ISSN 0105-8517

A PRACTICAL STATE SPLITTING ALGORITHM FOR CONSTRUCTING LR-PARSERS

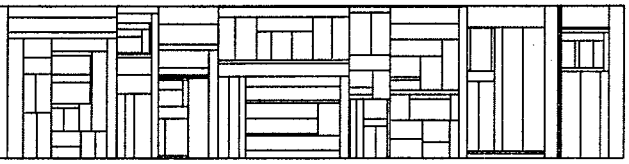
by

Bent Bruun Kristensen *
and
Ole Lehrmann Madsen

DAIMI PB-115
March 1980

* Aalborg University Center, Aalborg, Denmark

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



Abstract

A practical algorithm for constructing LR(k) parsers is given. The algorithm works by splitting those states in the LR(0)-machine that give rise to LALR(k)-conflicts. The algorithm takes a conflicting pair of items, say I, J in a state T, and performs a recursive backwards traversal of part of the predecessor tree of T. At each node pairs of items which contribute with lookahead to I and J in T are visited.

During the return from the recursion, states in the predecessor tree that give rise to LALR(k)-conflicts (between I and J in T) which are not LR(k)-conflicts, will be split. This splitting may involve unrolling of loops and separation of loops into several loops.

Contents

I. Introduction

In [De Remer 69 and 71] it is proposed to construct LR(k)-parsers indirectly by constructing the LR(0)-machine and then resolve possible LR(0)-conflicts by means of SLR(k)- or LALR(k)-lookahead. If this fails parsing conflicts may be resolved by splitting states in the LR(0)-machine. However, no practical solution for the state splitting approach is given.

The purpose of this paper is to present a practical algorithm for the state splitting approach. This algorithm will construct a parser for all LR(k)-grammars.

In [Kristensen & Madsen 79b] algorithms for testing LR(k)-ness of a grammar on the basis of its LR(0)-machine were given. These algorithms were a further development of the algorithms for computing LALR(k)-lookahead which were presented in [Kristensen & Madsen 79a]. In these algorithms LALR(k)-lookahead is computed by (recursively) tracing backwards in the predecessor tree of an item in a state T. The LR(k) testing is done by taking conflicting items two by two, say I, J in state T, and tracing backwards with both items simultaneously. In a predecessor state it is then checked whether the lookahead contribution from this state will give rise to LR-conflicts between I and J in T or not. In case of such a conflict there will be a corresponding conflict in the LR(k)-machine. If there are no such conflicts then the possible LALR conflicts are caused by the (implicit) merging of states with identical CORE.

This idea is developed further in this paper. If during the test for LR(k)-ness no conflicts are found it is possible to perform state splitting and avoid the conflicts caused by implicit merging.

In practice most (well prepared) grammars turn out to be LALR(k) and then it is not necessary to perform state splitting. Consequently very few parser generators accept more than LALR(k) grammars. However, the notion of an LALR(k)-grammar is hard to understand for a (non-LR expert) user of a parser generator system.

The experience with the BOBS-system ([Eriksen et. al. 73]) is that most of the LALR conflicts are caused by grammars which are not LR(k) (often because of ambiguities in the grammar). Such conflicts are relatively easy to repair as the notions of LR(k)-grammars and ambiguous grammars are understandable for most users of a parser-generator system. Conflicts caused by a grammar which is LR(k) but not LALR(k) are often hard to understand and repair as it requires detailed knowledge about the technique used to construct an LR-parser. Such conflicts may appear in practice as the following example shows :

```
<stmt-list> ::= <stmt> | <stmt-list> ; <stmt>
<stmt> ::= <proc-id> | <var> := <exp>
<var> ::= <identifier>
<proc-id> ::= <identifier>
<exp> ::= <var> | <proc-id> (<parameter-list>)
```

The problem for an LALR(1) parser is to decide whether an <identifier> is a <var> or a <proc-id>. This LALR-conflict can be repaired by replacing <proc-id> by <func-id> in the second alternative for <exp>, and introduce the additional production <func-id> ::= <identifier>.

All together we find it highly desirable that a parser generator accepts all LR(k)- (or LR(1)-) grammars. The state splitting algorithm presented in this paper is a sufficiently simple and practical way of achieving this.

In section 2 a summary of the necessary LR theory is given. The concept of a splitted LR(0)-machine is defined in section 3. The state splitting algorithm is presented in section 4 and proved correct in section 5. The paper is concluded in section 6 by discussing various improvements and making comparisons with related work. An example of using the state splitting algorithm is given in the appendix.

Acknowledgement

During the preparation of this paper we have received many helpful comments from Peter Kornerup.

2. Basic Terminology and Results

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

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 $S' \rightarrow S \dashv^k$, 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 M is a set of subsets of some set N then $\cup M$ means

$$\{x \in N \mid x \in m \ \& \ m \in M\}.$$

We shall repeat some definitions and theorems often in a slightly modified form.

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 $M_k^G \times (N \cup \Sigma) \rightarrow M_k^G$

□

For a given grammar G we shall assume the existence of its LRM_k^G on this form. The superscript G is omitted when this causes no confusion.

Let $T \in M_k$, then

$$KERNEL(T) = \begin{cases} \{[S' \rightarrow .S \dashv^k]\} & \text{if } T = IS_k \\ \{[A \rightarrow \alpha . \beta, u] \in T \mid |\alpha| > 0\} & \text{if } T \neq IS_k \end{cases}$$

The following definitions and theorems summarize the notions and results necessary in the following sections :

Definitions

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

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

$$LR_k([A \rightarrow \alpha.\beta], T) = \{u \mid [A \rightarrow \alpha.\beta, u] \in T\}.$$

(2.3) Let $[A \rightarrow \alpha.\beta, u]$ be a LR(k)-item and let $S \in M_k$, then

$CORE([A \rightarrow \alpha.\beta, u]) = [A \rightarrow \alpha.\beta]$, and
$CORE(S) = \{CORE(I) \mid I \in S\}$.

We shall not distinguish between the items $[A \rightarrow \alpha.\beta, e]$ and $[A \rightarrow \alpha.\beta]$.

(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

$$LALR_k([A \rightarrow \alpha.\beta], T) = \cup\{LR_k([A \rightarrow \alpha.\beta], S) \mid S \in URCORE_k(T)\}.$$

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

$$v \notin FIRST_k(EFF_k(\beta)\, u),$$

or equivalently that

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

(We shall use 2.6 as a definition. It may be found in [Aho & Ullman 72] as a theorem.)

(2.7) 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

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

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

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

□

Theorems

(2.9) Let $T \in M_0$ and let $[A \to \alpha . \beta] \neq [S' \to . S \dashv^k]$, then

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

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

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

where

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

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

(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

$$LR_k([A \to .\alpha], T) =$$

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

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

$$LR_k([A \to \alpha . \beta], T) = \{w \mid w \in FIRST_k(y) \wedge$$

$$S' \Rightarrow^*_{rm} \gamma Ay = \gamma\alpha\beta y \wedge GOTO_k(IS_k, \gamma\alpha) = T\}$$

Proofs

2.9 and 2.10 have been proved in [Kristensen & Madsen 79a]. 2.11, 2.12 and 2.13 follow directly from section 5.2.3 in [Aho & Ullman 72].

□

Notation 2.14

The following constructs are used in the algorithms:

ASSUME ...;

is used for name giving of (components of) structured variables.

FOR a ∈ M WHERE P_a DO S ENDFOR;

means

FOR a ∈ M DO

IF P_a THEN S ENDIF

ENDFOR ;

□

3. The k-splitted LR(0)-machine

In this section we introduce the operations used for splitting states in the LR(0)-machine. These operations are the ISOLATE- and the $MERGE_k$-operations.

The purpose of the ISOLATE-operation is as follows : Let $\mathbb{m} \subseteq PRED(T, X)$ for some state T accessed by X. ISOLATE may isolate $\mathbb{m}$ from the remaining predecessor states of T by creating a new copy of T, T', and letting $\mathbb{m}$ be the predecessor states of T'. $MERGE_k$ merges identical states, with respect to CORE and LA_k-lookahead, into a single state. ISOLATE, $MERGE_k$, and LA_k are formally defined below.

The state splitting algorithms use these two operations successively on the LR(0)-machine in order to perform the necessary state splitting. The resulting machine is called a k-splitted LR(0)-machine.

Definition 3.1

Let G be a CFG. A k-splitted LR(0)-machine for G, $SM_k = (M, IS, GOTO)$, is either :

- the LR(0)-machine for G, - or
- the result of applying either the ISOLATE- or the $MERGE_k$-operation to a k-splitted LR(0)-machine for G.

□

For the rest of this paper SM_k will denote the class of k-splitted LR(0)-machines. If $Q \in SM_k$, then Q will always have the form

$$Q = (M, IS, GOTO)$$

unless otherwise stated.

Definition 3.2

The function ISOLATE is defined as follows :

Let $Q \in SM_k$ and let $T \in M$; let $PRED(T, X) = \mathfrak{R} \cup \mathfrak{S}$ such that $\mathfrak{R} \neq \emptyset$, $\mathfrak{S} \neq \emptyset$, and $\mathfrak{R} \cap \mathfrak{S} = \emptyset$.

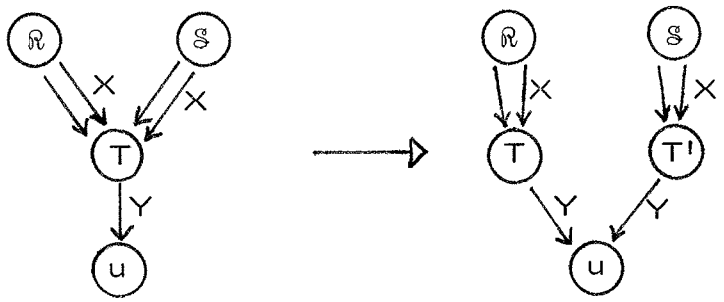
Let $(Q', T') = ISOLATE(Q, T, \mathfrak{S})$. Then $Q' = (M', IS, GOTO')$ is an SM_k which is identical to Q except that

- $M' = M \cup \{T'\}$ where T' is a new state not in M and $CORE(T') = CORE(T)$,
- $\forall S \in \mathfrak{S} : GOTO'(S, X) = T'$,
- $\forall Y \in N \cup \Sigma : GOTO'(T', Y) = GOTO(T, Y)$,
- If $T \in \mathfrak{S}$ then $GOTO'(T', X) = T'$

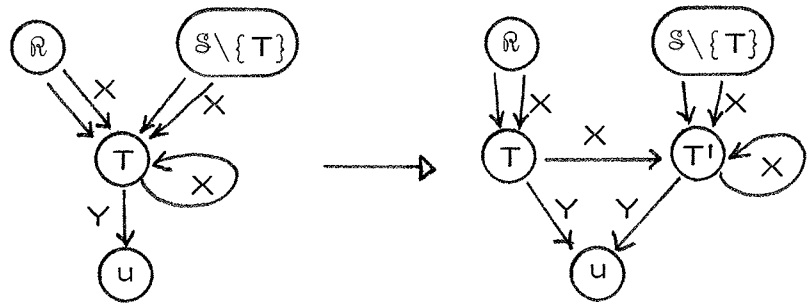
□

The following pictures illustrate the effect of an ISOLATE operation. [3.3a] shows the case when $T \notin \mathfrak{S}$ and [3.3b] shows the case where $T \in \mathfrak{S}$.

[3.3a]



[3.3b]



Definition 3.4

The function $MERGE_k$ is defined as follows :

Let G be an SM_k, let $T, T' \in M$, and let $CORE(T') = CORE(T)$. Let $Q' = MERGE_k(Q, T', T)$. Then $Q' = (M', IS, GOTO')$ is an SM_k which is identical to Q except that

- $M' = M \setminus \{T\}$,
- $\forall R \in PRED(T, X) : GOTO'(R, X) = T'$.

□

In theorem 2.9 and 2.10 the $LALR_k$-function is expressed in terms of the LR(0)-machine. An equivalent lookahead-function, LA_k, may be defined in terms of an SM_k.

Definition 3.5

Let Q be an SM_k and $T \in M$. The LA_k-lookahead (of length k) for an item $[A \to \alpha . \beta] \in T$, denoted $LA_k([A \to \alpha . \beta], T)$ is defined as the minimal solution to the following set of equations :

(i) $LA_k([S' \to . S \dashv^k], IS) = \{e\}$

(ii) Let $[A \to \alpha X . \beta] \in T$, then

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

(iii) Let $[A \to . \alpha] \in T$ and let $A \neq S'$, then

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

□

Remark 3.6

The definitions of GOTO, PRED, $URCORE_k$, CORE, and consistency (2.6, 2.7) may be extended in a straightforward way to an SM_k.

Definition 2.5 and the theorems 2.9, 2.11 and 2.13 may all be generalized to LA_k-lookahead and SM_ks.

Theorems

Let Q be an SM_k, $T \in M$, and $[A \to \alpha . \beta] \in T$, then

$$[3.7]\ LA_k([A \to \alpha . \beta], T) = \cup\{LA_k([A \to . \alpha\beta], S) \mid S \in PRED(T, \alpha)\}$$

$$[3.8]\ \exists\, \mathbb{m} \subseteq URCORE_k(T) :$$
$$LA_k([A \to \alpha . \beta], T) = \cup\{LR_k([A \to \alpha \circ \beta], S) \mid S \in \mathbb{m}\}.$$

$$[3.9]\ LA_k([A \to \alpha . \beta], T) = \{w \mid w \in FIRST_k(y) \wedge GOTO(IS, \gamma\alpha) = T \wedge$$
$$S' \Rightarrow^*_{rm} \gamma A y \Rightarrow \gamma \alpha \beta y\}$$

□

Now if all the states in an SM_k are consistent then it is straightforward to construct a set of corresponding LR(k)-tables which are equivalent to the canonical set of LR(k)-tables ([Aho & Ullman 72]), and thus to construct a valid parser.

4. The State Splitting Algorithm

In this section we shall step by step present a state splitting algorithm leading to an SM_k-machine which is consistent iff the grammar is LR(k).

Let $[A \to \alpha.\beta]$ and $[B \to \delta.]$ be two conflicting items in a state $T \in M$ of an SM_k. We have an LA_k-conflict if

$$EFF_k(\beta) \oplus_k LA_k([A \to \alpha.\beta], T) \cap LA_k([B \to \delta.], T) \neq \emptyset$$

(Initially we are thus considering an LALR-conflict in the LR(0)-machine).

