ISSN 0105-8517

A GENERAL ALGORITHM FOR SOLVING A SET OF RECURSIVE EQUATIONS (Exemplified by LR-theory)

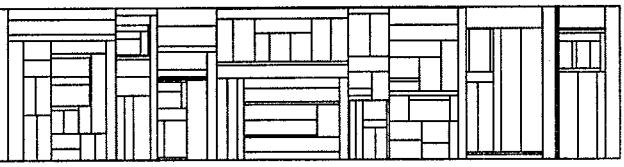
by

Bent Bruun Kristensen
and
Ole Lehrmann Madsen

DAIMI PB-110
February 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 general algorithm for computing the solutions to a set of recursive equations of a general form is given. The form of the recursive equations is closely related to results on LR-theory. The algorithm appears as intuitively understandable in an abstract form. The correctness of the algorithm is proved. An analysis of the complexity shows that the algorithm compares well to existing algorithms. Examples of application of the general algorithm to elements of LR-theory, as LALR(1)-lookahead computation and LR(1)-testing, are included.

Contents

1. Introduction 1

2. A General Algorithm 2

2.1 The basic Algorithm 2

2.2 An Improved Version 4

2.3 Complexity and Implementation 9

2.4 Comparison 11

2.5 A Proof of the Improved Algorithm 12

3. Examples from LR-theory 15

3.1 Definitions 15

3.2 LALR(1)-Lookahead Algorithm 17

3.3 Algorithm for LR(1)-testing 20

4. References 24

1. Introduction

A general algorithm for computing the minimal solution to a set of recursive equations of a general form is presented. The form of the recursive equations is closely related to results on LR-theory presented in [Kristensen & Madsen 79a, 79b]. The algorithm is based on an outline to an algorithm for computing LALR(k)-lookahead given in [Kristensen & Madsen 79a]. This outline is an improvement of a basic algorithm given in [Kristensen & Madsen 79a] which computes a solution for a single argument only, i.e. for an item, I, in a state, T, the LALR(k)-lookahead is computed. In the general algorithm solutions for a number of arguments are computed as a by-product, i.e. LALR(k)-lookahead may be computed for the initial argument, (I, T)$\in$ item x state as well as for any argument (J, S)$\in$ item x state on which the computation for (I, T) may depend, according to a set of recursive equations for LALR(k)-lookahead.

We give an informal introduction of the principle used in the general algorithm, and state the algorithm in abstract form. A theorem on the correctness of the algorithm is proved.

An analysis of the complexity of the algorithm is given, including remarks on implementation.

When interpreting the task of the algorithm in terms of strongly connected components of directed graphs, the complexity of the algorithm can compare with the results in [Tarjan 72].

As examples on the usability of the general algorithm, especially in LR-theory, we give algorithms for computing LALR(1)-lookahead and for LR(1)-testing.

Acknowledgement

We wish to thank Erik Meineche Schmidt for helpful discussions and Thomas J. Pennello for critical comments on a preliminary version of the complexity analysis.

2. A General Algorithm

We assume a set of recursive equations on the form defined as follows:

Definition 2.1

Let $F : D \to \mathcal{P}(R)$ be a function defined by the set of equations:

[2.2] Let $a \in D$ then

(*) $$F(a) = G(a) \cup \cup\{F(b) \mid b \in P_a\}$$

where G is a function

$$G : D \to \mathcal{P}(R)$$

and $\{P_a\}_{a \in D}$ is a family of powersets of D.

Let F_o denote the minimal solution to [2.2], in the sense that if F_1 is another solution to [2.2] then

$$\forall d \in D : F_o(d) \subseteq F_1(d).$$

□

2.1 The Basic Algorithm

The following algorithm computes $F_o(a)$. We use the construct

FOR $b \in P_a$ WHERE B DO S ENDFOR

meaning

FOR $b \in P_a$ DO

IF B THEN S ENDIF

ENDFOR

(*) $\cup\{F(b) \mid b \in P_a\}$ means $\bigcup_{b \in P_a} F(b)$

Algorithm 2.3 [Kristensen & Madsen 79a]

```
FUNCTION  F(a : D) : SET OF R;
VAR  RES : SET OF R;
  FIN, STACK : SET OF D;

  PROCEDURE  FF(a : D);
  BEGIN
      STACK  := STACK U{a};
      RES    := RES U G(a);
      FOR     b ∈ P_a WHERE  b ∉ STACK U FIN DO
            FF(b);
      ENDFOR;
      STACK  := STACK \ {a};
      FIN    := FIN U {a};
  END FF;

BEGIN
      RES  := FIN  := STACK  := Ø;
      FF(a);
      F  := RES;
END F;
```

□

Theorem 2.4 [Kristensen & Madsen 79a]

If the domain D is finite and the function G is correctly implemented then $F_0(a)$ is correctly computed by F(a) in algorithm 2.3. □

We remark that the sets FIN and STACK can be replaced by a single set, without affecting the results of algorithm 2.3.

