CIS511
A Survey of LR-Parsing Methods
The Graph Method For Computing Fixed Points
Computation of FIRST, FOLLOW, and LALR(1) Lookahead Sets

Jean Gallier

Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104, USA
jean@saul.cis.upenn.edu

June 27, 2008

Abstract. We give a brief survey on LR-parsing methods. We begin with the definition of characteristic strings and the construction of Knuth's $LR(0)$-characteristic automaton. Next, we describe the shift/reduce algorithm. The need for lookahead sets is motivated by the resolution of conflicts. A unified method for computing FIRST, FOLLOW and LALR(1) lookahead sets is presented. The method uses a same graph algorithm *Traverse* which finds all nodes reachable from a given node and computes the union of predefined sets assigned to these nodes. Hence, the only difference between the various algorithms for computing FIRST, FOLLOW and LALR(1) lookahead sets lies in the fact that the initial sets and the graphs are computed in different ways. The method can be viewed as an efficient way for solving a set of simultaneously recursive equations with set variables. The method is inspired by DeRemer and Pennello's method for computing LALR(1) lookahead sets. However, DeRemer and Pennello use a more sophisticated graph algorithm for finding strongly connected components. We use a slightly less efficient but simpler algorithm (a depth-first search). We conclude with a brief presentation of $LR(1)$ parsers.

1 $LR(0)$-Characteristic Automata

The purpose of *LR-parsing*, invented by D. Knuth in the mid sixties, is the following: Given a context-free grammar G, for any terminal string $w \in \Sigma^*$, find out whether w belongs to the language $L(G)$ generated by G, and if so, construct a rightmost derivation of w, in a deterministic fashion. Of course, this is not possible for all context-free grammars, but only for those that correspond to languages that can be recognized by a *deterministic* PDA (DPDA). Knuth's major discovery was that for a certain type of grammars, the $LR(k)$-grammars, a certain kind of DPDA could be constructed from the grammar (*shift/reduce parsers*). The k in $LR(k)$ refers to the amount of *lookahead* that is necessary in order to proceed deterministically. It turns out that $k = 1$ is sufficient, but even in this case, Knuth construction produces very large DPDA's, and his original $LR(1)$ method is not practical. Fortunately, around 1969, Frank DeRemer, in his MIT Ph.D. thesis, investigated a practical restriction of Knuth's method, known as $SLR(k)$, and soon after, the $LALR(k)$ method was discovered. The $SLR(k)$ and the $LALR(k)$ methods are both based on the construction of the $LR(0)$-*characteristic automaton* from a grammar G, and we begin by explaining this construction. The additional ingredient needed to obtain an $SLR(k)$ or an $LALR(k)$ parser from an $LR(0)$ parser is the computation of lookahead sets. In the SLR case, the FOLLOW sets are needed, and in the $LALR$ case, a more sophisticated version of the FOLLOW sets is needed. We will consider the construction of these sets in the case $k = 1$. We will discuss the shift/reduce algorithm and consider briefly ways of building $LR(1)$-parsing tables.

For simplicity of exposition, we first assume that grammars have no ϵ-rules. This restriction will be lifted in Section 10. Given a reduced context-free grammar $G = (V, \Sigma, P, S')$ augmented with start production $S' \to S$, where S' does not appear in any other productions, the set C_G of *characteristic strings of* G is the following subset of V^* (watch out, not Σ^*):

$$\begin{aligned} C_G = \{\alpha\beta \in V^* \mid S' &\overset{*}{\underset{rm}{\Longrightarrow}} \alpha B v \underset{rm}{\Longrightarrow} \alpha\beta v, \\ &\alpha, \beta \in V^*, v \in \Sigma^*, B \to \beta \in P\}. \end{aligned}$$

In words, C_G is a certain set of prefixes of sentential forms obtained in rightmost derivations: Those obtained by truncating the part of the sentential form immediately following the rightmost symbol in the righthand side of the production applied at the last step.

The fundamental property of LR-parsing, due to D. Knuth, is that C_G is a *regular language*. Furthermore, a DFA, DCG, accepting C_G, can be constructed from G.

Conceptually, it is simpler to construct the DFA accepting C_G in two steps:

(1) First, construct a nondeterministic automaton with ϵ-rules, NCG, accepting C_G.

(2) Apply the subset construction (Rabin and Scott's method) to NCG to obtain the DFA DCG.

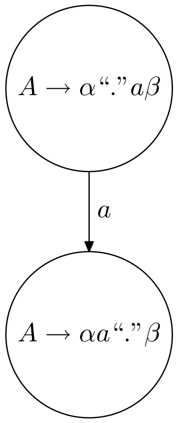

Figure 1: Transition on terminal input a

In fact, careful inspection of the two steps of this construction reveals that it is possible to construct DCG directly in a single step, and this is the construction usually found in most textbooks on parsing.

The nondeterministic automaton NCG accepting C_G is defined as follows:

The states of N_{C_G} are "marked productions", where a marked production is a string of the form $A \to \alpha``."\beta$, where $A \to \alpha\beta$ is a production, and "." is a symbol not in V called the "dot" and which can appear anywhere within $\alpha\beta$.

The start state is $S' \to ``."S$, and the transitions are defined as follows:

(a) For every terminal $a \in \Sigma$, if $A \to \alpha``."a\beta$ is a marked production, with $\alpha, \beta \in V^*$, then there is a transition on input $a$ from state $A \to \alpha``."a\beta$ to state $A \to \alpha a``."\beta$ obtained by "shifting the dot." Such a transition is shown in Figure 1.

(b) For every nonterminal $B \in N$, if $A \to \alpha``."B\beta$ is a marked production, with $\alpha, \beta \in V^*$, then there is a transition on input $B$ from state $A \to \alpha``."B\beta$ to state $A \to \alpha B``."\beta$ (obtained by "shifting the dot"), and transitions on input $\epsilon$ (the empty string) to all states $B \to ``."\gamma_i$, for all productions $B \to \gamma_i$ with left-hand side B. Such transitions are shown in Figure 2.

(c) A state is *final* if and only if it is of the form $A \to \beta``."$ (that is, the dot is in the rightmost position).

The above construction is illustrated by the following example:

Example 1. Consider the grammar G_1 given by:

$$\begin{aligned} S &\longrightarrow E \\ E &\longrightarrow aEb \\ E &\longrightarrow ab \end{aligned}$$

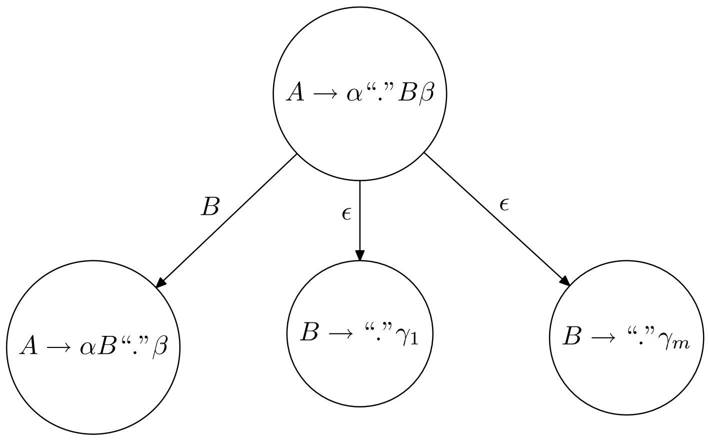

Figure 2: Transitions from a state $A \to \alpha``."B\beta$

The NFA for C_{G_1} is shown in Figure 3. The result of making the NFA for C_{G_1} deterministic is shown in Figure 4 (where transitions to the "dead state" have been omitted). The internal structure of the states $1, \ldots, 6$ is shown below:

$$\begin{aligned}
1 : S &\longrightarrow .E \\
E &\longrightarrow .aEb \\
E &\longrightarrow .ab \\
2 : E &\longrightarrow a.Eb \\
E &\longrightarrow a.b \\
E &\longrightarrow .aEb \\
E &\longrightarrow .ab \\
3 : E &\longrightarrow aE.b \\
4 : S &\longrightarrow E. \\
5 : E &\longrightarrow ab. \\
6 : E &\longrightarrow aEb.
\end{aligned}$$

The next example is slightly more complicated.