In the following we assume that T is always accessed by the symbol X. The LA_k-lookahead of the two items have contributions from all predecessor states of T (PRED(T, X)). These contributions have one of the following two forms :

[4.1a] Propagation :

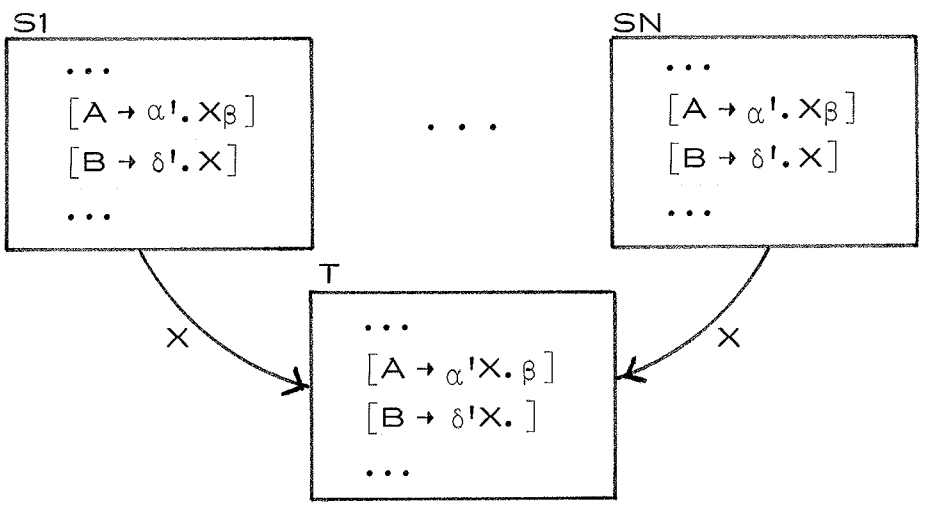
Let $\alpha = \alpha'X$ and $\delta = \delta'X$ then using 3.5 we have that

$$LA_k([A \to \alpha'X.\beta], T) = \cup\{LA_k([A \to \alpha'.X\beta], S) \mid S \in PRED(T, X)\}$$

and

$$LA_k([B \to \delta'X.], T) = \cup\{LA_k([B \to \delta'.X], S) \mid S \in PRED(T, X)\}.$$

Consider the following picture :



If for all $S \in PRED(T,X) = \{S1,\ldots,SN\}$ we have that

$$EFF_k(\beta) \oplus_k LA_k([A \to \alpha'.X\beta], S) \cap LA_k([B \to \delta'.X], S) = \emptyset$$

then there will be no conflicts between the two items in the corresponding states in the LR(k)-machine. $URCORE_k(T)$ will contain at least two states. We can solve the conflict in the LR(0)-machine by splitting T in as many copies as necessary (at most one for each predecessor of T) using the ISOLATE operation.

[4.1b] Spontaneity:

Let $\delta = e$ and $\alpha = \alpha'X$ (the cases where $\alpha = e$ or $\alpha = \delta = e$ may be treated similarly). Then it is easy to see that

$$(*) \quad LA_k([B \to \delta.], T) = \cup\{FIRST_k(\tau\Psi) \oplus_k LA_k([C \to \varphi.D\Psi], T) \mid [C \to \varphi.D\Psi] \in T \wedge B\tau \in EFF'_k(D) \wedge \varphi = \varphi'X\}$$

We now have a similar situation to [4.1a] in the following sense : We may eliminate possible LA_k-conflicts in T by splitting T if these conflicts are caused by the implicit merging of the X-successors of states in PRED(T,X). This is the case if the following predicate is true :

$$\forall S \in PRED(T,X):$$
$$EFF_k(\beta) \oplus_k LA_k([A \to \alpha'.X\beta], S) \cap$$
$$\cup\{FIRST(\tau\Psi) \oplus_k LA_k([C \to \varphi'.XD\Psi], S) \mid [C \to \varphi'.XD\Psi] \in S \wedge B\tau \in EFF'_k(D)\} = \emptyset$$

This is equivalent to the predicate :

$$\forall S \in PRED(T,X)\ \forall [C \to \varphi'.XD\Psi] \in S \text{ where } B\tau \in EFF'_k(D):$$
$$EFF_k(\beta) \oplus_k LA_k([A \to \alpha'.X\beta], S) \cap FIRST_k(\tau\Psi) \oplus_k LA_k([C \to \varphi'.XD\Psi], S) = \emptyset$$

(*) EFF'_k denotes the obvious extension of $EFF_k : (N \cup \Sigma)^* \to \Sigma^*$ to $EFF'_k : (N \cup \Sigma)^* \to (N \cup \Sigma)^*$.

If we have conflicts in some predecessors then we may in the same way attempt to (recursively) split those predecessors in order to remove the conflicts. In the general case propagation involves two items of the form $[A \to \alpha X.\ \beta]$ and $[B \to \delta X.\ \gamma]$ and spontaneity involves two items of the form $[A \to \alpha.\ \beta]$ and $[B \to .\ \delta]$.

Assume that we initially are considering the items $[A \to \alpha.\ \beta]$ and $[B \to \delta.\ ]$ in some state T. In case of spontaneity we must keep track of the sets $FIRST_k(\tau\Psi)$ when we trace backwards. At each level of the recursion we will thus have two items I, J , a state R and two sets $\mathfrak{A}, \mathfrak{B} \subseteq \Sigma^{*k}$ such that $\mathfrak{A} \oplus_k LA_k(I, R) \subseteq EFF_k(\beta) \oplus_k LA_k([A \to \alpha.\ \beta], T)$ and $\mathfrak{B} \oplus_k LA_k(J, R) \subseteq LA_k([B \to \delta.\ ], T)$.

If $\mathfrak{A} \oplus_k LA_k(I,R) \cap \mathfrak{B} \oplus_k LA_k(J,R) \neq \emptyset$ then we have an LA_k-conflict <u>involving</u> $(\mathfrak{A}, I, \mathfrak{B}, J, R)$ in the sense that there is an LA_k-conflict between $[A \to \alpha.\ \beta]$ and $[B \to \delta.\ ]$ in T.

The pairs $(\mathfrak{A}', I')$ where $\mathfrak{A}' \subseteq \Sigma^{*k}$ and $I' \in KERNEL(R)$ generated in case of spontaneity are captured by the following definition .

<u>Definition 4.2</u>

Let $T \in M$, $[A \to \alpha.\ \beta] \in T$ and $\mathfrak{A} \subseteq \Sigma^{*k}$, then

$$ORIGIN_k(\mathfrak{A}, [A \to \alpha.\ \beta], T) =$$

$$\begin{cases} \{(\mathfrak{A}, [A \to \alpha.\ \beta])\} & \text{if } \alpha \neq e \\ \cup\{ORIGIN_k(\mathfrak{A} \oplus_k FIRST_k(\Psi), [B \to \varphi.A\Psi], T) \mid [B \to \varphi.A\Psi] \in T\} & \text{if } \alpha = e \end{cases}$$

□

<u>Lemma 4.3</u>

Let $T \in M$, $I \in T$, and $\mathfrak{A} \subseteq \Sigma^{*k}$, then

$$\mathfrak{A} \oplus_k LA_k(I, T) = \cup\{\mathfrak{B} \oplus_k LA_k(J, T) \mid (\mathfrak{B}, J) \in ORIGIN_k(\mathfrak{A}, I, T)\}$$

□

As we mostly consider pairs of items we shall extend $ORIGIN_k$ in the following way

$$ORIGIN_k(\alpha, I, \beta, J, T) = \{(\alpha 1, I1, \beta 1, J1) \mid (\alpha 1, I1) \in ORIGIN_k(\alpha, I, T) \wedge (\beta 1, J1) \in ORIGIN_k(\beta, J, T)\}$$

In the rest of this paper $\mathcal{P}$ will always have the form $\mathcal{P} = (\alpha, I, \beta, J, T)$ where $\alpha, \beta \subseteq \Sigma^{*k}$, $I, J \in T$, and $T \in M$.

Notation 4.4

If I is an item of the form $[C \to \varphi Y . \Psi]$ then

$$\overleftarrow{I} = [C \to \varphi . Y\Psi]$$

□

The checks for conflicts in the predecessor states described in [4.1] may now be formulated as the following checks

$\forall S \in PRED(T, X)$:

$\forall (\alpha, I, \beta, J) \in ORIGIN_k(EFF_k(\beta), [A \to \alpha . \beta], \{e\}, [B \to \delta .], T)$:

$\alpha \oplus_k LA_k(\overleftarrow{I}, S) \cap \beta \oplus_k LA_k(\overleftarrow{J}, S) = \emptyset$

If there are conflicts in some predecessor state, S, then we may (as mentioned) attempt to split S by tracing further backwards to the predecessors of S. This may be expressed recursively by the following (yet incomplete) algorithm.

[4.5] Algorithm

```
VAR Q : SM_k ;
PROCEDURE LA-SPLIT (P) ;
BEGIN
  IF G ⊕_k LA_k(I, T) ∩ B ⊕_k LA_k(J, T) ≠ ∅ THEN
    FOR S ∈ PRED(T, X) ,
      (G1, I1, B1, J1) ∈ ORIGIN_k(P)
    DO
      LA-SPLIT(G1, ←I1, B1, ←J1, S)
    ENDFOR;
      Q : = Split(Q, T) ;
  ENDIF
END
```

The initial calls have the form

$$\text{LA-SPLIT}(\text{EFF}_k(\beta),\ [A \to \alpha.\beta],\ \{e\},\ [B \to \delta.\ ],\ T)$$

□

The function Split (defined in 4.7) isolates a number of new states from a given state T. Each new state T' has the property that no LA_k-conflict appears as the result of the implicit merge of the X-successor state of states in PRED(T', X) into T'. This property is expressed by the following predicate :

Definition 4.6

Let $\mathbb{M} \subseteq \text{PRED}(T, X)$. Then

$$\text{DISJOINT}(\mathbb{M}, P) \equiv$$

$$\forall S, S' \in \mathbb{M} \text{ such that } S \neq S' :$$
$$\forall (G1, I1, B1, J1) \in \text{ORIGIN}_k(P) :$$
$$G1 \oplus_k LA_k(\overleftarrow{I1}, S) \cap B1 \oplus_k LA_k(\overleftarrow{J1}, S') = \emptyset \quad \wedge$$
$$B1 \oplus_k LA_k(\overleftarrow{J1}, S) \cap G1 \oplus_k LA_k(\overleftarrow{I1}, S') = \emptyset$$

□

We may now define the function Split.

Definition 4.7

The function Split is defined as follows :

Let Q be an SM_k and let $T \in M$. Let $Q' = Split(Q, \mathbb{P})$. $Q' = (M', IS, GOTO')$ is identical to Q except as described below :

Let $\mathbb{m} = PRED(T, X)$.

(i) $\mathbb{m}$ is partioned into $\mathbb{m}_1 \cup \mathbb{m}_2 \cup \ldots \mathbb{m}_p$, $p \geq 1$, such that

- each set of states $\mathbb{m}_i$ ($i \in [1, p]$)satisfies $DISJOINT(\mathbb{m}_i, \mathbb{P})$, and

- for all $i, j \in [1, p]$, $i \neq j$ the value $DISJOINT(\mathbb{m}_i \cup \mathbb{m}_j, \mathbb{P})$ is false

  (this is in fact an optimization which is unnecessary, but which prevents an unnecessary splitting)

(ii) A set of new states is isolated from T by executing :

```
Q' : = Q ;
FOR i : = 1 TO p-1 DO
  (Q', T_i) : = ISOLATE(Q', T, m_i)
ENDFOR
```

- Consequently $M' = M \cup \{T_1, \ldots, T_{p-1}\}$

□

Algorithm [4.5] is incomplete for the following three reasons : (1) the initial state has no predecessors, (2) if the grammar happens to be non LR(k) then the recursion will never terminate, (3) if there are cycles in the predecessor tree then the procedure may loop forever.

For case (2) we note that if there exists an $R \in URCORE_k(T)$ where

$$\mathbb{G} \oplus_k LA_k(I, T) = \mathbb{G} \oplus_k LR_k(I, R)$$

then further splitting of T will not resolve any LA_k-conflicts involving $\mathcal{P}$. Such LA_k-conflicts will then appear as LR_k-conflicts in R. The following predicate $BOTTOM_k$ turns out to be a sufficient condition for this :

Definition 4.8 [Kristensen & Madsen 79b]

Let $T \in M$, $I \in T$ and $\mathcal{Q} \subseteq \Sigma^{*k}$, then

$$BOTTOM_k(\mathcal{Q}, I, T) \equiv \forall y \in \mathcal{Q} : |y| = k$$

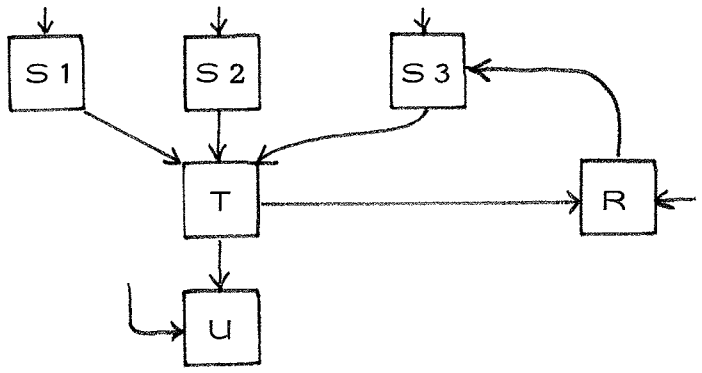
□

We extend $BOTTOM_k$ in the following way

$$BOTTOM_k(\mathcal{P}) \equiv BOTTOM_k(\mathcal{Q}, I, T) \vee BOTTOM_k(\mathcal{R}, J, T).$$

The recursion may now be stopped if we have $BOTTOM_k$. We shall in the following algorithms only use $BOTTOM_k$ and not the check for conflicts as in [4.5]. Note that this use of $BOTTOM_k$ will also handle case (1).

