McMaster University
Department of Computing and Software
Dr. W. Kahl

SFWR ENG 3E03
Exercise Sheet 2

Design and Selection of Programming Languages

20 September 2006, updated 21 September 2006

Exercise 2.1 — Context-Free Syntax: Exponents (Midterm 1, 2005)

For this question, the **abstract syntax** of expressions is defined by the following grammar:

$$\begin{array}{lll} \mathit{Expression} & \rightarrow & \mathit{Number} \mid \mathit{Expression}\ \mathit{Op}\ \mathit{Expression} \\ \mathit{Op} & \rightarrow & \texttt{*} \mid \texttt{/} \mid \texttt{\^{}} \end{array}$$

Define a **concrete syntax** for these expressions by giving a **context-free grammar (e.g. in EBNF)** such that

- the grammar is unambiguous,
- multiplication `*` and division `/` associate to the left,
- exponentiation `^` has higher precedence and associates to the right.

For example, the two strings "`2 / 3 ^ 4 ^ 5 / 7`" and "`(2 / (3 ^ (4 ^ 5))) / 7`" represent the same expression.

Solution Hints

$$\begin{array}{lll} \mathit{Expression} & \rightarrow & \mathit{Term} \mid \mathit{Expression}\ [\,\texttt{*} \mid \texttt{/}\,]\mathit{Term} \\ \mathit{Term} & \rightarrow & \mathit{Factor} \mid \mathit{Factor}\ \texttt{\^{}}\ \mathit{Term} \\ \mathit{Factor} & \rightarrow & \mathit{Number} \mid (\mathit{Expression}) \end{array}$$

This has a left-recursive rule for *Expression*, so is not directly suited for deriving a recursive-descent parser. To repair this, the first line would be changed into:

$\mathit{Expression} \rightarrow \mathit{Term}\ \{[\,\texttt{*} \mid \texttt{/}\,]\mathit{Term}\}^*$

For the second line, writing

$\mathit{Term} \rightarrow \mathit{Factor}\ \{\texttt{\^{}}\mathit{Factor}\}^*$

is acceptable, too — but the structure difference should be mentioned.

Exercise 2.2 — Expression Manipulation in Java

Substitution $e_1[v \mapsto e_2]$ of an expression e_2 for a variable v in an expression e_1 is defined as follows:

$$
\begin{array}{lcll}
v[v \mapsto e] & = & e & \\
w[v \mapsto e] & = & w & \text{if } v \neq w \\
k[v \mapsto e] & = & k & \text{for } k \in Num \\
(e_1 \oplus e_2)[v \mapsto e] & = & (e_1[v \mapsto e]) \oplus (e_2[v \mapsto e]) & \text{for } \oplus \in Op
\end{array}
$$

This exercise further modifies the expression classes of Exercise 1.2.

(a) Add an instance method *substituteVariable* that takes as arguments a variable, and an expression to be substituted into that variable, and **returns the result of the substitution** into the expression for which the method is called.

(b) Add an instance method *destructivelySubstituteVariable* that takes as arguments a variable, and an expression to be substituted into that variable, and **modifies the expression object for which the method is called** by performing the substitution.

(c) Discuss the difference between these two methods!

