ON EXTENDED CONTEXT FREE GRAMMARS AND LR-PARSING

by

Ole Lehrmann Madsen

Bent Bruun Kristensen

DAIMI PB-53

September 1975

ABSTRACT

To facilitate the specification of grammars, it is common to extend the Backus Naur Form (BNF) with the repetition and the alternation operator. These so-called extended context free grammars (ECFGs) are normal context free grammars (CFGs), where the right hand sides are regular expressions over the terminal and nonterminal symbols. It is described how ECFGs can be treated in connection with LR- parsing, partly by extending the LR- theory to cover the ECFGs - and partly by giving transformations from ECFGs to normal CFGs.

CONTENTS.

1. INTRODUCTION.

An extended context free grammar (ECFG) is a context free grammar (CFG) where the right hand sides of the production rules are regular expressions over the terminal and nonterminal symbols of the grammar. The topic treated here is how to parse strings generated by such grammars deterministically, from left to right, using a k-lookahead. The first step is to extend the LR(k) (Knuth [1965]) and the SLR(k) (DeRemer [1969],[1971]) definitions for normal CFGs to the ECFGs.

One solution is then to extend the LR- theory to cover the ECFGs, i.e. to specify both an LR- table construction algorithm and a corresponding parsing algorithm.

Another solution is to apply some obvious transformations to the ECFG to construct an equivalent CFG. Such transformations may introduce LR- problems. However if the language generated by the ECFG is deterministic then there exists an equivalent CFG, which is LR(k). Hence there exists also a corresponding transformation. A treatment of ECFGs at such a level is not of interest here. Instead an approach using rule to rule transformations is given, which keeps the phrase structure of the grammar. The main motivation is to use ECFGs as a practical tool, i.e. to use ECFGs as input to translator writing systems based on LR(k) grammars. By that reason the original structure of an ECFG has to be preserved.

In section 2 a proper notation is stated. Furthermore some preliminary definitions are given.

Section 3 contains general motivations for ECFGs and the problems with the obvious transformations are discussed.

A definition of LR(k)'ness in the ECFG case is given in section 4. The existing results concerning extensions of the LR- theory are summarized and are shown to be insufficient. Another approach is then given and the LR- theory is extended on the basis of this approach. Finally the SLR(k) concept is also extended to ECFGs.

In section 5 the transformations from ECFGs to normal CFGs are introduced. These transformations and their inverse transformations are then shown to preserve the LR(k) and the SLR(k) properties.

Finally section 6 contains concluding remarks about the results in the paper and some related topics are mentioned.

2. NOTATION.

The reader is assumed to be familiar with LR(k) grammars as they are treated in Aho & Ullman [1972] and [1973]. The following notions are used: LR-items,LR-tables,the GOTO function,the FIRST function, the e-free first function(EFF),the FOLLOW function,viable prefix, consistent set of items.

When specifying grammars we use capital Latin letters as nonterminal symbols, and lower case Latin letters near the beginning of the alphabet as terminal symbols. Lower case Latin letters near the end of the alphabet indicate strings of terminal symbols. The lower case Latin letter e denotes the empty string. Lower case Greek letters denoted by a(alpha), b(beta),g(gamma),d(delta),etc. are used for strings of terminal and nonterminal symbols. => , =>r , =>* , =>r* denote direct derivation ,direct rightmost derivation,and their reflexive and transitive closure, respectively.

We let $T^*\underline{k}$ denote the set containing all strings over T with length less than or equal to k, including e.

U,I,-,c, notin and x are used as set operators, denoting union, intersection, difference, membership, not membership and Cartesian product.

Underlined numbers and letters are also used as subscripts.

We now give a formal definition of an extended context free grammar.

Definition 2.1

An extended context free grammar (ECFG) is a 4-tuple G=(N,T,P,S) where

(1) N is a finite set of nonterminal symbols.

(2) T is a finite set of terminal symbols, disjoint from N.

(3) P is a (not necessarily finite) subset of N X (N U T)*. An element (A,a) in P will be written A->a and called a production. For all A c N, the set [o / A->o is in P] is a regular set.

(4) S is a distinguished symbol in N called the start symbol.
///

An obvious way to express an ECFG is to permit the right hand side of a production to be a regular expression over (N U T).

Definition 2.2

Let < , > , * , / be symbols not in (N U T). The set of regular expressions over (N U T) , denoted R(N U T) is defined inductively as follows:

(1) If a c (N U T), then a c R(N U T)

(2) If a,b c R(N U T), the ab c R(N U T)

(3) If a c R(N U T), then <a>* c R(N U T)

(4) If o1,o2,...,on c R(N U T),

then <o1/o2/.../on> c R(N U T)

(5) e c R(N U T)

(6) {} c R(N U T)

///

<a>* means that a can be repeated zero or more times. <o1/o2/.../on> means precisely one of o1,o2,...,on. e denotes the regular set {e} and {} denotes the empty set.

Notice, that we require explicit use of parentheses in regular expressions, since this will ease the treatment in the following.

From now on any ECFG is expressed by means of regular expressions.

3. MOTIVATION.

The most important reason for using the operators * and / to express grammars is the clearness and readability of the grammar. The normal way of expressing a CFG is to use BNF or the reversed BNF. The weakness of this method appears as soon as the language generated by the grammar becomes nontrivial. The number of productions and nonterminals then grows very fast and the structure of the language disappears slowly. Many of these productions and nonterminals come from simple repetitions and alternations and they are not part of the global structure of the grammar. To avoid such productions and nonterminals the use of the operators * and / is excellent.

Example 3.1

The following part of a grammar expressed in reversed BNF is taken from a PASCAL like language. The grammar generates the case- statement of the language.

```
STATEMENT = "case" EXPRESSION "of" STATEMENTLIST "end"
STATEMENTLIST = LABEL-STATEMENT
          / STATEMENTLIST ";" LABEL-STATEMENT
LABEL-STATEMENT = CASE-LABEL-LIST ":"
          / CASE-LABEL-LIST ":" STATEMENT
CASE-LABEL-LIST = CASE-LABEL
          / CASE-LABEL-LIST "," CASE-LABEL
CASE-LABEL = "name" / "konst"
```

In the following grammar the operators * and / have been introduced to eliminate the disturbing productions and nonterminals. The generated language is unchanged.

```
STATEMENT = "case" EXPRESSION "of" LABEL-STATEMENT
          <";" LABEL-STATEMENT>* "end"
LABEL-STATEMENT = CASE-LABEL <"," CASE-LABEL>* ":"
          <e/STATEMENT>
CASE-LABEL = <"name"/"konst">
```

///

Assuming that ECFGs are useful in practice, one has to solve the parsing problem. For a big class of normal CFGs this problem has already been solved satisfactorily by means of the invention of the LR(k),LALR(k) and SLR(k) grammars.

One way of solving the parsing problem for ECFGs is to rewrite the ECFG to an equivalent CFG, which can be parsed . In practice this rewriting to a CFG and parsing this CFG should take place secretly and all applications should always be in terms of the initial ECFG.

Two typical examples of productions involving *'s and /'s are

```
  A -> a <b>*g
  A -> a<o1/o2/.../on>b
```

The obvious rewritings of these productions appear to be

```
    A  -> aBg                  A  -> aBg
    B  -> Bb        or         B  -> bB
    B  -> e                    B  -> e
```

(left-and right-recursive, respectively) and

```
    A  -> aWb
    W  -> o1
    W  -> o2
       .
       .
       .
    W  -> on
```

However these rewriting rules are not satisfactory because some unnecessary LR(k) problems may be introduced.

Example 3.2

Consider the ECFG with the productions

```
    S  -> E
    E  -> a<b>*bd
    E  -> a<b>*bc
```

Applying the proposed rewriting rules, the result is the grammars (CFGs) with the productions

```
    S  -> E                    S  -> E
    E  -> aD1bd                E  -> aD1bd
    E  -> aD2bc                E  -> aD2bc
    D1  -> D1b      or         D1  -> bD1
    D1  -> e                   D1  -> e
    D2  -> D2b                 D2  -> bD2
    D2  -> e                   D2  -> e
```

Neither of these grammars are LR(1) or SLR(1). Two more intuitive, equivalent rewritings appear in turn both to be LR(1) and SLR(1). Namely the grammars with the productions

```
S -> E                      S -> E
E -> aBd                    E -> aBd
E -> aBc        or          E -> aBc
B -> Bb                     B -> bB
B -> e                      B -> e
```

///

Example 3.3

Consider the ECFG with the productions

```
S -> E
E -> a<b/c>df
E -> a<b/c>dg
```

Applying the trivial rewriting rules, the result is a grammar (CFG) with the productions

```
A -> E
E -> aW1df
E -> aW2dg
W1 -> b
W1 -> c
W2 -> b
W2 -> c
```

This grammar appears to be neither LR(1) nor SLR(1). A more intuitive , equivalent rewriting appears in turn to be both LR(1) and SLR(1). Namely the grammar with the productions

```
S -> E
E -> aBdf
E -> aBdg
B -> b
B -> c
```

///

4. EXTENDED LR(k) GRAMMARS.

4.1. LR(k) definition.

Our definition of an LR(k) grammar is the same as in Aho & Ullman [1972].

Definition 4.1

Let $G=(N,T,P,S)$ be a CFG and let $G'=(N',T,P',S')$ be its augmented grammar. We say that G is LR(k), $k \geq 0$, if the three conditions

(1) $S' \Rightarrow_r^* \underline{a}Aw \Rightarrow_r \underline{ab}w$

(2) $S' \Rightarrow_r^* \underline{g}Bx \Rightarrow_r \underline{ab}y$

(3) $FIRST\underline{k}(w)=FIRST\underline{k}(y)$

imply that $\underline{a}Ay=\underline{g}Bx$. (That is $\underline{a}=\underline{g}$, $A=B$ and $x=y$)

///

Using this definition of an LR(k) grammar in the CFG case we want to give a corresponding definition of LR(k)'ness in the ECFG case. For this purpose we have to specify what we mean by a derivation using the productions of an ECFG. A production A -> $\underline{a}$ defines a set of productions and each production may be derived from A -> $\underline{a}$ in more than one way, depending on the actual number of repetitions used in the involved *-clause and the actual selection in the /-clause.

Example 4.1

Consider the grammar:

A -> a<b>*<b>*c

The set of productions in the grammar :

A -> ac
A -> abc
A -> abbc
.
.
.
etc.

The right hand side in the rule A -> abc can be derived from a<b>*<b>*c by taking one b from the first *-clause and zero from the last or by taking zero from the first and one from the last, i.e. abc = a<b>$\underline{1}$ <b>$\underline{0}$ c = a<b>$\underline{0}$ <b>$\underline{1}$ c. In general we have a <b>$\underline{m}$ c = a <b>$\underline{i}$ <b>$\underline{j}$ c , $i+j=m$, $i,j \geq 0$.

///

Definition 4.2.

