## 3.2. Chop-and-Expand Context-Free Parser in Java

In this section we present the two Java programs: `List.java` and `CFParser.java`, which realize the context-free parsing using the Chop-and-Expand algorithm illustrated in Section 2.3.2 on page 31 and Section 3.1 on page 35. In those sections that algorithm is presented using a functional language.

The C++ versions of those programs can be found in [**19**, page 69]. A similar version of the program `List.java` can be found (with the same name) in Section 7.4 on page 190.

```
/**
 * ============================================================================
 *                  Generic Class List <T>    (class used for parsing)
 * Filename: "List.java"
 *
 * Every object of the class List<T> is a list of elements of type T.
 * The following methods are available:
 *
 *   cons(T d), head(), tail(), isSingleton(), isEmpty(),
 *   append(List<T> list2), makeEmpty(), copy(), and listPrint().
 *
 * Also cloneCopy() is available if we uncomment line (***) below and
 * comment the previous line.
 *
 * The head of the list is to the right, not to the left.
 *
 * Note that tail() is destructive.
 * After a tail operation, if we need the original value of the list, we
 * should reconstruct it by 'consing' the head of the list.
 * ============================================================================
 */

import java.util.*;        // needed for using ArrayList<T> and Iterator

 public class List<T> {    // it is ok if we do not use cloneCopy()

// public class List<T> implements Cloneable { // (***) In order to use
          // cloneCopy(), uncomment this line and comment the previous one.

   private ArrayList<T> list;

   public List() {                // constructor
      list = new ArrayList<T>();
   }

   public void cons(T datum) {    // the head of the list is to the right,
      list.add(datum);            // not to the left.
   }                              // add(_) is a method of ArrayList<T>


   public T head() {              // the head of the list is to the right.
      if (list.isEmpty())         // It is the last element: >------------+
         {System.out.println("Error: head of empty list!");};//           |
          T obj = list.get(list.size() - 1); // <-------------------------+
      return obj;                 // isEmpty(), size(), and get(int i)
   }                              // are methods of ArrayList<T>
```

```
   public void tail() {           // destructive tail() method
      if (list.isEmpty())
         {System.out.println("Error: tail of empty list!");};
      list.remove(list.size() - 1); // isEmpty(), size(), and remove(int i)
   }                                // are methods of ArrayList<T>

   public boolean isSingleton() {
      return list.size() == 1;    // size() is a method of ArrayList<T>
   }

   public boolean isEmpty() {
      return list.isEmpty();      // isEmpty() is a method of ArrayList<T>
   }

   // As usual, the head of the list 'this.append(list2)', after appending
   // the 'list2' to the list 'this', is the head of the list 'this', before
   // the append operation. Thus, 'this' = 'this' <> 'list2'.

   public List<T> append(List<T> list2) {
   List<T> l2 = list2.copy();     // l2 is an argument to a recursive call
                                  // to append. list2 should be copied!
   if (this.isEmpty()) {return l2;}
   else {T a = this.head(); this.tail();
         List<T> l = this.append(l2); l.cons(a);
         this.cons(a);            // reconstructing the list 'this'
         return l;}
   }
/**
 * ---------------- Examples of isEmpty() -------------------------------
 * System.out.println(new List<Integer>().isEmpty());   prints true
 * System.out.println(new List().isEmpty());            prints true
 * List <Integer> v = new List();
 * System.out.println(v.isEmpty());                     prints true
 * System.out.println(v == null);                       prints false
 * ----------------------------------------------------------------------
 */
   public void makeEmpty() {
      list.clear();               // clear() is a method of ArrayList<T>
   }
  // ------------------------- Printing a list ---------------------------
  // We assume that an element of the list has a toString() method.
   public void listPrint() {
      System.out.print("[ ");
      for (Iterator iter = list.iterator(); iter.hasNext(); ) {
          System.out.print((iter.next()).toString() + " ");
          };
      System.out.print("]");
   }

  // ------------------------- Making a copy of a list without clone() ----
   public List<T> copy() {
      List<T> copyList = new List<T>();
      for (Iterator<T> iter = list.iterator(); iter.hasNext(); ) {
           copyList.list.add(iter.next());
      };
      return copyList;
   }
```

```
  // -------------------------- Making a copy of a list with clone() -------
   public List<T> cloneCopy() {                          // see (***) above
      try { List<T> copyList = (List<T>)super.clone();
            copyList.list = (ArrayList<T>)list.clone();
            return copyList;
      } catch (CloneNotSupportedException e) {
        throw new InternalError();
      }
   }
 }
/**
 * ---------------------------------------------------------------------------
 * If we have the program fragment:
 *
 * Integer k = new Integer(6);
 * List<Integer> la = new List<Integer>();
 * la.cons(k);
 * System.out.print("\nla: "); la.listPrint();
 * la.cons(new Integer(4)); la.cons(new Integer(9));
 *              // in Java 1.5: we can write: 9, instead of: new Integer(9)
 * List<Integer> lb = new List<Integer>();
 * lb = la.copy();
 * System.out.print("\nlb: "); lb.listPrint();
 * System.out.print("\nla: "); la.listPrint();
 * lb.tail(); List<Integer> lc = la.append(lb.append(lb));
 * System.out.print("\nlc: "); lc.listPrint();
 *
 * After compilation we get:
 * Note: ./List.java uses unchecked or unsafe operations.
 * Note: Recompile with -Xlint:unchecked for details.
 *
 * When running the program we get:
 * la: [ 6 ]
 * lb: [ 6 4 9 ]
 * la: [ 6 4 9 ]
 * lc: [ 6 4 6 4 6 4 9 ]
 * ---------------------------------------------------------------------------
 * Note: the head of the list is to the right.
 * ---------------------------------------------------------------------------
 */
```