Example 2. Consider the grammar G_2 given by:

$$\begin{aligned}
S &\longrightarrow E \\
E &\longrightarrow E + T \\
E &\longrightarrow T \\
T &\longrightarrow T * a \\
T &\longrightarrow a
\end{aligned}$$

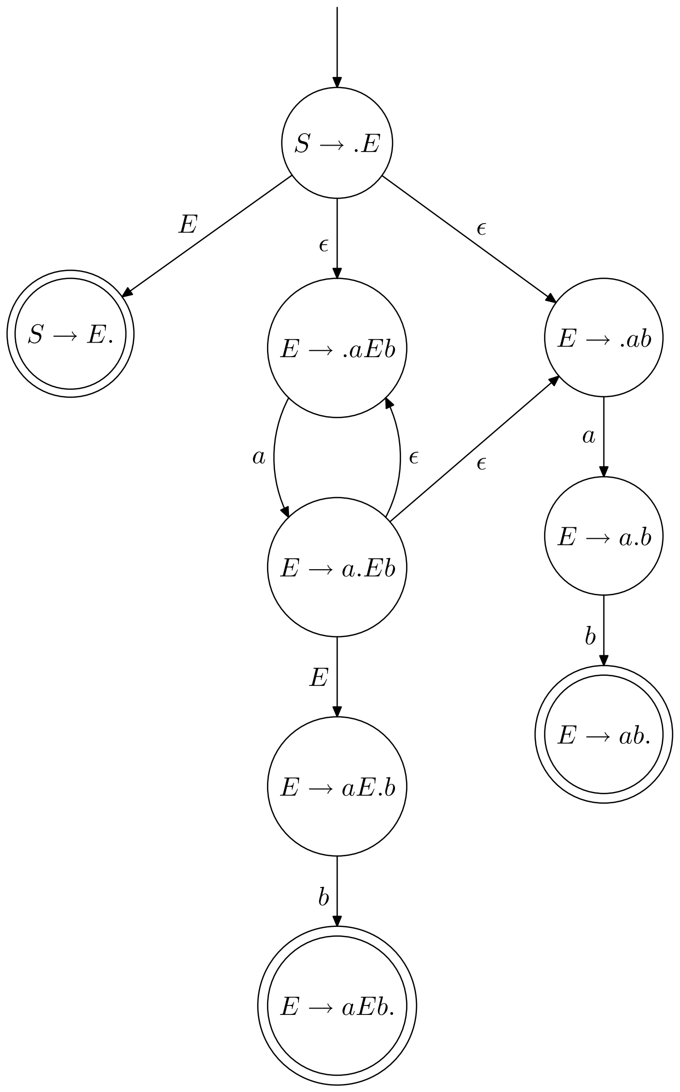

Figure 3: NFA for C_{G_1}

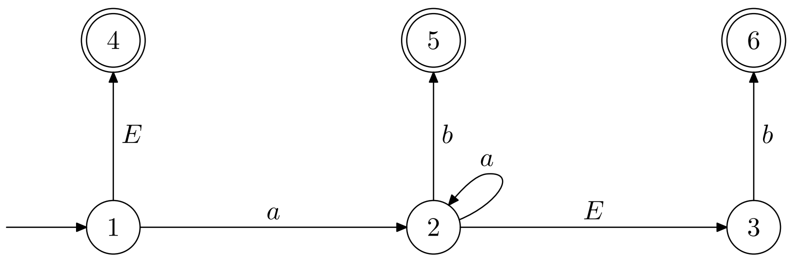

Figure 4: DFA for C_{G_1}

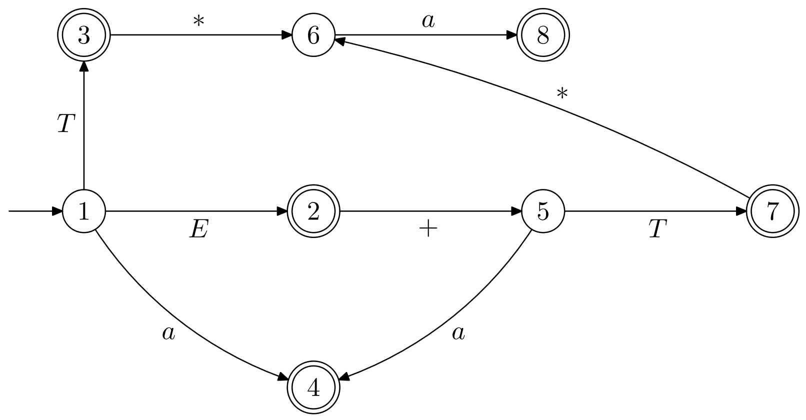

Figure 5: DFA for C_{G_2}

The result of making the NFA for C_{G_2} deterministic is shown in Figure 5 (where transitions to the "dead state" have been omitted). The internal structure of the states $1, \ldots, 8$ is shown below:

$$\begin{array}{rcl}
1: S & \longrightarrow & .E \\
E & \longrightarrow & .E + T \\
E & \longrightarrow & .T \\
T & \longrightarrow & .T * a \\
T & \longrightarrow & .a \\
2: E & \longrightarrow & E. + T \\
S & \longrightarrow & E. \\
3: E & \longrightarrow & T. \\
T & \longrightarrow & T. * a \\
4: T & \longrightarrow & a. \\
5: E & \longrightarrow & E + .T \\
T & \longrightarrow & .T * a \\
T & \longrightarrow & .a \\
6: T & \longrightarrow & T * .a \\
7: E & \longrightarrow & E + T. \\
T & \longrightarrow & T. * a \\
8: T & \longrightarrow & T * a.
\end{array}$$

Note that some of the marked productions are more important than others. For example, in state 5, the marked production $E \longrightarrow E + .T$ determines the state. The other two items $T \longrightarrow .T * a$ and $T \longrightarrow .a$ are obtained by ϵ-closure. We call a marked production of the form $A \longrightarrow \alpha.\beta$, where $\alpha \neq \epsilon$, a *core item*. If we also call $S' \longrightarrow .S$ a core item, we observe that every state is completely determined by its subset of core items. The other items in the state are obtained via ϵ-closure. We can take advantage of this fact to write a more efficient algorithm to construct in a single pass the $LR(0)$-automaton. Also observe the so-called *spelling property*: All the transitions entering any given state have the same label.

Given a state s, if s contains both a reduce item $A \longrightarrow \gamma.$ and a shift item $B \longrightarrow \alpha.a\beta$, where $a \in \Sigma$, we say that there is a *shift/reduce conflict* in state s on input a. If s contains two (distinct) reduce items $A_1 \longrightarrow \gamma_1.$ and $A_2 \longrightarrow \gamma_2.$, we say that there is a *reduce/reduce conflict* in state s. A grammar is said to be $LR(0)$ if the DFA DCG has no conflicts. This is the case for the grammar G_1. However, it should be emphasized that this is extremely rare in practice. The grammar G_1 is just very nice, and a toy example. In fact, G_2 is not $LR(0)$. To eliminate conflicts, one can either compute $SLR(1)$-lookahead sets, using FOLLOW sets (see Section 6), or sharper lookahead sets, the $LALR(1)$ sets (see Section 9). For example, the computation of $SLR(1)$-lookahead sets for G_2 will eliminate the conflicts. We will describe methods for computing $SLR(1)$-lookahead sets and $LALR(1)$-lookahead sets in Sections 6, 9, and 10. A more drastic measure is to compute the $LR(1)$-automaton, in which the states incoporate lookahead symbols (see Section 11). However, as we said before, this is not a practical methods for large grammars.