The problems with cycles in the predecessor tree may be illustrated by the following example

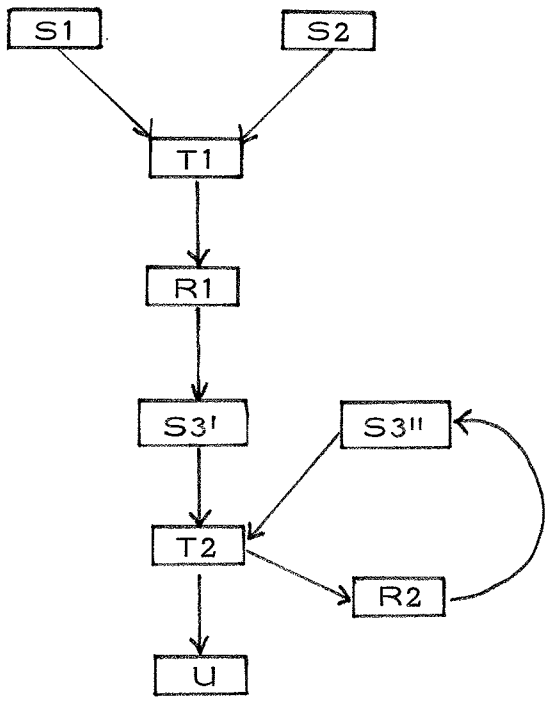


[4.9]

Suppose that we from the state U make an attempt to split T. This may imply attempts to split S 1, S 2 and S 3. The splitting of S 3 may imply an attempt to split R and then to split T again. We then have the risk of entering an infinite loop.

The loop in [4.9] is the result of a merge of corresponding loops in the LR(k)-machine. The loops in the LR(k)-machine may be of one of the following two different forms :

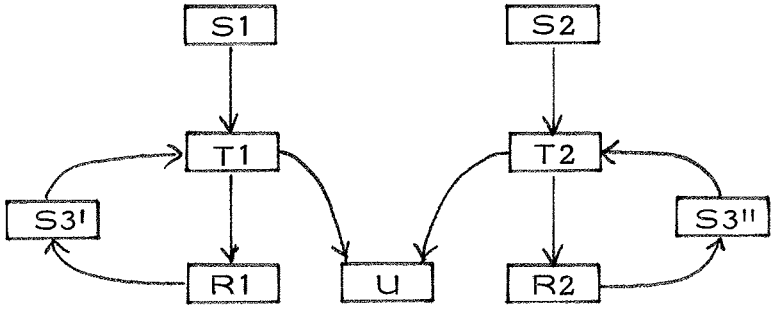
Loop unrolling



[4.10]

where CORE(T1) = CORE(T2) = T
CORE(S3') = CORE(S3'') = S3
CORE(R1) = CORE(R2) = R

Loop separation



[4.10b]

where CORE(T1) = CORE(T2) = T ,
CORE(S3') = CORE(S3'') = S3,
CORE(R1) = CORE(R2) = R

The loop involving T may be unrolled several times and at the same time the loop may be separated.

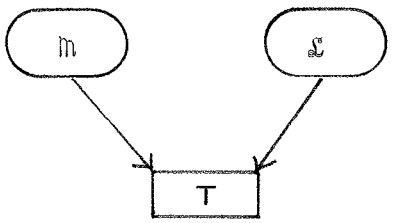
Using a stack we may check whether a cycle is entered or not. Similarly we may mark a predecessor which is part of a cycle. Algorithm 4.5 will be modified such that the recursion is stopped when a cycle is recognized.

In algorithm 4.11 we use a set Stack to collect all states "on the runtime stack". The set Blind is used to collect all predecessors that are part of a cycle. We do not make a recursive call on states which have been collected in Blind ∪ Stack. If PRED(T,X) ⊆ Blind ∪ Stack we put T in Blind. The next (yet incomplete) state-splitting algorithm will look as follows :

[4.11] Algorithm

```
VAR Q : SM_k ;
PROCEDURE LA-SPLIT(P) ;
BEGIN Stack := Stack ∪ {T} ;

  IF ¬BOTTOM_k(P) THEN

  L : FOR S ∈ PRED(T,X) \(Blind ∪ Stack) ,
                (Q1, I1, B1, J1) ∈ ORIGIN_k(P)
      DO
           LA-SPLIT(Q1, ←I1, B1, ←J1, S)
      ENDFOR ;

    IF PRED(T,X) ∩ (Blind ∪ Stack) = ∅ THEN Q := Split(Q,P)
    ELSEIF PRED(T,X) ⊆ (Blind ∪ Stack) = THEN Blind:=Blind ∪ {T}
    ELSE Split-loop
    ENDIF
  ENDIF ;

  Stack := Stack \ {T}

END LA-SPLIT ;
```

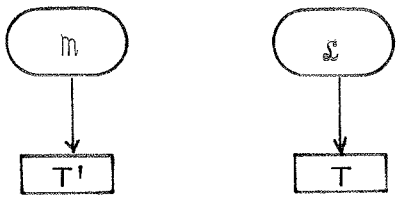
In the third case (Split-loop) we know that some of T's predecessors have been split properly, whereas others are in Blind ∪ Stack. This may be pictured as follows :



where $\mathbb{m} = PRED(T,X) \setminus (Blind \cup Stack)$
and $\mathfrak{L} = PRED(T,X) \cap (Blind \cup Stack)$.

Note that $T \in PRED(T,X)$ implies that $T \in \mathfrak{L}$.

We may first isolate $\mathbb{m}$:



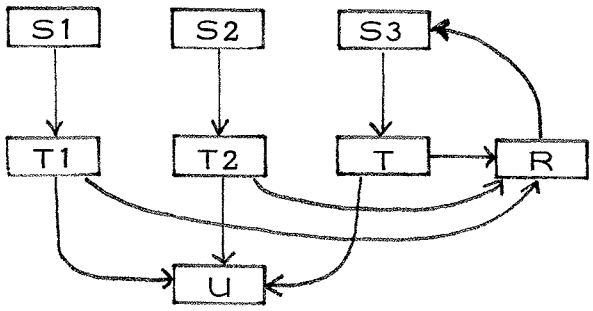
T' can now be properly split as none of its predecessors are in Blind ∪ Stack. Unfortunately we can in general do nothing with T. The predecessors of $\mathfrak{L}$ may reach T' if $GOTO(T',Y) \in Blind$ for some Y. This means that the states in Blind are no longer necessarily part of a cycle. We may, however, make a new attempt to split T. We shall thus make the following refinement of Split-loop :

[4.12] Split-loop

$(Q, T') := ISOLATE(Q, T, \mathbb{m})$;
$Q := Split(Q, (\mathbb{G}, I, \mathbb{B}, J, T'))$;
$Blind := \emptyset$;
GOTO L ;

□

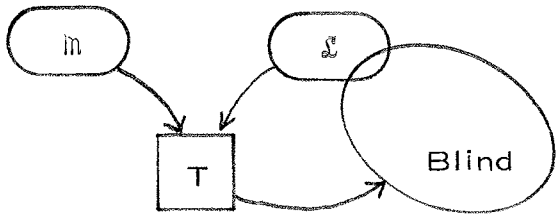
This may transform [4.9] into



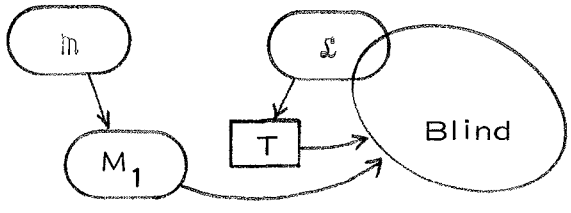
[4.13]

Repeating the splitting process of $\mathfrak{L}$ may imply a splitting of states which could not be split before (in [4.9], [4.13] S3 and R). It is, however, obvious that we will still have a cycle involving T. We may have to repeat the splitting process as the loop involving T may have to be unrolled a number of times.

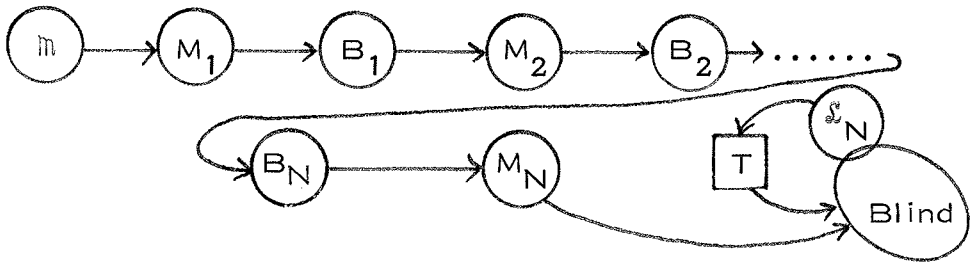
After the first execution of the repeat-loop we have the following situation :



An execution of Split-loop gives :



where M_1 are the states isolated from T. After N executions of the repeat-loop we have the following situation :



During the unrolling it will eventually happen that some state $T' \in M_i$ will be identical (with respect to LA_k-lookahead) to some state $T'' \in M_j$, $j < i$, and T'' is part of the predecessor tree for T'. If this is the case we may eliminate T' by using $MERGE_k(Q, T'', T')$ and in this way close a loop. The final refinement of Split-loop ([4.12]) will thus be as follows

[4.14] Split-loop

```
(Q, T') : = ISOLATE(Q, T, ℍ) ;
Q : = Split (Q, (ℂ, I, 𝔹, J, T')) ;
Blind : = ∅ ;  N : = N + 1,
Close-loop(N) ;
GOTO L ;
```

□

N has the initial value 0. Close-loop is defined as follows :

[4.15] Close-loop (N : INTEGER)

```
FOR (T', T'') ∈ (M_i, M_j)
  WHERE  i ∈ [1,N] , j ∈ [1, i-1]
  AND    T'' ∈ Predecessor tree (T')
  AND    ∀ I ∈ T' : LA_k(I, T') = LA_k(I, T'')
DO
  Q : = MERGE_k(Q, T'', T') ;
ENDFOR
```

□

For some N there will be an M_i which is empty after the execution of Close-loop(N). A succeding call of LA-SPLIT(ℙ) will imply that $PRED(T, X) \subseteq$ Blind. In this case T will be added to Blind, and the repeat-loop will terminate.

We are now ready to present the final algorithm. First we shall make one more change partly to simplify the proof and partly for optimizing the algorithm.

First we note that we do not treat arbitrary $\mathfrak{P}$'s, but only $\mathfrak{P}$'s that can be generated by LA-SPLIT from the initial calls. These $\mathfrak{P}$'s are characterized by the following definition and lemma.

Definition 4. 16

The predicate proper is defined inductively as follows:

(i) If $A \neq S'$ and $B \neq S'$ then
$(\{e\}, [A \to \alpha.], EFF_k(\delta), [B \to \beta.\delta], T)$ and
$(EFF_k(\pi), [A \to \alpha.\pi], \{e\}, [B \to \beta.], T)$
are both proper.

(ii) If $(\mathfrak{A}, [A \to \alpha X.\pi], \mathfrak{B}, [B \to \beta X.\delta], T)$ is proper and $S \in PRED(T,X)$ then $(\mathfrak{A}, [A \to \alpha.X\pi], \mathfrak{B}, [B \to \beta.X\delta], S)$ is proper.

(iii) If $\mathfrak{P} = (\mathfrak{A}, I, \mathfrak{B}, J, T)$ is proper and $(\mathfrak{A}1, I1, \mathfrak{B}1, J1) \in ORIGIN(\mathfrak{P})$ then $(\mathfrak{A}1, I1, \mathfrak{B}1, J1, T)$ is proper.

□

Lemma 4. 17

Let $\mathfrak{P}$ be proper. Then

$$\exists\ T' \in URCORE_k(T):$$
$$\exists\ R \in M_k:$$
$$\exists\ [A \to \alpha.\beta], [B \to \delta.] \in R:$$
$$\mathfrak{A} \oplus_k LR_k(I, T') \subseteq EFF_k(\beta) \oplus_k LR_k([A \to \alpha.\beta], R)$$
$$\wedge\ \mathfrak{B} \oplus_k LR_k(J, T') \subseteq LR_k([B \to \delta.], R)$$

□

We shall say that a $\mathfrak{P}$ is LA-DONE if, by means of LA-lookahead in T, possible LR-conflicts involving $\mathfrak{P}$ can be determined.

Definition 4. 18

Let $\mathcal{P}$ be proper. Then

LA-DONE($\mathcal{P}$) ≡

$\mathcal{G} \oplus_k LA_k(I, T) \cap \mathcal{B} \oplus_k LA_k(J, T) \neq \emptyset$

$\Downarrow$

G is not LR(k).

□

In the final algorithm we shall mark all proper parameter sets which are LA-DONE to avoid a lot of repeated backwards tracing.

LA-SPLIT will have the property that after a call LA-SPLIT($\mathcal{P}$) we are finished with $\mathcal{P}$ in the following sense:

[4. 19] T may have been splitted into a number of copies. For all such copies, T', either T' ∈ Blind or LA-DONE($\mathcal{G}$, I, $\mathcal{B}$, J, T') is true.

To keep track of the states being isolated during a call of LA-SPLIT we define the following concept:

Definition 4. 20

Let $\mathbb{m}$ be a set of states and let a statement, Stmt, be executed. COPY($\mathbb{m}$) and COPY^0($\mathbb{m}$) keep track of the states isolated from states in $\mathbb{m}$ during the execution of Stmt :

- Initially COPY^0($\mathbb{m}$) = $\mathbb{m}$.
- Let $T_1 \in$ COPY^0($\mathbb{m}$). If a statement (Q, T_2) := ISOLATE(Q, T_1, $\mathcal{S}$) is executed then T_2 is included in COPY^0($\mathbb{m}$).
- Let $T_2 \in$ COPY^0($\mathbb{m}$). If a statement Q := MERGE$_k$(Q, T_1, T_2) is executed then T_2 is removed from COPY^0($\mathbb{m}$).
- Finally COPY($\mathbb{m}$) = COPY^0($\mathbb{m}$)\ $\mathbb{m}$.

□

In the state-splitting algorithm we make use of $COPY^o$ in order to indicate in which sense we have finished the treatment of the conflict being considered. In practice it is not necessary to keep track of $COPY^o$ since the relevant information may be found in LA-Done.

