4.3. Visiting Trees While Looking for Good Nodes

In this section we will present some algorithms for visiting trees while looking for nodes which satisfy a given predicate, say p. We assume that the trees are generated in a dynamic fashion, that is, every tree is given by its root node and a function, say f, which for each node returns the list of its son-nodes. Any such function f is also called a *tree-generating* function.

If we assume that every node of a tree is of type α we have that:
(i) the predicate p is a function from α to *bool*, where *bool* $= \{true, \ false\}$, and
(ii) the tree-generating function f is a function from α to α *list*, where α *list* denotes the type of the lists whose elements are of type α.

In what follows the infix append function between two lists is also denoted by <>. Thus, $<>: (\alpha\, list) \times (\alpha\, list) \ \rightarrow \ (\alpha\, list)$.

4.3.1. Depth-first Visit. Version 1. In this section we present a function, called existsev (short for *exists eventually*), which given a predicate p, a tree-generating function f, and a list L of nodes, returns *true* if there exists a node n in L which is the root of a tree t_n generated by the function f, such that in t_n there exists a node m such that $p(m) = true$, otherwise it returns *false*.

The correctness of existsev is based on the assumption that for any node n the tree which is rooted in n is finite. This finiteness assumption is required because given any node n, the tree rooted in n is generated and visited in a *depth-first* manner (see the line marked with (†)).

Thus, if a node is the root of an infinite tree (because of the particular tree-generating function which is given), then the evaluation of existsev may not terminate even if in that infinite tree there exists a node which satisfies p.

The existsev function is as follows. In the first line we give first the type of the function and in the following lines we give its definition.

$$
\begin{array}{l}
\text{existsev} : (\alpha \rightarrow bool) \times (\alpha \rightarrow \alpha\, list) \times (\alpha\, list) \ \rightarrow \ bool \\
\text{existsev } p\, f\, L = \text{if } L\!=\![\,] \text{ then } false \\
\qquad\qquad\qquad\quad \text{else if } p(hd(L)) \text{ then } true \\
\qquad\qquad\qquad\quad \text{else existsev } p\, f \ \ (f(hd(L)) <> tl(L)) \qquad (\dagger)
\end{array}
$$

This function existsev can be used for visiting trees as follows. Suppose we are given a predicate p, a tree-generating function f, and a node n. Suppose also that f generates a finite tree rooted in n. Then, there exists a node m in the tree rooted in n generated by f such that $p(m)$ is *true* iff existsev$(p, f, [n]) = true$.

The inductive proof of this statement is left to the reader.

4.3.2. Depth-first Visit. Version 2. In this section we present a different algorithm for visiting trees in a depth-first manner. It has been suggested to us by Prof. R. M. Burstall.

As in the previous Section 4.3.1, we are given a predicate p, a tree-generating function f, and a node n. The following function existsev1(p, f, n) returns *true* if in the tree rooted in n generated from n by the function f there exists a node m such that $p(m) = true$, otherwise it returns *false*.

As for the function existsev of the previous section, the correctness of the existsev1 function is based on the assumption that for any node n, the tree rooted in n, is finite. This assumption is necessary because the tree rooted in n is generated and visited in a *depth-first* manner (see the line marked with (††)).

Given a predicate p and a list L of nodes, the function exists(p, L) returns *true* if in the list L there exists a node m such that $p(m)$ is *true*, otherwise it returns *false*.

exists : $(\alpha \to bool) \times (\alpha\, list) \to bool$
exists p L = if $L = [\,]$ then *false*
    else if $p(hd(L))$ then *true*
    else exists p $tl(L)$

existsev1 : $(\alpha \to bool) \times (\alpha \to \alpha\, list) \times \alpha \to bool$
existsev1 p f x = if $p(x)$ then *true*
    else exists (existsev1 p f) $f(x)$ (††)

Notice that in the above function we use the partial application tecnique: the predicate which is the first argument of exists in the line marked with (††), is the result of the application of the predicate p and the function f to the function existsev1 which is of arity 3. Thus, at line (††) the type of existsev $p\, f$ is $\alpha \to bool$, which is the type of a predicate on nodes, and the type of $f(x)$ is that of a list of α's, that is, $\alpha\, list$.