Let $\underline{a}$ be a regular expression over (N U T). Let < ,>$\underline{i}$ for i≥0 be symbols not in (N U T). The set of $\underline{\text{instances}}$ derived from $\underline{a}$, denoted I($\underline{a}$), is defined inductively as follows:

(1) If $\underline{a}$ c (N U T), then I($\underline{a}$)=[$\underline{a}$]

(2) If $\underline{a}$=$\underline{db}$ & $\underline{d}$,$\underline{b}$ c R(N U T), then
I($\underline{db}$)=[xy / x is in I($\underline{d}$) & y is in I($\underline{b}$)]

(3) If $\underline{a}$=<$\underline{b}$>* & $\underline{b}$ c R(N U T) , then
I(<$\underline{b}$>*)=[<$\underline{b1b2}$...$\underline{bi}$>$\underline{i}$ / $\underline{bj}$ c I($\underline{b}$),j=1,2,...,i & i≥0]

(4) If $\underline{a}$=<$\underline{o1}$/$\underline{o2}$/.../$\underline{on}$> & $\underline{oi}$ c R(N U T),i=1,2...n, then
I(<$\underline{o1}$/$\underline{o2}$/.../$\underline{on}$>)=[<$\underline{oi}$'>$\underline{i}$ / $\underline{oi}$' c I($\underline{oi}$),i c [1,n]]

///

An instance $\underline{a}$' derived from $\underline{a}$ defines exactly one string over (N U T) , namely the string obtained from $\underline{a}$' by removing all symbols which are not in (N U T).

In a corresponding way an instance of a production A -> $\underline{a}$ is a production A -> $\underline{a}$' in which $\underline{a}$' is an instance of $\underline{a}$.

Applying a production of an ECFG in a derivation is defined as applying an instance of the production. This will have the effect that the sentential forms of an ECFG may contain some of the special symbols used to denote instances. When we match sentential forms and other strings, all symbols but those in (N U T) are blind symbols.

It will always be pointed out in the text when it is necessary to distinguish between instances. Hopefully the reader will not be confused.

Two rightmost derivations of a sentential form in an ECFG are said to be identical if in each step the same instance of a production is applied in both derivations.

An ECFG is said to be ambiguous if for some string there exist two different rightmost derivations.

The reason for introducing instances is the following:

The LR-parser we construct for an ECFG will be able to announce when a reduction A -> $\underline{a}$ has to be applied . The next step will then be to detect which instance A -> $\underline{a}$' was actually used. At this step there may be more than one possibility. In our LR(k) definition we state that there must be exactly one.

Definition 4.3

Let G=(N,T,P,S) be an ECFG and let G'=(N',T,P',S') be its augmented grammar. For k$\geq$0, we say that G' is LR(k) if the three conditions

(1) S' =>r* $\underline{a}$Aw =>r $\underline{a}\underline{b}$w

(2) S' =>r* $\underline{g}$Bx =>r $\underline{g}\underline{p}$x = $\underline{a}\underline{b}$y (as strings over (N U T))

(3) FIRST$\underline{k}$(w) = FIRST$\underline{k}$(y)

imply that $\underline{a}$Ay = $\underline{g}$Bx (that is $\underline{a}$=$\underline{g}$, A=B and x=y) and that A -> $\underline{b}$,B -> $\underline{p}$ are the same instance of a production.

///

To explain the requirement that A -> $\underline{b}$ and B ->$\underline{p}$ must be the same instance of a production, consider the following example:

Example 4.2

Consider the ECFG with the productions :

S -> Ad

A -> <ab/a><b>*c

We construct derivations:

(1) S =>r $\underline{a}$Ad =>r $\underline{a}$<ab>$\underline{1}$<>$\underline{0}$cd = abcd

(2) S =>r $\underline{g}$Ad =>r $\underline{g}$<a>$\underline{2}$<b>$\underline{1}$cd = abcd

(3) FIRST$\underline{1}$(d)=FIRST$\underline{1}$(d)=d

where $\underline{a}$=$\underline{g}$=e , A=A , d=d , but
A -> <ab>$\underline{1}$<>$\underline{0}$c and A -> <a>$\underline{2}$<b>$\underline{1}$c are not the same instance of A -> <ab/a><b>*c, so the grammar is not LR(1).

///

4.2. Construction of the LR-tables for ECFG's.

One way of treating regular expressions in production rules is to extend the LR parser construction technique. This can be done by giving rules for moving the LR marker (dot) through the regular expressions for the purpose of computing the LR-items. The LR-items are constructed in the normal way using the following additional rules proposed by DeRemer [1974] and Early [1970] :

(1) Any item of the form

[A -> a .<b>*g,v]

is replaced by

[A -> a< .b>*g,v]

[A -> a<b>* .g,v]

(2) Any item of the form

[A -> a<b .>*g,v]

is replaced by

[A -> a<b>* .g,v]

[A -> a< .b>*g,v]

(3) Any item of the form

[A -> a .<o1/o2/.../on>b,v]

is replaced by

[A -> a<o1/o2/.../ .oi/.../on>b,v]

for i=1,2,...,n

(4) Any item of the form

[A -> a<o1/o2/.../oi ./.../on>b,v]

is replaced by

[A -> a<o1/o2/.../on> .b,v]

The idea behind these rules is to keep track of which parts of which productions are applicable at a given point in the input string, in a manner consistent with the LR construction technique and with the meanings of regular expressions.

Example 4.3

Consider the grammar with the productions:

```
S -> E

E -> T<<+ / ->T>*

T -> P<<* / />P>*

P -> a
```

The LR(O) items are:

```
0
  [S -> .E ,e]                        E->1
  [E -> .T<<+ / ->T>* ,e]             T->2
  [T -> .P<<* / />P>* ,e]             P->4
  [P -> .a ,e]                        a->6
-------------------------------------------
1
  [S -> E . ,e]
-------------------------------------------
2
  [E -> T<< .+ / ->T>* ,e]            +->3
  [E -> T<<+ /  .->T>* ,e]            -->3
  [E -> T<<+ / ->T>* . ,e]
-------------------------------------------
3
  [E -> T<<+ / -> .T>* ,e]            T->2
  [T -> .P<<* / />P>* ,e]             P->4
  [P -> .a ,e]                        a->6
-------------------------------------------
4
  [T -> P<< .* / />P>* ,e]            *->5
  [T -> P<<* / ./>P>* ,e]             /->5
  [T -> P<<* / />P>*  . ,e]
-------------------------------------------
5
  [T -> P<<* / /> .P>* ,e]            P->4
  [P -> .a ,e]                        a->6
-------------------------------------------
6
  [P -> a . ,e]
-------------------------------------------
```

///

From the LR-items the LR-tables can be constructed in the usual way , but unfortunately the LR-parsing algorithm is no longer valid. When parsing a string it can be announced when a reduction has to be applied, but it is no longer obvious how much to pop off the pushdown list, because the right hand side of a production has no fixed length. Another disadvantage is that even if the LR(k) items are consistent , the grammar may be ambiguous and the parsing algorithm cannot uniquely determine the length of the right hand side of the production applied.

Example 4.4

Consider the grammar with the productions:

```
S ->A

A -> Aa<bAa>*c

A -> z
```

The LR(O) items are :

```
 0
   [S -> .A ,e]                  A->1
   [A -> .Aa<bAa>*c ,e]
   [A -> .z ,e]                  z->5
---------------------------------------
 1
   [S -> A . ,e]
   [A -> A .a<bAa>*c ,e]         a->2
---------------------------------------
 2
   [A -> Aa< .bAa>*c ,e]         b->3
   [A -> Aa<bAa>* .c ,e]         c->6
---------------------------------------
 3
   [A -> Aa<b .Aa>*c ,e]         A->4
   [A -> .Aa<bAa>*c ,e]
   [A -> .z ,e]                  z->5
---------------------------------------
 4
   [A -> Aa<bA .a>*c ,e]         a->2
   [A -> A .a<bAa>*c ,e]
---------------------------------------
 5
   [A -> z . ,e]
---------------------------------------
 6
   [A -> Aa<bAa>*c . ,e]
---------------------------------------
```

The LR(O) items are consistent. However two parse trees exist for the string: z a b z a c a c ,namely

```
(1)           S
              I
          ----A----
          I     I I
    ------A---- I I
    I I I I I I I I
    A I I A I I I I
    I I I I I I I I
    z a b z a c a c


(2)         S
            I
    --------A------
    I I I   I   I I
    I I I --A-- I I
    I I I I I I I I
    A I I A I I I I
    I I I I I I I I
    z a b z a c a c
```

///

We shall now modify the algorithm for constructing the LR-items, such that the LR-parsing algorithm can be extended to determine the length of the right hand sides unambiguously, if the grammar is LR(k).

In the previous algorithm items of the form [A -> a .<b>*g,v] and [A -> a<b .>*g,v] are transformed into the same two items ([A -> a< .b>*g,v] , [A -> a<b>* .g,v]). This is where we loose information about whether the LR-table associated with the items is associated with a viable prefix da ([A -> a .<b>*g,v]) or with a viable prefix dab ([A -> a<b .>*g,v]). In the same way the n items of the form [A -> a<o1/o2/.../oi ./.../on>b,v] are transformed into one item [A -> a<o1/o2/.../on> .b,v] and we loose information about which oi actually accesses the LR-table. The modified algorithm keeps this information in the table.

Assume now that the production A -> a is applied. Let a' be the instance of a actually read. /a'/ symbols on top of the pushdown list have to be popped . Now a can be split into d1,d2,...dm for m$\geq$1 such that : a = d1d2...dm and each di has the form:

(1) di c (N U T)* or

(2) di = <b>* for some b or

(3) di = <o1/o2/.../on> for some oj , j=1,2...,n.

It then follows that a' = d1'd2'...dm' where di' is an instance of di. The popping of the pushdown list can be split into popping first /dm'/ symbols ,then /dm-1'/ symbols etc., and finally /d1'/ symbols. Let us investigate this process. Assume that /dm'/, /dm-1'/, ...,/di+1'/ symbols have been popped from the pushdown list, and that we are going to reduce an instance of di.

(a) If di has the form (1) , the only possible instance of di is di itself and we can pop /di/ symbols from the pushdown list.

(b) If di has the form (2) then di' is zero or more possibly different instances of b. The process of popping /di'/ symbols then consists of repeatedly popping /b1/,/b2/,...,/bl/ symbols (the sequence may be empty) with bi,i=1,2,...,l being instances of b. If the item [A -> d1...di-1<b .>*di+1...dm,v] is in the LR-table on top of the pushdown list and v matches the next k input symbols ,we can pop an instance of b. If the item [A -> d1...di-1 .<b>*di+1...dm,v] is in the topmost LR-table we can stop popping instances of b and we have recognized an instance of di. If only one of the items is in the topmost LR-table we have only one thing to do, but if both items are present we can continue popping instances of b or we can stop. In general we don't know what to do. If we shall pop an instance of b we must split b as we did with a and recursively perform the same process with b.

(c) If di has the form (3) then di' is an instance of either o1,o2,...,or on. If the topmost LR-table contains the item [A -> d1...di-1<o1/.../ol ./.../on>di+1...dm,v] and v matches the next k input symbols ,then an instance of ol is valid. If the LR-table also contains [A -> d1...di-1<o1/.../oj ./.../on>di+1...dm,v] with j≠l, then we in general don't know what to do. To pop an instance of ol consists of splitting ol as we did with a and recursively repeat the process.

We shall now formalize the rules for constructing LR-parsers for extended context free grammars. Doing so we have to generalize the LR(k) theory as described in Aho & Ullman [1972]. The generalization will not always be proved formally, as most of the proofs are straightforward. Only the parts which have to do with reducing right hand sides of applied productions are proved.