An upper bound on the number of activations of the procedure FF of algorithm 2.3 for a single activation of F is

$$O(|D|),$$

The corresponding upper bound for activations of F for all $a \in D$ is

$$O(|D|^{|D|})$$

In the remaining part of this section we shall modify algorithm 2.3 to improve this last bound considerably.

2.2 An Improved Version

We notice that in algorithm 2.3 $F_0(a)$ is computed by invoking FF(a) to deliver the result. Then FF may invoke itself recursively by arguments $b \in D$ to deliver partial results of $F_0(a)$. The total set of arguments for FF for an initial argument, a, may be denoted as $CLOSURE_0(a)$, - and is defined as the minimal solution to the following set of recursive equations:

Let $a \in D$. Then

$$CLOSURE(a) = \{a\} \cup \bigcup \{CLOSURE(b) \mid b \in P_a\}$$

The purpose of this section is to improve algorithm 2.3 to compute $F_0(b)$ for all $b \in CLOSURE_0(a)$ as a by-product, given the initial argument a.

We extend algorithm 2.3 to save the partial result computed by each activation of FF, by collecting the contributions to RES for each activation of FF and all its recursive activations.

For argument $a \in D$ let this partial result be denoted RES(a).

If

[2.5] $\nexists\ b \in CLOSURE_0(a) : a \in CLOSURE_0(b)$

then $RES(a) = F_0(a)$ and we may place a in FIN with its final value.

If [2.5] is false, we may have that $RES(a) \subset F_0(a)$. In this case we have that $CLOSURE_0(a) = CLOSURE_0(b)$ and $F_0(a) = F_0(b)$. In the algorithm this situation may be detected in a number of ways:

[2.6a] $b \in STACK$: we include a dependency marker $(\#b)$ in RES(a) to indicate that RES(b) must be added to RES(a), in order to complete RES(a) to $F_0(a)$.

[2.6b] $b \in FIN$, but RES(b) is incomplete, i.e. RES(b) contains a nonempty set of dependency markers corresponding to arguments in STACK. We include RES(b) in RES(a) in this case.

[2.6c] FF(b) is activated, but the result RES(b) is incomplete. Similar to [2.6b].

Finally when all $b \in P_a$ have been treated one of three cases exists (assume $D^{\#} = \{\#a \mid a \in D\}$):

Let $RES^{O}(a)$ denote $RES(a) \cap R$ and $RES^{\#}(a)$ denote $RES(a) \cap D^{\#}$, such that $RES(a) = RES^{O}(a) \cup RES^{\#}(a)$ and $RES^{O}(a) \cap RES^{\#}(a) = \emptyset$.

[2.7a] $RES^{\#}(a) = \emptyset$ then $RES(a) = F_0(a)$.

[2.7b] $RES^{\#}(a) = \{\#a\}$ then $RES^{O}(a) = F_0(a)$.

[2.7c] $RES^{\#}(a) \neq \emptyset \wedge RES^{\#}(a) \neq \{\#a\}$ then for all $\# c \in RES^{\#}(a) \setminus \{\#a\}$, RES(c) must be added to complete RES(a) to $F_0(a)$.

In [2.7b] and in [2.7c] if $\# a \in RES^{\#}(a)$, we use implicitly that F_0 is the <u>minimal</u> solution to [2.2], by including $\emptyset$. Any set may be included to have a solution to [2.2] in these cases.

In any of the cases [2.7] a is placed in FIN with RES(a) being $RES(a) \setminus \{\#a\}$, complete or incomplete.

In [2.6b] and [2.6c] it is assumed that $\sharp c \in RES^{\sharp}(b) \Rightarrow c \in STACK$. As a is removed from STACK we need to keep this condition invariant by replacing any occurrence of $\sharp a$ in $RES^{\sharp}(c)$ for $c \in FIN$ by the result of the activation FF(a), namely

[2.8] $RES(a) \setminus \{\sharp a\}$.

