Chapter 7

Parsing

7.1 Parser for a regular grammar

```
/* ==============================================================================
                 PARSER FOR A REGULAR GRAMMAR
==============================================================================
The input grammar is given as a string, named gg, of the form:
   terminal    ::= 'a'..'z'
   nonterminal ::= 'A'..'Z'
   rsides      ::= terminal '.' | terminal nonterminal (right recursion)
   symprod     ::= nonterminal  rsides
   grammar     ::= symprod | symprod grammar
The empty productions, also called e-rules, are not allowed. The given grammar
can be nondeterministic.
When writing the string gg which encodes the productions of the input grammar,
the productions relative to the same non-terminal need not be grouped
together. For instance, the string "Qb.PbQPa.QaQ" stands for the productions:
1.Q->b; 2.P->bQ; 3.P->a; 4.Q->aQ.
The function 'length' gives the length of a string.
The string to be parsed is in the string 'stp' of characters from 'a'..'z'.
The rsides of the form: nonterminal terminal | terminal '.' can always
be transformed in rsides of the form: terminal '.' | terminal nonterminal,
that is, left-recursion can be avoided in favour of right-recursion.
(see: Esercise 3.7 in Hopcroft-Ullmann: Formal Languages and Their Relation to
Automata. Addison Wesley. 1969.)
------------------------------------------------------------------------------
The value 9999 means either a vacuous position of a production for '.' or
the non existent next production of a nonterminal (because all productions for
that nonterminal have been already considered).
------------------------------------------------------------------------------
When printing the ancestorslist, each item is a pair: `number.nonterminal'.
The number identifies the production for the nonterminal within the string
'gg' which represents the grammar. This is used for denoting the productions
already explored. For any given nonterminal the productions are assumed to be
ordered from left to right according to their occurrence in 'gg'.
There is a global variable 'traceon'. If 'traceon' is set to 'true' then we
can trace various steps through the execution of the program.
============================================================================= */
#include <iostream>
#include <string>      // not <string.h>. File needed for using strings.
using namespace std;  typedef int Values;
#include "list_module.cpp"
/*-----------------------------------------------------------------------------
Now the type 'list' is the type of a list of integers. Now, the functions:
```

```
head, tail, cons, member, and printlist are available.
The global variables are: gg, stp, lgg, lstp1, traceon.
------------------------------------------------------------------------------- */
string gg, stp; // gg: given grammar, stp: string to parse.
int  lgg,lstp1;
/*------------------------------------------------------------------------------
lstp1 is the length of the string to parse -1.
-1 is due to the fact that indexing in c++ begins from 0:
       stp=abcab  length(stp)=5
index:     01234  lstp1=4
------------------------------------------------------------------------------- */
bool traceon;
//------------------------------------------------------------------------------
//                    printing a given production
void pprint(int i)
{cout << i/3+1 << "." << gg[i] << "->" << gg[i+1];
if (gg[i+2] != '.' ) {cout << gg[i+2];} cout << "; ";}
//------------------------------------------------------------------------------
//                    printing a given grammar
void gprint() {int i=0; while (i<lgg) {pprint(i); i=i+3;}; cout << "\n";}
//------------------------------------------------------------------------------
//                    printing the ancestorlist
void printnelist(list L)
{if (L!=NULL)
    {if (tail(L)==NULL)
     {cout << (head(L)/3)+1 << "." << gg[head(L)];}
     else {cout << (head(L)/3)+1 << "." << gg[head(L)] << ", ";
           printnelist(tail(L)); };};}

void print(list L)
{if (L==NULL) {cout << "[]";} else
{cout << "["; printnelist(L); cout << "]"; };}
//------------------------------------------------------------------------------
//                    tracing
void trace(string s, list al, int pg, int ps)
{if (traceon)
     {cout << "\n\nAncestorslist:      "; print(al); cout << ";  ";
     cout << "\nProduction at hand: ";
     if (pg==9999) {cout << "none;";} else {pprint(pg);};
     cout << "\nCharacter at hand:  " << stp[ps]; cout << "   => " << s;}}
//------------------------------------------------------------------------------
//                    next production
int findnextprod(int p)
{char s=gg[p];
do {p=p+3;} while (!((p>lgg-1) || (gg[p]==s)));
if (p<=lgg-1) {return p;} else {return 9999;};}