In the CFG case we know that an item [A -> $\underline{a}$.$\underline{b}$,v] is associated with a viable prefix $\underline{ga}$, i.e.

S =>r* $\underline{g}$Aw =>r $\underline{gab}$w

and FIRST$\underline{k}$(w) =v

In the ECFG case a similar statement holds, but because we are dealing with regular expressions, the dot in the $\underline{a}$.$\underline{b}$ may be inside some nesting in the regular expression. The LR-item [A -> $\underline{a}$.$\underline{b}$,v] is then associated with a viable prefix $\underline{ga}$' where $\underline{a}$' is obtained from $\underline{a}$. From $\underline{b}$ we get $\underline{b}$' such that

S =>r* $\underline{g}$Aw =>r $\underline{ga}$'$\underline{b}$'w

and FIRST$\underline{k}$(w)=v

and $\underline{a}$'$\underline{b}$' c I($\underline{ab}$)

We need to state formally what $\underline{a}$' and $\underline{b}$' can be.

Definition 4.4

If $\underline{ab}$ is a regular expression over (N U T), then the set of <u>head instances</u> obtained from $\underline{a}$.$\underline{b}$, denoted HI($\underline{a}$.$\underline{b}$) is defined inductively as follows

(1) If $\underline{a}$,$\underline{b}$ c R(N U T), then HI($\underline{a}$.$\underline{b}$)=I($\underline{a}$)

(2) If $\underline{a}$.$\underline{b}$=$\underline{d}$<$\underline{b1}$.$\underline{b2}$>*$\underline{g}$ & $\underline{b1}$,$\underline{b2}$ c R(N U T),then
HI($\underline{a}$.$\underline{b}$)=[$\underline{x}$<$\underline{h1h2}$...$\underline{hi1}$ / $\underline{x}$ c HI($\underline{d}$.<$\underline{b1b2}$>*$\underline{g}$),
$\underline{1}$ c I($\underline{b1}$) , $\underline{hj}$ c I($\underline{b1b2}$) , j=1,2,...,i]

(3) If $\underline{a}$.$\underline{b}$=$\underline{d}$<$\underline{o1}$/.../$\underline{oi1}$.$\underline{oi2}$/.../$\underline{on}$>$\underline{g}$
& $\underline{o1}$,...,$\underline{oi1}$,$\underline{oi2}$,...,$\underline{on}$ c R(N U T) , then
HI($\underline{a}$.$\underline{b}$)=[$\underline{x}$<$\underline{1}$ / $\underline{1}$ c I($\underline{oi1}$) & $\underline{x}$ c HI($\underline{d}$.<$\underline{o1}$/$\underline{o2}$/.../$\underline{on}$>$\underline{g}$)]
///

Definition 4.5

If $\underline{ab}$ is a regular expression over (N U T), then the set of tail instances obtained from $\underline{a}.\underline{b}$, denoted TI($\underline{a}.\underline{b}$) is defined as follows

TI($\underline{a}.\underline{b}$)=[$\underline{l}$ / there is an $\underline{x}$ c HI($\underline{a}.\underline{b}$) : $\underline{xl}$ c I($\underline{ab}$)]

///

Definition 4.6

If $\underline{ab}$ is a regular expression over (N U T), then the FIRST function is extended to $\underline{a}.\underline{b}$ as follows

FIRST$\underline{k}$($\underline{a}.\underline{b}$)=[FIRST$\underline{k}$($\underline{x}$) / $\underline{x}$ c TI($\underline{a}.\underline{b}$)]

///

Formally we should extend FIRST to instances, but according to previous remarks on instances , we consider symbols not in (N U T) to be blind. The e-free-first function (EFF) is extended in a similar way.

Definition 4.7

Let G=(N,T,P,S) be an ECFG, and let A -> $\underline{ab}$ be a production in P. An LR(k) item [A -> $\underline{a}$.$\underline{b}$,u] is valid for $\underline{pa}$', a viable prefix of G, if there is a derivation

S =>r* $\underline{p}$Aw =>r $\underline{pa}$'$\underline{b}$'w such that

u=FIRST$\underline{k}$(w) where

$\underline{a}$' c HI($\underline{a}.\underline{b}$), $\underline{b}$' c TI($\underline{a}.\underline{b}$), $\underline{a}$'$\underline{b}$' c I($\underline{ab}$)

///

Algorithm 4.1

Collection of sets of valid extended LR(k) items for G.

Input ECFG G=(N,T,P,S) and an integer k≥0.

Output $\underline{Sk}$, the collection of sets of extended LR(k) items valid for any viable prefix of G.

Method Use algorithms 5.8 and 5.9 from Aho & Ullman [1972]. To move the dot beyond the symbols <,> and /, use the following additional rules

(1) Any item of the form

```
            [A -> a .<b>*g,v]

        is replaced by

            [A -> a #<b>*g,v]

            [A -> a< .b>*g,v]

            [A -> a<b>* .g,v]

    (2) Any item of the form

            [A -> a<b .>*g,v]

        is replaced by

            [A -> a<b #>*g,v]

            [A -> a<b>* .g,v]

            [A -> a< .b>*g,v]

    (3) Any item of the form

            [A -> a .<o1/o2/.../on>b,v]

        is replaced by

            [A -> a<o1/o2/.../ .oi/.../on>b,v]

            for i=1,2...,n

    (4) Any item of the form

            [A -> a<o1/o2/.../oi ./.../on>b,v]

        is replaced by

            [A -> a<o1/o2/.../oi #/.../on>b,v]

            [A -> a<o1/o2/.../oi/.../on> .b,v]
```

An item containing the symbol # is not evaluated further (# is a special "dead" dot)

///

We claim that the above algorithm correctly constructs the set of LR(k) items for any viable prefix of an ECFG G.

Lemma 4.1

Let G be an ECFG and Sk the canonical collection of sets of extended LR(k) items for G. If [A -> a .b,v] and [B -> g .d,u] are in the same set of items in Sk, then there exist a' c HI(a.b), g' c HI(g.d) such that for all

b' c TI(a.b), d' c TI(g.d) with

a'b' c I(ab) ,g'd' c I(gd) and some p,t,x,w we have

(1) S =>r* pAw =>r pa'b'w

(2) S =>r* tBx =>r tg'd'x

(3) FIRSTk(w)=v & FIRSTk(x)=u

(4) pa' = tg' (as strings over (N U T))

///

Theorem 4.1

Let G be an ECFG and let Sk be the collection of sets of items constructed by algorithm 4.1. G is LR(k) if and only if

(1) If [A -> d .,v] and [B -> p .t,u] are in the same set of items in Sk then v is not in EFFk(p.tu).

(2) If [A -> a #<b>*g,v] and [A -> a<b #>*g,u] are in the same set of items in Sk then v≠u.

(3) If [A -> a<o1/o2/.../oi #/...on>b,v] and [A -> a<o1/o2/.../oj #/.../on>b,u] with i≠j are in the same set of items in Sk then v≠u.

Proof:

Only if. Let G be LR(k)

Case 1: The proof of (1) is a restatement of Theorem 5.9 in Aho & Ullman [1972].

case 2: Assume [A -> a #<b>*g,v] and [A -> a<b #>*g,v] are in the same set of items. We have (lemma 4.1) that there exist a',a",b',b" such that

a' c HI(a .<b>*g), a" c HI(a<b .>*g),

b' c TI(a .<b>*g), b" c TI(a<b .>*g),

a'b' , a"b" c I(a<b>*g).

We can choose

b' = <>og', b" = >ig', i>1 for some g' c TI(a<b>* .g).

For some p,t,w,x, we have

(1) S =>r* pAw =>r pa'b'w

(2) S =>r* tAx =>r ta"b"x

(3) FIRSTk(w)=FIRSTk(x)=v

(4) pa' = ta" (as strings over (N U T))

This implies that pa'b' = ta"b" (as strings over (N U T))

The LR(k) definition then gives

pAx=tAx and A -> a'b', A -> a"b" are the same instance of A -> a<b>*q. However a'b' is an instance with zero repetitions of b and a"b" has at least one repetition of b, so a'b' and a"b" cannot be the same instance of a<b>*q.

case 3: Assume that [A -> a<o1/.../oi #/.../on>b,v] and [A -> a<o1/.../oj #/.../on>b,v] (i≠j) both are in the same set of items. We then have that there exist a',a",b',b" such that

a' c HI(a<o1/.../oi ./.../on>b),

a" c HI(a<o1/.../oj ./.../on>b),

b' c TI(a<o1/.../oi ./.../on>b),

b" c TI(a<o1/.../oj ./.../on>b),

a'b',a"b" c I(a<o1/o2/.../on>b),

We choose

b' = >id, b" = >jd, for some d c TI(a<o1/o2/.../on> .b).

For some p,t,w,x, we have

(1) S =>r* pAw =>r pa'b'w

(2) S =>r* tAx =>r ta"b"x

(3) FIRSTk(w)=FIRSTk(x)=v

(4) pa' = ta" (as strings over (N U T)).

This implies that pa'b' = ta"b" (as strings over (N U T)).

The LR(k) definition then gives

pAx=tAx and A -> a'b', A -> a"b" are the same instance of A -> a<o1/o2/.../on>b, but this is a contradiction.

If part.

Assume that (1),(2) and (3) are satisfied, and that G is not LR(k), i.e. we can construct derivations such that

(c1) S =>r* pAw =>r pa'w

(c2) S =>r* tBx =>r tb'x = pa'y

(c3) FIRSTk(w) = FIRSTk(y) = v

Case 1 : pAx ≠ tBy

This case is a restatement of Theorem 5.9 in Aho & Ullman [1972]

Case 2 : pAx = tBy, but A -> a' , B -> b' are not the same instance of a production.

Case 2a : A -> a' is an instance of A -> a and B -> b' is an instance of B -> b with A -> a and B -> b being different productions in G.

We then have that [A -> a .,v] and [B -> b .,v] both are valid items for the viable prefix pa' = tb' and this violates condition (1).

Case 2b : A -> a', B -> b' are different instances of the production A -> a. Let B -> b' have the form A -> a".

Let a = d1d2...dm,

let a'= d1'd2'...dm',

let a"= d1"d2"...dm",

where di c (N U T)*, or di = <b>* for some b,

or di = <o1/o2/.../on>,for some oj, j=1,2,...,n,

and di',di" c I(di) , i=1,2,...,m.

Let di',di" be the first two instances from right to left which are different. We then have

di+1'...dm' = di+1"...dm" (as strings and instances)

If di = <b>*, then

di'=<b1'...bl'>l

di"=<b1"...bj">j. Let l≥j.

We can assume that either j=0 or bl' ≠ bj" as instances of b.

(We can successively from right to left remove identical instances of b until j=0 or the rightmost instances are different, and then construct similiar derivations).

If bl' ≠ bj" (as instances), we recursively treat b as we did with a.

If di = <o1/o2/.../on>, then

di' = <ol'>l

di" = <oj">j

If l=j, then we recursively treat oj as we did with a.