NOTE 4.3.1. The functions existsev and existsev1 are equivalent in the sense that for any given predicate p from α to *bool*, any tree generating function f from α to an α list, and any given initial node n of type α, we have that:
existsev p f $[n]$ = existsev1 p f n. □

As we will see in more detail in Chapter 5, one can use this depth-first visit algorithm for parsing. Here is an example where we consider the regular grammar G whose productions are:

$$\begin{array}{lll} P & \to & b \\ P & \to & bQ \\ Q & \to & a \\ Q & \to & aQ \end{array}$$

and whose axiom is P. We want to check whether or not the string ba belongs to the language of G. We can do so by constructing a tree whose root node is the pair (P, ba) of the axiom P and the string to parse ba. The tree-generating function

f generates the list of the son-nodes of any given node (*sentential-form*, *string-to-parse*) by applying the productions of the first symbol of the *sentential-form*. For instance, given the node $(P,\ ba)$, the list of the son-nodes is $[(b, ba),\ (bQ, ba)]$. The predicate p is the one which is *true* for a node of the form $(\varepsilon,\ \varepsilon)$. The following Figure 4.3.1 shows a tree of nodes with root $(P,\ ba)$. It indicates that the string ba belongs to the language of G because there is a node $(\varepsilon,\ \varepsilon)$.

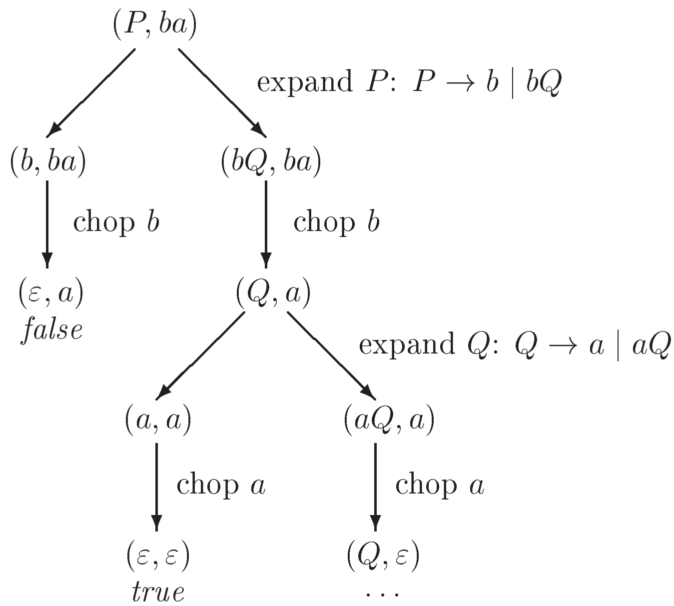

FIGURE 4.3.1. The tree of nodes which shows that the string ba is generated by the grammar G whose productions are: $P \to b \mid bQ$, $Q \to a \mid aQ$.

4.3.3. Breadth-first Visit. In this section we present a third algorithm for visiting trees. The visit is performed in a *breadth-first* manner so that we do not need for this algorithm the hypothesis that the tree generated from any given node by the given tree-generating function, is finite.

Given a function f from an element of type α to a list of α's and a list L of elements of type α, the function flatmap(f, L) returns the concatenation of the lists produced by applying the function f to every element of L.

The function exists is the one we have presented in the previous Section 4.3.2.

The function bf-existsev (short for *breadth-first exists eventually*) is very similar to the function existsev of Section 4.3.1. They have the same type. However, in the case of the function bf-existsev, the tree is generated and visited in a breadth-first manner (see line marked with (†††)). Indeed, at each recursive call of the function bf-existsev, we construct the list of son-nodes of all nodes in the list of the previous call of bf-existsev. Thus, the tree is generated *level-by-level*, by considering the nodes at distance $k{+}1$ from the root, only after all nodes at distance k, for any $k \geq 0$.