Algorithm 4.21

```
TYPE ℙ° = (SET OF Σ*^k × Item × SET OF Σ*^k × Item × State);
VAR  Q : SM_k;
     LA-Done : SET OF ℙ°; {initially LA-Done = ∅}
PROCEDURE LA-SPLIT-k(I, J: Item; T: State);
VAR Blind, Stack: SET OF State;
BEGIN
    Blind := Stack := ∅ ;
    ASSUME (I, J) ≈ ([A → α.β], [B → δ.]);
    LA-SPLIT(EFF_k(β), I, {e}, J, T);
    FOR T' ∈ COPY°({T}) \ Blind DO
        IF EFF_k(β) ⊕_k LA_k(I, T') ∩ LA_k(J, T') ≠ ∅
        THEN {(EFF_k(β), I, {e}, J, T') ∈ LA-Done}
            "Report that there is an LR(k)-conflict
            between I and J in some state in URCORE_k(T')".
        ENDIF
    ENDFOR ;
    IF Blind ≠ ∅ THEN
        "Remove from Q all states in Blind"
    ENDIF
END LA-SPLIT-k;
```

```
PROCEDURE LA-SPLIT(𝒫:𝒫^o);
VAR N: Integer; X: N ∪ Σ; S: State;
    I1, J1: Item;        𝒢1, ℬ1: SET OF Σ*^k;
    𝔪, 𝔏   : SET OF State;
BEGIN ASSUME 𝒫 = (𝒢, I, ℬ, J, T);
    Stack := Stack ∪ {T};
    IF ¬ BOTTOM_k(𝒫) THEN
            N := 0;

            REPEAT
                FOR S ∈ PRED(T,X)\ (Blind ∪ Stack),
                    (𝒢1,I1,ℬ1,J1) ∈ ORIGIN_k(𝒫)
                WHERE (𝒢1,←I1,ℬ1,←J1,S) ∉ LA-Done DO
                    LA-SPLIT(𝒢1,←I1,ℬ1,←J1,S)
                ENDFOR;
                𝔪 := PRED(T,X)\(Blind ∪ Stack);
                 𝔏 := PRED(T,X)∩(Blind ∪ Stack);
                IF 𝔪 = ∅ THEN Blind := Blind ∪ {T}
                ELSEIF 𝔏 = ∅ THEN Q := Split(Q,𝒫 )
                ELSE {𝔪 ≠ ∅ ∧ 𝔏 ≠ ∅}
                  (Q, T') := ISOLATE(Q, T, 𝔪) ;
                  Q := Split(Q, (𝒢, I,ℬ, J, T'));
                  Blind := ∅; N := N+1;
                  Close-loop(N)
                ENDIF
            UNTIL 𝔪 = ∅ ∨ 𝔏 = ∅;
    ENDIF ;
    LA-Done := LA-Done ∪ {𝒫};
    Stack := Stack\ {T};
END LA-SPLIT;
```

□

The operation Split(Q, (𝒢, I,ℬ, J, R)) has to update LA-Done properly, i.e.

> If R' is isolated from R, then (𝒢, I,ℬ, J, R') must be added to LA-Done and for all (𝒢', I',ℬ', J', R) ∈ LA-Done,(𝒢', I',ℬ', J', R') must be added to LA-Done.

The following theorem states the correctness of algorithm 4.21.

Theorem 4.22

Let

$$\text{PRE-SPLIT}(\mathfrak{P}) \equiv \{L = \{\mathfrak{P}1 \mid \text{LA-DONE}(\mathfrak{P}1)\} \wedge \text{proper}(\mathfrak{P})\}$$

$$\text{POST-SPLIT}(\mathfrak{P}) \equiv \{\text{proper}(\mathfrak{P}) \wedge$$
$$(\forall T' \in \text{COPY}^{o}(T) : T' \in \text{Blind} \vee \text{LA-DONE}(\mathfrak{A}, I, \mathfrak{B}, J, T')) \wedge$$
$$(\forall (\mathfrak{A}1, I1, \mathfrak{B}1, J1, T1) \in L : \forall T2 \in \text{COPY}^{o}(T1):$$
$$\text{LA-DONE}(\mathfrak{A}1, I1, \mathfrak{B}1, J1, T2))\}.$$

Let $\mathfrak{P} = (\text{EFF}_k(\beta), [A \to \alpha.\beta], \{e\}, [B \to \delta.], T)$ then

$\{\text{PRE-SPLIT}(\mathfrak{P})\}$
LA-SPLIT-k $([A \to \alpha.\beta], [B \to \delta.], T)$
$\{\text{POST-SPLIT}(\mathfrak{P})\}$
and LA-SPLIT-k terminates.

Proof: Follows from theorem 5.16.

□

5. A Proof of the State Splitting Algorithm

In this section a proof of the state splitting algorithm is given. We shall first define an auxiliary predicate which is needed in the proofs.

Definition 5.1

Let $\mathcal{P}$ be proper and let $\mathbb{m} \subseteq \text{PRED}(T, X)$. Then

$$\begin{aligned}&\text{PRED-DONE}(\mathbb{m}, \mathcal{P}) \equiv \\ &\quad \forall S \in \mathbb{m}\ \forall\ (\mathcal{A}1, I1, \mathcal{B}1, J1) \in \text{ORIGIN}_k(\mathcal{P}): \\ &\qquad \text{LA-DONE}(\mathcal{A}1, \overleftarrow{I1}, \mathcal{B}1, \overleftarrow{J1}, S)\end{aligned}$$

□

PRED-DONE($\mathbb{m}, \mathcal{P}$) is true iff it is possible to determine LR-conflicts involving $\mathcal{P}$ by means of the LA_k-sets for the items $\overleftarrow{I1}$ and $\overleftarrow{J1}$ in the states in $\mathbb{m}$.

We may now formulate the following lemmas:

Lemma

Let $\mathcal{P}$ be proper. Then

[5.2] If I or J is $[S \rightarrow .S' \dashv^k]$, then $\text{BOTTOM}_k(\mathcal{P})$ is true.

[5.3] $\text{BOTTOM}_k(\mathcal{P}) \Rightarrow \text{LA-DONE}(\mathcal{P})$.

[5.4] $\forall\, \mathbb{m} \subseteq \text{PRED}(T, X)$:
$\{\text{DISJOINT}(\mathbb{m}, \mathcal{P}) \wedge \text{PRED-DONE}(\mathbb{m}, \mathcal{P})\}$
$(Q, T') := \text{ISOLATE}(Q, T, \mathbb{m})$;
$\{\text{LA-DONE}(\mathcal{A}, I, \mathcal{B}, J, T')\}$

[5.4a] Let Stmt be $(Q, S2) := \text{ISOLATE}(Q, S1, \mathbb{m})$ or $Q := \text{MERGE}_k(Q, S1, S2)$. Then

$\{LP = \{\mathcal{P}1 \mid LA\text{-}DONE(\mathcal{P}1)\}\}$

Stmt

$\{\forall\ (\mathcal{A}1, I1, \mathcal{B}1, J1, T1) \in LP\ \forall T2 \in COPY^{o}(T1)$: $LA\text{-}DONE(\mathcal{A}1, I1, \mathcal{B}1, J1, T2)\}$

Proof

[5.2] Follows from the definition of LA_k (3.5 (i)).

[5.3] Follows from lemma 4.17 and the definition of $BOTTOM_k$([4.8]).

[5.4] If $\mathcal{A} \oplus_k LA_k(I, T') \cap \mathcal{B} \oplus_k LA_k(J, T') \neq \emptyset$ then there exist $S \in \mathbb{m}$ and $(\mathcal{A}1, I1, \mathcal{B}1, J1) \in ORIGIN_k(\mathcal{P})$ such that $\mathcal{A}1 \oplus_k LA_k(I1,S) \cap \mathcal{B}1 \oplus LA_k(J1,S) \neq \emptyset$. Since $LA\text{-}DONE(\mathcal{A}1, I1, \mathcal{B}1, J1, S)$ is true it follows that $LA\text{-}DONE(\mathcal{A}, I, \mathcal{B}, J, T')$ is true.

[5.4a] Assume that $LA\text{-}DONE(\mathcal{P}1)$ is true. LA_k-sets in $\mathcal{Q}$ cannot increase as the result of ISOLATE or $MERGE_k$, i.e. no new LA_k-conflict can be introduced. An LA_k-conflict involving $\mathcal{P}1$ cannot be removed since this will be a contradiction ($LA\text{-}DONE(\mathcal{P}1)$ implies that LA_k-conflicts involving $\mathcal{P}1$ are LR_k-conflicts). If $T2 \in COPY^{o}(T1)$ then clearly $LA\text{-}DONE(\mathcal{A}1, I1, \mathcal{B}1, J1, T2)$ is true.

□

If $\mathbb{m}$ is a set of states, then $ENTRY(\mathbb{m})$ is the set of states not in $\mathbb{m}$, that have a successor in $\mathbb{m}$:

Definition 5.5

$$ENTRY(\mathbb{m}) = \{S \mid S \notin \mathbb{m} \wedge \exists\ X\text{: } GOTO(S, X) \in \mathbb{m}\}$$

□

The set Blind is characterized by the following predicate:

Definition 5.6

$$BOUND \equiv ENTRY(Blind) \subseteq Stack$$

□

If Stack = $\emptyset$ then BOUND implies that no state in Blind can be reached from a state not in Blind. Note that the initial state (IS) will never be in Blind.

We shall prove that the set LA-Done keeps track of the $\mathcal{P}$'s that satisfy the predicate LA-DONE.

Lemma 5.7

Let PRE-SPLIT and POST-SPLIT be as defined in theorem 4.22 Then

$$\{\text{BOUND} \wedge \text{Stack} = \text{Stack}' \wedge T \notin \text{Blind} \wedge \mathcal{P} \notin \text{LA-Done} \wedge \text{PRE-SPLIT}(\mathcal{P})\}$$
$$\text{LA-SPLIT}(\mathcal{P})$$
$$\{\text{POST-SPLIT}(\mathcal{P}) \wedge \text{BOUND} \wedge \text{Stack} = \text{Stack}'\}$$

Proof

(a) We assume that all the inner calls of LA-SPLIT satisfy the lemma. Under this assumption we shall prove that an execution of the body of LA-SPLIT satisfies the lemma.

(b) If BOUND $\wedge$ Proper($\mathcal{P}$) $\wedge$ Stack = Stack' is true before the body of LA-SPLIT is executed then this is also the case after the execution of the body.

(c) [5.4a] also holds if Stmt is replaced by the body of LA-SPLIT.

(d) Let $\mathcal{N}$ be the set of states isolated from T during the execution of the body of LA-SPLIT ($\mathcal{N}$ does not include T). Initially $\mathcal{N} = \emptyset$, and $\mathcal{N}$ gets extended for each cycle of the repeat-loop. The following predicate is an invariant of the repeat loop.

$$\forall R \in \mathcal{N} : \text{LA-DONE}(\mathcal{G}, I, \mathcal{B}, J, R).$$

(e) After the for-loop and the separation of PRED(T, X) into $\mathfrak{m}$ and $\mathfrak{L}$, PRED-DONE($\mathfrak{m}$, $\mathfrak{P}$) is true and (by definition) $\mathfrak{L} \subseteq$ Blind $\cup$ Stack.

(f) The execution of the if-statement in the repeat-loop may imply that T $\in$ Blind ($\mathfrak{m} = \emptyset$) or LA-DONE($\mathfrak{P}$) ($\mathfrak{L} = \emptyset$). In both cases the predicate in (d) still holds.

(g) After the repeat-loop and the outermost if-statement we clearly have that

$$\forall\, R \in \mathfrak{n} \cup \{T\}: R \in \text{Blind} \vee \text{LA-DONE}(\mathfrak{G}, I, \mathfrak{B}, J, R).$$

□

We shall now prove that LA-SPLIT will terminate. The most difficult part of the termination proof is to assure that the repeat-loop terminates. For that purpose we need an invariant for the repeat-loop (lemma 5.14). In order to prove 5.14 we need a number of auxiliary lemmas which are given below.

The next lemma states that no state can get new predecessors other than those obtained by splitting the predecessors it had before the call of LA-SPLIT.

Lemma 5.8

$\{\mathfrak{m} = \text{PRED}(T', X)\}$
LA-SPLIT($\mathfrak{P}$)
$\{\text{PRED}(T', X) \subseteq \text{COPY}^0(\mathfrak{m})\}$

Proof:

Assume that the lemma is true for all inner calls of LA-SPLIT. In the body of LA-SPLIT, MERGE_k is only applied to copies of T' created during the execution of the body. This means that no state existing before the call can get a new predecessor.

□

The situation where the repeat-loop needs another activation is characterised by the next two lemmas in terms of the changes in ENTRY(Blind).

Lemma

[5.9] $\{\mathbb{m} = PRED(T,X) \setminus (Blind \cup Stack) \wedge$
$\mathbb{L} = PRED(T,X) \cap (Blind \cup Stack) \wedge BOUND$
$\wedge Q = Q' = (M', IS, GOTO') \wedge T \in Stack\}$

$(Q, T') := ISOLATE(Q', T, \mathbb{m});$
$Q := Split(Q, (\mathbb{G}, I, \mathbb{B}, J, T'))$;
ASSUME $Q = (M, IS, GOTO)$

$\{ENTRY(Blind) \subseteq (M \setminus M') \cup Stack\}$

Note that $M \setminus M'$ are the copies of T created by ISOLATE and Split.

[5.10] $\{PRED(T,X) = \mathbb{L} \subseteq B \cup Stack \wedge ENTRY(B) \subseteq \mathbb{m} \cup Stack\}$

LA-SPLIT($\mathbb{P}$)
$\{ENTRY(COPY(B)) \subseteq COPY^{O}(\mathbb{m})\}$

□

For the purpose of the invariant of the repeat-loop we introduce a number of auxiliary (shadow) variables as described below. In [5.11] a skeleton of the repeat-loop is given. Additional labels and statements defining the values of the auxiliary variables are inserted. During the proofs we shall refer to [5.11].

The auxiliary variables M_i, B_i ($i \in [1,N]$) are sets of states. Consider the situation after N executions of the repeat-loop:

The values of M_i, B_i ($i \in [1,N]$) may be described as follows:

M_i ($i \in [1,N]$) contains the copies of T being created during the i'th execution of the repeat-loop, including the possible further splitting during the succeeding executions of the repeat-loop.