At last we end up with

(d1) a = d<b>*q,

a'= d'<b1...bi>iq'

a"= d"<>0q"

with pd'<b1...bi = td" (as strings)

or

(d2) a = d<o1/o2/.../on>q

a'= d'<oi'>iq

a"= d"<oj">jq" , i≠j

with pd'<oi' = td"<oj" (as strings)

Case (d1) gives that [A -> d<b .>*q,v] and [A -> d .<b>*q,v] both are valid items for the viable prefix pd'<b1...bi = td", violating condition (2).

Case (d2) gives that [A -> d<o1/.../oi ./.../on>q,v] and [A -> d<o1/.../oj ./.../on>q,v] both are valid items for the viable prefix pd'<oi' = td"<oj" violating condition (3)

///

If the conditions (2) and (3) in theorem 4.1 are not satisfied, then it is in general not possible to determine how to reduce the stack.

Example 4.5

Consider the grammar with the productions

S -> A

A -> bB<v>*x

A -> B<v>*y

B -> <b>*

The grammar is clearly unambiguous. The two items [B -> #<b>* ,v] and [B -> <b #>* ,v] will be in the same set of items if we construct the LR(1) items, so the grammar is not LR(1). When performing the reduction B -> <b>* it is not possible by using k-lookahead to determine how to reduce the stack.

///

We now give the rules for constructing the LR-tables from a collection of sets of LR-items.

Definition 4.8

Let G=(N,T,P,S) be an ECFG and let SK be a collection of sets of LR(k) items for G. T(A), the LR(k) table associated with the set of items A in SK, is a triple <f,g,r> ,f is called the parsing action function, g the goto function and r the reduce function.

(1),(2) f,g are constructed as in Aho & Ullman [1972] p. 392

(3) r maps T*k into sets of so-called reduce items. A reduce item has the form [A -> a #b] where A -> ab is a production in G. If [A -> a #b,u] is in A, then [A -> a #b] is in r(u).
///

Example 4.6

Consider the ECFG with the productions

```
0 S -> T
1 T -> Aa
2 T -> Bb
3 A -> a<b<c>*>*d
4 B -> a<b/c>d
```

The LR(1) items are (items marked with - , are the items which have been replaced by other items)

```
0
   [S -> .T ,e]                    T -> 1
   [T -> .Aa ,e]                   A -> 2
   [T -> .Bb ,e]                   B -> 4
   [A -> .a<b<c>*>*d ,a]           a -> 6
   [B -> .a<b/c>d ,b]
----------------------------------------
1
   [S -> T . ,e]
----------------------------------------
2
   [T -> A .a ,e]                  a -> 3
----------------------------------------
3
   [T -> Aa .,e]
----------------------------------------
4
   [T -> B .b ,e]                  b -> 5
----------------------------------------
5
   [T -> Bb . ,e]
----------------------------------------
```

```
6
    [A -> a .<b<c>*>*d ,a]        -
    [B -> a .<b/c>d ,b]           -
    [A -> a #<b<c>*>*d ,a]
    [A -> a< .b<c>*>*d ,a]        b -> 7
    [A -> a<b<c>*>* .d ,a]        d -> 9
    [B -> a< .b/c>d ,b]
    [B -> a<b/ .c>d ,b]           c -> 12
---------------------------------------
7
    [A -> a<b .<c>*>*d ,a]        -
    [B -> a<b ./c>d ,b]           -
    [A -> a<b #<c>*>*d ,a]
    [A -> a<b< .c>*>*d ,a]        c -> 8
    [A -> a<b<c>* .>*d ,a]        -
    [A -> a<b<c>* #>*d ,a]
    [A -> a<b<c>*>* .d ,a]        d -> 11
    [A -> a< .b<c>*>*d ,a]        b -> 10
    [B -> a<b #/c>d ,b]
    [B -> a<b/c> .d ,b]
---------------------------------------
8
    [A -> a<b<c .>*>*d ,a]        -
    [A -> a<b<c #>*>*d ,a]
    [A -> a<b<c>* .>*d ,a]        -
    [A -> a<b< .c>*>*d ,a]        c -> 8
    [A -> a<b<c>* #>*d ,a]
    [A -> a<b<c>*>* .d ,a]        d -> 9
    [A -> a< .b<c>*>*d ,a]        b -> 10
---------------------------------------
9
    [A -> a<b<c>*>*d . ,a]
---------------------------------------
10
    [A -> a<b .<c>*>*d ,a]        -
    [A -> a<b #<c>*>*d ,a]
    [A -> a<b< .c>*>*d .a]        c -> 8
    [A -> a<b<c>* .>*d ,a]        -
    [A -> a<b<c>* #>*d ,a]
    [A -> a<b<c>*>* .d ,a]        d -> 9
    [A -> a< .b<c>*>*d ,a]        b -> 10
---------------------------------------
11
    [A -> a<b<c>*>*d . ,a]
    [B -> a<b/c>d . ,b]
---------------------------------------
12
    [B -> a<b/c .>d ,b]           -
    [B -> a<b/c #>d ,b]
    [B -> a<b/c> .d ,b]           d ->13
---------------------------------------
13
    [B -> a<b/c>d . ,b]
---------------------------------------
```

The LR(1) tables associated with the above items are:

	f					g							r				
	a	b	c	d	e	T	A	B	a	b	c	d	a	b	c	d	e
t0	s	x	x	x	x	t1	t2	t4	t6	x	x	x	E	E	E	E	E
t1	x	x	x	x	a	x	x	x	x	x	x	x	E	E	E	E	E
t2	s	x	x	x	x	x	x	x	t3	x	x	x	E	E	E	E	E
t3	x	x	x	x	1	x	x	x	x	x	x	x	E	E	E	E	E
t4	x	s	x	x	x	x	x	x	x	t5	x	x	E	E	E	E	E
t5	x	x	x	x	2	x	x	x	x	x	x	x	E	E	E	E	E
t6	x	s	s	s	x	x	x	x	x	t7	t12	t9	r1	E	E	E	E
t7	x	s	s	s	x	x	x	x	x	t10	t8	t11	r2	r3	E	E	E
t8	x	s	s	s	x	x	x	x	x	t10	t8	t9	r4	E	E	E	E
t9	3	x	x	x	x	x	x	x	x	x	x	x	E	E	E	E	E
t10	x	s	s	s	x	x	x	x	x	t10	t8	t9	r2	E	E	E	E
t11	3	4	x	x	x	x	x	x	x	x	x	x	E	E	E	E	E
t12	x	x	x	s	x	x	x	x	x	x	x	t13	E	r5	E	E	E
t13	x	4	x	x	x	x	x	x	x	x	x	x	E	E	E	E	E

s=shift , i=<u>reduce</u> i , a=accept , x=error, E=the empty set

```
r1 = [ [A -> a #<b<c>*>*d] ]
r2 = [ [A -> a<b #<<c>*>*d] , [A -> a<b<c>* #>*d] ]
r3 = [ [B -> a<b #/c>d] ]
r4 = [ [A -> a<b<c #>*>*d] , [A -> a<b<c>* #>*d] ]
r5 = [ [B -> a<b/c #>d] ]
```

///

The parsing algorithm for the extended LR(k) grammars is a modification of the algorithm given in Aho & Ullman [1972] p.375.

Algorithm. Extended LR(k) parsing algorithm.

Input. A set $\underline{J}$ of LR(k) tables for an extended LR(k) grammar G=(N,T,P,S), with $\underline{T0}$ c $\underline{J}$ designated as the initial table, and an input string z c T*$\underline{k}$, which is to be parsed.

Output. If z c L(G), the right parse of z. Otherwise, an error indication.

Method. Perform step (1) and (2) until acceptance occurs or an error is encountered. If acceptance occurs, the string in the output buffer is the right parse of z.

(1) The lookahead string u, consisting of the next k input symbols, is determined.

(2) When using the triple <f,g,r> in the following, it is always on the table on top of the pushdown list. The parsing action function f is applied to the lookahead string u.

(a) If f(u)=shift, then the next input symbol, say a, is removed from the input. The goto function g is applied to a to determine the new table to be placed on top of the pushdown list. If g(a) is undefined, or there is no next input symbol, halt and declare error.

(b) If f(u)= error, we halt parsing (and, in practice, transfer to an error recovery routine).

(c) If f(u)=accept, we halt and declare the string in the output buffer to be the right parse of the original input string.

(d) If f(u)=reduce i and production i is A->$\underline{a}$, then place production number i in the output buffer and call

REDUCE([A -> $\underline{a}$ #],$\underline{a}$).

The goto function g of the top table of the pushdown list is applied to A to determine the next table to be put on the pushdown list.

REDUCE is the recursive procedure defined as follows:

```
PROCEDURE REDUCE(VALUE IT:REDUCEITEM; VALUE d:STRING);
BEGIN
 LET IT=[A ->ad #q]
 LET d=d1d2...dm
 WHERE di c (N U T)*
 OR    di = <b>* for some b
 OR    di = <o1/o2/.../on> for some oj ,j=1,2...n;
 FOR I:=m DOWNTO 1 DO
  IF di c (N U T)* THEN
    remove /di/ tables from the pushdown list
  ELSE
    IF di=<b>* THEN
      BEGIN
        WHILE [A ->ad1...di-1<b #>*di+1...dmq] c r(u) DO
          REDUCE([A->ad1...di-1<b #>*di+1...dmq],b);
        UNLESS [A->ad1...di-1 #<b>*di+1...dmq] c r(u)
        THEN ERROR
      END
     ELSE
      IF there exist an item (j c [1,n]) such that
      [A->ad1...di-1<o1/.../oj #/.../on>di+1...dmq] c r(u)
      THEN REDUCE(
        [A->ad1...di-1<o1/.../oj #/.../on>di+1...dmq],oj)
      ELSE ERROR
END;                                                       ///
```

Example 4.7

The parsing algorithm is applied to the initial configuration (t0, abbccda,e) using the LR(k) tables in example 4.6. To facilitate the reading of the example , we also put the input symbols on pushdown list. The following sequence of moves are made:

```
(t0,abbccda,e)              =>shift

(t0at6,bbccda,e)            =>shift

(t0at6bt7,bccda,e)          =>shift

(t0at6bt7bt7,ccda,e)        =>shift

(t0at6bt7bt7ct8,cda,e)      =>shift

(t0at6bt7bt7ct8ct8,da,e)    =>shift

(t0at6bt7bt7ct8ct8dt9,a,e)=>reduce 3 (A -> a<b<c>*>*d)

                                reduce an instance of a<b<c>*>*d

                                  reduce an instance of d

(t0at6bt7bt7ct8ct8,a,3)     =>   reduce an instance of <b<c>*>*

                                    reduce an instance of b<c>*

                                      reduce an instance of <c>*

                                        reduce an instance of c

(t0at6bt7bt7ct8,a,3)        =>         reduce an instance of c

(t0at6bt7bt7,a,3)           =>       reduce an instance of b

(t0at6bt7,a,3)              =>     reduce an instance of b<c>*

                                      reduce an instance of <c>*

                                      reduce an instance of b

(t0at6,a,3)                 =>   reduce an instance of a

(t0,a,3)                    =>g(A)

(t0At2,a,3)                 =>shift

(t0At2at3,e,3)              =>reduce 1 (T -> Aa)

(t0St1,e,31)                accept
                                                              ///
```

4.3. Extended SLR(k) grammars.

Now we turn to the so-called simple LR(k) grammars. In the following section we shall need a definition of SLR(k)'ness in the ECFG case.