The functions flatmap, exists, and bf-existsev are defined as follows.

flatmap : $(\alpha \to \alpha\, list) \times (\alpha\, list) \to (\alpha\, list)$
flatmap $f\ L =$ if $L{=}[\,]$ then $[\,]$
else $(f(hd(L)) <> (\text{flatmap}\, f\ tl(L))$

exists : $(\alpha \to bool) \times (\alpha\, list) \to bool$
exists $p\ L =$ if $L{=}[\,]$ then $false$
else if $p(hd(L))$ then $true$
else exists $p\ tl(L)$

bf-existsev : $(\alpha \to bool) \times (\alpha \to \alpha\, list) \times (\alpha\, list) \to bool$
bf-existsev $p\ f\ L =$ if $L{=}[\,]$ then $false$
else if exists $p\ L$ then $true$
else bf-existsev $p\ f$ (flatmap $f\ L$) (†††)

Notice that the functions existsev and bf-existsev may not terminate, if starting from a given node, the iterated applications of the function f produce an infinite tree.

Exercise 4.3.2. Show that the following function definition of existsev is not correct:
existsev $p\ f\ L =$ if exists $p\ L$ then $true$
else existsev $p\ f\ (f(hd(L)) <> tl(L))$
Hint: Consider the case when $L{=}[\,]$.
□

Exercise 4.3.3. Show that the following function definition for bf-existsev is not correct:
bf-existsev $p\ f\ L =$ if $L{=}[\,]$ then $false$
else if $p(hd(L))$ then $true$
else bf-existsev $p\ f$ (flatmap $f\ L$)
Hint: Some nodes are used for generating their son-nodes, but never tested by p.
□

Exercise 4.3.4. Show that the following function definition for bf-existsev is not correct:
bf-existsev $p\ f\ L =$ if exists $p\ L$ then $true$
else bf-existsev $p\ f$ (flatmap $f\ L$)
Hint: Consider the case when $L{=}[\,]$.
□

CHAPTER 5

Chop-and-Expand Parser for Context-Free Languages

In this chapter we illustrate the Chop-and-Expand parser for context-free languages in a functional, an imperative language such as Java 1.5, and a logic language. The idea of this parser is to perform a depth-first visit of tree-like search space as indicated in Chapter 4.

5.1. Chop-and-Expand Context-Free Parser in a functional language

The varia ble 'sentform' contains the value of a sentential form. ε stands for the empty string. Any char list is also written as the string. For instance, [a,b,c] is also written as: abc.
The char list [a], made out of the character a only, is also written as the string: a.
Thus, $hd(\text{abc}) = \text{a}$ and $tl(\text{abc}) = \text{bc}$.

sentform *sf* : char list
string *s* : char list

state : ⟨sentform *sf*, string *s*⟩ pair	Example: ⟨PR, bb⟩
stl : state list	Example: [⟨PR, bb⟩ ⟨bPQ, ab⟩]
right-hand-sides *rhs* : nonterminal → string list	Example: given P → b | aQQ we have: *rhs*(P) = [b, aQQ]
grammar *gr* : ⟨symbol, right-hand-sides⟩ list	Example: [⟨P, [b, aQQ]⟩ ⟨Q, [ε, b, bP]⟩]

Given P → b | aQQ and *stl* = [⟨PR, bb⟩ ⟨bPQ, ab⟩],
after the execution of *expand*(*rhs*(P), R, bb, *new-stl*), we get:

new-stl=[⟨bR, bb⟩ ⟨aQQR, bb⟩ ⟨bPQ, ab⟩]

$exists\ stl\ gr =$
if stl=[ ] then $false$
else let $hd(stl) = \langle sf, s\rangle$ in
    if $sf = \varepsilon$ and $s = \varepsilon$ then $true$
    else if $sf \neq \varepsilon$ and $nonterminal(hd(sf))$ then
        begin // EXPAND
        $expand(rhs(hd(sf)),\ tl(sf),\ s,\ new\text{-}stl)$;
        $exists\ new\text{-}stl\ gr$
        end
(1) else if $sf \neq \varepsilon$ and $s \neq \varepsilon$ and $terminal(hd(sf))$ and $hd(sf) = hd(s)$ then // CHOP
        $exists\ (\langle tl(sf), tl(s)\rangle : tl(stl))\ gr$
    else $exists\ \ tl(stl)\ \ gr$

(1) ‘and $terminal(hd(sf))$’ is not necessary.