//------------------------------------------------------------------------------
//                    first production
int findfirstprod(int p)
{char s=gg[p]; int i=0;
while ((gg[i]!=s) && (i<=lgg-1)) {i=i+3;};
if (i<=lgg-1) {return i;} else {return 9999;};}
// -----------------------------------------------------------------------------
//                    parsing
bool parse(list al, int pg, int ps)
{list al1=NULL; int n=0,h=0,ps1=0;
if ((al==NULL) && (pg==9999))
   {trace("Fail.",al,pg,ps); return false;}
```

```
else if (pg==9999)
     {al1=tail(al); h=head(al); n=findnextprod(h); ps1=ps-1;
      trace("Alternative production from the father.",al,pg,ps);
      return parse(al1,n,ps1);}
else if ((gg[pg+1]!=stp[ps]) ||
        ((gg[pg+2]=='.') && (ps!=lstp1)) ||
        ((gg[pg+2]!='.') && (ps==lstp1)))
       {trace("Alternative production.",al,pg,ps);
        n=findnextprod(pg); return parse(al,n,ps);}
else if ((gg[pg+2]=='.') && (ps==lstp1))
     {trace("Success.\n",al,pg,ps); return true;}
else  {al1=cons(pg,al); n=findfirstprod(pg+2); ps1=ps+1;
       trace("Go down the string.",al,pg,ps);
       return parse(al1,n,ps1);};
}
//------------------------------------------------------------------------------
int main()                // this program runs with the C++ compiler Vers. 3.2
{traceon=true;
gg="Pa.PbQPbP";           // example 0
stp="ba";                 // true
lgg=gg.length(); lstp1=stp.length()-1;
cout << "\nThe given grammar is: "; gprint();
char axiom=gg[0];
cout << "The axiom is " << axiom << ".";
list al=NULL; int pg=0; int ps=0; bool ans=parse(al,pg,ps);
cout << "\nThe input string\n  " << stp << "\nis ";
if (!ans) {cout << "NOT ";}; cout << "generated by the given grammar.\n";
}
/* input: output:  traceon=true.
--------------------------------------------------------------------------------
          The given grammar is: 1.P->a; 2.P->bQ; 3.P->bP;
          The axiom is P.

          Ancestorslist:      [];
          Production at hand: 1.P->a;
          Character at hand:  b   => Alternative production.

          Ancestorslist:      [];
          Production at hand: 2.P->bQ;
          Character at hand:  b   => Go down the string.

          Ancestorslist:      [2.P];
          Production at hand: none;
          Character at hand:  a   => Alternative production from the father.

          Ancestorslist:      [];
          Production at hand: 3.P->bP;
          Character at hand:  b   => Go down the string.

          Ancestorslist:      [3.P];
          Production at hand: 1.P->a;
          Character at hand:  a   => Success.

          The input string
            ba
          is generated by the given grammar.
----------------------------------------------------------------------------- */
/* Other examples already prepared:
```

```
gg="PbPPa.";                     // example 1
stp="aba";                       // false
stp="bba";                       // true

gg="Pa.PaQQb.QbP";               // example 2
stp="ab";                        // true
stp="ababa";                     // true
stp="aaba";                      // false

gg="Pa.Qb.PbQQaQ";               // example 3
stp="baaab";                     // true
stp="baab";                      // true
stp="bbaaba";                    // false
------------------------------------------------------------------------- */
```

7.2 Parser for a context-free grammar

```
/* ==========================================================================
                  PARSER FOR A CONTEXT FREE GRAMMAR
Thanks to R. Burstall and E.W. Dijkstra. "the chop-and-expand parser".
=============================================================================
We use TEMPLATES to avoid the redefinition of null, isnull, head, tail, cons
functions for:
- list of chars          (type: charlist)
- list of lists of chars (type: rsides)
- list of states         (type: statelist)
- list of syRs           (type: 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 'grammar',
according to the following productions:
  character ::= 'a'..'z'  | 'A'..'Z'           ('e' is reserved: see below)
  charlist  ::= character | character charlist
  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 list of chars: (a,Q,Q)
     -----   b|aQQ is a list of lists of chars,i.e.,an rsides: ((b), (a,Q,Q))
   ------- P>b|aQQ is a syRs, i.e.,
                       a pair of a symbol and its rsides <P, ((b), (a,Q,Q))>
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 it is either a
non-empty sequence of characters in 'a'..'z' or it is the empty sequence '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 'parselistofchars' 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.
strlen(f) computes the length of the string f. For instance, if f="abcb"
strlen(f)=4.