Solution Hints

The following contributed (imperfect — see the notes below) Java file also contains answers to Exercise 1.2.

```
import java.io.*;

/**
  Basic Operator class
 */
class Operator {
  private char _op;
  public Operator (char c) { _op = c; }
  public String toString() { return String.valueOf(_op); }
  public String writeDotGraph()
  {
        return ("\"" + _op + "\" [shape=\"circle\"];");
  }
  public String rootString() { return String.valueOf(_op);}
}

/**
  Abstract Expression class
 */
abstract class Expression {
  /** toString() - convert this expression into string form */
  public abstract String toString();

  /** writeDotGraph() - convert this expression into string form */
  public abstract String writeDotGraph();
```

```
    /** rootString()
     * @return String representation of the root of this expression's parse tree
     */
    public abstract String rootString();

    /**
     * Substitutes 'newexp' into this expression wherever 'v' occurs.
     * Returns the new expression (the original is not modified)
     */
    public abstract Expression substituteVariable(Variable v, Expression newexp);


} // Expression = Value + Variable + Binary

/**
  * Concrete Expression class - containing (integer) values
  */
class Value extends Expression { // Value = int
  private int intValue;
  public Value(int v)
  {
    intValue = v;
  }
  public String toString()
  {
    return Integer.toString(intValue);
  }
  public String writeDotGraph()
  {
        return ("\"" + intValue + "\" [shape=\"box\"];");
  }
  public String rootString() { return String.valueOf(intValue);}

  /* A variable substitution? That can't affect us! We're mere values! */
  public Expression substituteVariable(Variable v, Expression newexp)
  {
    return this;
  }

} // end of class Value

/**
  * Concrete Expression class - containing (string) variables
  */
class Variable extends Expression { // Variable = String
```

```
  private String name;
  public Variable(String n)
  {
    name = n;
  }

  public String toString()
  {
    return name;
  }
  public String writeDotGraph()
  {
        return ("\"" + name + "\" [shape=\"triangle\"];");
  }
  public String rootString() { return name;}

  /* For the variable case, we either return this object unchanged,
   * or return the expression to be substituted */
  public Expression substituteVariable(Variable v, Expression newexp)
  {
    //check - is this object the same as v?
    if(this.equals(v))
    {
      return newexp;
    }
    else
    {
      return this; //probably don't need to clone() this..
    }
  }

  public boolean equals(Variable other)
  {
    if (this.name == other.name)
    {
      return true;
    }
    else return false;
  }

} // end of class Variable

/**
 * Concrete Expression class - containing (binary) expressions
 */
```

```
class Binary extends Expression { // Binary = Expression × Operator × Expression
  private Operator op;
  private Expression term1, term2;
  public Binary(Operator in_op, Expression in_t1, Expression in_t2)
  {
    op = in_op;
    term1 = in_t1;
    term2 = in_t2;
  }

  /* This contains some extraneous parentheses, but we'll include them anyway
   * ( just ( to ( make ( everything ( clear ) ) ) ) ). */
  public String toString()
  {
    return "( " + term1.toString() + " " + op.toString() + " " + term2.toString() + " )";
  }

  public String writeDotGraph()
  {
        String result = op.writeDotGraph() + "\n"
                      + term1.writeDotGraph() + "\n"
                     + term2.writeDotGraph() + "\n"
                     + "\"" + op.rootString() + "\""
                     + " -> "
                     + "\"" + term1.rootString() + "\";\n"
                     + "\"" + op.rootString() + "\""
                     + " -> "
                     + "\"" + term2.rootString() + "\";";
        return result;
  }

  public String rootString() { return op.toString();}

  /* For the non-destructive case, just chain the substitution down the line
   * (pass the buck! pass the buck!) */
  public Expression substituteVariable(Variable v, Expression newexp)
  {
    return new Binary(op, term1.substituteVariable(v,newexp),term2.substituteVariable(v,newexp));
  }

} // end of class Binary


/**
 * ExprHandle - a 'handle' class for expressions
```

```
 *
 * This class exists to abstract away 'kinds' of expressions while still
 * giving us a concrete interface to use.
 *
 * Of course this only works if we restrict users to using only this class
 */
class ExprHandle {

  // We have a 'secret' expression 'exp'. Note this is an abstract type,
  // which allows us to make it ANY kind of expression we want.
  private Expression exp;

  //For shorthand purposes, we can use these constructors to create
  //atomic expressions
  public ExprHandle(int v) { exp = new Value(v);}
  public ExprHandle(String v) { exp = new Variable(v);}

  //Our private constructor allows us to create new handles based on our
  //secret expression
  private ExprHandle(Expression e) { exp = e; }

  /* We also have some public 'constructors' to allow the user to make
   * different kinds of expressions. In general, the user should not need
   * to care (nor know) what kind of expression resides in their ExprHandle
   * object. They all have the same interface, after all.
   *
   * Note there is an exception later on, though :p
   */
  public static ExprHandle makeValue(int v)
  {
    return new ExprHandle( new Value(v));
  }

  public static ExprHandle makeVariable(String v)
  {
    return new ExprHandle( new Variable(v));
  }

  public static ExprHandle makeBinary(Operator op, ExprHandle t1, ExprHandle t2 )