Now we will present the program `CFParser.java` which, together with the above `List.java` program, implements the context-free parsing algorithm that we have presented using a functional language in Section 3.1 on page 35. In particular, the method `exists`, which is listed immediately above the main program, implements in Java the *exists* function of the Chop-and-Expand parser presented on page 38.

```
/**
 * ============================================================================
 *               CHOP-AND-EXPAND PARSER FOR CONTEXT FREE GRAMMARS
 * Filename:"CFParser.java". Thanks to R. M. Burstall and E. W. Dijkstra.
 *
 * ----------------------------------------------------------------------------
 *                              --- 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 the functions
 * head(), tail(), cons(), isEmpty(), makeEmpty() for:
 *
 * - list of String (type: List<String>). It is the type of the list of all
 *      right hand sides of a nonterminal. This type is also called: Rsides.
 * - list of States (type: List<State>)
 * - list of SyRs   (type: List<SyRs>). It is the type of a grammar. It is
 *      the type of the list of <nonterminal, list of its right hand sides>.
 *      This type is also called: Gram.
 * Redefinition of functions is necessary for printing lists of different
 * types.
 * ----------------------------------------------------------------------------
 * The input grammar is given as a string named 'gg' of type 'Gram',
 * according to the following productions:
 *
 * character ::= 'a'..'d'  |  'f'..'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 epsilon-rules, are allowed.
 * The character 'e' is reserved for denoting the right hand side of
 * an empty production, that is, 'e' denotes 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  we have that:
 *        \_/     aQQ is a string: "aQQ"
 *      \___/   b|aQQ is a list of elements of type String, i.e., an Rsides.
 *                     It is: ["b", "aQQ"]
 *    \_____/ P>b|aQQ is a SyRs, i.e., a pair of a symbol and its
 *                           right hand sides. It is: <'P', ["b", "aQQ"]>
 * The axiom of the grammar is the nonterminal of the left hand side
 * of the first production of the grammar.
 *
 * The string to parse, named 'stp', is either a non-empty sequence of
 * characters from the set {'a',...,'d','f',...,'z'} or is the empty
 * sequence denoted by 'e'.
 *
 * -------------------- ASSUMPTIONS ON THE INPUT --------------------------
 * (1) When writing the input grammar, the productions relative to the same
 *     nonterminal should be grouped together. For instance, we should
 *     write "P>a|aQQ;Q>e|b|bP", instead of "P>a;Q>e;P>aQQ;Q>b;Q>bP".
```

```
 * (2) The strings denoting the input grammars should be written according
 *     to the rules we have stated above for the nonterminal 'grammar'.
 * ------------------------------------------------------------------------
 * The grammar should not be left recursive, that is, it should not be the
 * case that a nonterminal A exists such that A r ->* A s, for some strings
 * r and s of terminal and/or nonterminal symbols of the grammar.
 * ------------------------------------------------------------------------
 * This 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 parse 'stp'.
 * ------------------------------------------------------------------------
 * There is a global variable 'traceon'. If it is set to 'true' then we
 * can see the various steps through the execution of the program, and in
 * particular, the sequence of the lists of states which are generated
 * during the depth first visit of search space.
 * ------------------------------------------------------------------------
 * After performing a tail() operation on a list we need to reconstruct
 * the list which, otherwise, is modified (see statements 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;      // index points to the next character to be read
   }
   public void back(){
      index--;
   }
   public char getCh() {
      char x;
      if (index < string.length()) { x = string.charAt(index); }
      else x = '.';
      index++; return x;    // index is incremented by one unit
   }
 }
//-------------------------------------------------------------------------
// 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> : printing functions
   public static void stlPrintNE(List<State> L) {
    if (!L.isEmpty())
       {State h = L.head(); h.statePrint();  L.tail();
        if (!L.isEmpty()) {System.out.print("   ");}; 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); System.out.print("]");}
   }
 }

// ---------------------------------------------------------------------------
// 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 {
//---------------------------------------------------------------------------
/*                        RECURSIVE DESCENT PARSING
 *                  for making the grammar from the string gg
 * -------------------------------------------------------------------------
 * The function 'parseString' gets one of the alternatives (as a list of
 * characters) of the right hand sides of the productions of a nonterminal.
 */
 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 = "";
          f.back(); }
   return chl; // At the end index points to either '>' or '|' or ';' or '.'