Definition 4.9

Let G=(N,T,P,S) be an ECFG. Let SO be the canonical collection of sets of extended LR(O) items for G. G is said to be a simple LR(k) grammar (SLR(k)) if the following conditions are satisfied, for any set of items A in SO.

(a) Whenever [A -> a .b,e] and [B -> g .d,e] are two distinct items in A, one of the following conditions is satisfied.

(1) Neither of b and d are e.

(2) b≠e , d=e and

FOLLOWk(B) I EFFk(<a .b>FOLLOWk(A)) = empty

(3) b=e , d≠e and

FOLLOWk(A) I EFFk(<d .g>FOLLOWk(B)) = empty

(4) b=d=e and

FOLLOWk(A) I FOLLOWk(B) = empty

(b) Two items of the form [A -> a #<b>*g,e] and [A -> a<b #>*g,e] are not both in A.

(c) Two items of the form [A -> a<o1/.../oi #/.../on>b,e] and [A -> a<o1/.../oj #/.../on>b,e] with i≠j are not both in A.

///

This definition is similar to the one given by Aho & Ullman [1972] and is the same for a normal CFG where condition (b) and (c) are superfluous. The conditions (b) and (c) assure that the right hand side of an applied production can be uniquely determined without using lookahead. Global lookahead on the left hand side will not help us. In case (b) it is not possible by means of global lookahead on A, to determine whether to continue or to stop reducing instances of b. Similarly in case (c) global lookahead on A cannot determine whether to reduce an instance of oi or oj.

5. TRANSFORMATION OF AN ECFG TO A CFG.

5.1. Introduction of the transformation rules.

In this section we show how to transform ECFGs into normal CFGs such that LR(k) and SLR(k) properties are preserved.

The transformations are introduced using two typical cases of productions which involve *'s and /'s , namely

p) $A \rightarrow \underline{a}<\underline{b}>^{*}\underline{g}$

q) $A \rightarrow \underline{a}<\underline{o1}/\underline{o2}/\ldots/\underline{on}>\underline{b}$

These productions are transformed into

p') $A \rightarrow \underline{a}D$

$D \rightarrow \underline{b}D$

$D \rightarrow \underline{g}$

(introducing the new nonterminal symbol D) and

q') $A \rightarrow \underline{a}W$

$W \rightarrow \underline{o1}D$

$W \rightarrow \underline{o2}D$

....

$W \rightarrow \underline{on}D$

$D \rightarrow \underline{b}$

(introducing the new nonterminal symbols W,D) ,respectively.

Intuitively the intention of production p is to recognize $\underline{a}$, an unknown number of $\underline{b}$'s and finally $\underline{g}$ and then reduce this string into the nonterminal symbol A. The right recursive production set p' prescribes this action sequence. Note that no reduction is performed before $\underline{g}$ has been recognized. An analogous remark holds for q.

Investigating the LR(O) machines (the GOTO graph in Aho & Ullman [1973]) of an ECFG and its transformed grammar the resemblance of the two machines is striking. Not surprisingly the number of states in the machine for the transformed grammar has increased, however the machine has kept the structure of the original machine.

To illustrate this fact consider again the two typical productions and the isolated parts of the machines related to these productions. All new states appear to be necessary in order to keep track of the reductions corresponding to the two possibilities

1) the number of repetitions of the string b

2) the selected string gi

*-case :

p) A -> a<b>*g

```
                +-----<---+ b
                I         I
+----+    a +----+ I b +----+ I
I 1 I---->I 2 I-+-->I 3 I-+
+----+    +----+ I   +----+ I
                I         I
                I-----<---+ g
                I
                I g +----+
                +-->I 4 I--> A->a<b>*g
                    +----+
```

p') A -> aD , D -> bD , D -> g

```
+----+    a +----+   D +----+
I 1 I---->I 2 I---->I D I--> A->aD
+----+    +----+ I   +----+
                I
                I-----<---+ b
                I         I
                I b +----+ I D +----+
                I-->I 3 I-+-->I D I--> D->bD
                I   +----+ I   +----+
                I         I
                I-----<---+ g
                I
                I g +----+
                +-->I 4 I--> D->g
                    +----+
```

/-case :

q) A -> a<o1/o2/.../on>b

```
+----+  a +----+ o1 +----+
I 1 I---->I 2 I---->I 3 I-+ b
+----+    +----+ I   +----+ I
                I         I
                I         I
                I         I
                Io2 +----+ I b +----+
                I-->I 3 I-+-->I 4 I--> A->a<o1/o2/.../on>b
                I   +----+ I   +----+
                I         I
                I         I
                I         I
                .         .
                .         .
                .         .
                I         I
                I         I
                I         I
                Ion +----+ I
                +-->I 3 I-+ b
                    +----+
```

q') A -> aW , W -> o1D , W -> o2D , ... , W -> onD , D -> b

```
+----+  a +----+  W +----+
I 1 I---->I 2 I---->I W I--> A->aW
+----+    +----+ I   +----+
                I
                I       b +----->-------------+
                I         I                   I
                Io1 +----+ I D +----+           I
                I-->I 3 I---->I D I--> W->o1D I
                I   +----+    +----+           I
                I                             I
                I       b +----->-------------I
                I         I                   I
                Io2 +----+ I D +----+           I b +----+
                I-->I 3 I---->I D I--> W->o2D I-->I 4 I--> D->b
                I   +----+    +----+           I   +----+
                I                             I
                I                             I
                .                             .
                .                             .
                .                             .
                I                             I
                I       b +----->-------------+
                I         I
                Ion +----+ I D +----+
                +-->I 3 I---->I D I--> W->onD
                    +----+    +----+
```

Example 5.1

Illustration of application of the transformation rules and illustration of the LR(O) machines before and after the transformations.

Consider the grammar (ECFG) with the production set,

```
S -> E

E -> T < + / - > T

T -> ( < P ↑ >* a )

P -> a
```

Applying the transformation rules on the grammar , the result is a grammar (CFG) with the production set,

```
S -> E

E -> T W

W -> + C

W -> - C

C -> T

T -> ( D

D -> P ↑ D

D -> a )

P -> a
```

The resemblance of the LR(O) machines for the grammars shown in fig. 5.1 is striking.

```
+---+   E +---+
I 0 I---->I 1 I--> S->E
+---+ I   +---+
      I
      I
      I
      I T +---+   + +---+             T +---+
      I-->I 2 I---->I 3 I------->------->I 4 I--> E->T<+/->T
      I   +---+ I   +---+ I         I   +---+
      I         I         I         I
      I-----<---U---------+ (       I
      I         I                   I
      I         I - +---+           I
      I         +-->I 3 I------->---+ T
      I             +---+ I
      I                   I
      I-----<-------------+ (
      I
      I         +----------<--------+ P
      I         I                   I
      I ( +---+ I P +---+   ↑ +---+ I
      +-->I 5 I-+-->I 6 I---->I 7 I-+
          +---+ I   +---+     +---+ I
                I                   I
                I----------<--------+ a
                I
                I a +---+
                +-->I 8 I----> P->a
                    +---+ I
                          I
                          I
                          I
                          I ) +---+
                          +-->I 9 I--> T->(<P↑>*a)
                              +---+
+---+   E +---+
I 0 I---->I 1 I--> S->E
+---+ I   +---+
      I
      I
      I
      I T +---+   W +---+
      I-->I 2 I---->I W I--> E->TW
      I   +---+ I   +---+
      I         I
      I         I
      I         I       T +----->-------------+
      I         I         I                   I
      I         I + +---+ I C +---+           I T +---+
      I         I-->I 3 I-+-->I C I--> W->+C  I-->I 4 I--> C->T
      I         I   +---+ I   +---+           I   +---+
      I         I         I                   I
      I-----<---U---------+ (                 I
      I         I                             I
      I         I       T +----->-------------+
      I         I         I
      I         I - +---+ I C +---+
      I         +-->I 3 I-+-->I C I--> W->-C
      I             +---+ I   +---+
      I                   I
      I-----<-------------+ (
      I
      I ( +---+   D +---+
      +-->I 5 I---->I D I--> T->(D
          +---+ I   +---+
                I
                I----------<--------+ P
                I                   I
                I P +---+   ↑ +---+ I D +---+
                I-->I 6 I---->I 7 I-+-->I D I--> D->P↑D
                I   +---+     +---+ I   +---+
                I                   I
                I----------<--------+ a
                I
                I a +---+
                +-->I 8 I----> P->a
                    +---+ I
                          I
                          I
                          I
                          I
                          I ) +---+
                          +-->I 9 I--> D->a)
                              +---+
```

Fig. 5.1 The LR(0) machines for the grammar and the transformed grammar.

///

In the following two sections we are going to prove that the proposed rewriting rules for productions of ECFGs preserve the SLR(k) and LR(k) properties. For this purpose let us specify the rewriting rules precisely.

Definition 5.1

Definition of the transformations T* and T/ .

*-case :

Given a production of the form

A -> $\underline{a}$<$\underline{b}$>*$\underline{g}$, where $\underline{a}$ c (N U T)*

define T* to

T* (A -> $\underline{a}$<$\underline{b}$>*$\underline{g}$) =

[A -> $\underline{a}$D , D -> $\underline{b}$D , D -> $\underline{g}$] , where D notin N

/-case :

Given a production of the form

A -> $\underline{a}$<$\underline{o1}$/$\underline{o2}$/.../$\underline{on}$>$\underline{b}$, where $\underline{a}$ c (N U T)*

define T/ to

T/ (A -> $\underline{a}$<$\underline{o1}$/$\underline{o2}$/.../$\underline{on}$>$\underline{b}$) =

[A -> $\underline{a}$W , W -> $\underline{o1}$D , W -> $\underline{o2}$D , ... ,

W -> $\underline{on}$D , D -> $\underline{b}$] , where W,D notin N

///

Starting with a production involving one or more applications of the operators * and/or / on its righthand side it is possible, by means of a number of repeated applications of the transformations T* and T/ in an appropriate order, to end up with a set of productions, free of all occurences of the operators * and / .

(Notice, the definitions of T* and T/ include the restriction, that the affected operator (* or /) is always found leftmost in order to make T* and T/ unambiguous).

Definition 5.2

Definition of the transformed grammars G* and G/ .

Applying either the T* or T/ transformation on a production from a grammar G (ECFG), a new grammar G* or G/ is produced which is also an ECFG. More precisely, let G = (N,T,P,S) be an ECFG, then

*-case :

Let p: A -> $\underline{a}$<$\underline{b}$>*$\underline{q}$ c P, where $\underline{a}$ c (N U T)*. Applying the transformation T* on p, we achieve a new grammar

G* = (N*,T,P*,S) , where

P* = P / [p] U T*(p) , (introducing D notin N)

N* = N U [D]

/-case :

Let q: A -> $\underline{a}$<$\underline{o1}$/$\underline{o2}$/.../$\underline{on}$>$\underline{b}$ c P, where $\underline{a}$ c (N U T)*. Applying the transformation T/ on q, we achieve a new grammar

G/ = (N/,T,P/,S) , where

P/ = P / [q] U T/(q) , (introducing W,D notin N)

N/ = N U [W,D]

///

5.2. The SLR(k) case.

In this section we are going to prove that, applying either the T* or the T/ transformation on a production of an ECFG which is SLR(k), we produce a new ECFG which has preserved the SLR(k) property.

