CHAPTER 3

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 2. This kind of parser works for context-free grammars which are not left recursive, that is, grammars without productions of the form: $A \to A\alpha$ for some nonterminal symbol A and some string α made of terminal and nonterminal symbols. This condition, however, is not a restrictive condition because any context-free grammar can be transformed into an equivalent one without left-recursive productions.

The reader will find in the companion book **[12]** the description of two general algorithm for parsing context-free languages:
(i) the Cocke-Younger-Kasami parser, and
(ii) the Earley parser.

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

The variable '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(abc) = a   and tl(abc) = 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 $\rightarrow$ 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-stl gr
    end
(1) **else if** $sf\neq\varepsilon$ and $s\neq\varepsilon$ and $terminal(hd(sf))$ and $hd(sf)=hd(s)$ **then**
    exists $(\langle tl(sf), tl(s)\rangle : tl(stl))$ *gr* // CHOP
  else *exists* $tl(stl)$ *gr*

In line (1) the condition ‘and $terminal(hd(sf))$’ is not necessary.

The initial call of *exists stl gr*
is of the form: *exists* [⟨S, string_to_parse⟩] given_grammar
where S is the axiom of the given_grammar.

3.2. 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 3.1. The C++ versions of these programs can be found in [**11**, page 69]. An similar version of the `List.java` program can be found (with the same name) in Section 5.4 on page 109.

```
/** =======================================================================
 *                  Generic Class List <T>    (for parsing)
 *
 * 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(), copy(), and
 *   listPrint().
 * Also cloneCopy() is available if we use (***)
 *
 * Notice that the tail() method 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 { //it is for using cloneCopy()
//                                                                   (***)

   private ArrayList<T> list;

   public List() {
      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.
   }

   public void tail() {           // destructive tail() method
      if (list.isEmpty())
         {System.out.println("Error: tail of empty list!");};
      list.remove(list.size() - 1);
   }                              // size() is a method of ArrayList<T>

   public T head() {
      if (list.isEmpty())
         {System.out.println("Error: head of empty list!");};
          T obj = list.get(list.size() - 1);
      return obj;                 // get(int i) is a method 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>
   }

   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();
 * List<Integer> lb = new List<Integer>();
 * la.cons(new Integer(4));
 * la.cons(9); // Java 1.5: no need for new Integer(9)
 * lb = la.cloneCopy();
 * System.out.print("\nlb: "); lb.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 ]
 * ------------------------------------------
 * Note: the head of the list is to the right.
 * -------------------------------------------------------------------------
 */
```

```
/** ========================================================================
 *                    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 search 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
* ---------------------------------------------------------------------
*/
```