  {
    return new ExprHandle( new Binary(op, t1.exp, t2.exp));
  }

  /* And we have a couple functions here that we just pass through to the
   * internal expression…
```

```
 */
public String toString()
{
  return exp.toString();
}

public String writeDotGraph()
{
  return exp.writeDotGraph();
}

/* This is where it gets interesting. We NEED the first expression to be
 * of type Variable. This leads us to some run-time type-checking, and also
 * means the user must know which of their expressions are variables.
 *
 * Oh well.
 */
public ExprHandle substituteVariable(ExprHandle v, ExprHandle newexp)
{
  //This function uses something of a hack to ensure that v really is a variable.
  //Not really recommended.
  Expression var_candidate = v.exp;
  if(var_candidate instanceof Variable)
  {
    ExprHandle newhandle = new
ExprHandle(exp.substituteVariable((Variable)var_candidate,newexp.exp));
    return newhandle;
  } else {
    System.out.println("Error - can't subtite an expression into a non-variable.");
    //just return this object unchanged..
    return this;
  }

}

/* Same deal here, except we assign the result of a substitution to our secret
 * expression…
 */
public void destructivelySubstituteVariable(ExprHandle v, ExprHandle newexp)
{
  //This function uses something of a hack to ensure that v really is a variable.
  //Not really recommended.
  Expression var_candidate = v.exp;
  if(var_candidate instanceof Variable)
  {
```

```
        exp = exp.substituteVariable((Variable)var_candidate,newexp.exp);
      } else {
        System.out.println("Error - can't substitute an expression into a non-variable.");
      }
    }
} // end of class ExprHandle



/**
  Expression2 - Our 'main' class
 */
public class Expression2
{
  /* This is a hack variable to 'generate unique temp file names' so we
   * can display several graphs in one run.
   */
  private static int file_count = 0;

  /* Basic file output function */
  private static void writeFile(String filename, String contents)
  {
    //taken from http://www.javacoffeebreak.com/java103/java103.html#output
    FileOutputStream out; // declare a file output object
    PrintStream p; // declare a print stream object

    try
    {
      // Create a new file output stream
      // connected to "filename"
      out = new FileOutputStream(filename);

      // Connect print stream to the output stream
      p = new PrintStream( out );

      p.print(contents);

      p.close();
    }
    catch (Exception e)
    {
      System.err.println ("Error writing to file"); // My, how descriptive!
    }
  }
```

```
/* Our top-level graph display function.
 * It does the best it can at faking a .dot format file, writing the header
 * and footer on its own (and calling upon the ExprHandle to write itself)
 */
private static void showDotGraph(ExprHandle e)
{

        String header = new String();
        header = "digraph MT1a { \n"
                        + "node [fontsize=\"30\"];\n"
                      + "edge [labeldistance=\"1\",fontsize=\"30\"];" ;
        String footer = new String( "}");

        String contents = header + "\n" + e.writeDotGraph() + "\n" + footer + "\n";
  String filename = "temp" + file_count +".dot";

        writeFile(filename,contents);
  file_count++; //so the next filename is different :p

        try {
                Runtime.getRuntime().exec("dotty " + filename);
        }
        catch (Exception ex)
        {
   // Not the best error handling! Oh well :)
                System.out.print("Exception. You probably don't have 'dotty' installed.\n");
   System.out.print("Make sure you're running Linux and have the graphviz package installed.\n");
        }
  finally {
     // if you wanted to delete 'temp.dot', you could do that here.
     // For the time being I'll leave it alone so you can see the file
     // again after the program ends.
  }
}

/* Finally, our main function. It's kinda cobbled together, but… */
public static void main(String[] args)
{
  //Create several operators and subexpressions, building up piece-wise:

  //Some of these operators are 'creative' since having the same operator
  // used in multiple expressions kinda makes the graph come out odd.
  Operator a1 = new Operator('*');
  Operator a2 = new Operator('+');
  Operator a3 = new Operator('^');
```

```
    Operator a4 = new Operator('-');
    Operator a5 = new Operator('/');
    Operator a6 = new Operator('.');
    ExprHandle e1 = ExprHandle.makeBinary(a1, new ExprHandle(3), new ExprHandle("x"));
    ExprHandle e2 = ExprHandle.makeBinary(a2, e1, new ExprHandle("y"));
    ExprHandle e3 = ExprHandle.makeBinary(a4, new ExprHandle(10), new ExprHandle(–1));
    ExprHandle e4 = ExprHandle.makeBinary(a3, new ExprHandle(0), new ExprHandle("y"));
    ExprHandle e5 = ExprHandle.makeBinary(a6, e3, e4);
    ExprHandle e6 = ExprHandle.makeBinary(a5, e2, e5);

    //Print the original graph
    System.out.println(e6.toString());
    showDotGraph(e6);

    //do some substitutions. Of course 'zz' is the new expression, but 'e6'
    // hasn't been changed yet..
    ExprHandle v1 = new ExprHandle("y");
    ExprHandle v2 = new ExprHandle("z");
    ExprHandle v3 = new ExprHandle(42);
    Operator a7 = new Operator('%');
    ExprHandle v4 = ExprHandle.makeBinary(a7,v2,v3);
    ExprHandle zz = e6.substituteVariable(v1,v4);
    System.out.println(zz.toString());
    showDotGraph(zz);

    //A destructive substitution! Now 'e6' certainly has changed:
    ExprHandle v5 = new ExprHandle("x");
    e6.destructivelySubstituteVariable(v5,v4);
    System.out.println(e6.toString());
    showDotGraph(e6);
  }
} // end of class Expression2
```

