Chapter 6

Symbolic Evaluation of Expressions

6.1 A Theorem Prover for Propositional Calculus

```
/* ==========================================================================
           CHECKING TAUTOLOGIES IN PROPOSITIONAL CALCULUS
-----------------------------------------------------------------------------
Propositional variables are named: 0, 1,..., max-1. The input propositional
expression is given according to the following (non left-recursive) grammar:
   element ::= (top)|'0'|...|'9'
   primary ::= element | - primary            - stands for not
   term    ::= primary | primary * term       * stands for and
   expr    ::= term | term + expr             + stands for or
   top     ::= expr | expr > top              > stands for implies
The internal representation of the expression is a tree.
-----------------------------------------------------------------------------
Type ';' and carriage return (CR) at the end of input expression. In the
output expression > is printed as ->. The internal representation of the input
expression has the integers 0,...,9 corresponding to the characters
'0',...,'9'.
The input expression is assumed to be correct according to the given grammar.
Priorities for bindings are:  -, *, +, > (to the left the strongest binding).
Right associativity is assumed. Parentheses overrule priority.
========================================================================== */
#include <iostream.h>
#include <stdlib.h>   // file needed for the EXIT_SUCCESS and the EXIT_FAILURE
const int max=4;
// max is the maximum number of variables in use: from 0 to max-1.
// max can be at most 10.
//---------------------------------------------------------------------------
enum binop {land,lor,impl};
enum Tag {variabletree, unarytree, binarytree};
struct tree {
       struct Variable {int var;};
       struct Unary    {tree* lnot;};
       struct Binary   {binop bop; tree* left; tree* right;};
       union {Variable l; Unary u; Binary b;};
       Tag tag;};

typedef tree* treeptr;
//---------------------------------------------------------------------------
treeptr ntp;
char x;         // x is a global variable
bool A[max]={}; // each element of the array A is initialized to false
//---------------------------------------------------------------------------
```

```
//                     from characters to integers
int v(char c) {return c-48;};
//---------------------------------------------------------------------------
//                     printing the array A
void printar(bool A[], int max)
{for (int i=0; i<max; i=i+1)
      {cout << i << "=" << (A[i]? "true":"false") << "; ";};
cout << "\n";}
//---------------------------------------------------------------------------
//                     printing the expression
void printtree(treeptr t)
{if (t==NULL) {cout << "nil";}
else {switch (t->tag){
case variabletree: cout << t->l.var; break;
case unarytree:    cout << "(-"; printtree(t->u.lnot); cout << ")"; break;
case binarytree:   cout << "("; printtree(t->b.left);
        switch (t->b.bop){
          case land: cout << " * "; break;
          case lor:  cout << " + "; break;
          case impl: cout << " -> "; break;
        default: cout << "Error 1 in printtree!";};
        printtree(t->b.right); cout << ")"; break;
default: cout << " Error 2 in printtree!"; exit(EXIT_FAILURE);}};}
//---------------------------------------------------------------------------
// declarations for mutual recursion
void parsetop    (treeptr &t);
void parseexpr   (treeptr &t);
void parseterm   (treeptr &t);
void parseprimary(treeptr &t);
void parseelem   (treeptr &t);
//---------------------------------------------------------------------------
void parsetop(treeptr &t)
{treeptr ntpD1=new tree; parseexpr(ntpD1);
if (x=='>') {treeptr ntpD=new tree; ntpD->tag=binarytree;
             ntpD->b.left=ntpD1; ntpD->b.bop=impl;
             cin >> x; parsetop(ntpD->b.right); t=ntpD;}
else {t=ntpD1;};}
//-------
void parseexpr(treeptr &t)
{treeptr ntpE1=new tree; parseterm(ntpE1);
if (x=='+') {treeptr ntpE=new tree; ntpE->tag=binarytree;
             ntpE->b.left=ntpE1; ntpE->b.bop=lor;
             cin >> x; parseexpr(ntpE->b.right); t=ntpE;}
else {t=ntpE1;};}
//-------
void parseterm(treeptr &t)
{treeptr ntpT1=new tree; parseprimary(ntpT1);
if (x=='*') {treeptr ntpT=new tree;
              ntpT->tag=binarytree;
              ntpT->b.left=ntpT1; ntpT->b.bop=land;
              cin >> x; parseterm(ntpT->b.right); t=ntpT;}
else {t=ntpT1;};}
//-------
void parseprimary(treeptr &t)
{treeptr ntpP=new tree;
if (x =='-') {ntpP->tag=unarytree; cin >> x;
           treeptr ntpP1=new tree;
           parseprimary(ntpP1);
```

```
           ntpP->u.lnot=ntpP1;}
else {parseelem(ntpP);}; t=ntpP;}
//-------
void parseelem(treeptr &t)
{treeptr ntpV=new tree;
if (('0' <= x) && (x <= '9'))
  {ntpV->tag=variabletree; ntpV->l.var=v(x); cin >> x;}
else {if (x=='(') {cin >> x; parsetop(ntpV);
      if (x==')') {cin >> x;};};};
t=ntpV;}
//-------------------------------------------------------------------------
//                    evaluation of a boolean expression
bool bval(treeptr t)
{bool e; switch (t->tag){
case variabletree: return A[t->l.var]; break;
case unarytree:    return !bval(t->u.lnot); break;
case binarytree:   e=bval(t->b.left);
                   switch (t->b.bop){
                   case land: if (e) {return bval(t->b.right);}
                              else {return false;}; break;
                   case lor:  if (e) {return true;}
                              else {return bval(t->b.right);}; break;
                   case impl: if (e) {return bval(t->b.right);}
                              else {return true;}; break;
                   default: cout << "Error 1 in bval!";};
default: cout << "Error 2 in bval!"; exit(EXIT_FAILURE);}};
//-------------------------------------------------------------------------
void main()
{cout << "Write an expression with variables from 0 to "
      << max-1 << ", please.\nThen type ; and CR.\n";
cout << "You may use the connectives -(not), +(or), *(and), ";
cout << "\n>(implies), and the parentheses (,).\n";
cin >> x; parsetop(ntp);
cout << "Boolean expression: "; printtree(ntp); cout << "\n";
/*------------------------------------------------------------------------
The following fragment computes the dispositions for A[0],..., A[max-1] from
false,...,false to true,...,true until it finds one which makes bval(ntp)=
false. Then it stops and says that the given formula is NOT a tautology. If,
otherwise, all dispositions make bval(ntp)=true, then it stops and says that
the given formula is a tautology.
------------------------------------------------------------------------ */
int lim=1; for (int i=0; i<max; i++) {lim=lim*2;};
bool res=true; int h=0;
while (h<lim) {if (bval(ntp)==false) {res=false; goto lab1;};
      for (int k=0; k<max; k++)
      {if (A[k]) {A[k]=false;} else {A[k]=true; goto lab2;};};
      lab2: h++;};
//-------------------------------------------------------------------------
lab1: cout << "\nIs it a tautology? ";
   if (res) {cout << "YES\n";}  else {cout << "NO\n";};
   if (! res) {cout << "Falsity assignment:\n";
      printar(A,max);};
}
/*  input:  output: (for max=4)
-------------------------------------------------------------------------
         Write an expression with variables from 0 to 3, please.
         You may use the connectives -(not), +(or), *(and),
         >(implies), and the parentheses (,).
```

```
0>1>0;CR
        Boolean expression: (0 -> (1 -> 0))

        Is it a tautology? YES
--------------------------------------------------------------------------------
0>-1;CR
        Boolean expression: (0 -> (-1))

        Is it a tautology? NO
        Falsity assignment:
        0=true; 1=true; 2=false; 3=false;
------------------------------------------------------------------------------ */
```