A number of improvements are immediately possible to the above outline:

[2.9a] In [2.7c] let the runtimestack corresponding to STACK have the form

$\boxed{\alpha\ d\ \beta}$.

such that

$d \in RES^{\sharp}(a) \setminus \{\sharp a\} \wedge \nexists\, c \in \alpha : \sharp\, c \in RES^{\sharp}(a) \setminus \{\sharp a\}$

i.e. d is lowest in STACK.

We define

$MIN_{\sharp}(RES^{\sharp}(a) \setminus \{\sharp a\}) = \{\sharp d\}$.

When the execution of FF with argument d is continued, then for all $\sharp c \in RES^{\sharp}(a) \setminus \{\sharp a\}$, such that $c \neq d$, we have that $RES(c) \subseteq RES(d)$ and that the activation of FF with argument c has been terminated. Thus in [2.7c] the result may be restricted to $RES^{O}(a) \cup \{\sharp d\}$.

[2.9b] Again in [2.7c] we may, based on [2.9a], place a in FIN with the value $\{\sharp d\}$ but return $RES^{O}(a) \cup \{\sharp d\}$ as the result of the activation FF(a).

[2.9c] In [2.8] any occurrence of $\sharp a$ may now be replaced by $\sharp d$ in case [2.7c].

The optimizations in [2.9] have been included in the following algorithm based on the above outline.

Algorithm 2.10

```
1   TYPE D# = {#a | a ∈ D};
2        R# = SET OF(R ∪ D#);
3
4   VAR  RES : TABLE(INDEX D, ELEMENT R#);
5        FIN : SET OF D;
6
7   FUNCTION F(a : D) : SET OF R;
8   VAR  STACK : SET OF D;
9
10       FUNCTION FF (a : D) : R#;
11       BEGIN
12          STACK := STACK ∪ {a};
13          RES(a) := G(a);
14          FOR b ∈ P_a DO
15            IF b ∈ FIN THEN RES(a) := RES(a) ∪ RES(b)
16            ELSEIF b ∈ STACK THEN RES(a) := RES(a) ∪ {#b}
17            ELSE RES(a) := RES(a) ∪ FF(b)
18            ENDIF;
19          ENDFOR;
```

```
20          ASSUME RES°(a) = RES(a) ∩ R, RES♯(a) = RES(a) ∩ D♯;
21          IF RES♯(a) = ∅ THEN FF := RES°(a)
22          ELSE
23            IF RES♯(a) = {♯a} THEN
24               FF := RES°(a);
25               RES(a) := RES°(a);
26            ELSE
27               ♯d := MIN♯(RES♯(a));
28               FF := RES°(a) ∪ {♯d};
29               RES(a) := {♯d};
30            ENDIF;
31            ASSUME FIN_a = {b∈FIN | RES(b) = {♯a}};
32            FOR b∈ FIN_a DO RES(b) := RES(a); ENDFOR;
33          ENDIF;
34          STACK := STACK ∖ {a};
35          FIN := FIN ∪ {a};
36        END FF;
37
38    BEGIN
39          IF a ∈ FIN THEN F := RES(a)
40          ELSE
41            STACK := ∅;
42            F := FF(a);
43          ENDIF;
44    END F;                                                    □
```

Let $P \equiv$

$$\forall\ b \in FIN : RES(b) = F_0(b) \wedge CLOSURE_0(b) \subseteq FIN$$

The following theorem states that algorithm 2.10 computes the results discussed:

Theorem 2.11

Let F be given as in algorithm 2.10.
Then

$\{ P \}$

L := F(a)

$\{ P \wedge a \in FIN \wedge L = F_0(a) \}$

Proof: Straightforward using lemma 2.13 in section 2.5. □

2.3 Complexity and Implementation.

An upper bound on the number of activations of the procedure FF of algorithm 2.10, when activating F for a single argument $a \in D$, is

$$O(|D|)$$

The same bound is valid when activating F for all $a \in D$, i.e. we may compute $F_0(a)$ for all $a \in D$ within this bound on the activations of FF.