Notes:

- dot output is generated here at the string level, i.e., at the level of concrete dot syntax. More separation of concerns could be achieved by having a separate class that handles generation of concrete dot syntax and exposes an interface oriented at an appropriate abstract syntax of dot files.
- The comments in the *main* method explain that multiple occurrences of the same operator make the *dot* graph come out "odd" — each node of the abstract syntax tree should of course produce a different node in the *dot* output, and for this you need to create node identities. This could usefully be done in the special dot class. A direct solution would have *writeDotGraph* accept a counter reference as argument, increment this for every node, use it for node identities in the dot output,

and take care that the top node of each subgraph obtains the last counter value so that the parent knows which node to link to.

- The *ExprHandle* class mainly exists to enable destructive substitution, which requires a reference to a reference to an object of a subclass of *Expression*.
- The first argument of *substituteVariable* could also have been chosen to be a *Variable* or even just a variable name, namely a *String*.

Exercise 2.3 — Expression Parsing and Manipulation in C

Extend the C datatype for expressions and the simple bison-based calculator presented in the lecture (source files are available on the course page) with the following functionality — carefully define and document the interfaces:

(a) Add a function for producing string representations from expressions.

(b) Add an exponentiation operator.

(c) Add destructive and non-destructive substitution functions as in Exercise 2.2.

(d) Further modify the simple calculator presented in class so that it accepts definitions of variables, introduced by the keyword "let":

```
let x = 4
let y = 5
x+y
 = 9
```

(e) Further modify the simple calculator presented in class so that it produces step-wise evaluation traces:

```
(4+3) * 8 - 2*7
 = (4 + 3) * 8 - 2 * 7
 = 7 * 8 - 2 * 7
 = 56 - 2 * 7
 = 56 - 14
 = 42
```

Solution Hints

`Expr.h`

```
// pointer types need not have declared destination struct type!
typedef struct ExprStruct * Expr ;                    // Expr.h
typedef struct SymRecStruct * SymRec;

extern Expr exprInt(long int n);
```

```
extern Expr exprVar(char * ident);
extern Expr exprBin(char * op, Expr e1, Expr e2);
extern long int exprEval(Expr e);

extern SymRec mkSymrec(const char* varname, Expr exp);
```