Because the definition 4.9 of SLR(k) for an ECFG is based on the canonical collection of sets of LR(O) items for the grammar , the proof shall also be in terms of LR(O) items.

For the proof we shall need some lemmas. These are concerned with equalities of the LR(O) items before and after applying one of the transformations T* or T/ to the grammar.

First let us state a proper notation.

Notation.

Let G = (N,T,P,S) be an ECFG and let G*, G/ be as in definition 5.2 . Let $\underline{SO}$, $\underline{S*}$ and $\underline{S/}$ be the canonical sets of LR(O) items for G, G* and G/ ,respectively.

Furthermore define $\underline{M*}$, $\underline{M/}$ to be the following sets of sets of items,

$\underline{M*}$ = [[[A -> $\underline{a}$D. ,e]],

[[D -> $\underline{b}$D. ,e]],

[[A -> $\underline{a}$D. ,e] , [D -> $\underline{b}$D. ,e]]]

$\underline{M/}$ = [[[A -> $\underline{a}$W. ,e]]] U

[[[W -> $\underline{oi1}$D. ,e] , [W -> $\underline{oi2}$D. ,e] ,

...,[W -> $\underline{oik}$D. ,e] / k c [1,n], i$\underline{j}$ c [1,n]]

///

We are now going to define some relations between the items in $\underline{SO}$ and items in either $\underline{S*}$ or $\underline{S/}$.

Definition 5.3

*-case :

If X->$\underline{r}$ is a production in P then for any $\underline{r}$',$\underline{r}$", where $\underline{r}$ = $\underline{r}$'$\underline{r}$" and for the item [X -> $\underline{r}$'.$\underline{r}$" ,e] in a set in $\underline{S*}$ we define

⌊X -> r'.r" ,e⌋ = [X -> r'.r" ,e]

For any a',a" with a = a'a" and for the item [A -> a'.a"D ,e] in a set in S*, we define

⌊A -> a'.a"D ,e⌋ = [A -> a'.a"<b>*g ,e]

For any b',b" with b = b'b" and for the item [D -> b'.b"D ,e] in a set in S*, we define

⌊D -> b'.b"D ,e⌋ = [A -> a<b'.b">*g ,e]

For any g',g" with g = g'g" and for the item [D -> g'.g" ,e] in a set in S*, we define

⌊D -> g'.g" ,e⌋ = [A -> a<b>*g'.g" ,e]

We shall not define associated items for the items [A -> aD. ,e] and [D -> bD. ,e]

/-case :

If X->r is a production in P then for any r',r", where r = r'r" and for the item [X -> r'.r" ,e] in a set in S/ we define

⌊X -> r'.r" ,e⌋ = [X -> r'.r" ,e]

For any a',a" with a = a'a" and for the item [A -> a'.a"W ,e] in a set in S/, we define

⌊A -> a'.a"W ,e⌋ = [A -> a'.a"<o1/o2/.../on>b ,e]

For any oi',oi" with oi = oi'oi" and for the item [W -> oi'.oi"D ,e] in a set in S/, we define

⌊W -> oi'.oi"D ,e⌋ = [A -> a<o1/o2/.../oi'.oi"/.../on>b ,e]

For any b',b" with b = b'b" and for the item [D -> b'.b" ,e] in a set in S/, we define

⌊D -> b'.b" ,e⌋ = [A -> a<o1/o2/.../on>b'.b" ,e]

We shall not define associated items for the items [A -> aW. ,e] and [W -> oiD. ,e] ,for i c [1,n].

///

Notice, some of the following lemmas contain notational difficulties. The problems may exist when we try to move the dot in an LR(0) item backwards over the symbol, which accesses the set of items. In this case there may exist some disturbing parentheses from the regular expressions, due to the extended algorithm 4.1. However, when this difficulty arises, one gets over it by using the following rules of interpretation :
In case that an item is of the form [C -> k<.l>*m ,e] or [C -> k<l>*.m ,e] it can be interpreted as either [C -> k.<l>*m ,e] or

$[C \to \underline{k}\langle\underline{l}.\rangle{*}\underline{m}\ ,e]$, depending on whether $[C \to \underline{k}\#\langle\underline{l}\rangle{*}\underline{m}\ ,e]$ or $[C \to \underline{k}\langle\underline{l}\#\rangle{*}\underline{m}\ ,e]$ also exists in the same set of items.
Similarly, if an item is of the form $[C \to \underline{k}\langle\underline{l1}/\underline{l2}/\ldots/.li/\ldots/\underline{ln}\rangle\underline{m}\ ,e]$ it can be interpreted as $[C \to \underline{k}.\langle\underline{l1}/\underline{l2}/\ldots/\underline{ln}\rangle\underline{m}\ ,e]$ and finally - if an item is of the form $[C \to \underline{k}\langle\underline{l1}/\underline{l2}/\ldots/\underline{ln}\rangle.\underline{m}\ ,e]$ it can be interpreted as $[C \to \underline{k}\langle\underline{l1}/\underline{l2}/\ldots/\underline{li}./\ldots/\underline{ln}\rangle\underline{m}\ ,e]$ if the item $[C \to \underline{k}\langle\underline{l1}/\underline{l2}/\ldots/\underline{li}\#/\ldots/\underline{ln}\rangle\underline{m}\ ,e]$ also exists in the same set of items. This interpretation proceeds until none of the mentioned cases are applicable.

Lemma 5.1

Let x c (N U E). If $[C \to \underline{q}x.\underline{t}\ ,e]$ is an item in a set of items in $\underline{S*} - \underline{M*}$ (in the T*-case - or in $\underline{S/} - \underline{M/}$ in the T/-case) and $\underline{[}C \to \underline{q}x.\underline{t}\ ,e\underline{]} = [B \to \underline{p}.\underline{d}\ ,e]$ then

(1) $\underline{p} = \underline{p}'x$ for some $\underline{p}'$

(2) $\underline{[}C \to \underline{q}.x\underline{t}\ ,e\underline{]} = [B \to \underline{p}'.x\underline{d}\ ,e]$

Proof.

T*-case :

There are three nontrivial cases of the item $[C \to \underline{q}x.\underline{t}\ ,e]$, namely

Case 1 : $[A \to \underline{o}x.\underline{t}'D\ ,e]$ with $\underline{t} = \underline{t}'D$ and $\underline{a} = \underline{o}x\underline{t}'$. This implies that

$\underline{[}A \to \underline{o}x.\underline{t}'D\ ,e\underline{]} = [A \to \underline{o}x.\underline{t}'\langle\underline{b}\rangle{*}\underline{g}\ ,e]$
$\underline{[}A \to \underline{o}.x\underline{t}'D\ ,e\underline{]} = [A \to \underline{o}.x\underline{t}'\langle\underline{b}\rangle{*}\underline{g}\ ,e]$.

Case 2 : $[D \to \underline{o}x.\underline{t}'D\ ,e]$ with $\underline{t} = \underline{t}'D$ and $\underline{b} = \underline{o}x\underline{t}'$. This implies that

$\underline{[}D \to \underline{o}x.\underline{t}'D\ ,e\underline{]} = [A \to \underline{x}\langle\underline{o}x.\underline{t}'\rangle{*}\underline{g}\ ,e]$
$\underline{[}D \to \underline{o}.x\underline{t}'D\ ,e\underline{]} = [A \to \underline{x}\langle\underline{o}.x\underline{t}'\rangle{*}\underline{g}\ ,e]$

Case 3 : $[D \to \underline{o}x.\underline{t}\ ,e]$ with $\underline{q} = \underline{o}x\underline{t}$. This implies that

$\underline{[}D \to \underline{o}x.\underline{t}\ ,e\underline{]} = [A \to \underline{a}\langle\underline{b}\rangle{*}\underline{o}x.\underline{t}\ ,e]$
$\underline{[}D \to \underline{o}.x\underline{t}\ ,e\underline{]} = [A \to \underline{a}\langle\underline{b}\rangle{*}\underline{o}.x\underline{t}\ ,e]$.

T/-case :

There are three nontrivial cases of the item $[C \to \underline{q}x.\underline{t}\ ,e]$, namely

Case 1 : $[A \to \underline{q}x.\underline{t}'W\ ,e]$ with $\underline{t} = \underline{t}'W$ and $\underline{a} = \underline{q}x\underline{t}'$. this implies that

```
    [A -> gx.t'W ,e] = [A -> gx.t'<o1/o2/.../on>b ,e]
    [A -> g.xt'W ,e] = [A -> g.xt'<o1/o2/.../on>b ,e]
```

Case 2 : [W -> gx.t'D ,e] with t = t'D and oi = gxt' , (fixed i c [1,n]). This implies that

```
    [W -> gx.t'D ,e] = [A -> a<o1/o2/.../gx.t'/.../on>b ,e]
    [W -> g.xt'D ,e] = [A -> a<o1/o2/.../g.xt'/.../on>b ,e] .
```

Case 3 : [D -> gx.t ,e] ,with b = gxt . This implies that

```
    [D -> gx.t ,e] = [A -> a<o1/o2/.../on>gx.t ,e]
    [D -> g.xt ,e] = [A -> a<o1/o2/.../on>g.xt ,e] .
```

///

Lemma 5.2

Let Y,V c N* (or N/) and Y =>* Vu for some u c E*. If the items [X -> r.Ys ,e] and [V -> .d ,e] are in the same set of items in S* - M* (in the T*-case - or in S/ - M/ in the T/-case), then the items [X -> r.Ys ,e] and [V -> .d ,e] are in the same set of items in SO.

Proof.

T*-case :

There are four nontrivial cases of the items [X -> r.Ys ,e] and [V -> .d ,e] in a set in S* - M*, namely

Case 1 : [A -> a.D ,e], [D -> .bD ,e], then

```
    [A -> a.D ,e] = [A -> a.<b>*g ,e] c S in SO, implying that
    [A -> a<.b>*g ,e] = [D -> .bD ,e] is in S.
```

Case 2 : [A -> a.D ,e], [D -> .g ,e], then

```
    [A -> a.D ,e] = [A -> a.<b>*g ,e] c S in SO, implying that
    [A -> a<b>*.g ,e] = [D -> .g ,e] is in S.
```

Case 3 : [D -> b.D ,e], [D -> .bD ,e], then

```
    [D -> b.D ,e] = [A -> a<b.>*g ,e] c S in SO, implying that
    [A -> a<.b>*g ,e] = [D -> .bD ,e] is in S.
```

Case 4 : [D -> b.D ,e], [D -> .g ,e], then

```
    [D -> b.D ,e] = [A -> a<b.>*g ,e] c S in SO, implying that
    [A -> a<b>*.g ,e] = [D -> .g ,e] is in S.
```

T/-case :

There are two nontrivial cases of the items [X -> r.Ys ,e] and [V -> .d ,e] in a set in S/ - M/, namely

Case 1 : [A -> a.W ,e], [W -> .oiD ,e], (fixed i c [1,n]), then

[A -> a.W ,e] = [A -> a.<o1/o2/.../on>b ,e] c S in SO, implying that
[A -> a<o1/o2/.../.oi/.../on>b ,e] = [W -> .oiD ,e] is in S.