The initial call of $exists\ \ stl\ \ gr$
is of the form: $exists$ [$\langle S$, string_to_parse$\rangle$] given_grammar
where S is the axiom of the given_grammar.

5.2. Chop-and-Expand Context-Free Parser in Java

In this section we list the two programs: `List.java` and `CFParser.java`, which realize the context-free parsing using the Chop-and-Expand idea illustrated in the Section 5.1. The C++ versions of these programs can be found in [**10**, page 69].

```
/** =========================================================================
 *                    PARSER FOR A CONTEXT FREE GRAMMAR
 * Thanks to R. Burstall and E.W. Dijkstra. "the chop-and-expand parser".
 * -------------------------------------------------------------------------
 *                          --- TYPES ---
 *  List<char>  = String
 *  State       = <sentential_form, string>
 *  List<State> = list of states
 *  Rsides      = list of String : list of right_hand_sides
 *  SyRs        = <Sy,Rs> = <char:symbol, Rsides:Rs = its right_hand_sides>
 *  Gram        = List<SyRs> : list of SyRs pairs (type of a grammar)
 * -------------------------------------------------------------------------
 * We use GENERIC TYPES to avoid the redefinition of isEmpty(),
 * makeEmpty(), head(), tail(), cons() functions for:
 *
 * - list of String (type: List<String>). It is the type of the
 *                   right-hand-sides. It is also called: Rsides.
 * - list of States (type: List<State>)
 * - list of SyRs   (type: List<SyRs>). It is the type of every grammar.
 *                   It is also called: Gram.
 *
 * Different functions are used for different types only for printing and
 * appending lists.
 * -------------------------------------------------------------------------
 * The input grammar is given as a string named 'gg' of the type 'Gram',
 * according to the following productions:
 *
 * character ::= 'a'..'z'  | 'A'..'Z'          ('e' is reserved: see below)
 * charlist  ::= character | character charlist  (that is, String)
 * Rsides    ::= 'e' | charlist | charlist '|' Rsides
 * SyRs      ::= char '>' Rsides
 * grammar   ::= syRs | syRs ';' grammar
 *
 * The empty productions, also called e-rules, are allowed.
 * The character 'e' is reserved for denoting the right hand side of an
 * empty production, that is, the empty charlist. For instance:
 * gg="P>b|aQQ;Q>e|b|bP" stands for: P->b, P->aQQ, Q->e, Q->b, Q->bP
 * In  P>b|aQQ;Q>e|b|bP:
 *         \_/     aQQ is a string: "aQQ"
 *       \___/   b|aQQ is a list of String
 *                         i.e., an Rsides: ("b", "aQQ")
 *     \_____/ P>b|aQQ is a SyRs, i.e., a pair of a symbol and its rsides
 *                         <P, ("b", "aQQ")>
 * When writing the input grammar, the productions relative to the same
 * non-terminal should be grouped together. For instance, we have to write
 * "P>a|aQQ" (not "P>a;P>aQQ").
 * The string to be parsed is the string named 'stp', and
 * stp is either a non-empty sequence of characters in 'a'..'z' or
 * stp is the empty sequence named 'e'.
 * -------------------------------------------------------------------------
 * The grammar should not be left recursive (that is, without productions
 * of the form P>P...). This is not a restriction, because any left
 * recursive context-free grammar can be transformed in an equivalent one
 * which is not left recursive.(see:Lemma 4.3 of Hopcroft-Ullmann: Formal
```

```
 * Languages and Their Relation to Automata, Addison Wesley, 1969)
 * ------------------------------------------------------------------------
 * The function 'parseString' is used both for:
 *  (i)  getting one of the alternatives of the right hand sides of the
 *       productions, and
 *  (ii) generating the list of characters 'stringtoparse' from the string
 *       'stp'.
 * In both cases, in fact, we look for a charlist. We assume that the input
 * strings are written according to the rules we stated above.
 * ------------------------------------------------------------------------
 * There is a global variable 'traceon'. If it is set to 'true' then one
 * can see the various steps through the execution of the program, and in
 * particular, the backtracking through the history of the list of states
 * examined or expanded. The axiom is the nonterminal of the left_hand_side
 * of the first production of the input grammar.
 * ------------------------------------------------------------------------
 * The context-free parser works by exploring in a depth-first fashion all
 * possible words generated by the grammar and consistent with the
 * character at hand in the string to be parsed.
 * ------------------------------------------------------------------------
 * After performing a tail() operation we need to reconstruct the list
 * which, otherwise, is modified (see statement marked by (***)).
 * ========================================================================
 */
import java.util.*;
// ------------------------------------------------------------------------