If B_i' ($i \in [1,N]$) is the value of Blind at label L3 in [5.11] during the i'th execution of the repeat-loop, then B_i ($i \in [1,N]$) is the set of states which have been isolated from B_i' during the (i+1)'th execution of the repeat-loop, including the possible further splitting during the succeeding executions of the repeat-loop.

```
[5.11]   N := 0;
         REPEAT L1:
              FOR ... DO
                 . . .
                    ┌──────────────────────────────────────────┐
                    │ FOR i ∈ [1,N] DO                         │
                    │       M_i := COPY°(M_i); B_i := COPY°(B_i)│
                    │ ENDFOR                                   │
                    └──────────────────────────────────────────┘
              L2:
              ENDFOR;

              ┌──────────────────┐
              │ B_N := B_N\Blind; │
              └──────────────────┘
              L3:
              𝔪 := ... ; 𝔏 := ... ;
              IF ... ELSEIF
              ELSE
                   L4:
                      (Q, T') := ISOLATE(Q, T, 𝔪);
                      Q := Split(Q, (ℂ, I, 𝔅, J, T'));
                         ┌──────────────────────────────────────────┐
                         │ B_N+1 := Blind;                          │
                         │ M_N+1 := {T'} ∪ {the states isolated from T'│
                         │                    by Q := Split(Q,(..., T'))};│
                         └──────────────────────────────────────────┘
                   L5:
                      Blind := ∅; N := N+1; Closeloop(N);
                   L6:
              ENDIF
         UNTIL 𝔪 = ∅ ∨ 𝔏 = ∅;
```

Introducing the following predicates

$$[5.12] \quad P(m) \equiv \{\forall\ i \in [1, m-1]: (ENTRY(B_i) \subseteq M_i) \wedge (ENTRY(M_{i+1}) \subseteq \cup\{B_j \mid j = i, \ldots, m\} \cup M_i)\}$$

$$[5.13] \quad Q(m) \equiv P(m) \wedge (ENTRY(B_m) \subseteq M_m \cup Stack),$$

the following lemma is true.

Lemma 5.14

Q(N) is an invariant of the repeat-loop in the sense that Q(N) is always true at L1.

Proof

Q(N) is clearly true at L1 when the repeat-loop is entered since N = 0. Assume that Q(N) is true at L1. Then using 5.8-5.10 we may prove that

(a) $P(N) \wedge (ENTRY(B_N) \subseteq M_N)$ is true at L3;

(b) Q(N+1) is true at L5,

(c) Q(N) is true at L6.

□

We may now prove that LA-SPLIT terminates.

Lemma 5.15

LA-SPLIT(℘) terminates.

Proof

(a) The recursion stops as we either will obtain $BOTTOM_k$ or meet a state in Stack ∪ Blind.

(b) The for-loop must terminate, even if PRED(T, X) grows for each iteration, because S, and any new copy of S, are handled for the given parameter set, i.e. after LA-SPLIT(𝔊1, $\overleftarrow{\mathfrak{I}}$1, ℬ1, $\overleftarrow{\mathfrak{J}}$1, S)

we have $\forall S' \in COPY^{o}(\{S\}) : (S' \in Blind) \vee LA\text{-}DONE(\mathfrak{A}1, \overleftarrow{\mathfrak{I}1}, \mathfrak{B}1, \overleftarrow{\mathfrak{J}1}, S')$.

(c) The repeat-loop must terminate. Assume that it does not. Then for any $N \geq 0$ there will exist a sequence of sets of states $M_1, B_1, M_2, B_2, \ldots, M_N, B_N$ as defined by [5.11] and satisfying P(N) ([5.12]). Furthermore all $M_i \neq \emptyset$ $(i \in [1,N])$ otherwise an application of LA-SPLIT($\mathfrak{P}$) will imply that $PRED(T,X) \subseteq Blind \cup Stack$.

This is a contradiction, since then there will for any N be a sequence of states $T_i \in M_i, S_{i_1}, \ldots, S_{i_m} \in B_i$ $(i \in [1,N])$ such that there is a path from T_i via $S_{i_1}, \ldots, S_{i_m}$ to T_{i+1} $(i \in [1,N-1])$. If N is large enough there will exist $i,j \in [1,N]$ $i \neq j$ such that T_i and T_j may be merged.

□

We may now formulate the main result.

Theorem 5.16

Let PRE-SPLIT and POST-SPLIT be as defined in theorem 4.22, and let $\mathfrak{P} = (EFF_k(\beta), [A \rightarrow \alpha.\beta], \{e\}, [B \rightarrow \delta.], T)$. Then

$\{Blind = Stack = \emptyset \wedge PRE\text{-}SPLIT(\mathfrak{P})\}$

LA-SPLIT($\mathfrak{P}$)

$\{Stack = \emptyset \wedge BOUND \wedge POST\text{-}SPLIT(\mathfrak{P})\}$
and LA-SPLIT($\mathfrak{P}$) terminates.

□

Theorem 4.22 follows immediately from theorem 5.16.

6. Concluding Remarks

We conclude the paper by giving suggestions for using LA-SPLIT in practice. This includes integration of LA_k-lookahead into LA-SPLIT, various optimizations, and the case k = 1. Finally a comparison with related work is made.

6.1 A strategy for using the state-splitting algorithm

Algorithm 4.22 is expressed (in Split by DISJOINT and in Close-loop by $MERGE_k$) by means of LA_k-sets computed on the actual SM_k. It is possible to integrate the computation of LA_k-lookahead sets into LA-SPLIT in the same way as is done in [Kristensen & Madsen 79b] for the function testing LR(k)-ness. In addition it would be desirable to save LA_k-lookahead sets once computed, for later reuse. Applications of the $MERGE_k$ and ISOLATE operations may however change the LA_k-lookahead sets, such that a recomputation seems necessary. The following strategy integrates the computation of LA_k-lookahead sets into LA-SPLIT and prevents recomputation of LA_k-sets once computed.

Let $LA'_k(I, T)$ denote the lookahead currently saved for (I, T). The idea is that if the LA'_k-sets for the interesting items in the states in PRED(T, X) are available then these LA'_k-sets may be used to determine the LA'_k-sets for the items being considered in T :

[6.1a] Initially $LALR_k$ lookahead is added to the LR(0)-machine, i.e. $LA'_k(I, T) = LALR_k(I, T)$ for all I, T.

[6.1b] If $BOTTOM_k(\mathcal{P})$ is true in the body of LA-SPLIT then $LA_k(I, T)$ and $LA_k(J, T)$ are computed, i.e. the LA'_k-sets are updated to the actual LA_k-sets.

[6.1c] Having executed an operation of the form

$$(Q, T') := ISOLATE(Q, T, \mathcal{M})$$

LA'_k is updated for I, J in T and T' by propagating LA'_k-sets from $\mathcal{M} = PRED(T', X)$ to T' and from PRED(T, X) to T.

[6.1d] The comparisons made by Close-loop is based on the $LA^!_k$-sets. An execution of $MERGE_k$ makes no changes to the $LA^!_k$-sets. □

We shall later in this section improve on [6.1].

We shall now argue that the strategy works in the sense that

[6.2] If $\mathcal{P} \in$ LA-Done then LR_k-conflicts involving $\mathcal{P}$ may be determined using the $LA^!_k$-sets for I, J in T, and LA-SPLIT terminates.

This is supported by the following observations :

The LA_k-sets cannot increase as the result of an ISOLATE or $MERGE_k$-operation :

[6.2a] $\forall\ T \in M_k\ \forall\ I \in T :\ LA_k(I, T) \subseteq LA^!_k(I, T)$.

LA_k-sets for items visited during the recursive traversal by LA-SPLIT will always be recomputed since LA-SPLIT continues until $BOTTOM_k$ is true :

[6.2b] After an execution of LA-Split($\mathcal{P}$) we have that

$$LA^!_k(I, T) = LA_k(I, T) \wedge LA^!_k(J, T) = LA_k(J, T)$$

[6.2c] The partition made by Split ([4.7]) using DISJOINT on the $LA^!_k$-sets cannot introduce conflicts because of an implicit merge ([6.2a]).

[6.2d] The strategy in [6.1] will make no changes to the $LA^!_k$-sets for the predecessor states of T which are in Blind ∪ Stack. Let $\mathcal{L}$ = PRED(T, X) ∩ (Blind ∪ Stack). The $LA^!_k$-sets for states in $\mathcal{L}$ have not been updated. This will, however, cause no problems since in a succeeding ISOLATE-operation $\mathcal{L}$ will be predecessors of T and we have not finished the treatment of T.

[6.2e] Consider an attempt to execute $MERGE_k(Q, T_1, T_2)$ where $CORE(T_1) = CORE(T_2) = T$.

It is only necessary to require identity of the LA_k-sets for items that appear in LA-Done ; i.e. items $I, J \in T$ where $(\mathfrak{A}, I, \mathfrak{B}, J, T_i) \in$ LA-Done for some $\mathfrak{A}$ and $\mathfrak{B}$ and for $i \in [1,2]$. Items that do not appear in LA-Done have not been part of a state-splitting process and whether or not the LA_k-sets are changed does not matter (their LA_k-sets will in the worst case be $LALR_k$-sets).

We have three situations :

(1) $LA'_k(I, T_i) = LA_k(I, T_i)$, $i = 1, 2$,

(2) $LA'_k(I, T_1) = LA'_k(I, T_2)$ and
$LA_k(I, T_1) \neq LA_k(I, T_2)$

(3) $LA'_k(I, T_1) \neq LA'_k(I, T_2)$ and
$LA_k(I, T_1) = LA_k(I, T_2)$

In case (1) there are no problems. In case (2) we should not have merged and in case (3) we could have merged.

Case (2) might violate LA-DONE, but all $\mathfrak{P}$'s in LA-Done are LA-DONE according to $LA'_k(I, T_1)$ and $LA_k(I, T_1)$; i.e. an increase in $LA_k(I, T_1)$ to $LA'(I, T_1)$ makes no changes.

Case (3) might violate the termination proof ; but this is based on the fact that Σ^{*k} is finite and not that the LA_k-sets are correct.

The termination part of [6.2] follows from [6.2e]. The remaining part of [6.2] follows from the above remarks and may be formulated as follows :

[6.2f] $\forall \mathfrak{P} \in$ LA-Done ;

$$\mathfrak{A} \oplus_k LA'_k(I, T) \cap \mathfrak{B} \oplus_k LA'_k(J, T) = \emptyset$$

$$\Updownarrow$$

$$\mathfrak{A} \oplus_k LA_k(I, T) \cap \mathfrak{B} \oplus_k LA_k(J, T) = \emptyset$$

This expresses that using LA'-sets instead of LA_k-sets does not introduce new conflicts and does not remove LR_k-conflicts.

We shall improve on the strategy in [6.1]. If $\mathfrak{P} \in$ LA-Done then only

(*) the sets $LA_i(I, T)$ and $LA_j(J, T)$ where $i = k - |\mathfrak{A}|_{min}$ and $j = k - |\mathfrak{B}|_{min}$ are necessary in order to check LR_k-conflicts involving $\mathfrak{P}$.

This may be used to improve on [6.1]. The points [6.3a-3d] give the changes to the corresponding points in [6.1a-1d]:

[6.3a] Initially $LALR_0$-sets are added, i.e. $LA'_0(I, T) = \{e\}$ for all I, T.

[6.3b] In case of $BOTTOM_k(\mathfrak{P})$, $LA_i(I, T)$ and $LA_j(J, T)$ are computed where i and j are determined as above. Note that either i = 0 or j = 0. If $LA'_r(I, T)$, where $r > i$, is already saved then $LA_r(I, T)$ must be computed. Similarly for the LA'-set for (J, T).

[6.3c] This is similar to [6.3b]. LA'_i-sets for I in T and T' and LA'_j-sets for J in T and T' are updated by propagating from the LA'-sets in the predecessors of T and T'. If $LA'_r(I, T)$ where $r > i$ is already saved then $LA'_r(I, T)$ must be updated by recomputing $LA_r(I, T)$.

[6.3d] In Close-loop the comparision is based on the LA'_i-sets saved for the largest i. If $LA'_i(I, T_1)$ and $LA'_r(I, T_2)$, where $r > i$, are saved and the LA'-sets for (I, T_1) and (I, T_2) need to be compared then $LA_r(I, T_1)$ must be recomputed.

(*) Let $\mathfrak{A} \subseteq \Sigma^{*k}$ then $|\mathfrak{A}|_{min}$ is the length of the shortest string in $\mathfrak{A}$, if $\mathfrak{A} \neq \emptyset$, and 0 if $\mathfrak{A} = \emptyset$.

6.2 Further Improvements

It is possible to add an extra condition for terminating the repeat-loop. The reason for not terminating the repeat-loop in the case where $\mathbb{M} \neq \emptyset$ and $\mathbb{L} \neq \emptyset$ is that PRED-DONE($\mathbb{L}$, T) is in general not true. If, however, we have that

$$\forall S \in \mathbb{L} \quad \forall (\mathbb{Q}1,\ I1,\ \mathbb{B}1,\ J1) \in \text{ORIGIN}_k(\mathbb{P}) :$$
$$\mathbb{Q}1 \oplus_k \text{LA}_k(\overleftarrow{I1}, S) \cap \mathbb{B}1 \oplus_k \text{LA}_k(\overleftarrow{J1}, S) = \emptyset$$

then PRED-DONE($\mathbb{L}$, T) is true and we may split T and terminate the repeat-loop.

Concerning Close-loop it may seem complex that the sets M_i are tested against the sets $M_{i-1}, \ldots, M_1$ $(i = n, \ldots, 2)$. For practical grammars in the case k = 1, this is no serious problem since n will be small. However, one may limit the attempts to try to merge. Assume that the states in the sets $M_i, \ldots, M_1$ $(i \in [0, n-1])$ have not had their LA_k-lookahead sets changed since the previous Close-loop operation. (If using the strategy in [6.1] no state in these sets have been split) then one needs only consider states in M_j $(j \in [i+1, n])$ against those in $M_{j-1}, \ldots, M_1$. By doing this then only newly created states are considered for a merge. This will then correspond to the LR(k) construction algorithm where it is tested if a newly generated state already exists.

Concerning Split (4.8) : The third requirement in point (b) is in fact not necessary and without this requirement a simpel partition can be made. It is, however, straight-forward to make a partition satisfying the requirement. It is more complex to make a partition where p is minimal. In practice it does not seem necessary to do this. Even if one does minimize p it will not guarantee that this gives the minimal number of states.

The primitives ISOLATE, MERGE_k, ORIGIN_k and BOTTOM_k are straight forward to implement. Concerning ORIGIN_k, it may pay to compute $\text{ORIGIN}_k(\{e\}, I, \{e\}, J, T)$ when $\text{ORIGIN}_k(\mathbb{Q}, I, \mathbb{B}, J, T)$ is desired and then save it.

6.3 The k = 1 Case

The LALR(k) and $LRCOND_k$ algorithms in [Kristensen & Madsen 79a, 79b] have all been improved for the important practical case with k = 1. In this case it is possible to eliminate the parameters of type SET OF Σ^{*k}. This can also be done for LA-SPLIT. The following lemma is useful in this connection.

