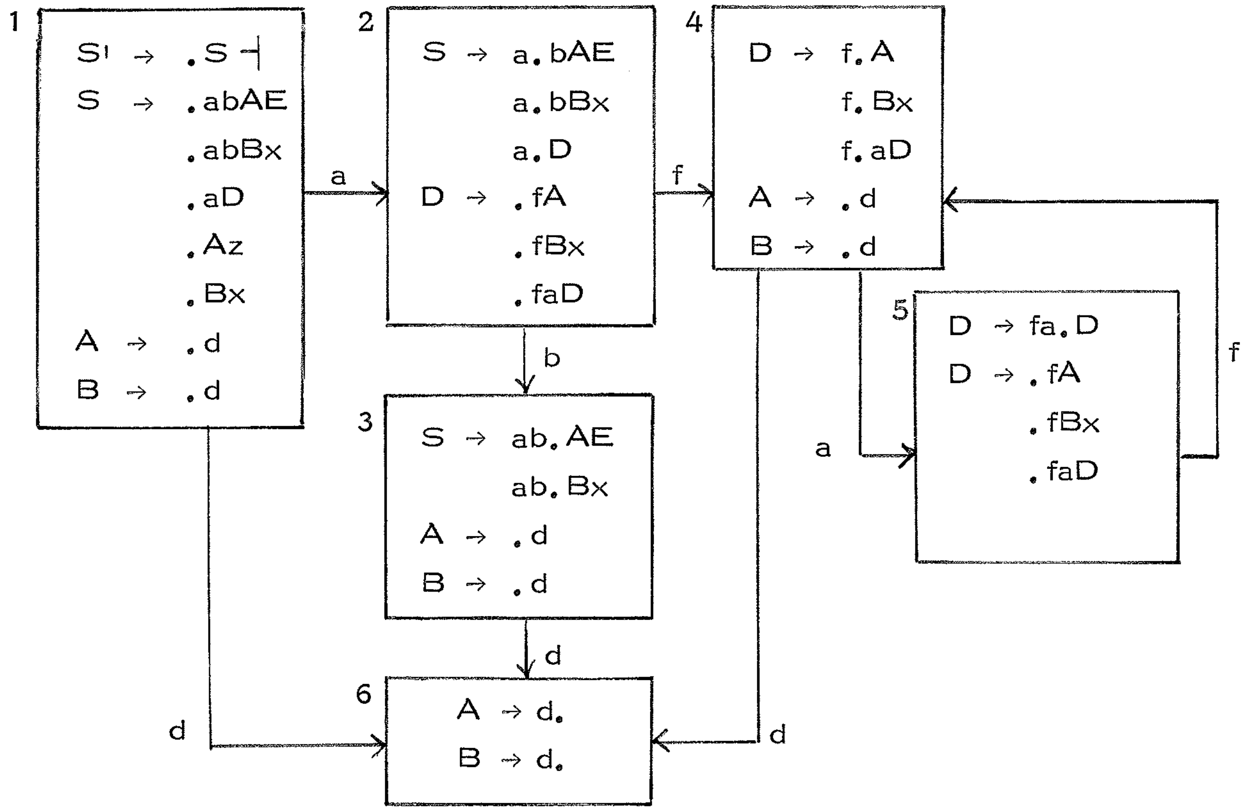
□

Appendix B Example

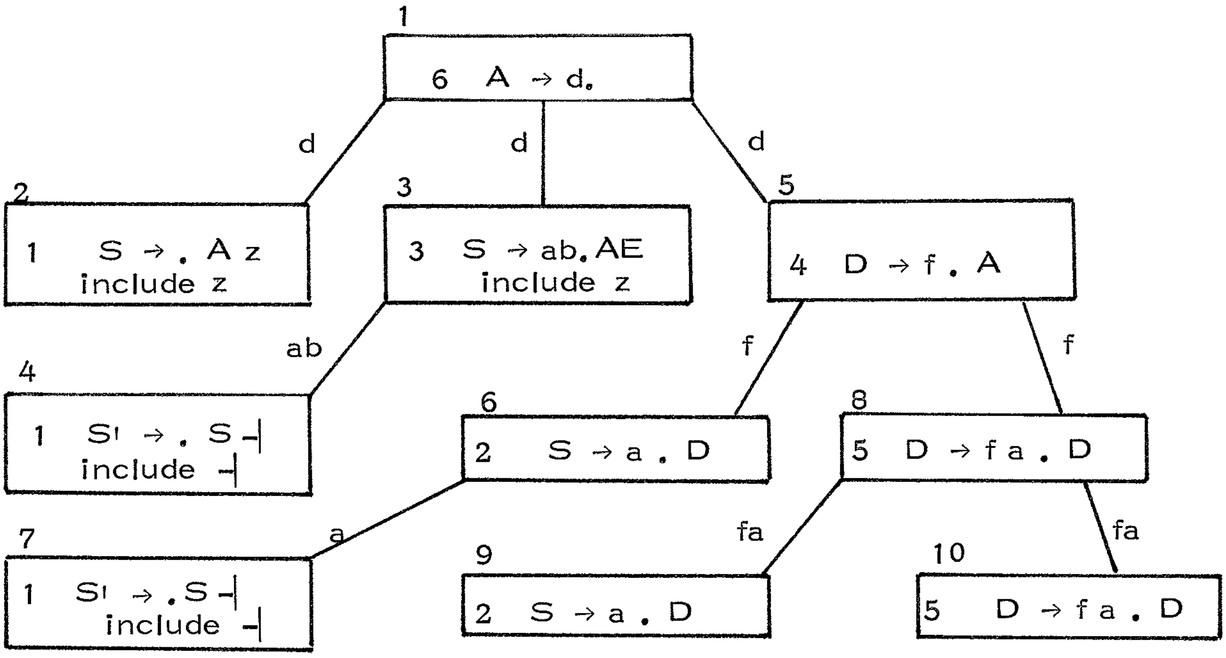
Here we give an example of a computation of LALR(1)-lookahead
Consider the grammar $\mathcal{G}$ defined by the productions

$S' \to S \dashv$
$S \to abAE \mid abBx \mid aD \mid Az \mid Bx$
$A \to d$
$B \to d$
$D \to fA \mid fBx \mid faD$
$E \to z \mid e$ (the empty string)

The interesting parts of the LR(0)-machine for $\mathcal{G}$ are given below



A call LALR-1([A → d.], 6) may traverse the following parts of the predecessor tree in the order given by the number of the boxes.



A box contains the number of the state and the considered item and the lookahead added in this state. The interior nodes in the predecessor tree corresponds to states where a recursive call is made. The leaves correspond to states where the recursion is stopped either because no item is of the form [B → φ . A Ψ] where Ψ ⇒* e (box 2, 4 and 7) or because the considered item and state is in Done (box 9 and 10).