 }
                                // ----------------------------------------
 public static List<String> parseRsides(IndexString f) {
  List<String> rs = new List<String>();
  String chl2 = parseString(f);
                                //this is the case of the epsilon production
  if (chl2.equals("e")) {chl2 = "";};
  char x = f.getCh();
  if (x=='|') { rs = parseRsides(f); rs.cons(chl2); return rs; }
  else {//Here x can be either ';' or '.'. No other characters are possible.
        f.back();   // ready to get again either ';' or '.'.
        rs.cons(chl2); return rs; }
           // At the end in rs we have all the productions of a nonterminal.
 }
                                // ----------------------------------------
 public static SyRs parseProds(IndexString f)
                                                  throws RuntimeException {
  SyRs sRs1 = new SyRs(); sRs1.symb = f.getCh();
  char x = f.getCh();
  if (x=='>') { List<String> rs1 = new List<String>();
                rs1 = parseRsides(f); sRs1.rhss=rs1; return sRs1; }
  else { // x should have been '>'. If it is not '>' we throw an exception.
         throw new RuntimeException("Error in production!\n"); }
 }
```

```
                                  // ---------------------------------------
 public static List<SyRs> parseGram(IndexString f)
                                                 throws RuntimeException {
  SyRs sRs1 = parseProds(f);
  char x = f.getCh();
  // Here x can be either ';' or '.'.  No other characters are possible.
  if (x==';') { List<SyRs> gram2 = new List<SyRs>();
                gram2 = parseGram(f); gram2.cons(sRs1);
                return gram2; }
  else if (x=='.') { List<SyRs> gram1 = new List<SyRs>(); gram1.cons(sRs1);
                     return gram1; }
  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) {// The state st is modified. The first
                                    // characters of the strings are chopped
   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
//
// This code is directly derived from the code of the exists function
// in functional style. The search space is visited in a depth first manner.
//
 private static boolean exists (boolean traceon, List<State> stl1,
                                List<SyRs> gr) {
  if (stl1.isEmpty()) { return false; }
  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 : rhs's of the nonterminal symbol
         rs = rightSides(st1sentForm.charAt(0),gr);
                                     // delete the nonterminal symbol
         st1sentForm = st1sentForm.substring(1, st1sentForm.length());
                                     // EXPAND the nonterminal symbol
         stl2 = expand(rs,st1sentForm,st1inString);
         stl1 = stl2.append(stl1);        // stl1 = stl2 <> stl1
         trace(traceon,"expand",stl1);    // tracing ---
         return exists(traceon,stl1,gr); }
  else if (!(st1sentForm.equals("")) && !(st1inString.equals(""))
       && ((('a'<=st1sentForm.charAt(0)) && (st1sentForm.charAt(0)<='d')) ||
           (('f'<=st1sentForm.charAt(0)) && (st1sentForm.charAt(0)<='z')))
       && (st1sentForm.charAt(0) == st1inString.charAt(0)))
       { chop(st1); stl1.tail(); stl1.cons(st1); // CHOP the terminal symbol
         trace(traceon,"chop",stl1);      // tracing ---
         return exists(traceon,stl1,gr); }
  else { stl1.tail();
         trace(traceon,"fail",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);
 *                           [<symb,    rhss>      <symb,      rhss>      ]
 *                              |       /  \          |       /  |  \
 * gr0 is the list of pairs: [<'P', ["b", "aQQ"]>,  <'Q', ["", "b", "bP"]>]
 * "" is the empty string.
 * -------------------------------------------------------------------------
 */
      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 String (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
}
```

```
/**
 * When printing the statelist, the head is to the left.
 * input:   output:
 * ------------------------------------------------------------------------
 * javac CFParser.java
 * java  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)]
 *          expand:
 *            [(b, ab)   (aQQ, ab)]
 *          fail:
 *            [(aQQ, ab)]
 *          chop:
 *            [(QQ, b)
 *          expand:
 *            [(Q, b)   (bQ, b)   (bPQ, b)]
 *          expand:
 *            [(e, b)   (b, b)   (bP, b)   (bQ, b)   (bPQ, b)]
 *          fail:
 *            [(b, b)   (bP, b)   (bQ, b)   (bPQ, b)]
 *          chop:
 *            [(e, e)   (bP, b)   (bQ, b)   (bPQ, b)]
 *
 *          The following string of characters:
 *            ab
 *          is generated by the given grammar.
 * ------------------------------------------------------------------------
 *                           Various Examples
 *
 * String gg="P>b|aQQ;Q>e|b|bP";       // Example 0 (empty production in gg)
 * 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>a|aAS;A>SbA|ba|SS";    // Example 2.7 of Hopcroft-Ullman ---
 * String stp="aabaaa";                // true
 * String stp="aabba";                 // false
 * String stp="aabbaa";                // true
 * 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
 * ------------------------------------------------------------------------
 */
```