Case 2 : [W -> oi.D ,e], [D -> .b ,e], (fixed i c [1,n]), then

[W -> oi.D ,e] = [A -> a<o1/o2/.../oi./.../on>b ,e] c S in SO, implying that
[A -> a<o1/o2/.../on>.b ,e] = [D -> .b ,e] is in S.

///

Lemma 5.3

Let [C -> q.t ,e] and [B -> p.d ,e] be items in a set T in S* - M* (in the T*-case - or in S/ - M/ in the T/-case) then there exists a set S in SO containing the items [C -> q.t ,e] and [B -> p.d ,e] .

Proof.

T*-case :

Let S* = [T1,T2,...,Tm] be ordered such that, for a set of items Ti c S*, i c [2,n] there exists a set of items Tj c S* and an x such that GOTO(Tj,x) = Ti and j < i .
Let SO = [S1,S2,...,Sn] be ordered similarly.
The ordering is obtained directly from the order of creation of the sets in the LR-construction algorithm.
The proof is by induction on Ti :

Basis : i=1

When [C -> q.t ,e] and [B -> p.d ,e] are items in T1, then q=p=e. Furthermore the item [S' -> .S ,e] is in T1.
Obviously, the items [C -> q.t ,e] and [B -> p.d ,e] are both in S1.

Inductive step:

Assume that the implication is true for all Ti, i < k+1.
Let the items [C -> q.t ,e] and [B -> p.d ,e] both be in Tk+1.
Three possibilities exist for q,p, namely

Case 1 : q≠e , p≠e .
According to the ordering in S*, there exists Ti c S* , i < k+1, containing the items [C -> q'.xt ,e] and [B -> p'.xd ,e] such that GOTO(Ti,x) = Tk+1, where q = q'x and p = p'x .

By the inductive hypothesis, there exists S1 c S0 containing the items [C -> q'.xt ,e] and [B -> p'.xd ,e] .

An application of lemma 5.1 implies there exists Sj c S0 containing the items [C -> q.t ,e] and [B -> p.d ,e] such that GOTO(S1,x) = Sj .

Case 2 : q=p=e .
In this case there exist items in Tk+1, [X1 -> r1x.Y1s1 ,e] and [X2 -> r2x.Y2s2 ,e] such that Y1 =>* Cv1, Y2 =>* Bv2 for some v1,v2.

According to the ordering in S* there exists Ti c S*, i < k+1, containing the items [X1 -> r1.xY1s1 ,e] and [X2 -> r2.xY2s2 ,e] such that GOTO(Ti,x) = Tk+1.

By the inductive hypothesis there exists S1 c S0 containing the items [X1 -> r1.xY1s1 ,e] and [X2 -> r2.xY2s2 ,e].

An application of lemma 5.1 implies then, that there exists Sj c S0, containing the items [X1 -> r1x.Y1s1 ,e] and [X2 -> r2x.Y2s2 ,e] such that GOTO(S1,x) = Sj.

An application of lemma 5.2 implies now that Sj also contains the items [C -> q.t ,e] and [B -> p.d ,e].

Case 3 : either q=e and p≠e or q≠e and p=e .
Assume that q=e and p≠e. (The other case is equivalent).
In this case there exists an item in Tk+1, [X -> rx.Ys ,e] such that Y => Cv for some v.

According to the ordering in S*, there exists Ti, i < k+1, containing the items [X -> r.xYs ,e] and [B -> p'.xd ,e] such that GOTO(Ti,x) = Tk+1, where p = p'x .

By the inductive hypothesis, there exists S1 c S0, containing the items [X -> r.xYs ,e] and [B -> p'.xd ,e].

An application of lemma 5.1 implies then, there exists Sj c S0, containing the items [X -> rx.Ys ,e] and [B -> p'x.d ,e] such that GOTO(S1,x) = Sj.

An application of lemma 5.2 implies now that Sj also contains the item [C -> q.t ,e].

T/-case :

The case is equivalent to the T*-case.

///

Lemma 5.4

T*-case :
If the items [A -> aD. ,e] and [D -> bD. ,e] are in the same set of items in S*, then G is not SLR(k).

T/-case :
If the items [A -> oiW. ,e] and [A -> ojW. ,e] , (i≠j), are in the same set of items in S/, then G is not SLR(k).

Proof.

T*-case :
The assumption implies that there exists a set of items in S* containing the items [A -> a.D ,e] and [D -> b.D ,e]. From lemma 5.3 it follows, that there exists a set of items in SO containing the items [A -> a.<b>*g ,e] and [A -> a<b.>*g ,e]. However this violates the condition (b) in definition 4.9.

T/-case :
The assumption implies that there exists a set of items in S/ containing the items [A -> oi.W ,e] and [A -> oj.W ,e]. From lemma 5.3 it follows, that there exists a set of items in SO containing the items [A -> a<o1/o2/.../oi./.../on>b ,e] and [A -> a<o1/o2/.../oj./.../on>b ,e]. However this violates the condition (c) in definition 4.9.

///

Lemma 5.5

Let the item [A -> r.s ,e] be in S* - M* (in the T*-case - or in S/ - M/ in the T/-case).
If [A -> r.s ,e] = [B -> e.g ,e] then
FOLLOWk(A) = FOLLOWk(B) .

Proof.
The proof is straightforward and is omitted.

///

Lemma 5.6

Let the item [A -> r.s ,e] be in S* - M* (in the T*-case - or in S/ - M/ in the T/-case).
If [A -> r.s ,e] = [B -> e.g ,e] then
EFFk(r.s) = EFFk(e.g) .

Proof.
The proof is straightforward and is omitted.

///

Theorem 5.1

The SLR(k) property under T* and T/ transformation.

Let G = (N,T,P,S) be an ECFG and let G* and G/ be given by definition 5.2 . Then

*-case :
G is SLR(k) if and only if G* is SLR(k).

/-case :
G is SLR(k) if and only if G/ is SLR(k).

Proof.

Only if :

The proof is straightforward by application of lemma 5.3, 5.4, 5.5 and 5.6. Therefore we shall only give an outline of the proof.

*-case :
We have to show, that the conditions (a), (b) and (c) of definition 4.9 are satisfied for G*.
Let [B -> $\underline{x}$.$\underline{t}$,e] and [C -> $\underline{q}$.$\underline{d}$,e] be two distinct items in a set A in $\underline{S*}$.
Lemma 5.4 assures us that these items are not of the form [A -> $\underline{a}$D. ,e] , [D -> $\underline{b}$D. ,e], therefore A c $\underline{S*}$ - $\underline{M*}$.
Then lemma 5.3 enables us to establish the proof by means of the conditions (a), (b) and (c) for the items in $\underline{sO}$ for G and appropriate applications of lemma 5.5 and 5.6.

/-case :
The proof is equivalent to the *-case.

If :

The proof is almost similar to the only if -case and is omitted.

///

5.3. The LR(k) case.

In this section we are going to prove that, applying either the T* or the T/ transformation on a production of an ECFG which is LR(k), we produce a new ECFG, which has preserved the LR(k) property.

The proof of the equivalent theorem for the SLR(k) case (theorem 5.1) given in the previous section is based on the canonical collection of sets of LR(0) items for the grammar and the transformed grammar. The proof of theorem 5.1 can be extended to cover the LR(k) case. However we prefer to give a proof in terms of the grammars, because the definition 4.1 of LR(k)'ness of an ECFG is based only on the grammar.

Theorem 5.2

The LR(k) property under T* and T/ transformation.

Let G = (N,T,P,S) be an ECFG and let G* and G/ be given by definition 5.2 .

*-case :
G is LR(k) if and only if G* is LR(k).

/-case :
G is LR(k) if and only if G/ is LR(k).

Proof.

To facilitate the reading of the proof we assume that G, G* and G/ are already the augmented grammars.

Proof of *-case :

Only if :

Assuming that G is LR(k), prove that G* is LR(k), that is, prove that the three conditions

c1) $S^* \Rightarrow_r^* \underline{d}Bw \Rightarrow_r \underline{d}\underline{m}w$

c2) $S^* \Rightarrow_r^* \underline{l}Cx \Rightarrow_r \underline{l}\underline{p}x = \underline{d}\underline{m}y$

c3) $\mathrm{FIRST}\underline{k}(w) = \mathrm{FIRST}\underline{k}(y)$

imply that $\underline{d}Bx = \underline{l}Cy$, (that is $\underline{d}=\underline{l}$, B=C and x=y) and that $B \rightarrow \underline{m}$, $C \rightarrow \underline{p}$ are the same instance of a production.

The derivations c1, c2 applying productions of G* are directly obtainable in G as long as the productions $A \rightarrow \underline{a}D$, $D \rightarrow \underline{b}D$ and $D \rightarrow \underline{g}$ are not included in the derivations.

If sequences of these are applied and completed (i.e. terminated with $D \rightarrow \underline{g}$) in the $\Rightarrow^*$ -parts of c1, c2 then the same effect is obtainable in G by means of choosing an appropriate instance of the production $A \rightarrow \underline{a}\langle\underline{b}\rangle^*\underline{g}$.

If sequences of these are applied and not completed, then we have to investigate the possible cases more explicitly.

The only interesting productions of which B -> m is an instance, are then

    i1) A -> aD
    i2) D -> bD
    i3) D -> g

Case i1 :
The only possible productions, of which C -> p is an instance, are then

    i11) A -> aD
    i12) D -> bD

Both i11 and i12 imply directly that x=y.

Case i11 :
B -> m, C -> p are both instances of A -> aD, (of the form A -> a'D, A -> a"D) i.e. B=C=A.
Turning to G we use c1, c2 and c3 to construct three conditions

    d1) S =>r* dAw =>r da'xw
    d2) S =>r* lAx =>r la"xx = da'xy
    d3) FIRSTk(w) = FIRSTk(y)

The instances of the production `A -> a<b>*g` applied in d1, d2 are A -> a'x, A -> a"x, where x is an instance of `<b>*g` .
Using that G is LR(k), the conditions imply that d=l and that A -> a'x, A -> a"x are the same instance of `A -> a<b>*g`.
Hence a',a" are the same instance of a, implying that A -> a'D, A -> a"D are the same instance of the production A -> aD.

Case i12 :
B -> m is an instance of A -> aD, (of the form A -> a'D) and C -> p is an instance of D -> bD, (of the form D -> b'D).
The string l can be split l=l'a"b1b2...bi, i ≥ 0, where a" is an instance of a and bj is an instance of b, for j c [1,i].
Turning to G we use c1, c2 and c3 to construct three conditions

    d1) S =>r* dAw =>r da'<>0ew
    d2) S =>r* l'Ax =>r l'a"<b1b2...bib'>i+1ex = da'<>0ey
    d3) FIRSTk(w) = FIRSTk(y)

The instances of the production `A -> a<b>*g` applied in d1, d2 are `A -> a'<>0e`, `A -> a"<b1b2...bib'>i+1e`, where e is an instance of g.
Using that G is LR(k), the conditions imply that d=l' and that `A -> a'<>0e`, `A -> a"<b1b2...bib'>i+1e` are the same instance of `A -> a<b>*g`.
Hence `a'<>0e`, `a"<b1b2...bib'>i+1e` are the same instance of `a<b>*g`.
This is a contradiction, since this implies that i+1=0, where i ≥ 0.