2 Shift/Reduce Parsers

A shift/reduce parser is a modified kind of DPDA. Firstly, push moves, called *shift moves*, are restricted so that exactly one symbol is pushed on top of the stack. Secondly, more powerful kinds of pop moves, called *reduce moves*, are allowed. During a reduce move, a finite number of stack symbols may be popped off the stack, and the last step of a reduce move, called a *goto move*, consists of pushing one symbol on top of new topmost symbol in the stack. Shift/reduce parsers use *parsing tables* constructed from the $LR(0)$-characteristic automaton DCG associated with the grammar. The shift and goto moves come directly from the transition table of DCG, but the determination of the reduce moves requires the computation of *lookahead sets*. The $SLR(1)$ lookahead sets are obtained from some sets called the FOLLOW sets (see Section 6), and the $LALR(1)$ lookahead sets $\mathrm{LA}(s, A \longrightarrow \gamma)$ require fancier FOLLOW sets (see Section 9).

The construction of shift/reduce parsers is made simpler by assuming that the end of input strings $w \in \Sigma^*$ is indicated by the presence of an *endmarker*, usually denoted \$, and assumed not to belong to Σ.

Consider the grammar G_1 of Example 1, where we have numbered the productions $0, 1, 2$:

$$\begin{aligned} 0: S &\longrightarrow E \\ 1: E &\longrightarrow aEb \\ 2: E &\longrightarrow ab \end{aligned}$$

The parsing tables associated with the grammar G_1 are shown below:

	a	b	$\$$	E
1	$s2$			4
2	$s2$	$s5$		3
3		$s6$		
4			acc	
5	$r2$	$r2$	$r2$	
6	$r1$	$r1$	$r1$	

Entries of the form si are *shift actions*, where i denotes one of the states, and entries of the form rn are *reduce actions*, where n denotes a production number (*not* a state). The special action acc means accept, and signals the successful completion of the parse. Entries of the form i, in the rightmost column, are *goto actions*. All blank entries are **error** entries, and mean that the parse should be aborted.

We will use the notation $\mathrm{action}(s, a)$ for the entry corresponding to state s and terminal $a \in \Sigma \cup \{\$\}$, and $\mathrm{goto}(s, A)$ for the entry corresponding to state s and nonterminal $A \in N - \{S'\}$.

Assuming that the input is $w\$$, we now describe in more detail how a shift/reduce parser proceeds. The parser uses a stack in which states are pushed and popped. Initially, the stack contains state 1 and the cursor pointing to the input is positioned on the leftmost symbol. There are four possibilities:

(1) If $\mathrm{action}(s, a) = sj$, then push state j on top of the stack, and advance to the next input symbol in $w\$$. This is a *shift move*.

(2) If $\mathrm{action}(s, a) = rn$, then do the following: First, determine the length $k = |\gamma|$ of the righthand side of the production $n\colon A \longrightarrow \gamma$. Then, pop the topmost k symbols off the stack (if $k = 0$, no symbols are popped). If p is the new top state on the stack (after the k pop moves), push the state $\mathrm{goto}(p, A)$ on top of the stack, where A is the lefthand side of the "reducing production" $A \longrightarrow \gamma$. Do not advance the cursor in the current input. This is a *reduce move*.

(3) If $\mathrm{action}(s, \$) = \mathrm{acc}$, then accept. The input string w belongs to $L(G)$.

(4) In all other cases, **error**, abort the parse. The input string w does not belong to $L(G)$.

Observe that no explicit state control is needed. The current state is always the current topmost state in the stack. We illustrate below a parse of the input $aaabbb\$$.

stack	remaining input	action
1	$aaabbb\$$	$s2$
12	$aabbb\$$	$s2$

122	$abbb\$$	$s2$
1222	$bbb\$$	$s5$
12225	$bb\$$	$r2$
1223	$bb\$$	$s6$
12236	$b\$$	$r1$
123	$b\$$	$s6$
1236	$\$$	$r1$
14	$\$$	acc

Observe that the sequence of reductions read from bottom-up yields a rightmost derivation of $aaabbb$ from E (or from S, if we view the action acc as the reduction by the production $S \longrightarrow E$). This is a general property of LR-parsers.

The $SLR(1)$ reduce entries in the parsing tables are determined as follows: For every state s containing a reduce item $B \longrightarrow \gamma.$, if $B \longrightarrow \gamma$ is the production number n, enter the action rn for state s and every terminal $a \in \mathrm{FOLLOW}(B)$. If the resulting shift/reduce parser has no conflicts, we say that the grammar is $SLR(1)$. For the $LALR(1)$ reduce entries, enter the action rn for state s and production $n\colon B \longrightarrow \gamma$, for all $a \in \mathrm{LA}(s, B \longrightarrow \gamma)$. Similarly, if the shift/reduce parser obtained using $LALR(1)$-lookahead sets has no conflicts, we say that the grammar is $LALR(1)$.

3 Computation of FIRST

In order to compute the FOLLOW sets, we first need to to compute the FIRST sets! For simplicity of exposition, we first assume that grammars have no ϵ-rules. The general case will be treated in Section 10.

Given a context-free grammar $G = (V, \Sigma, P, S')$ (augmented with a start production $S' \longrightarrow S$), for every nonterminal $A \in N = V - \Sigma$, let

$$\mathrm{FIRST}(A) = \{a \mid a \in \Sigma,\ A \overset{+}{\Longrightarrow} a\alpha, \text{ for some } \alpha \in V^*\}.$$

For a terminal $a \in \Sigma$, let $\mathrm{FIRST}(a) = \{a\}$. The key to the computation of $\mathrm{FIRST}(A)$ is the following observation: a is in $\mathrm{FIRST}(A)$ if either a is in

$$\mathrm{INITFIRST}(A) = \{a \mid a \in \Sigma,\ A \longrightarrow a\alpha \in P, \text{ for some } \alpha \in V^*\},$$

or a is in

$$\{a \mid a \in \mathrm{FIRST}(B),\ A \longrightarrow B\alpha \in P, \text{ for some } \alpha \in V^*,\ B \neq A\}.$$

Note that the second assertion is true because, if $B \overset{+}{\Longrightarrow} a\delta$, then $A \Longrightarrow B\alpha \overset{+}{\Longrightarrow} a\delta\alpha$, and so, $\mathrm{FIRST}(B) \subseteq \mathrm{FIRST}(A)$ whenever $A \longrightarrow B\alpha \in P$, with $A \neq B$. Hence, the FIRST sets are the least solution of the following set of recursive equations: For each nonterminal A,

$$\mathrm{FIRST}(A) = \mathrm{INITFIRST}(A) \cup \bigcup \{\mathrm{FIRST}(B) \mid A \longrightarrow B\alpha \in P,\ A \neq B\}.$$

In order to explain the method for solving such systems, we will formulate the problem in more general terms, but first, we describe a "naive" version of the shift/reduce algorithm that hopefully demystifies the "'optimized version" described in Section 2.

4 The Intuition Behind the Shift/Reduce Algorithm

Let $DCG = (K, V, \delta, q_0, F)$ be the DFA accepting the regular language C_G, and let δ^* be the extension of δ to $K \times V^*$. Let us assume that the grammar G is either $SLR(1)$ or $LALR(1)$, which implies that it has no shift/reduce or reduce/reduce conflicts. We can use the DFA DCG accepting C_G recursively to parse $L(G)$. The function CG is defined as follows: Given any string $\mu \in V^*$,

$$CG(\mu) = \begin{cases} error & \text{if } \delta^*(q_0, \mu) = error; \\ (\delta^*(q_0, \theta), \theta, v) & \text{if } \delta^*(q_0, \theta) \in F,\ \mu = \theta v \text{ and } \theta \text{ is the} \\ & \text{shortest prefix of } \mu \text{ s.t. } \delta^*(q_0, \theta) \in F. \end{cases}$$

The naive shift-reduce algorithm is shown below:

```
begin
  $accept$ := true;
  $stop$ := false;
  $\mu := w\$$;   {input string}
  while $\neg stop$ do
    if $CG(\mu) = error$ then
      $stop$ := true; $accept$ := false
    else
      Let $(q, \theta, v) = CG(\mu)$
      Let $B \rightarrow \beta$ be the production so that
      action$(q, \mathrm{FIRST}(v)) = B \rightarrow \beta$ and let $\theta = \alpha\beta$
      if $B \rightarrow \beta = S' \rightarrow S$ then
        $stop$ := true
      else
        $\mu := \alpha B v$   {reduction}
      endif
    endif
  endwhile
end
```

The idea is to recursively run the DFA DCG on the sentential form μ, until the first final state q is hit. Then, the sentential form μ must be of the form $\alpha\beta v$, where v is a terminal string ending in \$, and the final state q contains a reduce item of the form $B \longrightarrow \beta$, with $\text{action}(q, \text{FIRST}(v)) = B \longrightarrow \beta$. Thus, we can reduce $\mu = \alpha\beta v$ to $\alpha B v$, since we have found a rightmost derivation step, and repeat the process.