`Expr.c`

```
#include <stdlib.h>                    //  Expr.c
#include <stdio.h>
#include <string.h>
#include <math.h>          //  for pow

// Two structure 'handle's : one for expressions, one for records
// in a symbol table
typedef struct ExprStruct * Expr;
typedef struct SymRecStruct * SymRec;

/* Here we use the infamous 'enum hack' to allow us to reference this value
  inside array declarations. This is now obsolete and should not be used
  for C++ ! */
enum { MAX_OP_LEN = 4 };


typedef struct {   Expr left;
                   char op[MAX_OP_LEN];        // only short operators!
                   Expr right;
           } BinRec;

typedef enum { tagNum, tagVar, tagBin } Tag;

struct ExprStruct {  // record containing tagged union
  Tag tag;
  union {          long int num;  // for tagNum
                   char * name;   // for tagVar
                   BinRec bin;    // for tagBin
         } u;                                    // Note the struct field label  "u"
};

/* Data type for records in the global symbol table.  */
struct SymRecStruct
 {
   char* name;
   Expr bound_exp;
 } ;
```

```
/* Symbol table */
// Fixed allocation, but should be all right for demonstration purposes
// (don't try this at home, kids!)
SymRec symbolTable[100];
int symbolsUsed = 0;

// -------------------------------------
// addToTable(rec)
//   - adds a single symbol to the global symbol table.
//
//  rec: The record to add
//  Result: N/A
// ----------------------------------------
void addToTable(SymRec rec)
{
  //first check to make sure the symbol is not already there
  int i;
  for(i = 0; i < symbolsUsed; i++)
  {
    if(strcmp(symbolTable[i]→name,rec→name) == 0)
    {
      // then overwrite current definition
      free(symbolTable[i]);
      symbolTable[i] = rec;
      return;
    }
  }
  //otherwise, it's new.
  symbolTable[symbolsUsed] = rec;
  symbolsUsed++;
}

// -------------------------------------
// mkSymrec(varname, exp)
//   - creates a record for use in the global symbol table
//
// varname: The symbol's string representation (i.e. variable name)
// exp: The expression that the above symbol represents
// Result: The created symbol table record
// ----------------------------------------
SymRec mkSymrec(const char* varname, Expr exp)
{
  SymRec new_record = (SymRec)malloc(sizeof(struct SymRecStruct));
  new_record→name = (char*)malloc((strlen(varname) + 1) * sizeof(char));
```

```
  strcpy(new_record→name,varname);
  new_record→bound_exp = exp;
  printf("%s\n",varname);
  return new_record;
}

// ------------------------------------
// freeExpr(exp)
//   - frees memory allocated to the given expression
//
// exp: The expression to be removed from memory.
// Result: N/A
// ----------------------------------------
void freeExpr(Expr exp)
{
  if(exp == NULL) return; //nothing to do..
  if(exp→tag ≠ tagBin)
  {
    //then just free the given expression
    free(exp);
    exp = NULL;
  }
  else
  {
    //here we have to call freeExp on the binary components.
    freeExpr(exp→u.bin.left);
    freeExpr(exp→u.bin.right);
    //and then free the toplevel
    free(exp);
    exp=NULL;
  }
}
// ------------------------------------
// toString(exp)
//   - converts exp to its string representation (exp is not modified)
//
//  exp: The expression to be 'stringified'
//  Result: A string containing the 'stringified' expression. Caller frees this!
// ----------------------------------------
char* toString(const Expr exp)
{
  char* newstr;
  int numchars;
  switch( exp→tag)
  {
```

```
  case tagNum:
    {
      numchars = (int) (log10(abs(exp→u.num))) + 1;
      newstr = (char *)malloc((numchars + 1) * sizeof(char));
      snprintf(newstr,(numchars + 1), "%d",exp→u.num);
      break;
    }
  case tagVar:
    {
      newstr = strdup(exp→u.name);
      break;
    }
  case tagBin:
    {
      char* leftstr = toString(exp→u.bin.left);