Case i2 :
The only possible productions, of which C -> p is an instance, are then

    i21) A -> aD
    i22) D -> bD

Both i21 and i22 imply directly that x=y.

Case i21 :
The case is equivalent to case i12.

Case i22 :
B -> m, C -> p are both instances of D -> bD, (of the form D -> b'D, D -> b"D), i.e. B=C=D.
The strings d, l can be split d=d'a'b1'b2'...bi', l=l'a"b1"b2"...bj", i,j ≥ 0, where a', a" are instances of a and br', bs" are instances of b, for r c [1,i] and s c [1,j].
Turning to G we use c1, c2 and c3 to construct three conditions

d1) S =>r* d'Aw =>r d'a'<b1'b2'...bi'b'>i+1tw

d2) S =>r* l'Ax =>r l'a"<b1"b2"...bj"b">j+1tx = d'a'<b1'b2'...bi'b'>i+1ty

d3) FIRSTk(w) = FIRSTk(y)

The instances of the production A -> a<b>*q applied in d1, d2 are A -> a'<b1'b2'...bi'b'>i+1t, A -> a"<b1"b2"...bj"b">j+1t, where t is an instance of q.
Using that G is LR(k), the conditions imply that d'=l' and that A -> a'<b1'b2'...bi'b'>i+1t, A -> a"<b1"b2"...bj"b">j+1t are the same instance of A -> a<b>*q.
Hence a'<b1'b2'...bi'b'>i+1, a"<b1"b2"...bj"b">j+1 are the same instance of a<b>*. This implies, that b', b" are the same instance of b and that a'b1'b2'...bi', a"b1"b2"...bj" are the same instance, i.e. i=j.
Thus d=l and D -> b'D, D -> b"D are the same instance of the production D -> bD.

Case i3 :
There are two possibilities for the nonterminal C, namely

i31) C=D

i32) C≠D

Case i31 :
C -> p cannot be an instance of D -> bD, therefore B -> m, C -> p are both instances of D -> q, i.e. B=C=D.
The strings d, l can be split d=d'x', l=l'x", where x', x" are both instances of a<b>*.
Turning to G we use c1, c2 and c3 to construct three conditions

d1) S =>r* d'Aw =>r d'x'mw = dmw

d2) S =>r* l'Ax =>r l'x"px = dmy

d3) FIRSTk(w) = FIRSTk(y)

The instances of the production A -> a<b>*q applied in d1, d2 are A -> x'm, A -> x"p.
Using that G is LR(k), the conditions imply that d'=l' and that A -> x'm, A -> x"p are the same instance of A -> a<b>*q.
Hence x'm, x"p are the same instance of a<b>*q. This implies, that m, p are the same instance of q and that x', x" are the same instance of a<b>*.
Thus d=l and D -> m, D -> p are the same instance of the production D -> q.

Case i32 :
A -> m is an instance of D -> q . Because C≠D and because C -> p cannot be an instance of A -> aD, the production C -> p exists in G.
The string d can be split d=d'x, where x is an instance of

a<b>*.
Turning to G we use c1, c2 and c3 to construct three conditions
d1) S =>r* d'Aw =>r d'xmw = dmw
d2) S =>r* lCx =>r lpx = dmy
d3) FIRSTk(w) = FIRSTk(y)
The instance of the production A -> a<b>*g applied in d1 is A -> xm.
Using that G is LR(k), the conditions imply that d'=l, C=A, x=y and that A -> xm, C -> p are the same instance of A ->_a≤b>*g.
This is a contradiction, since A -> a<b>*g is not available in G*, but C -> p is nevertheless an instance of this production.

If :

Assuming that G* is LR(k), prove that G is LR(k), that is, prove that the three conditions
c1) S =>r* dBw =>r dmw
c2) S =>r* lCx =>r lpx = dmy
c3) FIRSTk(w) = FIRSTk(y)
imply that dBx = lCy, (that is d=l, B=C and x=y) and that B -> m, C -> p are the same instance of a production.

The derivations in c1,c2 applying productions of G are directly obtainable in G* as long as the production A -> a<b>*g is not included in the derivations. If this production is applied in the =>* -parts of c1, c2 then the same effect is obtainable in G* by means of choosing an appropriate sequence of instances of the productions A -> aD, D -> bD and D -> g . If the production is applied in the => -parts we have to investigate the case more explicitly.

Assume that B -> m is an instance of A -> a<b>*g, of the form A -> a'<b1'b2'...bi'>ig' where a' is an instance of a, bk' is an instance of b, k c [1,i] and g' is an instance of g.
There are two possibilities for the instance C -> p, namely
i1) C -> p is an instance of A -> a<b>*g.
i2) C -> p is not an instance of A -> a<b>*g.

Case i1 :
Let C -> p be of the form A -> a"<b1"b2"...bj">jg", where a", bk" (k c [1,j]) and g" are instances of a, b and g, respectively.
Turning to G* we use c1,c2 and c3 to construct three conditions
d1) S* =>r* dAw
=>r da'Dw
=>r da'b1'Dw
=>r* da'b1'b2'...bi'Dw
=>r da'b1'b2'...bi'g'w
= dmw
d2) S* =>r* lAx
=>r la"Dx
=>r la"b1"Dx
=>r* la"b1"b2"...bj"Dx
=>r la"b1"b2"...bj"g"x
= lpx = dmy

d3) FIRSTk(w) = FIRSTk(y)
The instances applied in d1, d2 are obvious instances of the productions A -> aD, D -> bD and D -> g.
Using that G* is LR(k) the conditions imply that da'b1'b2'...bi' = la"b1"b2"...bj" and x=y and that D -> g', D -> g" are the same instance of D -> g.
Applying this result (and that G* is LR(k)) the conditions now imply that da'b1'b2'...bi-1' = la"b1"b2"...bj-1" and that D -> bi'D, D -> bj"D are the same instance of D -> bD.
Continuing in this fashion, the conditions imply that d = l and that A -> a'D, A -> a"D are the same instance of A -> aD and finally that i=j and D -> bk'D, D -> bk"D are the same instance of D -> bD for k c [1,i].
Hence B -> m, C -> l are the same instance of A -> a<b>*g.

Case i2 :
Turning to G* we use c1,c2 and c3 to construct three conditions

```
    d1) S* =>r* dAw
           =>r* da'b1'b2'...bi'Dw
           =>r da'b1'b2'...bi'g'w
           = dmw
    d2) S* =>r* lCx
           =>r lpx = dmy
    d3) FIRSTk(w) = FIRSTk(y)
```

The instances applied in d1 are obvious instances of the productions A -> aD, D -> bD and D -> g.
Using that G* is LR(k), the conditions imply that da'b1'b2'...bi' = l, D=C, x=y and that D -> g', C -> p are the same instance of D -> g.
This is a contradiction since D -> g is not available in G, but C -> p is nevertheless an instance of this production.

Proof of /-case :

The proof is similar to the *-case and is omitted.

///

6. EVALUATION.

In a practical implementation of an LR-generator system for ECFGs, it seems to be most appropriate to rewrite the source grammar as discussed in section 5, instead of constructing the LR-items directly. One can then use the standard techniques and needs not modify the parsing algorithm.

Transforming the grammar is a relatively simple job. If the rewriting rules are applied directly, a lot of single productions may be introduced and special consideration can be taken in order to avoid such productions. However, if a system is available which automatically removes single productions, one needs not care about these special cases.

A point to consider is how and when to call semantic routines. Usually in LR-parsers a routine is called each time a reduction is performed , using the number of the applied production as a parameter.

In the ECFG case there is the possibility to give more or less detailed information about which instance of the production has been applied.

Semantic routines should not always be called exactly at the moment of performing reductions in the transformed grammar. Instead the parser has to act as if the grammar has not been transformed i.e. a reduction in the original grammar corresponds to a series of reductions in the transformed grammar.

We have required explicit use of parentheses in regular expressions. In a practical implementation, one can introduce precedence rules between *, / and concatenation in order to avoid some of the parentheses. The precedence rules for the operators can then be: repetition is most binding, then concatenation, then alternation.

The extra requirements in the LR(k) and SLR(k) definitions state the conditions necessary to perform a unique reduction of the right hand side of an applied production. If one does not care about how the right hand side is determined , but just wants one of the possiblities , one has to resolve ambiguity problems in the transformed grammar.

Suppose that conditions (2) and (3) in theorem 4.1 are not satisfied. The transformed grammar will then in the *-case give rise to a conflict between the productions A -> $\underline{a}$D and D -> $\underline{b}$D on a given lookahead symbol. Similarly in the /-case we get conflicts between productions of the form W -> $\underline{oi}$D and A -> $\underline{oj}$D, $i \neq j$.

Some simple ambiguity problems may be solved, but this subject is not treated further here. The reader is referred to Aho, Johnston & Ullman [1973].

We have not generalized to LALR(k), but this seems to be straightforward. If conflicts in definition 4.9 may be solved by using lookahead, then the transformed grammar will be LALR(k).

Another frequently used operator is the repetition operator +, which means at least one repetition. To construct the LR-items the following rules can be used.

Any item of the form

A -> $\underline{a}$.<$\underline{b}$>+$\underline{g}$

is replaced by

A -> $\underline{a}$ #<$\underline{b}$>+$\underline{g}$

A -> $\underline{a}$< .$\underline{b}$>+$\underline{g}$

Any item of the form

A -> $\underline{a}$<$\underline{b}$.>+$\underline{g}$

is replaced by

A -> $\underline{a}$<$\underline{b}$ #>+$\underline{g}$

A -> $\underline{a}$<$\underline{b}$>+ .$\underline{g}$

A -> $\underline{a}$< .$\underline{b}$>+$\underline{g}$

Testing for LR(k), condition (2) in theorem 4.1 is also valid with the *-operator replaced by the +-operator. Similarly in definition 4.9, condition (b) is valid for the +-operator.

To eliminate the +-operator, the production A-> $\underline{a}$<$\underline{b}$>+$\underline{g}$, with $\underline{a}$ c R(N U T), may be replaced by

A -> $\underline{a}\underline{b}$D, D -> $\underline{b}$D, D -> $\underline{g}$

keeping possible LR-conditions invariant.

The authors want to thank Erik Meineche Schmidt and Arto Salomaa for valuable criticism and comments on the manuscript.

7. REFERENCES.

Aho, A. V., S. C. Johnson, and J. D. Ullman [1973].
Deterministic parsing of ambiguous grammars.
Conference Record of ACM Symposium on principles of Programming Languages, October 1973, pp. 1-21.

Aho, A. V., and J. D. Ullman [1972].
The theory of parsing, translation and compiling.
Volume 1: parsing.
Prentice-Hall, Englewood Cliffs, N.J.

Aho, A. V., and J D. Ullman [1973].
The theory of parsing, translation and compiling.
Volume 2: compiling.
Prentice-Hall, Englewood Cliffs, N.J.

DeRemer, F. L. [1969].
Practical translators for LR(k) languages.
Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, Mass.

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

DeRemer, F. L. [1974].
Lexical analysis.
Compiler construction, an advanced course.
Lecture Notes in Computer Science nr. 21, Springer Verlag.

Early, J. [1970].
An efficient context-free parsing algorithm.
Comm. ACM 13:2, 94-102.

Knuth D. E. [1965].
On the translation of languages from left to right.
Information and Control 8:6, 607-639.