Lemma 6.4

Let $I, J \in T$, then

$$\text{LA-DONE}_1(\{e\}, I, \{e\}, J, T)$$

$$\Updownarrow$$

$$\forall S \in \text{PRED}(T, X):$$

$$\forall (\mathbb{A}, I1, \mathbb{B}, J1) \in \text{ORIGIN}, \{e\}, I, \{e\}, J, T):$$

$$\text{BOTTOM}_1(\mathbb{A}, \overleftarrow{I1}, \mathbb{B}, \overleftarrow{J1}, S) \vee \text{LA-DONE}_1(\{e\}, \overleftarrow{I1}, \{e\}, \overleftarrow{J1}, S)$$

□

BOTTOM_1 appears when either $e \notin \mathbb{A}$ or $e \notin \mathbb{B}$. The set parameters can be eliminated by testing for BOTTOM_1 before the internal recursive calls of LA-SPLIT in algorithm 4.20 instead of the BOTTOM-test in the beginning of the body of procedure LA-SPLIT. It is then also necessary to test for BOTTOM_1 before the initial call of LA-SPLIT in the body of LA-SPLIT-k. If BOTTOM_1 is true before the initial call, then we have a shift/reduce conflict. It is well known that an LALR(1) shift/reduce conflict is also an LR(1) shift/reduce conflict, i.e. state-splitting cannot remove the conflict.

6.4 Comparison

The idea of using state splitting for constructing LR(k) parsers for non-LALR(k) grammars was suggested by De Remer [De Remer 69]. The LR(0) machine is viewed as a so-called characteristic finite state machine, CFSM, which recognices the regular set of characteristic strings (cf. [Aho & Ullman 72]). The CFSM is converted into a nondeterministic FSM by duplicating all pathes leading to state with LALR(k) conflicts. If such

a path contains loops then each loop is unrolled k times in order to stabilize the lookahead propagation along the path. The nondeterministic FSM is then made deterministic using standard techniques.

It has been shown in [Pager 72] that it is not enough to unroll loops k times in order to stabilize lookahead propagation. According to Pager this must be done $k \cdot n + c$ times where n is the number of states in the smallest KERNEL in the loop and $c = 0$ if the first state in the loop has a KERNEL of size n, and $c = 1$ otherwise.

The above approach is more theoretical than practical. The amount of loop unrolling and loop separation is in general too high. The part of the LR(0) machine being split this way may have a size that is larger than the corresponding parts in the LR(k)-machine. The reason for this is that the splitting process does not capture the situations where lookahead propagation stabilizes much further than in the worst case.
No practical use of the method has been reported.

Another approach to state splitting is reported in [Pager 77a]. Here the approach is to regenerate the part of the LR(0) machine in conflict by constructing sets of LR(k) items. By doing this one avoids the worst case loop unrolling and loop splitting as in De Remer's approach. The sets of LR(k)-items are regenerated by a method described in [Pager 77b] which uses two merge criteria to merge LR(k)-states during the generation process in order to reduce the space needed by the standard algorithm for constructing LR(k)-parsers.

A weak and a strong merge criterion are used to check if a newly generated state T can be merged into an existing state T' without introducing a conflict in T' or any state reachable from T'. If T and T' satisfy one of the merge criteria then T is merged into T'. The lookaheads of the items in T are then added to the corresponding items in T'. This newly added lookahead in T' must then be propagated to those successors of T' which have been generated. Thus merging involves one or two tests and a lookahead propagation.

A disadvantage of Pager's method seems to be that not only the states that need to be split are regenerated but also all of their successors. These superflous states are then eliminated by the merging process. Alltogether this method seems to perform unnecessary work. However, it is difficult to evaluate if there is any practical differences between Pager's method and ours. The size of the resulting parsers seems to be the same.

The arbitrariness in the size of p when making the partition in [4.8] appears in Pager's method in the arbitrariness in which of several possible states that is selected for a merge.

The method in [Pager 77b] can be used for directly constructing an LR(k)-parser. This also appears to be complex because of the merging process. For most practical grammars these will be LALR(k) and if not then only a small amount of state splitting is necessary. It thus seems to pay to generate the LR(0)-machine first and then make the state splitting. Again it is hard to evaluate if this has any practical differences.

It is difficult to compare the two methods because a worst case analysis is a poor measure for realistic examples. We know of no way of making an analysis that shows the differences for realistic examples other than by impiric results.

We shall conclude by considering the situation where a grammar is LALR(k) and the situation where no merge is possible in the LR(k)-machine.

Let PS be the state splitting approach of Pager and let PD be the direct approach of Pager. Let KMS be the algorithm LA-SPLIT of this paper.

Case 1. Let G be an LALR(k)-grammar. There is no principal difference between PS and KMS. There is a difference between PD and the state splitting approaches (PS and KMS). The time spent in PD for testing that two states can be merged (they always can since G is LALR(k) must be compared with the time used for computing LALR(k)-lookahead).

Case 2. Let G be an extreme LR(k)-grammar where no states in M_k with identical CORE can be merged. Here PD is more expensive than the standard LR(k) constructor algorithm because of the attempts to try to merge states that never can be merged. PS is more expensive than PD since it includes a complete regeneration of M_k using PD. KMS will continue to split M_0 until M_k is reached. This is probably also more expensive than the standard LR(k) constructor algorithm. How it compares with PD is difficult to see.

Case 2 is extreme in the sense that for practical grammars we are close to case 1. In that situation the state splitting approaches seem to be the simplest ones. According to Pager,PS is more efficient than PD when there are few simple LALR(k)-conflicts in the grammar, whereas PD is faster when there are many 'complex' LALR(k)-conflicts. This is because the more the PD-algorithm is used as part of PS to resolve conflicts the less favourable is PS compared to using PD directly.

As KMS is simpler than using PD to handle non LALR(k)-grammars we believe that KMS may compare favourable with both PD and PS in practical situations.

7. References

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

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).

Kristensen, B.B. and Madsen, O.L. [1979a]
"Methods for Computing LALR(k)-lookahead"
Computer Science Department, Aarhus University, 1979.
DAIMI PB-101.

Kristensen, B.B. and Madsen, O.L. [1979b]
"Methods for LR(k) Testing"
(Informative Diagnostics on LALR(k)-Conflicts)"
Computer Science Department, Aarhus University, 1979.
DAIMI PB-106.

Pager, D. [1972]
"On the Incremental Approach to Left-to-right Parsing"
Tech. Report No. PE 238, Information and Computer Sciences Dept.,
University of Hawaii, Honolulu, Jan. 1972.

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

Examples illustrating the state-splitting process

The state splitting algorithm is illustrated by two examples. This is done by showing the steps of the state splitting algorithm when transforming the LR(0)-machine of a grammar into its LR(1)-machine. The first example includes splitting a loop into more loops. The grammar of the second example is a variant of the first one. Here the splitting process furthermore includes unrolling loops.

We use the version of LA-SPLIT-k with k = 1 as described in section 6.3, in order to avoid the two set parameters.

Example A.1

Let $G_1 = (\{S', S, A, B\}, \{a, b, c, d, x, y, z, \dashv\}, P, S')$ be a CFG, where P consists of :

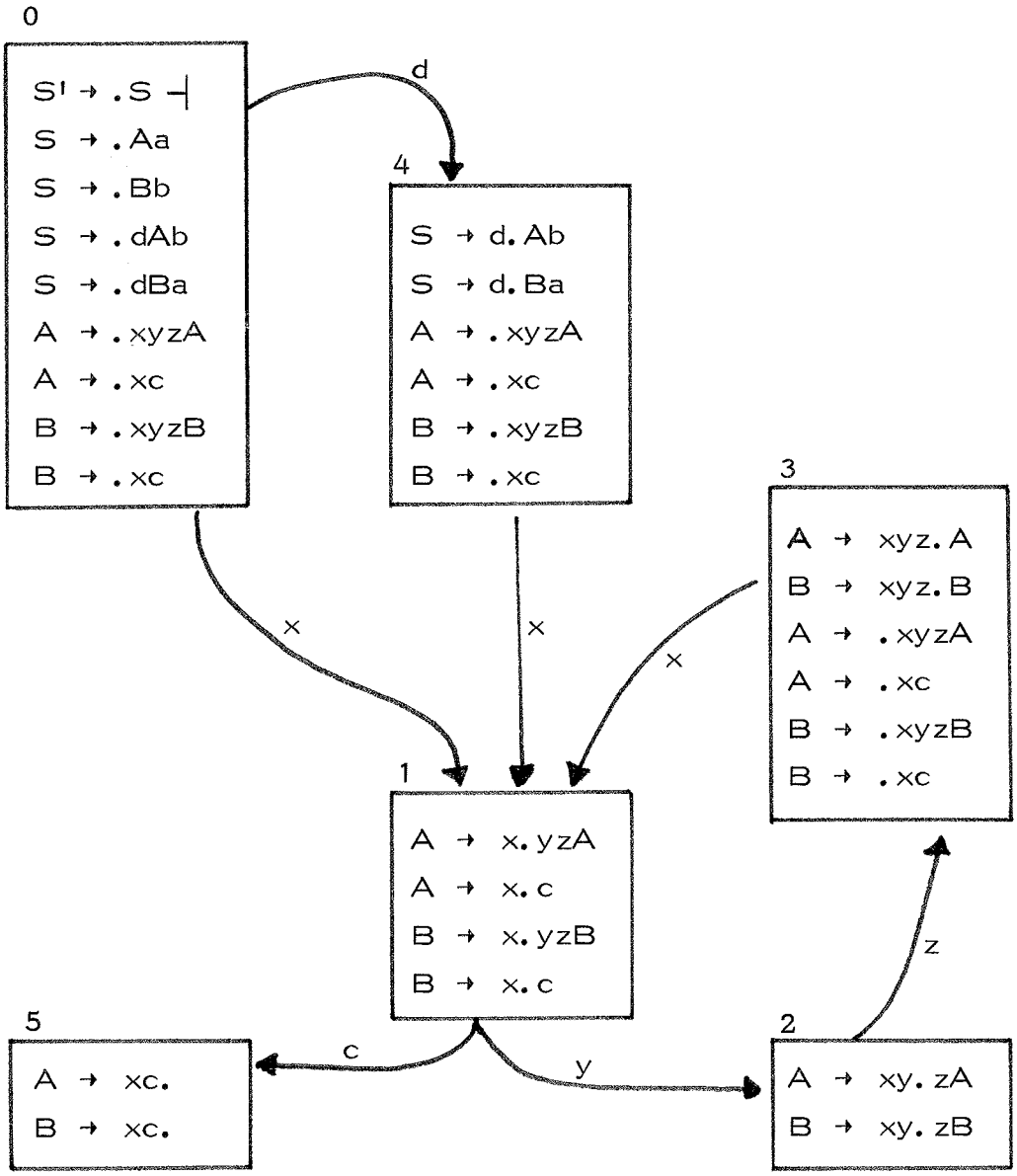
$S' \rightarrow S \dashv$
$S \rightarrow Aa \mid Bb \mid dAb \mid dBa$
$A \rightarrow xyzA \mid xc$
$B \rightarrow xyzB \mid xc$

The interesting parts of respectively the LR(0) - and the LR(1) - machines for G_1 are shown in [A.1] and [A.2]

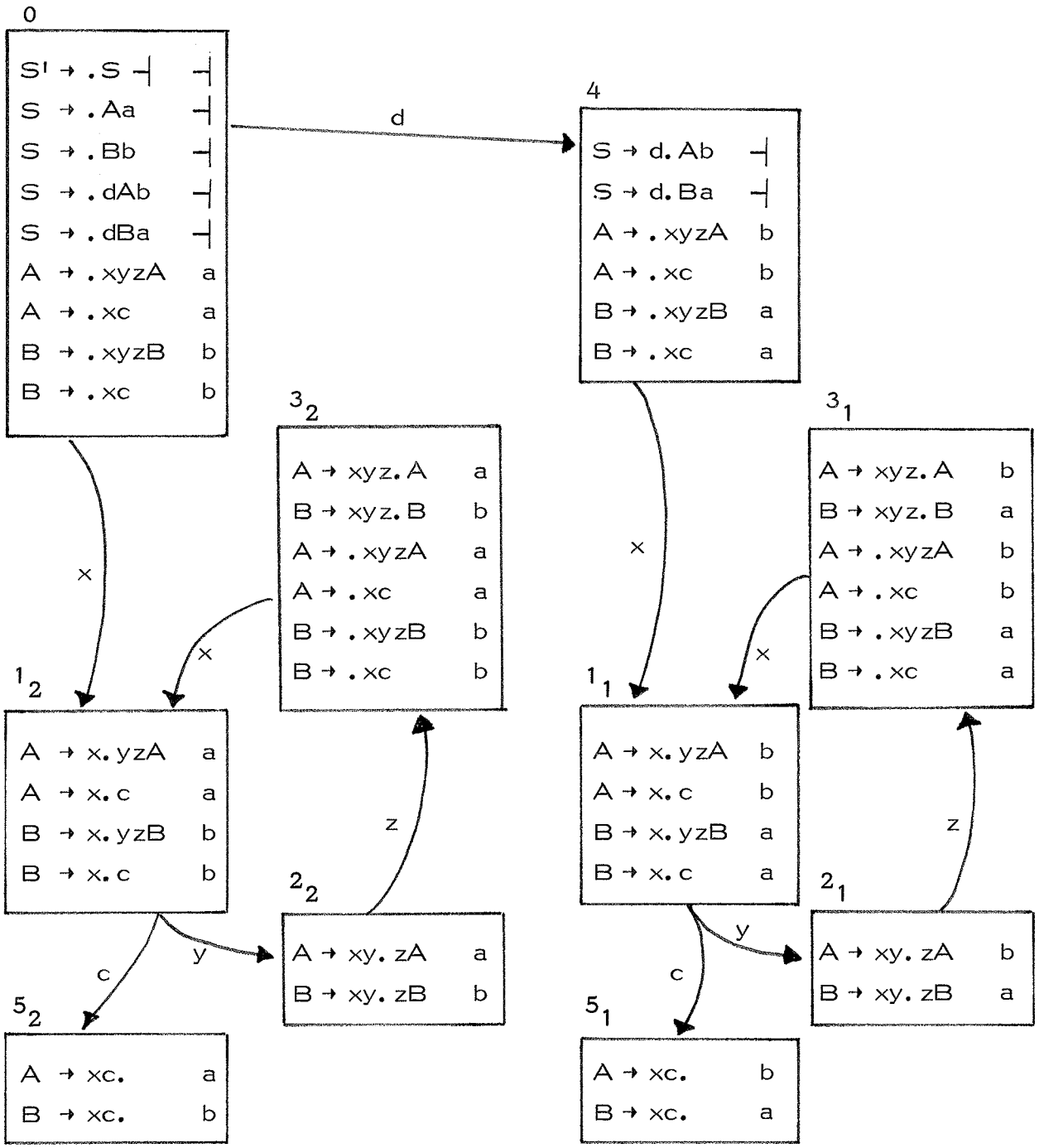
State number 5 of [A.1] is inconsistent and may therefore serve as a base for the initial activation of LA-SPLIT-1.