For each activation of FF the number of U-operations of the FOR-loop in 14-19 is linear in $|P_a|$, i.e. for $O(|D|)$ activations of FF we have $O(\sum_{a \in D} |P_a|)$ U-operations. The number of := operations of the FOR-loop at 32 is linear in $|D|$, i.e. for $O(|D|)$ activations of FF we have $O(|D|^2)$:= operations. Because $|p_a| \leq |D|$ for all $a \in D$, we may conclude that to compute $F_0(a)$ for all $a \in D$ we use at most

$$O(|D|^2)$$

U- and := - operations.

We remark that the ELSE-part in 22-33 is only used when circularity exists in the recursive equations [2.2], i.e. when [2.5] does not hold.

In the following we give an implementation of the sets $RES^{\#}(a)$ for $a \in D$, reducing these bounds. By implementing the sets FIN_a (see line 31 of algorithm 2.10) directly, the problem may be transformed into the UNION-FIND problem discussed in [Aho, Hopcroft & Ullman 76]. The changes to the algorithm may then be outlined as:

For $b \in FIN$, we let $RES(b) = \{\#\}$ denote that $b \in FIN_c$ for some c.

The FIND-operation is then used at 15 when $RES(b) = \{\#\}$:

```
LET b ∈ FIN_c;                "c := FIND(b)"
RES(a) := RES(a) ∪ {#c};
```

Line 31 is deleted.

Line 32 is inserted as line 25.1. The TRAVERSAL-operation of the elements of a given set may be added to the UNION-FIND implementation, e.g. by chaining the elements of a set. We remark that the sets are disjoint and that any element is visited in a TRAVERSAL-operation at most one time.

Line 29 is replaced by

```
RES(a) := {#};
FIN_d := FIN_d ∪ FIN_a;    "UNION(a,d,d)"
FIN_d := FIN_d ∪ {a};      "INSERT(a,d)"
```

Let $n_o = \sum_{a \in D} |P_a|$.

For $O(|D|)$ activations of FF we have that the changes in algorithm 2.10 outlined above imply that

the number of FIND-operations is $O(n_o)$.

the number of UNION-operations is $O(|D|)$,

and the number of :=, ∪, etc-operations is $O(n_o)$.

Using the efficent implementation of UNION-FIND as given in [Aho, Hopcroft & Ullman 76] then, when F of algorithm 2.10

is activated for all a $\in$ D, we have at most

$$O(n_o G(n_o))$$

executions of elementary operations (i.e. :=, $\cup$ and the operations used to implement FIND and UNION). The G-function may be treated as constant for all pratical values given as argument, e.g. $G(n) \leq 5$ for all $n \leq 2^{65536}$.

2.4 Comparison

After the publication of [Kristensen & Madsen 79a] an independent work on a method for efficient computation of LALR(1)-look-ahead sets has been published in [DeRemer & Pennello 79]. This method is based on an efficient algorithm for computing the transitive closure of a relation given in [Eve & Kurki-Suonio 77]. This method is itself based on an algorithm in [Tarjan 72] for finding the strongly connected components (SCC's) in a directed graph. This SCC-algorithm is treated thoroughly in [Aho, Hopcroft & Ullman 76].

On this background we shall compare algorithm 2.10 to the above mentioned algorithms by interpreting algorithm 2.10 in terms of directed graphs. By letting the elements a $\in$ D be vertices, then an edge, (a,b), is given in the directed graph if and only if b $\in P_a$. In this terminology the effect of algorithm 2.10 is to traverse a connected subgraph according to the depth first search principle, and to compute some results based on the values at the vertices encountered on the traversal. The presence of strongly connected components implies circularity and is the reason for the introduction of the dependency markings.

Interpreted in this way algorithm 2.10 may be seen as an approach to the SCC-problem, distinct to the method in [Tarjan 72]. In that method the SCC's of the graph are handled by an extended stack-concept during the traversal: The elements of a strongly connected component are kept on the stack until all its elements are known; then the entire set is popped off the stack. The dependency marking is performed by means of indices in the stack. It seems that these differences from our approach have the effect on the com-

plexity that the factor $G(n_o)$ may be omitted. We notice that in the applications of algorithm 2.10 on LR-theory, the SCC's will appear very infrequent and that the dependency marking in algorithm 2.10 is not involved if no SCC is present. In this case algorithm 2.10 directly gives the upper bound $O(n_o)$, as mentioned in section 2.3.