// string with an index pointing at the next available character.

 class IndexString {
   public String string;
   public int index;

   public IndexString(String string) {  // constructor
      this.string = string;
      this.index  = 0;
   }

   public void back(){
      index--;
   }

   public char getCh() {
      char x;
      if (index < string.length()) { x = string.charAt(index); }
      else x = '$'; index++; return x;
   }
 }
//-------------------------------------------------------------------------
// state : <String:sentential_form,  String:input_string>

 class State {
   public String sentForm;
   public String inString;
```

```
  public State() {
    this.sentForm = "";
    this.inString = "";
  }
                              // state : printing function
  // for reasons of readability the empty string is printed as "e"
  public void statePrint() {
    String sentForm1 = sentForm.equals("") ? "e" : sentForm;
    String inString1 = inString.equals("") ? "e" : inString;
    System.out.print("(" + sentForm1 + ", " + inString1 + ")");
  }
                              // -----------------------
                              // List<State> : append function
  public static List<State> stAppend(List<State> list1,
                                                List<State> list2) {
   List<State> stl = new List<State>();
   if (list1.isEmpty()) {stl = list2.copy(); return stl;}
   else {State st = list1.head(); list1.tail();
         stl = stAppend(list1,list2); stl.cons(st);
         list1.cons(st);      // reconstructing list1              (***)
         return stl;}
  }
                              // -----------------------
                              // List<State> : printing functions
  public static void stlPrintNE(List<State> L) {
   if (!L.isEmpty())
      {State h = L.head(); h.statePrint();
       System.out.print("   "); L.tail(); stlPrintNE(L);
       L.cons(h);             //reconstructing the list L of states (***)
      };
  }

  public static void stlPrint(List<State> L) {
   if (L.isEmpty()){System.out.print("[]");}
   else {System.out.print("  "); stlPrintNE(L); }
  }
 }
// -----------------------------------------------------------------------
// SyRs: <char:symb (left-hand-side), List<String>:rhss (right-hand-sides)>
 class SyRs {
  public char symb;
  public List<String> rhss;

  public SyRs() {
    this.symb = '-';
    this.rhss = null;
  }
                              // -----------------------
                              // List<String> = Rsides:printing functions
  public static void rPrintNE(List<String> L) {
   if (!L.isEmpty()) {
          String h = L.head();
          if (h.equals("")) {System.out.print("e");}
          else {System.out.print(h);};
```

```
          L.tail();
          if (!L.isEmpty()) {System.out.print(" |  ");};
          rPrintNE(L);
          L.cons(h);           // reconstructing the list L          (***)
                    }
   }

   public static void rPrint(List<String> L) {
    if (L.isEmpty()) {System.out.print("<>");}
    else {System.out.print(" "); rPrintNE(L); System.out.print(" ");}
   }
                               // ------------------------
                               // SyRs : printing function
   public static void SyRsPrint(SyRs r) {
   System.out.print(r.symb + " -> "); rPrint(r.rhss);
   }
 }
// ---------------------------------------------------------------------------
// Gram : List<SyRs>  (it is the type of a grammar)

 class Gram {
   public List<SyRs> gg;

   public Gram() {
     this.gg = new List<SyRs>();
   }
                               // ------------------------
                               // List<SyRs> = Gram:  printing functions
   public static void printGramNE(List<SyRs> L) {
    if (!L.isEmpty())
       {SyRs h = L.head(); SyRs.SyRsPrint(h);
        System.out.print("\n"); L.tail();
        if (!L.isEmpty()) {System.out.print("  ");}; printGramNE(L);
        L.cons(h);
       }
   }

   public static void printGram(List<SyRs> L) {
    if (L.isEmpty()) {System.out.print("{ }");}
    else {System.out.print("{ "); printGramNE(L); System.out.print("}");}
   }
 }
// ---------------------------------------------------------------------------
public class CFParser {
//----------------------------------------------------------------------------
//                    Making the grammar from the string gg

 public static String parseString(IndexString f) {
   char x = f.getCh();
   String chl;
   if (('a'<=x && x<='z')||('A'<=x && x<='Z'))
        { chl = x + parseString(f); }
   else { chl = ""; };
   return chl;
```

```
}
                                // ----------------------------------------
public static List<String> parseRsides(IndexString f) {
 List<String> rs = new List<String>();
 String chl2 = parseString(f); f.back();
 // this is the case of the epsilon production
 if (chl2.equals("e")) {chl2 = "";};
 char x = f.getCh();
 // x can be: either ';' or '$' or '|'. No other values are possible.