For strings: array notation     char  gg[]="P>b|aQQ;Q>e|b|bP"
             pointer notation   char*   gg="P>b|aQQ;Q>e|b|bP"
gg[i] in array notation is *(gg+i) in pointer notation.
--------------------------------------------------------------------------------
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.
============================================================================== */
#include <iostream.h>
//-----------------------------------------------------------------------------
template<class T> class List
{struct cell {T atom; cell* next;
              cell(T at, cell* nx) : atom(at), next(nx) { };};
              cell* ptr;
public :
List(): ptr(NULL){}
void null() {ptr=NULL;}
bool isnull() {return (ptr == NULL);}
T head()
  {if (ptr == NULL) {cout << "\nError: head of empty list\n";}
  else {return ptr->atom;}}
void tail()
  {if (ptr == NULL) {cout << "\nError: tail of empty list\n";}
  else {ptr=ptr->next;}}
void cons(T x) {ptr= new cell(x,ptr);}};
//                    end of template
//------------------------------------------------------------------------------
typedef List<char> charlist;       // list of chars
class state                        // state : <sentential_form, string>
   {public : charlist sentform; charlist instring;};
typedef List<state> statelist;     // list of states
typedef List< List<char> > rsides; // right_hand_sides: list of lists of chars
class syRs                    // <sy,Rs>, i.e., <symbol, its right_hand_sides>
   {public : char symb; rsides rhss;};
typedef List<syRs> gram;      // grammar: list of syRs pairs
/*------------------------------------------------------------------------------
                      global variables
gg   : given_grammar
stp  : string_to_parse
f    : string (as a char*)
ind  : pointer on a string
x    : character at f[ind], that is, *(f+ind)
```

```
-------------------------------------------------------------------------- */
char* gg ="P>b|aQQ;Q>e|b|bP"; // example 0 (with an empty production)
char* stp="ab";               // true
//char* stp="abaa";           // false
char* f;                      // string to store gg and stp (see main program)
int ind; char x;
gram gr0;              // gr0 is the internal representation of the grammar gg
bool traceon=true;
//---------------------------------------------------------------------------
//                      append and print functions for charlist
charlist append(charlist list1, charlist list2)
{if (list1.isnull()) {return list2;}
 else {char c=list1.head(); list1.tail();
      charlist cs=append(list1,list2); cs.cons(c);
      return cs;};}

void printnelist(charlist L)
{if (!L.isnull()) {cout << L.head() <<" "; L.tail(); printnelist(L);}}

void print(charlist L)
{if (L.isnull()) {cout <<"e ";} else {cout <<""; printnelist(L); cout <<"";}}
//---------------------------------------------------------------------------
//                     printing function for state
void stprint(state st)
{cout <<"("; print(st.sentform); cout <<", "; print(st.instring);
cout <<")"; };
//---------------------------------------------------------------------------
//                    printing functions for statelist
statelist stappend(statelist list1, statelist list2)
{if (list1.isnull()) {return list2;}
 else {state st=list1.head(); list1.tail();
       statelist stl; stl=stappend(list1,list2); stl.cons(st);
       return stl;};}

void stlprintne(statelist L)
{if (!L.isnull())
   {state st; st=L.head(); stprint(st); cout <<"   ";
    L.tail(); stlprintne(L);};};

void stlprint(statelist L)
{if (L.isnull()){cout <<"[]";}
 else {cout <<"  "; stlprintne(L); cout <<"";}};
//---------------------------------------------------------------------------
//                   printing functions for rsides
void rprintne(rsides L)
{if (!L.isnull()) {charlist cs; cs=L.head(); print(cs); L.tail();
 if (!L.isnull()) {cout <<" |  ";}; rprintne(L);}}

void rprint(rsides L)
{if (L.isnull()) {cout <<"<>";}
 else {cout <<" "; rprintne(L); cout <<" ";} };
//---------------------------------------------------------------------------
//                    printing functions for syRs
void sRsprint(syRs r)
{cout << r.symb;cout <<" -> "; rprint(r.rhss);}
//---------------------------------------------------------------------------
//                   printing functions for gram
void printgramne(gram L)
```

```
{if (!L.isnull())