The following sequence of calls will appear : (with the level of recursion indicated in the leftmost column)

0 LA-SPLIT ([A → xc.], [B → xc.], 5)
1 LA-SPLIT ([A → x.c], [B → x.c], 1)
2 LA-SPLIT ([A → .xc], [B → .xc], 0)
2 LA-SPLIT ([A → .xc], [B → .xc], 4)
2 LA-SPLIT ([A → .xc], [B → .xc], 3)
3 LA-SPLIT ([A → xy.zA], [B → xy.zB], 2)

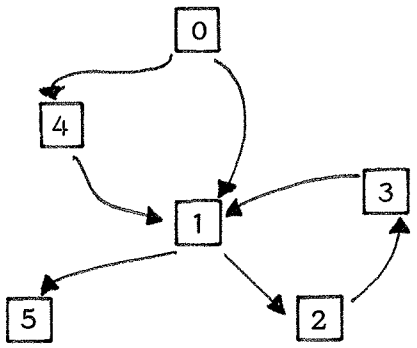

0
S' → .S ⊣
S → .Aa
S → .Bb
S → .dAb
S → .dBa
A → .xyzA
A → .xc
B → .xyzB
B → .xc
d
4
S → d.Ab
S → d.Ba
A → .xyzA
A → .xc
B → .xyzB
B → .xc
3
A → xyz.A
B → xyz.B
A → .xyzA
A → .xc
B → .xyzB
B → .xc
x
x
x
1
A → x.yzA
A → x.c
B → x.yzB
B → x.c
z
5
A → xc.
B → xc.
c
y
2
A → xy.zA
B → xy.zB

[A.1]


0
S' → .S ⊣ ⊣
S → .Aa ⊣
S → .Bb ⊣
S → .dAb ⊣
S → .dBa ⊣
A → .xyzA a
A → .xc a
B → .xyzB b
B → .xc b
d
4
S → d.Ab ⊣
S → d.Ba ⊣
A → .xyzA b
A → .xc b
B → .xyzB a
B → .xc a
x
x
3_2
A → xyz.A a
B → xyz.B b
A → .xyzA a
A → .xc a
B → .xyzB b
B → .xc b
3_1
A → xyz.A b
B → xyz.B a
A → .xyzA b
A → .xc b
B → .xyzB a
B → .xc a
1_2
A → x.yzA a
A → x.c a
B → x.yzB b
B → x.c b
1_1
A → x.yzA b
A → x.c b
B → x.yzB a
B → x.c a
z
z
2_2
A → xy.zA a
B → xy.zB b
2_1
A → xy.zA b
B → xy.zB a
y
y
c
c
5_2
A → xc. a
B → xc. b
5_1
A → xc. b
B → xc. a

[A. 2]

The two first calls on level 2 stop because BOTTOM becomes true; the call on level 3 stops because of PRED(2, y) = {1}, and 1 ∈ stack. The recursion then returns to level 1 and we arrive with the following snapshot where the contents of Stack, Blind and parts of LA-Done are given:

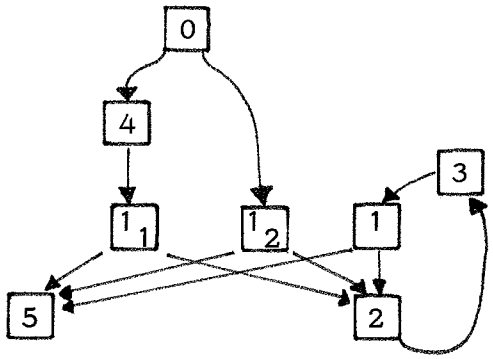


[A. 3a]

{([A → . xc] , [B → . xc], 0) , ([A → . xc] , [B → . xc], 4)} ⊆ LA-Done,

{5, 1} = stack , {2, 3} = Blind.

We have that PRED(1,x) = {0, 4} ∪ {3} where {3} ⊆ Blind, i. e. we cannot finish the treatment of state 1. We may perform a splitting of state 1 and reactivate LASPLIT ([A → . xc] , [B → . xc], 3) as shown by the next picture :

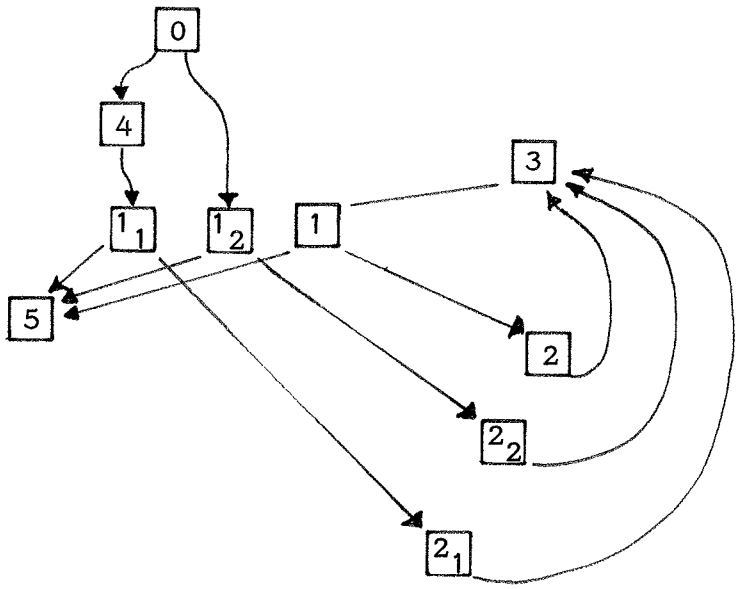


[A. 3b]

{([A → x. c] , [B → x. c], 1_1) , ([A → x. c] , [B → x. c], 1_2)} ⊆ LA-Done,

{5, 1, 3, 2} = Stack, Blind = ∅

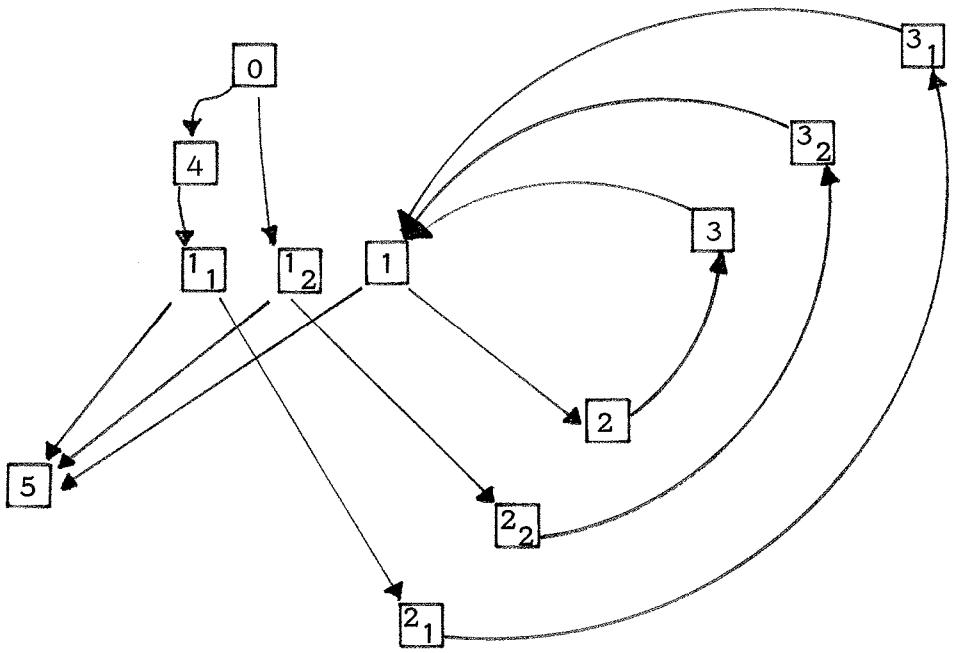
The next steps involve a splitting of states 2 and 3 as shown by the pictures [A. 3c] and [A. 3d].



[A. 3c]

$\{([A \to xy.zA], [B \to xy.zB], 2_1), ([A \to xy.zA], [B \to xy.zB], 2_2)\} \subseteq$ LA-Done,

Stack = $\{5, 1, 3\}$, Blind = $\{2\}$

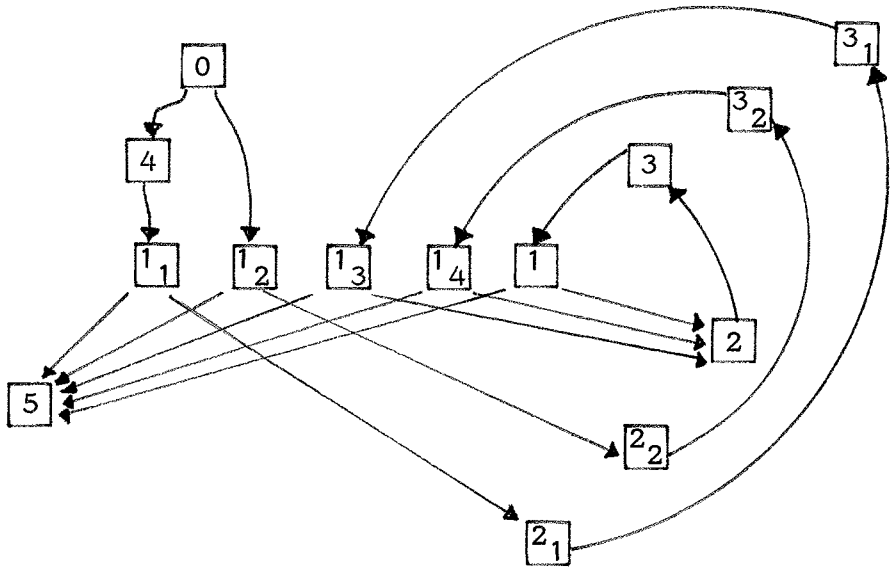


[A. 3d]

$\{([A \to .xc], [B \to .xc], 3_1), ([A \to .xc], [B \to .xc], 3_2)\} \subseteq$ LA-Done

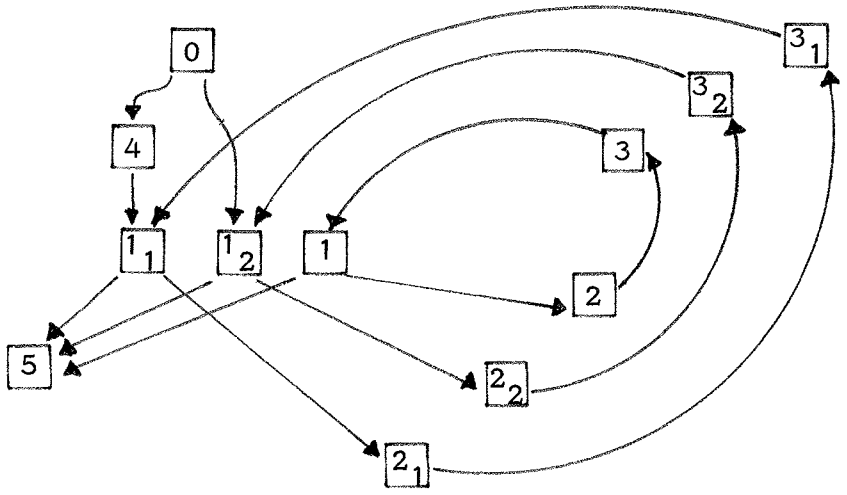
Stack = $\{5, 1\}$, Blind = $\{2, 3\}$

We have now finished a reactivation of the REPEAT-loop and may perform a new splitting of state 1 :



[A. 3e]

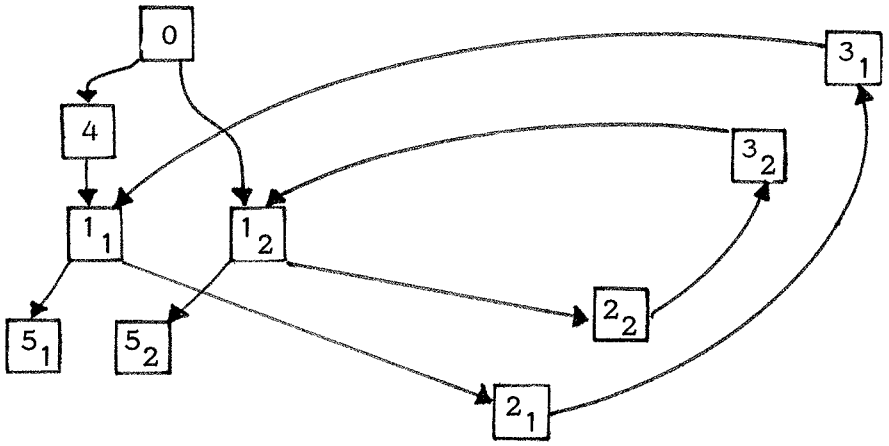
Now we find that the LA_1-lookahead of items in 1_1 and 1_2 is identical to the LA_1-lookahead of corresponding items in 1_3 and 1_4 respectively. We may thus merge 1_3 into 1_1 and 1_4 into 1_2 :



[A. 3f]

Stack = $\{5\}$, Blind = $\{1, 2, 3\}$.

Returning to the initial call we may make a splitting of state 5 and remove the states in Blind : The final result is shown by picture [A. 3g].