 if ((x==';') || (x=='$')) {rs.cons(chl2); return rs; }
 else { // here we have: x=='|'
       rs = parseRsides(f); rs.cons(chl2); return rs; }
}
                                // ----------------------------------------
public static SyRs parseprod(IndexString f) {
 SyRs sRs1 = new SyRs(); sRs1.symb = f.getCh();
 char x = f.getCh();
 if (x!='>') {System.out.print("Error in production.\n");};
 List<String> rs1 = new List<String>();
 rs1 = parseRsides(f); sRs1.rhss=rs1; return sRs1;
}
                                // ----------------------------------------
public static List<SyRs> parseGram(IndexString f)
                                             throws RuntimeException {
 SyRs sRs1 = parseprod(f); f.back();
 char x = f.getCh();
 // x can be: either ';' or '$'.  No other values are possible.
 if (x=='$') { List<SyRs> gram1 = new List<SyRs>(); gram1.cons(sRs1);
              return gram1; }
 else if (x==';') {List<SyRs> gram2 = new List<SyRs>();
                   gram2 = parseGram(f); gram2.cons(sRs1);
                   return gram2; }
 else { throw new RuntimeException("Error parseGram!"); }
}
//-----------------------------------------------------------------------
//                  trace function
public static void trace(boolean traceon, String s, List<State> stl) {
 if (traceon)
    {System.out.print("\n" + s +":\n"); State.stlPrint(stl);};
}
//-----------------------------------------------------------------------
//  visiting the search space: constructing new lists of states
public static List<String> rightSides(char s, List<SyRs> gr)
                                      throws RuntimeException {
 if (gr.isEmpty())
    { throw new RuntimeException ("No production for "+ s +
                                   " in the grammar.\n");}
 else {SyRs sR = new SyRs(); sR = gr.head();
       if (sR.symb == s) { return sR.rhss; }
       else {List<String> rs = new List<String>();
             gr.tail(); rs = rightSides(s,gr);
             gr.cons(sR);           //reconstructing the grammar gr (***)
             return rs; }
      }
```

```
 }
//-----------------------------------------------------------------------
//                  final function
 public static boolean isFinal(State st) {
   return (st.sentForm.equals("")) && (st.inString.equals(""));
 }
//-----------------------------------------------------------------------
//                  chop function
 public static void chop(State st) {    //  st : the state st is modified
   st.sentForm = st.sentForm.substring(1, st.sentForm.length());
   st.inString = st.inString.substring(1, st.inString.length());
 }
//-----------------------------------------------------------------------
//                  expand function
 public static List<State> expand(List<String> rs, String v, String w) {
   State st = new State();
   List<State> stl = new List<State>();
   if (rs.isEmpty()) { stl.makeEmpty(); return stl; }
   else {String cs = rs.head();
         rs.tail();  stl = expand(rs,v,w);
         rs.cons(cs);                    // reconstructing the list rs (***)
         st.sentForm = cs.concat(v);
         st.inString = w;
         stl.cons(st); return stl; }
 }
//-----------------------------------------------------------------------
//                  exists function
 private static boolean exists (boolean traceon, List<State> stl1,
                               List<SyRs> gr) {
  if (stl1.isEmpty()) { return false; }
                                  // depth first visit of the seach space
  else {State st1 = new State(); st1 = stl1.head();
        String st1sentForm = st1.sentForm;
        String st1inString = st1.inString;
       if (isFinal(st1)) { return true; }
  else if (!(st1sentForm.equals("")) &&
          ('A'<=st1sentForm.charAt(0)) && (st1sentForm.charAt(0)<='Z'))
          {List<State> stl2 = new List<State>();
           List<String> rs  = new List<String>();
           stl1.tail();
           rs = rightSides(st1sentForm.charAt(0),gr);
                                        // first character chopped
           st1sentForm = st1sentForm.substring(1, st1sentForm.length());
           stl2 = expand(rs,st1sentForm,st1inString);
           stl1 = State.stAppend(stl2,stl1);
           trace(traceon,"expanded",stl1);   // tracing
           return exists(traceon,stl1,gr);}
  else if (!(st1sentForm.equals("")) && !(st1inString.equals(""))
       && (('a' <= st1sentForm.charAt(0)) && (st1sentForm.charAt(0) <='z'))
       && (st1sentForm.charAt(0) == st1inString.charAt(0)))
          {chop(st1); stl1.tail(); stl1.cons(st1);
           trace(traceon,"chopped",stl1);   // tracing
           return exists(traceon,stl1,gr);}
  else    {stl1.tail();
```

```
            trace(traceon,"failed",stl1);    // tracing
            return exists(traceon,stl1,gr);}
       }
 }
//-----------------------------------------------------------------------
//                  main program
public static void main(String[] args) {
/** ---------------------------------------------------------------------