2.5 A Proof of the Improved Algorithm

We introduce a list of definitions:

[2.12a] $$P_{FIN} \equiv \forall\, b \in FIN:$$
$$((F_o(b) = RES(b))$$
$$\vee(RES(b) = \{\#c\} \wedge c \in STACK \wedge F_o(b) = F_o(c)))$$
$$\wedge(P_b \subseteq STACK \cup FIN))$$

[2.12b] Let $b \in D$ and let the runtimestack corresponding to STACK have the form

$$\boxed{\alpha\ b_1 b_2 \ldots b_n}$$

If $b = b_1$ then

$$TOP(b,STACK) = \{b_1, b_2, \ldots, b_n\}$$
$$\cup \cup\{TOP(c,STACK) \mid \#c \in RES^{\#}(b_i) \wedge i \in [1,n]\}$$

As TOP is defined by a set of recursive equations, we shall assume TOP to be the minimal solution to these equations. TOP captures a topmost part of the runtimestack, namely the largest part reachable through $RES^{\#}(b_i)$, recursively.

[2.12c] Let Q_a for $a \in D$ denote the subset of P_a for which the statement of the FOR-loop at 14-19 has been executed. Precisely, Q_a may be added to the algorithm as follows:

insert line 10.1 : VAR Q_a : SET OF D;
insert line 12.1 : $Q_a := \emptyset$;
insert line 18.1 : $Q_a := Q_a \cup \{b\}$;

[2.12d] Let $b \in D$ such that $b \in STACK$. Then

$$\tilde{F}(b, STACK) = \bigcup\{\hat{F}(c) \mid c \in TOP(b, STACK)\}$$

where

$$\hat{F}(c) = RES^{O}(c) \cup \bigcup\{F_o(d) \mid d \in P_c \setminus Q_c\}$$

[2.12e] Let $L \subseteq R$. Then

$$P_{STACK}(L) \equiv \forall\, b \in STACK: (F_o(b) = L \cup \tilde{F}(b, STACK)) \wedge (\forall\, \sharp c \in RES^{\sharp}(b) : c \in STACK)$$

Lemma 2.13

Let $L \subseteq R^{\sharp}$ and let L^{O}, $L^{\sharp}$ be defined as for RES. Let FF be given as in algorithm 2.10 and let

$$P \equiv P_{FIN} \wedge STACK = STACK' \wedge STACK \cap FIN = \emptyset.$$

Then

$$\{P \wedge P_{STACK}(\emptyset) \wedge FIN = FIN' \wedge a \notin STACK \cup FIN\}$$

$$L := FF(a)$$

$$\{P \wedge FIN' \subset FIN \wedge a \in FIN \wedge ((L = F_o(a) \wedge P_{STACK}(\emptyset)) \vee (L^{\sharp} = \{\sharp c\} \wedge P_{STACK}(L^{O}) \wedge c \in STACK \wedge F_o(a) = F_o(c)))\}$$

Proof:

Let $P_1 \equiv P_{FIN} \wedge STACK \cap FIN = \emptyset \wedge STACK = STACK' \cup \{a\} \wedge FIN' \subseteq FIN$

After line 13, $Q_a = \emptyset$ and RES(a) = G(a) and thus we have that P_2 is valid:

$$P_2 \equiv F_o(a) = \tilde{F}(a, STACK) \wedge P_1 \wedge P_{STACK}(\emptyset).$$

At line 14-19 a cycle of the FOR-loop includes $F_o(b)$ in RES(a) using an expression of $F_o(b)$ given by either P_{FIN}, $P_{STACK}(\emptyset)$ or the invariant for FF. At line 17, $P_{STACK}(L^o)$ is restored to $P_{STACK}(\emptyset)$ by including $L^o = RES^o(b)$ in RES(a). Before line 19, P_2 is still valid.

After line 19 P_3 is valid:

$$P_3 \equiv P_2 \wedge Q_a = P_a.$$

The cases at line 21 and line 23-25 are treated using P_3. At line 27-29 the identity

$$TOP(MIN_{\#}(RES^{\#}(a), STACK)) = \cup\{TOP(c, STACK) \mid \#c \in RES^{\#}(a)\}$$

may be used.

The remaining steps are then straightforward, using at line 34 that

$$F_o(a) = FF^o \cup \tilde{F}(a, STACK \setminus \{a\})$$

where FF^o, $FF^{\#}$ may be defined as for RES.