      char* rightstr = toString(exp→u.bin.right);
      numchars = strlen(leftstr) + strlen(rightstr) + MAX_OP_LEN + 3; // allow 3 for parentheses
      newstr = (char*)malloc(numchars * sizeof(char));
      strcpy(newstr,"(");
      strcat(newstr,leftstr);
      strcat(newstr,exp→u.bin.op);
      strcat(newstr,rightstr);
      strcat(newstr,")");
      free(leftstr);
      free(rightstr);
      break;
    }
  default:
    {
      const char errorexp[] = "error";
      newstr = (char*)malloc(strlen(errorexp) * sizeof(char));
      strcpy(newstr,errorexp);
      break;
    }
  }
  return newstr;
}

// --------------------------------------
// printSymbolTable()
//   - displays the current symbol table on-screen
//    (useful for demonstration purposes)
//
//  Result: N/A
// ----------------------------------------
```

```
void printSymbolTable()
{
   int i = 0;
   SymRec recptr;

   printf("Symbol table : \n");
   for(i = 0; i < symbolsUsed; i++)
   {
      recptr = symbolTable[i];
      printf("[ %s , %s ]\n", recptr→name, toString(recptr→bound_exp));
   }

}

/* *********************************************
 * exprX()
 *   - creates an expression from its argument.
 ********************************************* */
Expr exprInt(long int n) {
  Expr result = malloc(sizeof(struct ExprStruct));
  if ( result == NULL) return NULL;
  result→tag = tagNum;
  result→u.num = n;
  return result;
}
Expr exprVar(const char * ident) {
  Expr result = malloc(sizeof(struct ExprStruct));
  if ( result == NULL) return NULL;
  result→tag = tagVar;
  result→u.name = strdup(ident);
  return result;
}

Expr exprBin(const char * op, Expr e1, Expr e2) {
  if ( op == NULL || strlen(op) > 3 ) return NULL;
  Expr result = malloc(sizeof(struct ExprStruct));
  if ( result == NULL ) return NULL;
  result→tag = tagBin;
  result→u.bin.left = e1;
  result→u.bin.right = e2;
  strcpy(result→u.bin.op, op);
  return result;
}

// ------------------------------------
```

```
// substituteVariable(thisexp, var,newexp)
//   - substitutes one expression into another wherever 'var' occurs
//
//  thisexp: The base expression to be operated upon (here, the original
//      version of 'thisexp' is not modified)
//  var : The variable to perform substitution for
//  newexp: The expression to replace 'var' in 'thisexp'
//  Result: A new expression holding the substituted expression. (Caller frees!)
// ----------------------------------------
Expr substituteVariable(Expr thisexp, Expr var, Expr newexp)
{
  if(var→tag ≠ tagVar)
  {
    printf("Error encountered - must substitute expression for variable!\n");
    //return passed-in expression as default.
    return thisexp;
  }

  switch(thisexp→tag)
  {
  case tagNum:
    {
      return thisexp;
      break;
    }
  case tagVar:
    {
      if(strcmp(thisexp→u.name,var→u.name) == 0)
      {
        return newexp;
      }
      else
      {
        return thisexp; //no change
      }
      break;
    }
  case tagBin:
    {
      return exprBin(thisexp→u.bin.op,
            substituteVariable(thisexp→u.bin.left, var, newexp),
            substituteVariable(thisexp→u.bin.right, var, newexp) );
      break;
    }
  default:
```

```
    {
      printf("Error - invalid expression type.\n");
      return thisexp;
    }
  }

}


// ------------------------------------
// destructivelySubstituteVariable(thisexp, var,newexp)
//   - substitutes one expression into another wherever 'var' occurs
//
//  thisexp: A reference to the base expression to be operated upon
//      (here, the original version of '*thisexp' IS modified)
//  var : The variable to perform substitution for
//  newexp: The expression to replace 'var' in '*thisexp'
//  Result: N/A
// ----------------------------------------
void destructivelySubstituteVariable(Expr * thisexp, Expr var, Expr newexp)
{
  if(var→tag ≠ tagVar)
  {