 *  gg   : given_grammar
 *  stp  : string_to_parse
 * ----------------------------------------------------------------------
 */
  String gg ="P>b|aQQ;Q>e|b|bP";  // example 0 (with an empty production)
  String stp="ab";                // true
  boolean traceon = true;         // variable for tracing the execution

//-----------------------------------------------------------------------
// From the string gg to the List<SyRs> gr0.
// We first construct the IndexString f from the String gg.
// gr0 is the grammar encoded by the given string gg.
//-----------------------------------------------------------------------
 IndexString f  = new IndexString(gg);
 List<SyRs> gr0 = new List<SyRs>();

 char axiom = gg.charAt(0);
 if (('A'<=axiom) && (axiom<='Z'))
    {System.out.print("\nThe input grammar is ");
     System.out.print("(\"e\" denotes the empty string in the rhs):\n");
     gr0 = parseGram(f);
/**
 * ----------------------------------------------------------------------
 * In order to avoid parsing, the above statement 'gr0 = parseGram(f);' can
 * be replaced by the following statements (erase the *'s, please):
 *
 * SyRs Prs = new SyRs(); Prs.symb='P';                         // P
 * String chl1P = "b";                                          //   b
 * String chl2P = "aQQ";                                        //    |aQQ
 * List<String> rsP = new List<String>();
 *              rsP.cons(chl2P); rsP.cons(chl1P);               //   b|aQQ
 * Prs.rhss = rsP;                                              // P>b|aQQ
 *
 * SyRs Qrs = new SyRs(); Qrs.symb='Q';                         // Q
 * String chl1Q = "";                                           //   e
 * String chl2Q = "b";                                          //    |b
 * String chl3Q = "bP";                                         //      |bP
 * List<String> rsQ = new List<String>();
 *              rsQ.cons(chl3Q); rsQ.cons(chl2Q); rsQ.cons(chl1Q);
 *                                                              //   e|b|bP
 * Qrs.rhss = rsQ;
 *                              //gr0 corresponding to gg="P>b|aQQ;Q>e|b|bP"
 * gr0.cons(Qrs); gr0.cons(Prs);
 * ----------------------------------------------------------------------
 */
     Gram.printGram(gr0); }
```

```
 else System.out.print("Not a grammar in input.");

 System.out.print("\n\nThe axiom is " + axiom + ".\n");
 State st = new State();
 st.sentForm = axiom + ""; // from char to Sytring of length 1.
 st.inString = stp;
 List<State> stl = new List<State>();
 stl.cons(st);
 System.out.print("The initial list of states is:\n");
 State.stlPrint(stl);
 boolean ans = exists(traceon,stl,gr0);
 System.out.print("\n\nThe following string of characters:\n  ");
 System.out.print(stp + "\nis ");
 if (!ans) {System.out.print("NOT ");};
 System.out.print("generated by the given grammar.\n");
 } // end of main
}
/**
 * input:   output:
 * ------------------------------------------------------------------------
 * java5c CFParser.java
 * java5  CFParser
 *
 *          The input grammar is ("e" denotes the empty string in the rhs):
 *          { P ->  b |  aQQ
 *            Q ->  e |  b |  bP
 *          }
 *
 *          The axiom is P.
 *          The initial statelist is:
 *            (P, ab)
 *          expanded:
 *            (b, ab)   (aQQ, ab)
 *          failed:
 *            (aQQ, ab)
 *          chopped:
 *            (QQ, b)
 *          expanded:
 *            (Q, b)   (bQ, b)   (bPQ, b)
 *          expanded:
 *            (e, b)   (b, b)   (bP, b)   (bQ, b)   (bPQ, b)
 *          failed:
 *            (b, b)   (bP, b)   (bQ, b)   (bPQ, b)
 *          chopped:
 *            (e, e)   (bP, b)   (bQ, b)   (bPQ, b)
 *
 *          The following string of characters:
 *            ab
 *          is generated by the given grammar.
 * ------------------------------------------------------------------------
*/
/**
 * Examples already prepared:
 * String gg="P>b|aQQ;Q>e|b|bP";       // example 0 - with empty production
```

```
* String stp="a";                        // true
* String stp="abaa";                     // false
* String stp="aba";                      // true

* String gg="P>a|aQ;Q>b|bP";             // example 1
* String stp="ab";                       // true
* String stp="ababa";                    // true
* String stp="aaba";                     // false

* String gg="P>a|bQ;Q>b|aQ";             // example 2
* String stp="baab";                     // true
* String stp="bbaaba";                   // false

* String gg="S>aB|bA;A>bAA|a|aS;B>aBB|b|bS";// same numbers of a's and b's
* String stp="bbabaaab";                 // true
* String stp="babaaab";                  // false

* String gg="S>a|aAS;A>SbA|ba|SS";       // example 2.7 of Hopcroft-Ullman
* String stp="aabaaa";                   // true
* String stp="aabba";                    // false
* String stp="aabbaa";                   // true
* ---------------------------------------------------------------------------
*/
```