Note that the major inefficiency of the algorithm is that when a reduction is performed, the prefix α of μ is reparsed entirely by DCG. Since DCG is deterministic, the sequence of states obtained on input α is uniquely determined. If we keep the sequence of states produced on input θ by DCG in a stack, then it is possible to avoid reparsing α. Indeed, all we have to do is update the stack so that just before applying DCG to αAv, the sequence of states in the stack is the sequence obtained after parsing α. This stack is obtained by popping the $|\beta|$ topmost states and performing an update which is just a goto move. This is the standard version of the shift/reduce algorithm!

5 The Graph Method for Computing Fixed Points

Let X be a finite set representing the domain of the problem (in Section 3 above, $X = \Sigma$), let $F(1), \ldots, F(N)$ be N sets to be computed and let $I(1), \ldots, I(N)$ be N given subsets of X. The sets $I(1), \ldots, I(N)$ are the initial sets. We also have a directed graph G whose set of nodes is $\{1, \ldots, N\}$ and which represents relationships among the sets $F(i)$, where $1 \leq i \leq N$. The graph G has no parallel edges and no loops, but it may have cycles. If there is an edge from i to j, this is denoted by iGj (note that the absense of loops means that iGi never holds). Also, the existence of a path from i to j is denoted by iG^+j. The graph G represents a relation, and G^+ is the graph of the transitive closure of this relation. The existence of a path from i to j, including the null path, is denoted by iG^*j. Hence, G^* is the reflexive and transitive closure of G. We want to solve for the least solution of the system of recursive equations:

$$F(i) = I(i) \cup \{F(j) \mid iGj,\, i \neq j\}, \quad 1 \leq i \leq N.$$

Since $(2^X)^N$ is a complete lattice under the inclusion ordering (which means that every family of subsets has a least upper bound, namely, the union of this family), it is an ω-complete poset, and since the function $F\colon (2^X)^N \to (2^X)^N$ induced by the system of equations is easily seen to preserve least upper bounds of ω-chains, the least solution of the system can be computed by the standard fixed point technique (as explained in Section 3.7 of the class notes). We simply compute the sequence of approximations $(F^k(1), \ldots, F^k(N))$, where

$$F^0(i) = \emptyset, \quad 1 \leq i \leq N,$$

and

$$F^{k+1}(i) = I(i) \cup \bigcup \{F^k(j) \mid iGj,\, i \neq j\}, \quad 1 \leq i \leq N.$$

It is easily seen that we can stop at $k = N - 1$, and the least solution is given by

$$F(i) = F^1(i) \cup F^2(i) \cup \cdots \cup F^N(i), \quad 1 \leq i \leq N.$$

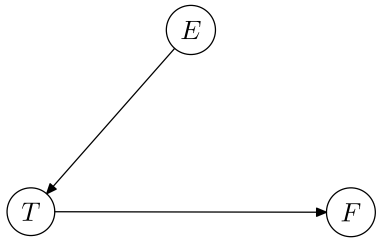

Figure 6: Graph GFIRST for G_1

However, the above expression can be simplified to

$$F(i) = \bigcup \{I(j) \mid iG^* j\}, \quad 1 \leq i \leq N.$$

This last expression shows that in order to compute $F(i)$, it is necessary to compute the union of all the initial sets $I(j)$ reachable from i (including i). Hence, any transitive closure algorithm or graph traversal algorithm will do. For simplicity and for pedagogical reasons, we use a depth-first search algorithm.

Going back to FIRST, we see that all we have to do is to compute the INITFIRST sets, the graph GFIRST, and then use the graph traversal algorithm. The graph GFIRST is computed as follows: The nodes are the nonterminals and there is an edge from A to B ($A \neq B$) if and only if there is a production of the form $A \longrightarrow B\alpha$, for some $\alpha \in V^*$.

Example 1. Computation of the FIRST sets for the grammar G_1 given by the rules:

$$\begin{array}{lcl} S & \longrightarrow & E\$ \\ E & \longrightarrow & E + T \\ E & \longrightarrow & T \\ T & \longrightarrow & T * F \\ T & \longrightarrow & F \\ F & \longrightarrow & (E) \\ F & \longrightarrow & -T \\ F & \longrightarrow & a. \end{array}$$

We get

$$\text{INITFIRST}(E) = \emptyset, \quad \text{INITFIRST}(T) = \emptyset, \quad \text{INITFIRST}(F) = \{(, -, a\}.$$

The graph GFIRST is shown in Figure 6.

We obtain the following FIRST sets:

$$\text{FIRST}(E) = \text{FIRST}(T) = \text{FIRST}(F) = \{(, -, a\}.$$

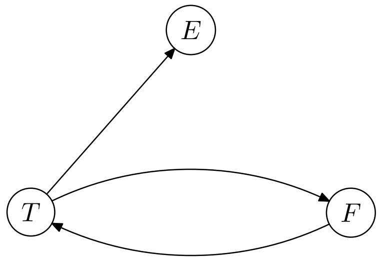

Figure 7: Graph GFOLLOW for G_1

6 Computation of FOLLOW

Recall the definition of FOLLOW(A) for a nonterminal A:

$$\text{FOLLOW}(A) = \{a \mid a \in \Sigma,\ S \overset{+}{\Longrightarrow} \alpha A a \beta, \quad \text{for some } \alpha, \beta \in V^*\}.$$

Note that a is in FOLLOW(A) if either a is in

$$\text{INITFOLLOW}(A) = \{a \mid a \in \Sigma,\ B \longrightarrow \alpha A X \beta \in P,\ a \in \text{FIRST}(X),\ \alpha, \beta \in V^*\}$$

or a is in

$$\{a \mid a \in \text{FOLLOW}(B),\ B \longrightarrow \alpha A \in P,\ \alpha \in V^*,\ A \neq B\}.$$

Indeed, if $S \overset{+}{\Longrightarrow} \lambda B a \rho$, then $S \overset{+}{\Longrightarrow} \lambda B a \rho \Longrightarrow \lambda \alpha A a \rho$, and so,

$$\text{FOLLOW}(B) \subseteq \text{FOLLOW}(A)$$

whenever $B \longrightarrow \alpha A$ is in P, with $A \neq B$. Hence, the FOLLOW sets are the least solution of the set of recursive equations: For all nonterminals A,

$$\text{FOLLOW}(A) = \text{INITFOLLOW}(A) \cup \bigcup \{\text{FOLLOW}(B) \mid B \longrightarrow \alpha A \in P,\ \alpha \in V^*,\ A \neq B\}.$$

According to the method explained above, we just have to compute the INITFOLLOW sets (using FIRST) and the graph GFOLLOW, which is computed as follows: The nodes are the nonterminals and there is an edge from A to B ($A \neq B$) if and only if there is a production of the form $B \longrightarrow \alpha A$ in P, for some $\alpha \in V^*$. Note the duality between the construction of the graph GFIRST and the graph GFOLLOW.