The recursion will terminate, because the set FIN is increased with one element for each activation of FF.

□

3. Examples from LR-theory

Before we give the algorithms, we restate the necessary definitions and notation as used in [Kristensen & Madsen 79a, 79b].

3.1 Definitions

We use the terminology and conventions from [Aho & Ullman 72, 73]. Any CFG, G = (N, Σ, P, S) is assumed to be extended with a new start symbol S' and the production S' → S⊣^{k} where ⊣ ∉ (N ∪ Σ).

Definition 3.1

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 will 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 $(M_k) \times (N \cup \Sigma)^* \rightarrow M_k$.

Definitions

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

[3.2] Let $T \in M_k$, then

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

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

$$CORE([A \rightarrow \alpha.\beta, u]) = [A \rightarrow \alpha.\beta], \text{ and}$$

$$CORE(S) = \{CORE(I) \mid I \in S\}.$$

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

[3.4] Let $T \in M_0$, then

$URCORE_k(T) = \{S \in M_k \mid CORE(S) = T\}$.

[3.5] Let $T \in M_0$, then

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

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

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

[3.7] Let $T \in M_0$ and $[A \to \alpha.\pi], [B \to \beta.\delta] \in T$ and $A, B \subseteq \Sigma^{*^k}$, then

$LRCOND_k(A, [A \to \alpha.\pi], B, [B \to \beta.\delta], T) \equiv$

$\forall S \in URCORE_k(T)$:

(*) $A \oplus_k LR_k([A \to \alpha.\pi], S) \cap B \oplus_k LR_k([B \to \beta.\delta], S) = \emptyset$

[3.8] Let $T \in M_0$ and $[A \to \alpha.\beta] \in T$ and $A \subseteq \Sigma^{*^k}$, then

$BOTTOM_k(A, [A \to \alpha.\beta], T) \equiv$

(**) $(\alpha \neq e \Rightarrow |A|_{min} \geq k) \wedge$

$(\alpha = e \Rightarrow$

$\forall [B \to \varphi.A\Psi] \in T$:

$BOTTOM_K(A \oplus_k FIRST_k(\Psi), [B \to \varphi.A\Psi], T))$

□

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

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

3.2 LALR(1)-Lookahead Algorithm

In [Kristensen & Madsen 79a] we have proved that the following set of recursive equations is valid for $LALR_k$ as defined in [3.5]:

[3.9] Let $T \in M_0$ and $[A \to \alpha.\beta] \in T$ such that $[A \to \alpha.\beta] \neq [S' \to .S \dashv^k]$. Then

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

In the pratical case for k=1, [Kristensen & Madsen 79a] includes the result, that $LALR_1$ is the minimal solution to the following variant of [3.9]:

[3.10] Let $T \in M_0$ and $[A \to \alpha.\beta] \in T$. Then

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

By theorem 2.11 we may then use the general algorithm 2.10 to produce a special algorithm for computing $LALR_1$-lookahead based on [3.10].

The equations in [3.10] have not exactly the form assumed in [2.2] but a simple transformation will do.

The $LALR_1$-lookahead computation is realized by the following algorithm:

Algorithm 3.11

```
TYPE D = item × state;
     D# = {#(I,T) | (I,T) ∈ D};
     Σ# = SET OF (Σ ∪ D#),
VAR  RES : TABLE(INDEX D, ELEMENT Σ#);
     FIN : SET OF D;

FUNCTION LALR-1 ((I,T):D) : SET OF Σ;
VAR  STACK : SET OF D;
     FUNCTION LALR((I,T):D) : Σ#;
     BEGIN
        STACK := STACK ∪ {(I,T)};
        ASSUME I = [A→α.β];
        RES(I,T) := ∅;
        FOR S ∈ PRED(T,α) DO
          FOR [B→φ.AΨ] ∈ S DO
             RES(I,T) := RES(I,T) ∪ (FIRST1(Ψ) ∖ {e});
             IF e ∈ FIRST1(Ψ) THEN
                ASSUME J = [B→φ.AΨ];
                IF (J,S) ∈ FIN THEN RES(I,T) := RES(I,T) ∪ RES(J,S)
                ELSEIF (J,S) ∈ STACK THEN
                    RES(I,T) := RES(I,T) ∪ {#(J,S)}
                ELSE RES(I,T) := RES(I,T) ∪ LALR(J,S);
                ENDIF;
             ENDIF;
          ENDFOR;
        ENDFOR;
```

```
        ASSUME RES°(I,T) = RES(I,T) ∩ Σ, RES#(I,T) = RES(I,T) ∩ D#;
        IF  RES#(I,T) = ∅ THEN LALR := RES°(I,T)
        ELSE
            IF RES#(I,T) = {#(I,T)} THEN
                LALR := RES°(I,T);
                RES(I,T) := RES°(I,T);
            ELSE
                #(I',T') := MIN#(RES#(I,T));
                LALR := RES°(I,T) ∪ {#(I',T')};
                RES(I,T) := {#(I',T')};
            ENDIF;
            ASSUME FIN(I,T) = {(I',T')∈FIN | RES(I',T')={#(I,T)}};
            FOR(I',T') ∈ FIN(I,T) DO
                RES(I',T') := RES(I,T);
            ENDFOR;
        ENDIF;
        STACK := STACK ∖ {(I,T)};
        FIN := FIN ∪ {(I,T)};
    END LALR;
BEGIN
    IF (I,T) ∈ FIN THEN LALR-1 := RES(I,T)
    ELSE
        STACK := ∅·
        LALR-1 := LALR(I,T);
    ENDIF;
END LALR-1;
```

3.3 Algorithm for LR(1)-Testing

In [Kristensen & Madsen 79b] we have proved that the following set of recursive equations is valid for $LRCOND_k$ as defined in [3.7]:

[3.12] Let $T \in M_0$ and $[A \to \alpha\beta.\pi]$, $[B \to \beta.\delta] \in T$ and $\mathcal{A}, \mathcal{B} \subseteq \Sigma^{*^k}$. Then

$$LRCOND_k(\mathcal{A}, [A \to \alpha\beta.\pi], \mathcal{B}, [B \to \beta.\delta], T) \equiv$$

$$(BOTTOM_k(\mathcal{B}, [B \to \beta.\delta], T) \Rightarrow$$

$$\mathcal{A} \oplus_k LALR_k([A \to \alpha\beta.\pi], T) \cap \mathcal{B} \oplus_k LALR_k([B \to \beta.\delta], T) = \emptyset)$$

$$\wedge$$

$$(\neg BOTTOM_k(\mathcal{B}, [B \to \beta.\delta], T) \Rightarrow$$

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

$$\forall [C \to \varphi.B\Psi] \in S:$$

$$LRCOND_k(\mathcal{A}, [A \to \alpha.\beta\pi], \mathcal{B} \oplus_k FIRST_k(\Psi), [C \to \varphi.B\Psi], S))$$

Again we restrict the discussion to the practical case for k=1. In [Kristensen & Madsen 79b] it is proved that $LRCOND_1$ is the minimal solution to the following variant of [3.12]:

[3.13] Let $T \in M_0$ and $[A \to \alpha\beta.\pi]$, $[B \to \beta.\delta] \in T$. Then

$$LRCOND_1(\{e\}, [A \to \alpha\beta.\pi], \{e\}, [B \to \beta.\delta], T) \equiv$$

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

$$(\forall [C \to \varphi.B\Psi] \in S \text{ where } \Psi \Rightarrow^* e:$$

$$LRCOND_1(\{e\}, [A \to \alpha.\beta\pi], \{e\}, [C \to \varphi.B\Psi], S))$$

$$\wedge$$

$$(LALR_1([A \to \alpha.\beta\pi], S) \cap (\cup\{FIRST_1(\Psi) \mid [C \to \varphi.B\Psi] \in S\} \setminus \{e\}) = \emptyset)$$

Using the following result, which is an immediate consequence of combining the LR(k)-definition and [3.7]:

[3.14] G is LR(k) if and only if

$\forall$ T $\in$ M_0:

$\forall$ [A→α.π], [B→β.] $\in$ T:

$LRCOND_k(EFF_k(\pi), [A\rightarrow\alpha.\pi], \{e\}, [B\rightarrow\beta.], T)$

we may then base an algorithm for LR(1)-testing according to [3.13], on the scheme given by algorithm 2.10.

The equations in [2.2] are formulated in terms of sets, while the equations for $LRCOND_1$ in [3.13] are in terms of boolean values. We may resolve this problem by implementing boolean as a set. The type boolean$^\sharp$ is a combination of boolean and the dependency markers, defined as follows:

Let the set of dependency markers be described by $D^\sharp$, then

boolean$^\sharp$ = {TRUE,FALSE} $\cup$ SET OF $D^\sharp$.