### 3.3. Chop-and-Expand Context-Free Parser in a Logic Language

Here is a logic program which realizes a Chop-and-Expand Parser.

Chop-and-Expand Parser

1. $parse(G, [\,], [\,]) \leftarrow$
2. $parse(G, [A|X], [A|Y]) \leftarrow terminal(A),\ parse(G, X, Y)$ // CHOP
3. $parse(G, [A|X], Y) \leftarrow nonterminal(A),\ member(A \rightarrow B,\ G),\ append(B, X, Z),\ parse(G, Z, Y)$ // EXPAND
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 these seven clauses we also need the clauses which define the terminal symbols and the nonterminal symbols. Recall that in logic programming $[A|X]$ represents the list whose head is the element $A$ and whose tail is the list $X$. In particular, if $[A|X] = [2, 3, 1, 1]$ then $A = 2$ and $X = [3, 1, 1]$. All constants in Prolog should begin with a lower case letter, so that, in particular, the axiom $S$ is represented by the character $s$.

The first argument of *parse* is a context-free grammar, the second argument is a list of terminal symbols and nonterminal symbols (that is, 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 $x \rightarrow y$, 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 $G$, that is, $W$ is a word of the language generated by $G$.

EXAMPLE 3.3.1. Let us consider the context-free grammar $G = \langle \{a, b\},\ \{S\},\ \{S \rightarrow aSb,\ S \rightarrow ab\},\ S \rangle$. It is represented by the clauses:

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

together with the list $[s \rightarrow [a, s, b],\ s \rightarrow [a, b]]$ which represents its productions. The left hand side of the first production is assumed to be the start symbol $s$. For this grammar $G$ the query $\leftarrow parse([s \rightarrow [a, s, b],\ s \rightarrow [a, b]],\ [s],\ [a, a, b, b])$ evaluates to true, and this means that the word $aabb$ belongs to the language generated by $G$.

Note that in the clause $member(A, [B|X]) \leftarrow member(A, X)$, we do not require that $A$ is different from $B$. Indeed with this clause, the query of the form $\leftarrow member(A, l)$, where $A$ is a variable and $l$ is a ground list, generates by backtracking all members of the list $l$ with their multiplicity. □

More examples of Prolog programs for parsing words generated by grammars can be found in Section 10.1 on page 249.