     {syRs sRs; sRs=L.head(); sRsprint(sRs); cout <<"\n"; L.tail();
      if (!L.isnull()) {cout <<"  ";}; printgramne(L);};};

void printgram(gram L)
{if (L.isnull()) {cout <<"{ }";}
else {cout <<"{ "; printgramne(L); cout <<"}";}};
//-------------------------------------------------------------------------------
//                      Making the grammar from the string gg
void getch(char &x)
{if (ind < strlen(f)) {x=*(f+ind); ind++;} else {x='$';};}

bool epsilon(charlist chl)
{char c=chl.head(); chl.tail(); return ((c=='e') && (chl.isnull()));};

void parselistofchars(charlist &chl)
{char x1=x; if (('a'<=x && x<='z')||('A'<=x && x<='Z'))
                 {getch(x); charlist chl1;
                  parselistofchars(chl1); chl1.cons(x1); chl=chl1;}
             else {chl.null();};};

void parsersides(rsides &rs)
{charlist chl2; chl2.null();
 parselistofchars(chl2);
 if (epsilon(chl2)) {chl2.null();};
 if ((x==';') || (x=='$')) {rs.null(); rs.cons(chl2);}
 else {if (x=='|') {getch(x);};
       rsides rs1; rs1.null(); parsersides(rs1);
       rs1.cons(chl2); rs=rs1;}};

void parseprod(syRs  &sRs1)
{sRs1.symb=x; getch(x);
   if (x!='>') {cout <<"Error in production.\n";};
   getch(x); rsides rs1; rs1.null(); parsersides(rs1); sRs1.rhss=rs1;};

void parsegram(gram &gram1)
{syRs sRs1; parseprod(sRs1);
 if (x=='$') {gram1.null(); gram1.cons(sRs1);}
 else if (x==';') {getch(x); gram gram2; gram2.null();
                   parsegram(gram2); gram2.cons(sRs1); gram1=gram2;};};
//-------------------------------------------------------------------------------
//                      trace function
void trace(char* s, statelist stl)
{if (traceon) {cout <<"\n"; cout << s <<":\n"; stlprint(stl);};};
//-------------------------------------------------------------------------------
// visiting the search space: constructing new lists of states
void rightsides(char s, gram gr, rsides &rs)
{if (gr.isnull())
     {cout <<"No production for "<<s<<" in the grammar.\n";}
 else {syRs sR; sR=gr.head();
      if (sR.symb==s) {rs=sR.rhss;}
 else {gr.tail(); rightsides(s,gr,rs);};};};
//-------------------------------------------------------------------------------
//                      final function
bool final(state st)
{return (((st.sentform).isnull()) && ((st.instring).isnull()));}
//-------------------------------------------------------------------------------
//                      chop function
```

```
void chop(state &st)                //  &st : the state st is modified
{(st.sentform).tail(); (st.instring).tail();};
//-----------------------------------------------------------------------------
//                    expand function
void expand(rsides rs, charlist v, charlist w, statelist &stl)
{state st;
 if (rs.isnull()) {stl.null();}
 else {charlist cs; cs=rs.head();
       statelist stl1; stl1.null();
       rs.tail(); expand(rs,v,w,stl1);
       st.sentform=append(cs,v);
       st.instring=w;
       stl1.cons(st); stl=stl1;};};
//-----------------------------------------------------------------------------
//                    exists function
bool exists (statelist stl1, gram gr)
{if (stl1.isnull()) {return false;} else
{state st1; st1=stl1.head();         // depth first visit of the seach space
if (final(st1)) {return true;}
else if ((st1.sentform).isnull())
            {stl1.tail();
            trace("failed",stl1);    // tracing
            return exists(stl1,gr);}
else if (('A'<=(st1.sentform).head()) && ((st1.sentform).head()<='Z'))
            {statelist stl2; rsides rs; stl1.tail();
            rightsides((st1.sentform).head(),gr,rs);
            (st1.sentform).tail();
            expand(rs,st1.sentform,st1.instring,stl2);
            stl1=stappend(stl2,stl1);
            trace("expanded",stl1);  // tracing
            return exists(stl1,gr);}
else if ((('a'<=(st1.sentform).head()) && ((st1.sentform).head()<='z'))
        && ((st1.instring).isnull()))
            {stl1.tail();
            trace("failed",stl1);    // tracing
            return exists(stl1,gr);}
else if ((('a'<=(st1.sentform).head()) && ((st1.sentform).head()<='z'))
        && ((st1.sentform).head()==(st1.instring).head()))
            {chop(st1); stl1.tail(); stl1.cons(st1);
            trace("chopped",stl1);   // tracing
            return exists(stl1,gr);}
else if ((('a'<=(st1.sentform).head()) && ((st1.sentform).head()<='z'))
        && ((st1.sentform).head()!=(st1.instring).head()))