5.3. Chop-and-Expand Context-Free Parser in a Logic language

Recall that in logic programming $[A|X]$ represents the list whose head is the element A and whose tail is the list X.

Here is a logic program which realizes a Chop-and-Expand parser:

1. $parse(G, [\,], [\,]) \leftarrow$
2. $parse(G, [A|X], [A|Y]) \leftarrow terminal(A),\ parse(G, X, Y)$ // CHOP
3. $parse(G, [A|X], Y) \leftarrow nonterminal(A),\ member(A \to B,\ G),$ // EXPAND
   $append(B, X, Z),\ parse(G, Z, Y)$
4. $member(A, [A|X]) \leftarrow$
5. $member(A, [B|X]) \leftarrow member(A, X)$
6. $append([\,], L, L) \leftarrow$
7. $append([A|X], Y, [A|Z]) \leftarrow append(X, Y, Z)$

together with the clauses which define which are the terminal symbols and the nonterminal symbols.

The first argument of *parse* is a context-free grammar, the second argument is a list of terminal symbols and nonterminal symbols (i.e., a sentential form), and the third argument is a word represented as a list of terminal symbols. We assume that context-free grammars are represented as lists of productions of the form $x \to y$, where x is a nonterminal symbol and y is a list of terminal and nonterminal symbols.

We have that $parse(G, [s], W)$ holds iff from the symbol s we can derive the word W using the grammar G.

Example 5.3.1. The grammar $G = \langle\{a, b\},\ \{S\},\ \{S \to aSb,\ S \to ab\},\ S\rangle$ is represented by the clauses:

$terminal(a) \leftarrow$
$terminal(b) \leftarrow$
$nonterminal(s) \leftarrow$

together with the list $[s \to [a, s, b],\ s \to [a, b]]$ which represents its productions. The left hand side of the first production is assumed to be the start symbol. For this grammar G the goal $\leftarrow parse([s \to [a, s, b],\ s \to [a, b]],\ [s],\ [a, a, b, b])$ is true. □