[A. 3g]

This is in fact the machine shown in [A. 2]. Note, however, that in [A. 1] and [A. 2] the A-successors of the states 3, 3_1 and 3_2 are not shown. In [A. 3g] 3_1 and 3_2 will have a common A-successor whereas this will not be the case in [A. 2]. The same is the case for the B-successor of states 3, 3_1 and 3_2

□

Example A. 2

Let G_2 be as G_1 except that the production

$$A \rightarrow xyzA$$

has been replaced by

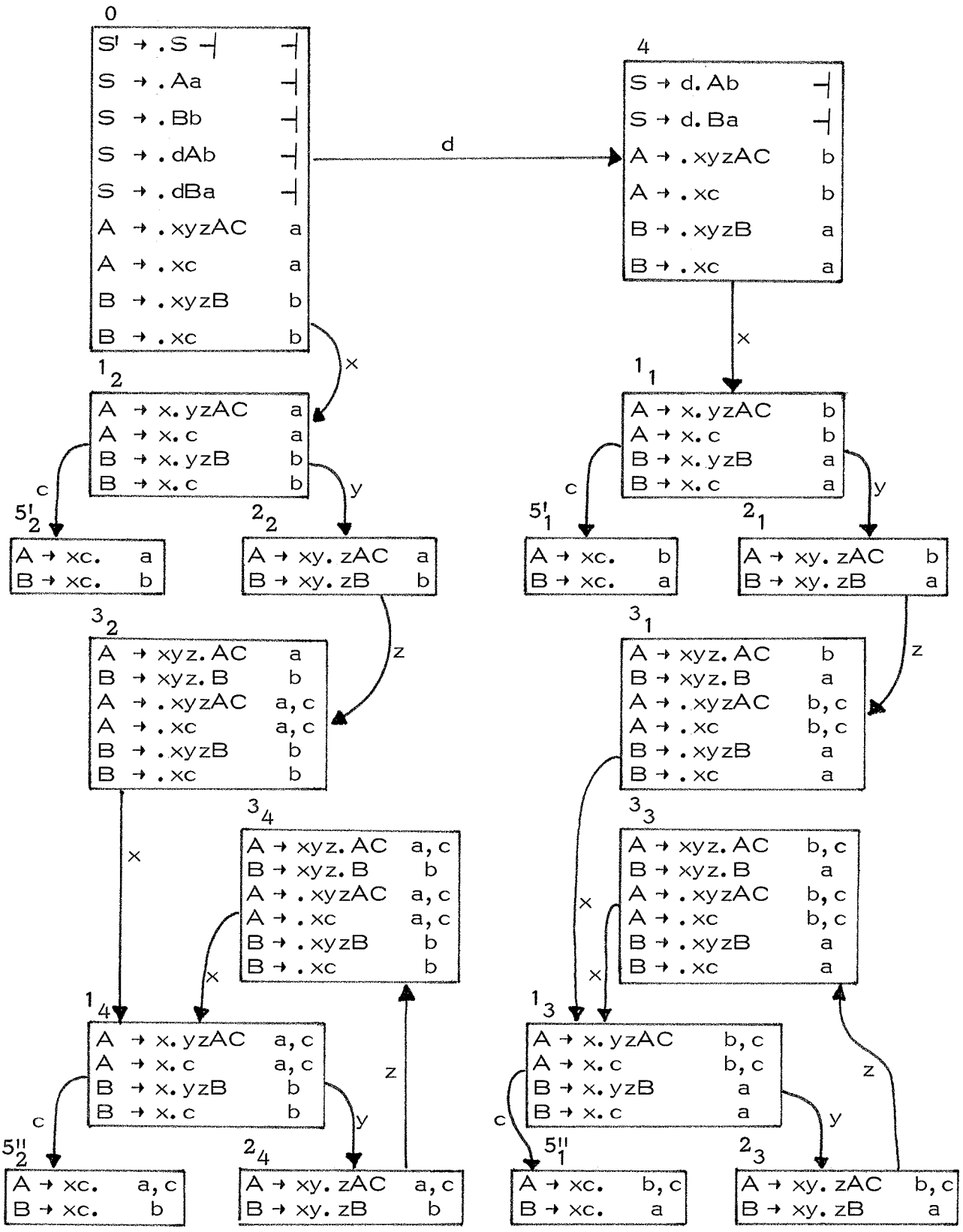
$$A \rightarrow xyzAC$$
$$C \rightarrow c$$
$$C \rightarrow e$$

The interesting part of the LR(0)-machine for G_2 is nearly unchanged from G_1 (the picture in [A. 1] may still be used). [A.4] gives the interesting part of the LR(1)-machine for G_2.

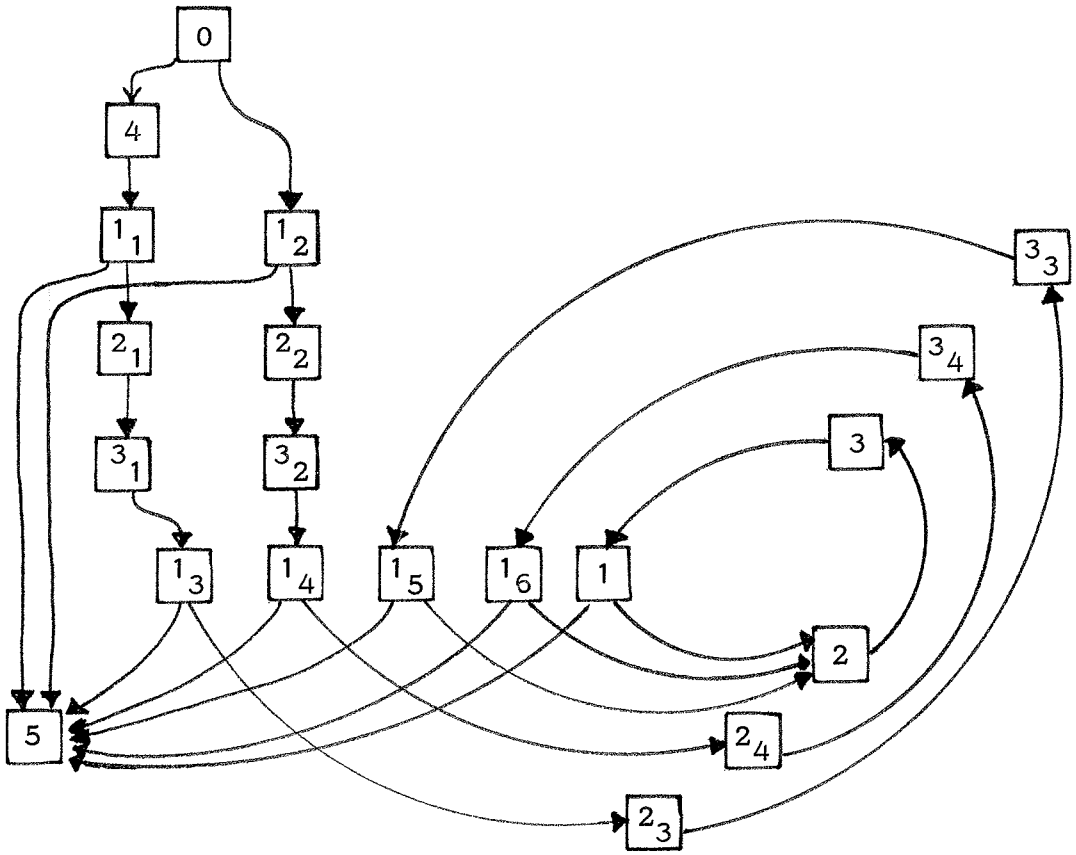
The state splitting process of the LR(0)-machine for G_2 differs at step [A. 3e]. Here it is not possible to merge 1_1 with 1_3 and 1_2 with 1_4.
It is necessary to perform yet another activation of

LA SPLIT ([A → . xc], [B → . xc], 3).

This implies a splitting of states 2, 3 and 1 as shown in the picture [A. 5f].

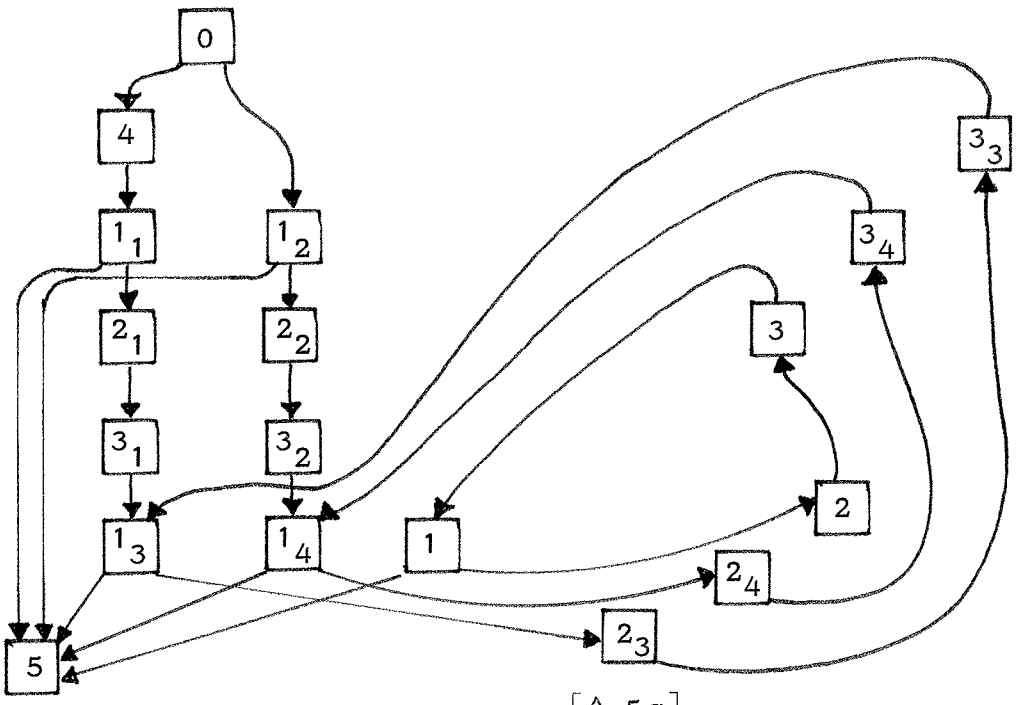

0
S' → .S ⊣ ⊣
S → .Aa ⊣
S → .Bb ⊣
S → .dAb ⊣
S → .dBa ⊣
A → .xyzAC a
A → .xc a
B → .xyzB b
B → .xc b
d
4
S → d.Ab ⊣
S → d.Ba ⊣
A → .xyzAC b
A → .xc b
B → .xyzB a
B → .xc a
x
1 2
A → x.yzAC a
A → x.c a
B → x.yzB b
B → x.c b
1 1
A → x.yzAC b
A → x.c b
B → x.yzB a
B → x.c a
c
y
5' 2
A → xc. a
B → xc. b
2 2
A → xy.zAC a
B → xy.zB b
5' 1
A → xc. b
B → xc. a
2 1
A → xy.zAC b
B → xy.zB a
z
3 2
A → xyz.AC a
B → xyz.B b
A → .xyzAC a, c
A → .xc a, c
B → .xyzB b
B → .xc b
3 1
A → xyz.AC b
B → xyz.B a
A → .xyzAC b, c
A → .xc b, c
B → .xyzB a
B → .xc a
3 4
A → xyz.AC a, c
B → xyz.B b
A → .xyzAC a, c
A → .xc a, c
B → .xyzB b
B → .xc b
3 3
A → xyz.AC b, c
B → xyz.B a
A → .xyzAC b, c
A → .xc b, c
B → .xyzB a
B → .xc a
1 4
A → x.yzAC a, c
A → x.c a, c
B → x.yzB b
B → x.c b
1 3
A → x.yzAC b, c
A → x.c b, c
B → x.yzB a
B → x.c a
5'' 2
A → xc. a, c
B → xc. b
2 4
A → xy.zAC a, c
B → xy.zB b
5'' 1
A → xc. b, c
B → xc. a
2 3
A → xy.zAC b, c
B → xy.zB a

[A.4]



[A.5f]

1_5 and 1_6 may now be merged into 1_3 and 1_4 respectively as shown by picture [A.5g]. Next state 5 is split and Blind is removed as shown by the final picture :



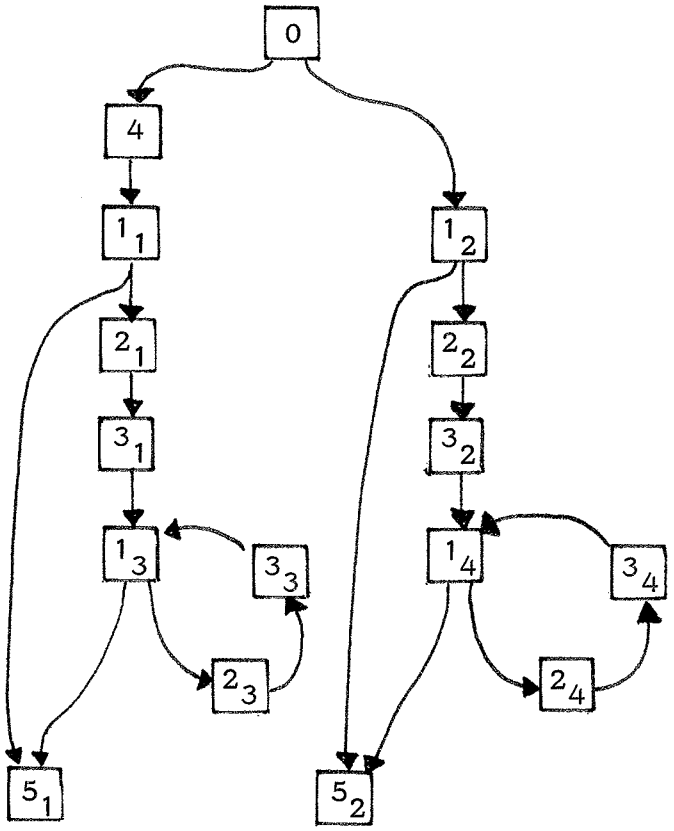
[A.5g]

$\{(I, J, 1), (I, J, 1_2), (I, J, 1_3), (I, J, 1_4)\} \subseteq$ LA-Done

where $I = [A \to x.c]$ and $J = [B \to x.c]$,

Stack = {5}, Blind = {1, 2, 3}

$PRED(5, c) = \{1_1, 1_2\} \cup \{1_3, 1_4\} \cup \{1\}$



[A.5h]

$\{([A \to xc.], [B \to xc.], 5_1), ([A \to xc.], [B \to xc.], 5_2)\} \subseteq$ LA-Done

This is close to the machine shown in [A.4]. The remark following [A.3g] about A- and B-successors also holds here. Note also that 5_1 (5_2) in [A.5b] is a merge at $5'_1$ and $5''_1$ ($5'_2$ and $5''_2$) in [A.4].

□