6.2 Formal Differentiation

```
/*==============================================================================
                     FORMAL DIFFERENTIATION
The input expression to be differentiated is given according to the following
grammar:
   element ::= (top)|'0'|...|'9'|'a'|...|'z'
   primary ::= element   | - primary                     (unary minus)
   term    ::= primary   | primary ^ term             (exponentiation)
   expr    ::= term      | term * expr    | term / expr  (times and division)
   top    ::= expr      | expr + top     | expr - top   (plus and minus)
The binding priorities are: most binding   -      (unary minus)
                                 |         ^      (exponentiation)
                                 |         *   /  (times and division)
                            least binding  +   -  (plus and binary minus))
In the exponentiation the exponent should not be dependent on the variable
w.r.t. which we differentiate.
Type semicolon and carriage return at the end of input expression. In the
output expression > is printed as ->.
Right associativity is assumed. This is usual for times (e.g., a*b*c =
a*(b*c)), but it is not usual for minus. Thus, a-b-c = a-(b-c), not (a-b)-c.
The input expression is assumed to be correct according to the given grammar.
--------------------------------------------------------------------------------
No use of the 'delete' instruction is made for improving storage utilization.
This is not a big disadvantage because when exiting a recursive call all local
variables are discarded. The output is not simplified: for instance, 0*x, y*1,
and 0+1 are not simplified to x, y, and 1, respectively.
============================================================================== */
#include <iostream.h>
#include <stdlib.h>   // file needed for the EXIT_SUCCESS and the EXIT_FAILURE
enum binop {exp,times,over,plus,minus};
enum Tag {variabletree, constanttree, unarymintree, binarytree};
struct tree {
       struct Variable {char vname;};
       struct Constant {char cname;};
       struct Unary    {tree* son;};
       struct Binary   {binop bop; tree* left; tree* right;};
       union {Variable v; Constant c; Unary u; Binary b;};
       Tag tag;};
typedef tree* treeptr;
//------------------------------------------------------------------------------
char x; char y;
//------------------------------------------------------------------------------
//                    constanttree from character z
```