Example 2. Computation of the FOLLOW sets for the grammar G_1.

$$\text{INITFOLLOW}(E) = \{+,), \$\},\ \text{INITFOLLOW}(T) = \{*\},\ \text{INITFOLLOW}(F) = \emptyset.$$

The graph GFOLLOW is shown in Figure 7. We have

$$\begin{aligned}
\text{FOLLOW}(E) &= \text{INITFOLLOW}(E),\\
\text{FOLLOW}(T) &= \text{INITFOLLOW}(T) \cup \text{INITFOLLOW}(E) \cup \text{INITFOLLOW}(F),\\
\text{FOLLOW}(F) &= \text{INITFOLLOW}(F) \cup \text{INITFOLLOW}(T) \cup \text{INITFOLLOW}(E),
\end{aligned}$$

and so

$$\text{FOLLOW}(E) = \{+,), \$\}, \quad \text{FOLLOW}(T) = \{+, *,), \$\}, \quad \text{FOLLOW}(F) = \{+, *,), \$\}.$$

7 Algorithm $Traverse$

The input is a directed graph Gr having N nodes, and a family of initial sets $I[i]$, $1 \leq i \leq N$. We assume that a function $successors$ is available, which returns for each node n in the graph, the list $successors[n]$ of all immediate successors of n. The output is the list of sets $F[i]$, $1 \leq i \leq N$, solution of the system of recursive equations of Section 5. Hence,

$$F[i] = \bigcup \{I[j] \mid iG^* j\}, \quad 1 \leq i \leq N.$$

The procedure *Reachable* visits all nodes reachable from a given node. It uses a stack $STACK$ and a boolean array $VISITED$ to keep track of which nodes have been visited. The procedures *Reachable* and *traverse* are shown in Figure 8.

8 More on $LR(0)$-Characteristic Automata

Let $G = (V, \Sigma, P, S')$ be an augmented context-free grammar with augmented start production $S' \longrightarrow S\$$ (where S' only occurs in the augmented production). The righmost derivation relation is denoted by $\underset{rm}{\Longrightarrow}$.

Recall that the *set* C_G *of characteristic strings* for the grammar G is defined by

$$C_G = \{\alpha\beta \in V^* \mid S' \overset{*}{\underset{rm}{\Longrightarrow}} \alpha A v \underset{rm}{\Longrightarrow} \alpha\beta v,\, \alpha\beta \in V^*,\, v \in \Sigma^*\}.$$

The fundamental property of LR-parsing, due to D. Knuth, is stated in the following theorem:

Theorem 8.1 *Let G be a context-free grammar and assume that every nonterminal derives some terminal string. The language C_G (over V^*) is a regular language. Furthermore, a deterministic automaton DCG accepting C_G can be constructed from G.*

The construction of DCG can be found in various places, including the book on Compilers by Aho, Sethi and Ullman. We explained this construction in Section 1. The proof that the NFA NCG constructed as indicated in Section 1 is correct, i.e., that it accepts precisely C_G, is nontrivial, but not really hard either. This will be the object of a homework assignment!

```
**Procedure** $Reachable(Gr : graph; startnode : node; I : listofsets;$
                    $\textbf{var} F : listofsets);$
**var** $currentnode, succnode, i : node; STACK : stack;$
                    $VISITED$ : **array**$[1..N]$ **of boolean**;
  **begin**
    **for** $i := 1$ **to** $N$ **do**
      $VISITED[i] :=$ **false**;
    $STACK := EMPTY;$
    $push(STACK, startnode);$
    **while** $STACK \neq EMPTY$ **do**
      **begin**
        $currentnode := top(STACK); pop(STACK);$
        $VISITED[currentnode] :=$ **true**;
        **for each** $succnode \in successors(currentnode)$ **do**
          **if** $\neg VISITED[succnode]$ **then**
            **begin**
              $push(STACK, succnode);$
              $F[startnode] := F[startnode] \cup I[succnode]$
            **end**
      **end**
  **end**
```

The sets $F[i]$, $1 \leq i \leq N$, are computed as follows:

```
**begin**
  **for** $i := 1$ **to** $N$ **do**
    $F[i] := I[i];$
  **for** $startnode := 1$ **to** $N$ **do**
    $Reachable(Gr, startnode, I, F)$
**end**
```

Figure 8: Algorithm *traverse*

However, note a subtle point: The construction of NCG is only correct under the assumption that every nonterminal derives some terminal string. Otherwise, the construction could yield an NFA NCG accepting strings **not in** C_G.

Recall that the states of the characteristic automaton CGA are sets of *items* (or *marked productions*), where an item is a production with a dot anywhere in its right-hand side. Note that in constructing CGA, it is not necessary to include the state $\{S' \longrightarrow S\$.\}$ (the endmarker \$ is only needed to compute the lookahead sets). If a state p contains a marked production of the form $A \longrightarrow \beta.$, where the dot is the rightmost symbol, state p is called a *reduce state* and $A \longrightarrow \beta$ is called a *reducing production* for p. Given any state q, we say that a string $\beta \in V^*$ *accesses* q if there is a path from some state p to the state q on input β in the automaton CGA. Given any two states $p, q \in CGA$, for any $\beta \in V^*$, if there is a sequence of transitions in CGA from p to q on input β, this is denoted by

$$p \xrightarrow{\beta} q.$$

The initial state which is the closure of the item $S' \longrightarrow .S\$$ is denoted by 1. The LALR(1)-lookahead sets are defined in the next section.

9 LALR(1)-Lookahead Sets

For any reduce state q and any reducing production $A \longrightarrow \beta$ for q, let

$$\mathrm{LA}(q, A \longrightarrow \beta) = \{a \mid a \in \Sigma,\ S' \overset{*}{\underset{rm}{\Longrightarrow}} \alpha A a v \underset{rm}{\Longrightarrow} \alpha\beta a v,\ \alpha, \beta \in V^*,\ v \in \Sigma^*,\ \alpha\beta\ \textit{accesses}\ q\}.$$

In words, $LA(q, A \longrightarrow \beta)$ consists of the terminal symbols for which the reduction by production $A \longrightarrow \beta$ in state q is the correct action (that is, for which the parse will terminate successfully). The LA sets can be computed using the FOLLOW sets defined below.

For any state p and any nonterminal A, let

$$\mathrm{FOLLOW}(p, A) = \{a \mid a \in \Sigma,\ S' \overset{*}{\underset{rm}{\Longrightarrow}} \alpha A a v,\ \alpha \in V^*,\ v \in \Sigma^* \text{ and } \alpha \text{ accesses } p\}.$$

Since for any derivation

$$S' \overset{*}{\underset{rm}{\Longrightarrow}} \alpha A a v \underset{rm}{\Longrightarrow} \alpha\beta a v$$

where $\alpha\beta$ accesses q, there is a state p such that $p \xrightarrow{\beta} q$ and α accesses p, it is easy to see that the following result holds:

Proposition 9.1 *For every reduce state q and any reducing production $A \longrightarrow \beta$ for q, we have*

$$\mathrm{LA}(q, A \longrightarrow \beta) = \bigcup \{\mathrm{FOLLOW}(p, A) \mid p \xrightarrow{\beta} q\}.$$

Also, we let

$$\text{LA}(\{S' \longrightarrow S.\$\}, S' \longrightarrow S\$) = \text{FOLLOW}(1, S).$$

Intuitively, when the parser makes the reduction by production $A \longrightarrow \beta$ in state q, each state p as above is a possible top of stack after the states corresponding to β are popped. Then the parser must read A in state p, and the next input symbol will be one of the symbols in FOLLOW(p, A).

The computation of FOLLOW(p, A) is similar to that of FOLLOW(A). First, we compute INITFOLLOW(p, A), given by

$$\text{INITFOLLOW}(p, A) = \{a \mid a \in \Sigma,\ \exists q, r,\ p \xrightarrow{A} q \xrightarrow{a} r\}.$$

These are the terminals that can be read in CGA after the "goto transition" on nonterminal A has been performed from p. These sets can be easily computed from CGA.