Let Q be of type boolean$^\sharp$ and b of type boolean, then

b := Q means b := (FALSE $\notin$ Q)

Q := Q $\cup$ b means Q := Q $\cup$ {TRUE} if b
{FALSE} if $\neg$b.

The LR(1)-testing is realized by the following algorithm:

Algorothm 3.15

```
TYPE D = item × item × state;
     D# = {♯(I,J,T) | (I,J,T) ∈ D};
VAR  RES : TABLE(INDEX D, ELEMENT boolean♯);
     FIN : SET OF D;

FUNCTION LRCOND-1 ((I,J,T):D):boolean;
VAR  STACK : SET OF D;
     FUNCTION LRCOND((I,J,T):D):boolean♯;
     BEGIN
        STACK := STACK ∪ {(I,J,T)};
        ASSUME I = [A→αβ.π], J = [B→β.δ];
        RES(I,J,T) := {TRUE};
        FOR S ∈ PRED(T,β) DO
           RES(I,J,T) := RES(I,J,T) ∪ (LALR1([A→α.βπ], S) ∩
                (∪ {FIRST1(Ψ) | [C→φ.BΨ] ∈ S}\ {e}) = ∅);
           FOR [C→φ.BΨ] ∈ S DO
             IF e ∈ FIRST1(Ψ) THEN
                ASSUME I' = [A→α.βπ], J' = [C→φ.BΨ];
                IF  (I',J',S) ∈ FIN THEN
                    RES(I,J,T) := RES(I,J,T) ∪ RES(I',J',S)
                ELSEIF (I',J',S) ∈ STACK THEN
                    RES(I,J,T) := RES(I,J,T) ∪ {♯(I',J',S)}
                ELSE RES(I,J,T) := RES(I,J,T) ∪ LRCOND(I',J',S)
                ENDIF;
             ENDIF;
           ENDFOR;
        ENDFOR;
```

```
      ASSUME RES°(I,J,T) = RES(I,J,T) ∩ {TRUE, FALSE},
             RES#(I,J,T) = RES(I,J,T) ∩ D#;
      IF RES#(I,J,T) = ∅ THEN LRCOND := RES°(I,J,T)
      ELSE
         IF RES#(I,J,T) = {#(I,J,T)} THEN
            LRCOND := RES°(I,J,T);
            RES(I,J,T) := RES°(I,J,T);
         ELSE
            #(I",J",T") := MIN#(RES#(I,J,T));
            LRCOND := RES°(I,J,T) ∪ {#(I",J",T")};
            RES(I,J,T) := {#(I",J",T")};
         ENDIF;
         ASSUME FIN(I,J,T)={(I",J",T")∈FIN
                              | RES(I",J",T")={#(I,J,T)}};
         FOR (I",J",T") ∈ FIN(I,J,T) DO
            RES(I",J",T") := RES(I,J,T);
         ENDFOR;
      ENDIF;
      STACK := STACK ∖ {(I,J,T)};
      FIN := FIN ∪ {(I,J,T)};
   END LRCOND;
BEGIN
      IF(I,J,T) ∈ FIN THEN LRCOND-1 := RES(I,J,T)
      ELSE
         STACK := ∅;
         ASSUME I = [A→α.π], J = [B→β.];
         LRCOND-1 := ((EFF1(π) ∖ {e}) ∩ LALR1(J,T) = ∅)
                        ∧ LRCOND(I,J,T);
      ENDIF;
END LRCOND-1;
```

□

4 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. [1972, 1973]
"The Theory of Parsing, Translation and Compiling"
Vol. I & II, Prentice-Hall, Englewood Cliffs, N.J., 1973.

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

Eve, J. and Kurki-Suonio, R. [1977]
"On Computing the Transitive Closure of a Relation"
ACTA Informatica 8, 303-314 (1977).

Kristensen, B.B. and Madsen, O.L. [1979a]
"Methods for Computing LALR(k)-lookahead"
Computer Science Department, Århus 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, Århus University, 1979.
DAIMI PB-106.

Tarjan, R.E. [1972]
"Depth First Search and Linear Graph Algorithms"
SIAM J. Computing, 1 : 2, 146-160 (1972).