            {stl1.tail();
            trace("failed",stl1);    // tracing
            return exists(stl1,gr);}
else {cout <<"This should not happen!\n";}; };
};
//-----------------------------------------------------------------------------
//                    main program
void main(){
charlist stringtoparse;
stringtoparse.null();
//-----------------------------------------------------------------------------
//  from the string gg to the list gr0 of type gram which encodes
//  the given grammar
//-----------------------------------------------------------------------------
ind=0; f=gg; getch(x);
```

```
if (('A'<=x) && (x<='Z'))
   {cout <<"\nThe input grammar is ";
    cout <<"(`e' is reserved for the empty string):\n";
    parsegram(gr0);
/*--------------------------------------------------------------------------------
// In order to avoid parsing, the above statement 'parsegram(gr0)' can be
// replaced by the following statements:

syRs Prs; Prs.symb='P';                                                 // P
charlist chl1P; chl1P.cons('b');                                        //   b
charlist chl2P; chl2P.cons('Q'); chl2P.cons('Q'); chl2P.cons('a'); //    |aQQ
rsides rsP; rsP.cons(chl2P); rsP.cons(chl1P);                           //   b|aQQ
Prs.rhss=rsP;                                                           // P>b|aQQ

syRs Qrs; Qrs.symb = 'Q';                                               // Q
charlist chl1Q;                                                         //   e
charlist chl2Q; chl2Q.cons('b');                                        //    |b
charlist chl3Q; chl3Q.cons('P'); chl3Q.cons('b');                       //      |bP
rsides rsQ; rsQ.cons(chl3Q); rsQ.cons(chl2Q); rsQ.cons(chl1Q);          //   e|b|bP
Qrs.rhss=rsQ;                                                           // Q>e|b|bP

gr0.cons(Qrs); gr0.cons(Prs);  // gr0 corresponding to gg="P>b|aQQ;Q>e|b|bP"
//------------------------------------------------------------------------------ */
     printgram(gr0);}
else cout <<"Not a grammar in input.";
char axiom=gg[0];
cout <<"\n\nThe axiom is " << axiom <<".\n";
/*--------------------------------------------------------------------------------
 from the string stp to the list of characters stringtoparse of type charlist.
-------------------------------------------------------------------------------- */
ind=0; f=stp; getch(x);
parselistofchars(stringtoparse); // stringtoparse is produced
                                 // in output as list of chars
state st;
(st.sentform).null();
(st.sentform).cons(axiom);       // st.sentform=cons(axiom,NULL);
st.instring=stringtoparse;
statelist stl;
stl.null(); stl.cons(st);        // statelist stl=cons(st,NULL);
cout <<"The initial statelist is:\n"; stlprint(stl);
bool ans=exists(stl,gr0);
cout <<"\n\nThe following string of characters:\n  ";
print(stringtoparse); cout <<"\nis ";
if (!ans) {cout <<"NOT ";}; cout <<"generated by the given grammar.\n";
} // end of main
/* input: output:
--------------------------------------------------------------------
          The input grammar is ('e' is reserved for the empty string):
          { P ->  b  |  a Q Q
            Q ->  e  |  b  |  b P
          }

          The axiom is P.
          The initial statelist is:
            (P , a b )
          expanded:
            (b , a b )   (a Q Q , a b )
          failed:
```

```
          (a Q Q , a b )
        chopped:
          (Q Q , b )
        expanded:
          (Q , b )   (b Q , b )   (b P Q , b )
        expanded:
          (e , b )   (b , b )   (b P , b )   (b Q , b )   (b P Q , b )
        failed:
          (b , b )   (b P , b )   (b Q , b )   (b P Q , b )
        chopped:
          (e , e )   (b P , b )   (b Q , b )   (b P Q , b )

        The following string of characters:
          a b
        is generated by the given grammar.
------------------------------------------------------------------------ */
/* Examples already prepared:
char* gg="P>b|aQQ;Q>e|b|bP";       // example 0   - with empty production -
char* stp="a";                     // true
char* stp="abaa";                  // false
char* stp="aba";                   // true

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

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

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

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