Note that for the state p whose core item is $S' \longrightarrow S.\$$, we have

$$\text{INITFOLLOW}(p, S) = \{\$\}.$$

Next, observe that if $B \longrightarrow \alpha A$ is a production and if

$$S' \underset{rm}{\overset{*}{\Longrightarrow}} \lambda B a v$$

where λ accesses p', then

$$S' \underset{rm}{\overset{*}{\Longrightarrow}} \lambda B a v \underset{rm}{\Longrightarrow} \lambda\alpha A a v$$

where λ accesses p' and $p' \xrightarrow{\alpha} p$. Hence $\lambda\alpha$ accesses p and

$$\text{FOLLOW}(p', B) \subseteq \text{FOLLOW}(p, A)$$

whenever there is a production $B \longrightarrow \alpha A$ and $p' \xrightarrow{\alpha} p$. From this, the following recursive equations are easily obtained: For all p and all A,

$$\text{FOLLOW}(p, A) = \text{INITFOLLOW}(p, A) \cup \bigcup \{\text{FOLLOW}(p', B) \mid B \longrightarrow \alpha A \in P,\ \alpha \in V^* \text{ and } p' \xrightarrow{\alpha} p\}.$$

From Section 5, we know that these sets can be computed by using the algorithm *traverse*. All we need is to compute the graph GLA.

The nodes of the graph GLA are the pairs (p, A), where p is a state and A is a nonterminal. There is an edge from (p, A) to (p', B) if and only if there is a production of the form $B \longrightarrow \alpha A$ in P for some $\alpha \in V^*$ and $p' \xrightarrow{\alpha} p$ in CGA. Note that it is only necessary to consider nodes (p, A) for which there is a nonterminal transition on A from p. Such pairs can be obtained from the parsing table. Also, using the *spelling property*, that is, the fact that all transitions entering a given state have the same label, it is possible to compute the relation *lookback* defined as follows:

$$(q, A)\ \mathit{lookback}\ (p, A) \quad \text{iff} \quad p \xrightarrow{\beta} q$$

for some reduce state q and reducing production $A \longrightarrow \beta$. The above considerations show that the FOLLOW sets of Section 6 are obtained by ignoring the state component from FOLLOW(p, A). We now consider the changes that have to be made when ϵ-rules are allowed.

10 Computing FIRST, FOLLOW and LA$(q, A \longrightarrow \beta)$ in the Presence of ϵ-Rules

First, it is necessary to compute the set E of *erasable nonterminals*, that is, the set of nonterminals A such that $A \overset{+}{\Longrightarrow} \epsilon$.

We let E be a boolean array and *change* be a boolean flag. An algorithm for computing E is shown in Figure 9. Then, in order to compute FIRST, we compute

$$\begin{aligned}\text{INITFIRST}(A) = \{a \mid a \in \Sigma,\ &A \longrightarrow a\alpha \in P, \text{ or} \\ &A \longrightarrow A_1 \cdots A_k a\alpha \in P, \text{ for some } \alpha \in V^*, \text{ and } E(A_1) = \cdots = E(A_k) = \mathbf{true}\}.\end{aligned}$$

The graph GFIRST is obtained as follows: The nodes are the nonterminals, and there is an edge from A to B if and only if either there is a production $A \longrightarrow B\alpha$, or a production $A \longrightarrow A_1 \cdots A_k B\alpha$, for some $\alpha \in V^*$, with $E(A_1) = \cdots = E(A_k) = \mathbf{true}$. Then, we extend FIRST to strings in V^+, in the obvious way. Given any string $\alpha \in V^+$, if $|\alpha| = 1$, then $\beta = X$ for some $X \in V$, and

$$\text{FIRST}(\beta) = \text{FIRST}(X)$$

as before, else if $\beta = X_1 \cdots X_n$ with $n \geq 2$ and $X_i \in V$, then

$$\text{FIRST}(\beta) = \text{FIRST}(X_1) \cup \cdots \cup \text{FIRST}(X_k),$$

where k, $1 \leq k \leq n$, is the largest integer so that

$$E(X_1) = \cdots = E(X_k) = \mathbf{true}.$$

To compute FOLLOW, we first compute

$$\text{INITFOLLOW}(A) = \{a \mid a \in \Sigma,\ B \longrightarrow \alpha A\beta \in P,\ \alpha \in V^*,\ \beta \in V^+, \text{ and } a \in \text{FIRST}(\beta)\}.$$

The graph GFOLLOW is computed as follows: The nodes are the nonterminals. There is an edge from A to B if either there is a production of the form $B \longrightarrow \alpha A$, or $B \longrightarrow \alpha A A_1 \cdots A_k$, for some $\alpha \in V^*$, and with $E(A_1) = \cdots = E(A_k) = \mathbf{true}$.

```
begin
    for each nonterminal $A$ do
      $E(A) :=$ false;
    for each nonterminal $A$ such that $A \longrightarrow \epsilon \in P$ do
      $E(A) :=$ true;
    $change :=$ true;
    while $change$ do
      begin
        $change :=$ false;
        for each $A \longrightarrow A_1 \cdots A_n \in P$
            s.t. $E(A_1) = \cdots = E(A_n) =$ true do
          if $E(A) =$ false then
            begin
              $E(A) :=$ true;
              $change :=$ true
            end
      end
end
```

Figure 9: Algorithm for computing E

The computation of the LALR(1) lookahead sets is also more complicated because another graph is needed in order to compute INITFOLLOW(p, A). First, the graph GLA is defined in the following way: The nodes are still the pairs (p, A), as before, but there is an edge from (p, A) to (p', B) if and only if either there is some production $B \longrightarrow \alpha A$, for some $\alpha \in V^*$ and $p' \xrightarrow{\alpha} p$, or a production $B \longrightarrow \alpha A\beta$, for some $\alpha \in V^*$, $\beta \in V^+$, $\beta \overset{+}{\Longrightarrow} \epsilon$, and $p' \xrightarrow{\alpha} p$. The sets INITFOLLOW$(p, A)$ are computed in the following way: First, let

$$\mathrm{DR}(p, A) = \{a \mid a \in \Sigma,\ \exists q, r,\ p \xrightarrow{A} q \xrightarrow{a} r\}.$$

The sets DR(p, A) are the *direct read* sets. Note that for the state p whose core item is $S' \longrightarrow S.\$$, we have

$$\mathrm{DR}(p, S) = \{\$\}.$$

Then,

$$\begin{aligned}&\mathrm{INITFOLLOW}(p, A) = \mathrm{DR(p, A)} \cup \\ &\qquad \bigcup \{a \mid a \in \Sigma,\ S' \underset{rm}{\overset{*}{\Longrightarrow}} \alpha A\beta av \underset{rm}{\Longrightarrow} \alpha Aav,\ \alpha \in V^*,\ \beta \in V^+,\ \beta \overset{+}{\Longrightarrow} \epsilon,\ \alpha\ \textit{accesses}\ p\}.\end{aligned}$$

The set INITFOLLOW(p, A) is the set of terminals that can be read before any handle containing A is reduced. The graph $GREAD$ is defined as follows: The nodes are the pairs (p, A), and there is an edge from (p, A) to (r, C) if and only if $p \xrightarrow{A} r$ and $r \xrightarrow{C} s$, for some s, with $E(C) = \mathbf{true}$.

Then, it is not difficult to show that the INITFOLLOW sets are the least solution of the set of recursive equations:

$$\mathrm{INITFOLLOW}(p, A) = DR(p, A) \cup \bigcup \{\mathrm{INITFOLLOW}(r, C) \mid (p, A)\ GREAD\ (r, C)\}.$$

Hence the INITFOLLOW sets can be computed using the algorithm traverse on the graph $GREAD$ and the sets $DR(p, A)$, and then, the FOLLOW sets can be computed using traverse again, with the graph GLA and sets INITFOLLOW. Finally, the sets LA$(q, A \longrightarrow \beta)$ are computed from the FOLLOW sets using the graph *lookback*.

From section 5, we note that $F(i) = F(j)$ whenever there is a path from i to j and a path from j to i, that is, whenever i and j are *strongly connected*. Hence, the solution of the system of recursive equations can be computed more efficiently by finding the maximal strongly connected components of the graph G, since F has a same value on each strongly connected component. This is the approach followed by DeRemer and Pennello in: Efficient Computation of LALR(1) Lookahead sets, by F. DeRemer and T. Pennello, *TOPLAS*, Vol. 4, No. 4, October 1982, pp. 615-649.

We now give an example of grammar which is LALR(1) but not SLR(1).

Example 3. The grammar G_2 is given by:

$$S' \longrightarrow S\$$$

$$\begin{array}{rcl}
S & \longrightarrow & L = R \\
S & \longrightarrow & R \\
L & \longrightarrow & *R \\
L & \longrightarrow & id \\
R & \longrightarrow & L
\end{array}$$

The states of the characteristic automaton CGA_2 are:

$$\begin{array}{rcl}
1: S' & \longrightarrow & .S\$ \\
S & \longrightarrow & .L = R \\
S & \longrightarrow & .R \\
L & \longrightarrow & . * R \\
L & \longrightarrow & .id \\
R & \longrightarrow & .L \\
2: S' & \longrightarrow & S.\$ \\
3: S & \longrightarrow & L. = R \\
R & \longrightarrow & L. \\
4: S & \longrightarrow & R. \\
5: L & \longrightarrow & *.R \\
R & \longrightarrow & .L \\
L & \longrightarrow & . * R \\
L & \longrightarrow & .id \\
6: L & \longrightarrow & id. \\
7: S & \longrightarrow & L = .R \\
R & \longrightarrow & .L \\
L & \longrightarrow & . * R \\
L & \longrightarrow & .id \\
8: L & \longrightarrow & *R. \\
9: R & \longrightarrow & L. \\
10: S & \longrightarrow & L = R.
\end{array}$$

We find that

$$\begin{array}{rcl}
\text{INITFIRST}(S) & = & \emptyset \\
\text{INITFIRST}(L) & = & \{*, id\} \\
\text{INITFIRST}(R) & = & \emptyset.
\end{array}$$

The graph GFIRST is shown in Figure 10.

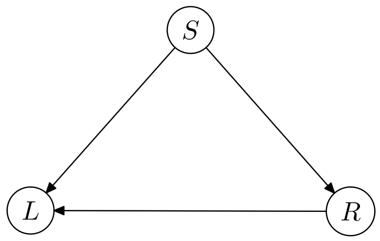

Figure 10: The graph GFIRST

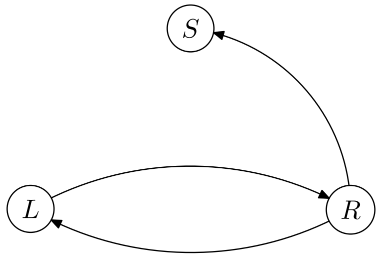

Figure 11: The graph GFOLLOW

Then, we find that

$$\begin{array}{rcl} \text{FIRST}(S) & = & \{*, id\} \\ \text{FIRST}(L) & = & \{*, id\} \\ \text{FIRST}(R) & = & \{*, id\}. \end{array}$$

We also have

$$\begin{array}{rcl} \text{INITFOLLOW}(S) & = & \{\$\} \\ \text{INITFOLLOW}(L) & = & \{=\} \\ \text{INITFOLLOW}(R) & = & \emptyset. \end{array}$$

The graph GFOLLOW is shown in Figure 11.
Then, we find that

$$\begin{array}{rcl} \text{FOLLOW}(S) & = & \{\$\} \\ \text{FOLLOW}(L) & = & \{=, \$\} \\ \text{FOLLOW}(R) & = & \{=, \$\}. \end{array}$$

Note that there is a shift/reduce conflict in state 3 on input $=$, since there is a shift on input $=$ (since $S \longrightarrow L. = R$ is in state 3), and a reduce for $R \rightarrow L$, since $=$ is in FOLLOW(R). However, as we shall see, the conflict is resolved if the LALR(1) lookahead sets are computed.

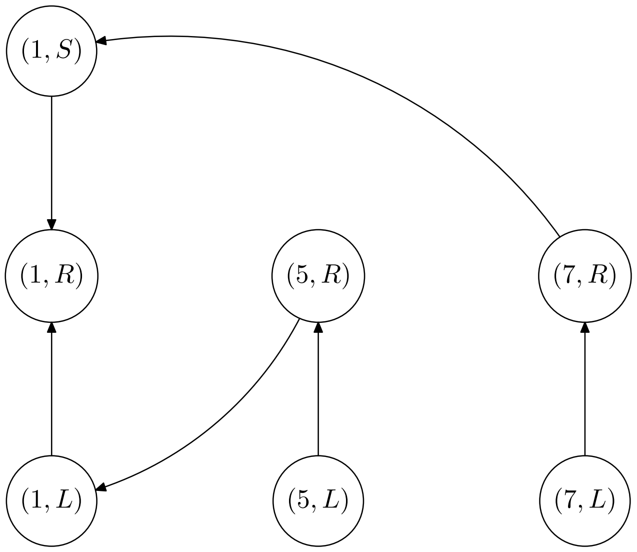

Figure 12: The graph GLA

The graph GLA is shown in Figure 12.
We get the following INITFOLLOW and FOLLOW sets:

$$\begin{array}{rcrc}
\mathrm{INITFOLLOW}(1,S) = & \{\$\} & \mathrm{INITFOLLOW}(1,S) = & \{\$\} \\
\mathrm{INITFOLLOW}(1,R) = & \emptyset & \mathrm{INITFOLLOW}(1,R) = & \{\$\} \\
\mathrm{INITFOLLOW}(1,L) = & \{=\} & \mathrm{INITFOLLOW}(1,L) = & \{=,\$\} \\
\mathrm{INITFOLLOW}(5,R) = & \emptyset & \mathrm{INITFOLLOW}(5,R) = & \{=,\$\} \\
\mathrm{INITFOLLOW}(5,L) = & \emptyset & \mathrm{INITFOLLOW}(5,L) = & \{=,\$\} \\
\mathrm{INITFOLLOW}(7,R) = & \emptyset & \mathrm{INITFOLLOW}(7,R) = & \{\$\} \\
\mathrm{INITFOLLOW}(7,L) = & \emptyset & \mathrm{INITFOLLOW}(7,L) = & \{\$\}.
\end{array}$$

Thus, we get

$$\begin{array}{rcl}
\mathrm{LA}(2, S' \longrightarrow S\$) & = & \mathrm{FOLLOW}(1,S) = \{\$\} \\
\mathrm{LA}(3, R \longrightarrow L) & = & \mathrm{FOLLOW}(1,R) = \{\$\} \\
\mathrm{LA}(4, S \longrightarrow R) & = & \mathrm{FOLLOW}(1,S) = \{\$\} \\
\mathrm{LA}(6, L \longrightarrow id) & = & \mathrm{FOLLOW}(1,L) \cup \mathrm{FOLLOW}(5,L) \cup \mathrm{FOLLOW}(7,L) = \{=,\$\} \\
\mathrm{LA}(8, L \longrightarrow *R) & = & \mathrm{FOLLOW}(1,L) \cup \mathrm{FOLLOW}(5,L) \cup \mathrm{FOLLOW}(7,L) = \{=,\$\} \\
\mathrm{LA}(9, R \longrightarrow L) & = & \mathrm{FOLLOW}(5,R) \cup \mathrm{FOLLOW}(7,R) = \{=,\$\} \\
\mathrm{LA}(10, S \longrightarrow L = R) & = & \mathrm{FOLLOW}(1,S) = \{\$\}.
\end{array}$$

Since $\mathrm{LA}(3, R \longrightarrow L)$ does not contain $=$, the conflict is resolved.

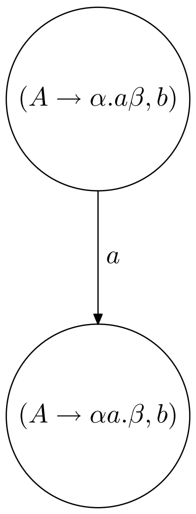

Figure 13: Transition on terminal input a

11 $LR(1)$-Characteristic Automata

We conclude this brief survey on LR-parsing by describing the construction of $LR(1)$-parsers. The new ingredient is that when we construct an NFA accepting C_G, we incorporate lookahead symbols into the states. Thus, a state is a pair $(A \longrightarrow \alpha.\beta, b)$, where $A \longrightarrow \alpha.\beta$ is a marked production, as before, and $b \in \Sigma \cup \{\$\}$ is a *lookahead symbol*. The new twist in the construction of the nondeterministic characteristic automaton is the following:

The start state is $(S' \to .S, \$)$, and the transitions are defined as follows:

(a) For every terminal $a \in \Sigma$, then there is a transition on input a from state $(A \to \alpha.a\beta, b)$ to the state $(A \to \alpha a.\beta, b)$ obtained by "shifting the dot" (where $a = b$ is possible). Such a transition is shown in Figure 13.

(b) For every nonterminal $B \in N$, there is a transition on input B from state $(A \to \alpha.B\beta, b)$ to state $(A \to \alpha B.\beta, b)$ (obtained by "shifting the dot"), and transitions on input ϵ (the empty string) to all states $(B \to .\gamma, a)$, for all productions $B \to \gamma$ with left-hand side B and all $a \in \mathrm{FIRST}(\beta b)$. Such transitions are shown in Figure 14.

(c) A state is *final* if and only if it is of the form $(A \to \beta., b)$ (that is, the dot is in the rightmost position).

Example 3. Consider the grammar G_3 given by:

$$\begin{aligned} 0\colon S &\longrightarrow E \\ 1\colon E &\longrightarrow aEb \\ 2\colon E &\longrightarrow \epsilon \end{aligned}$$

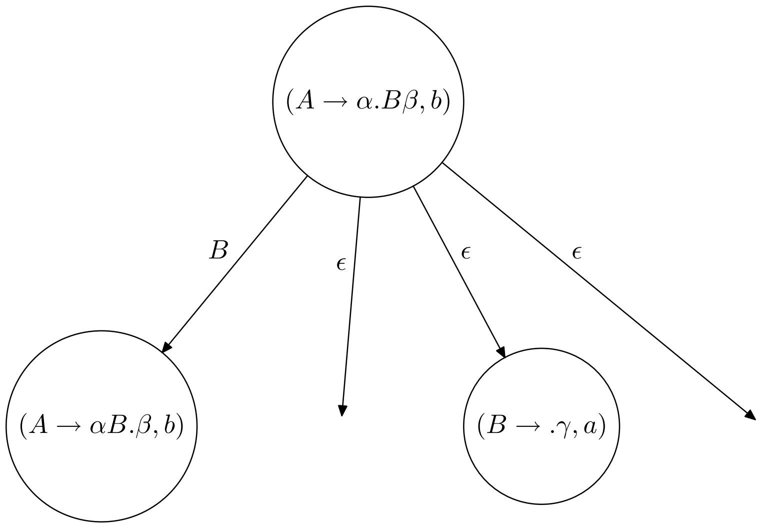

Figure 14: Transitions from a state $(A \to \alpha.B\beta, b)$

The result of making the NFA for C_{G_3} deterministic is shown in Figure 15 (where transitions to the "dead state" have been omitted). The internal structure of the states $1, \ldots, 8$ is shown below:

$$\begin{array}{rcl}
1: S & \longrightarrow & .E, \$ \\
E & \longrightarrow & .aEb, \$ \\
E & \longrightarrow & ., \$ \\
2: E & \longrightarrow & a.Eb, \$ \\
E & \longrightarrow & .aEb, b \\
E & \longrightarrow & ., b \\
3: E & \longrightarrow & a.Eb, b \\
E & \longrightarrow & .aEb, b \\
E & \longrightarrow & ., b \\
4: E & \longrightarrow & aE.b, \$ \\
5: E & \longrightarrow & aEb., \$ \\
6: E & \longrightarrow & aE.b, b \\
7: E & \longrightarrow & aEb., b \\
8: S & \longrightarrow & E., \$
\end{array}$$

The $LR(1)$-shift/reduce parser associated with DCG is built as follows: The shift and goto entries come directly from the transitions of DCG, and for every state s, for every item

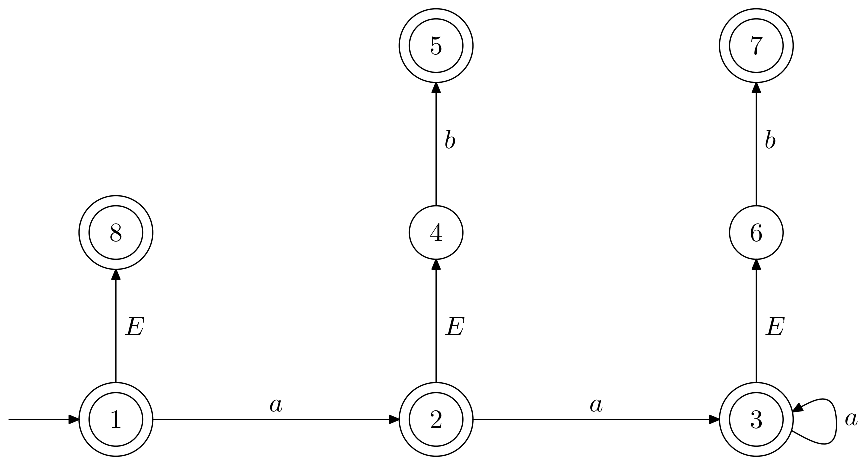

Figure 15: DFA for C_{G_3}

$(A \longrightarrow \gamma, b)$ in s, enter an entry rn for state s and input b, where $A \longrightarrow \gamma$ is production number n. If the resulting parser has no conflicts, we say that the grammar is an $LR(1)$ grammar. The $LR(1)$-shift/reduce parser for G_3 is shown below:

	a	b	$\$$	E
1	$s2$		$r2$	8
2	$s3$	$r2$		4
3	$s3$	$r2$		6
4		$r5$		
5			$r1$	
6	$r1$	$s7$		
7		$r1$		
8			acc	

Observe that there are three pairs of states, $(2, 3)$, $(4, 6)$, and $(5, 7)$, where both states in a common pair only differ by the lookahead symbols. We can merge the states corresponding to each pair, because the marked items are the same, but now, we have to allow lookahead sets. Thus, the merging of $(2, 3)$ yields

$$\begin{aligned} 2'\colon E &\longrightarrow a.Eb, \{b, \$\} \\ E &\longrightarrow .aEb, \{b\} \\ E &\longrightarrow ., \{b\}, \end{aligned}$$

the merging of $(4, 6)$ yields

$$3'\colon E \longrightarrow aE.b, \{b, \$\},$$

the merging of $(5, 7)$ yields

$$4'\colon E \longrightarrow aEb., \{b, \$\}.$$

We obtain a merged DFA with only five states, and the corresponding shift/reduce parser is given below:

	a	b	$\$$	E
1	$s2'$		$r2$	8
$2'$	$s2'$	$r2$		$3'$
$3'$		$s4'$		
$4'$		$r1$	$r1$	
8			acc	

The reader should verify that this is the $LALR(1)$-parser. The reader should also check that that the $SLR(1)$-parser is given below:

	a	b	$\$$	E
1	$s2$	$r2$	$r2$	5
2	$s2$	$r2$	$r2$	3
3		$s4$		
4		$r1$	$r1$	
5			acc	

The difference between the two parsing tables is that the $LALR(1)$-lookahead sets are sharper than the $SLR(1)$-lookahead sets. This is because the computation of the $LALR(1)$-lookahead sets uses a sharper version of FOLLOW sets. It can also be shown that if a grammar is $LALR(1)$, then the merging of states of an $LR(1)$-parser always succeeds and yields the $LALR(1)$ parser. Of course, this is a very inefficient way of producing $LALR(1)$ parsers, and much better methods exist, such as the graph method described in these notes. However, there are cases where the merging fails. Sufficient conditions for successful merging have been investigated, but there is still room for research in this area.