    printf("Error encountered - must substitute expression for variable!\n");
    //and leave:
    return;
  }

  switch((*thisexp)→tag)
  {
  case tagNum:
    {
      return;
      break;
    }
  case tagVar:
    {
      if(strcmp((*thisexp)→u.name,var→u.name) == 0)
      {
        freeExpr(*thisexp); //delete old variable
        *thisexp = newexp;
        return;
      }
      else
      {
```

```
            return; //no change
          }
          break;
        }
      case tagBin:
        {
          destructivelySubstituteVariable(&((*thisexp)→u.bin.left), var, newexp);
          destructivelySubstituteVariable(&((*thisexp)→u.bin.right), var, newexp);
          break;
        }
      default:
        {
          printf("Error - invalid expression type.\n");
          return; //leave in a huff.
        }
      }

}


// ------------------------------------
// exprEval(e)
//   - evaluates an expression completely
//
//  e : The expression to evaluate
//  Result: The numeric value of the expression
//  Exception: Program terminates upon an invalid expression,
//    or encountering an unitialized variable.
//--------------------------------------
long int exprEval(Expr e) {
  switch (e→tag) {
    case tagNum:
      return e→u.num;
    case tagBin:
      {
        long int val1 = exprEval(e→u.bin.left);
        long int val2 = exprEval(e→u.bin.right);
        switch ( e→u.bin.op[0] ) {              // only for demonstration!
          case '+':   return val1 + val2;
          case '-':   return val1 – val2;
          case '*':   return val1 * val2;
          case '/':   return val1 / val2;
          case '^':   return (int) pow((double)val1,(double)val2);
          default:   fprintf(stderr,"exprEval: illegal operator '%s'\n", e→u.bin.op);
        }
```

```
      }
      break;
    case tagVar:
      {
        //look up in symbol table
        int i;
        for(i = 0 ; i < symbolsUsed; i++)
        {
          if(strcmp(symbolTable[i]→name,e→u.name) == 0)
          {
            return exprEval(symbolTable[i]→bound_exp);
          }
        }
        //otherwise we're screwed.
        fprintf(stderr,"exprEval: unexpected variable '%s'\n", e→u.name);
      }
      break;
    default:
      fprintf(stderr,"exprEval: illegal tag\n");
  }
  exit(1);
}                                   // all error exit goes through this



// ------------------------------------
// exprReduce(Expr e)
//   - reduce an expression by at most one step. If a reduction
//  is performed, print the result.
//
//  e : The expression to evaluate
//  Result: Either the expression reduced one step, or the original expression
//    (if it cannot be reduced further)
//  Exception: Program terminates upon an invalid expression,
//    or encountering an unitialized variable.
//
//  There are three main cases: Either a substitution takes place,
//   part of a binary expression is reduced, or all possible reduction has
//   been done.
//--------------------------------------
Expr exprReduce(Expr e) {
  switch (e→tag) {
    case tagNum:
      return e;
    case tagBin:
```

```
    {
      //check the left half..
      if(e→u.bin.left→tag ≠ tagNum)
      {
        return exprBin(e→u.bin.op, exprReduce(e→u.bin.left), e→u.bin.right);
      }
      else
      {
        //check the right half..
        if(e→u.bin.right→tag ≠ tagNum)
        {
          return exprBin(e→u.bin.op, e→u.bin.left, exprReduce(e→u.bin.right));
        }
        else
        {
          //if here, *both* sides are reduced already. So we can
          // fully evaluate them.
          return exprInt(exprEval(e));
        }
      }

    }
    break;
  case tagVar:
    {
      //look up in symbol table
      int i;
      for(i = 0 ; i < symbolsUsed; i++)
      {
        if(strcmp(symbolTable[i]→name,e→u.name) == 0)
        {
          return symbolTable[i]→bound_exp;
        }
      }
      //otherwise we’re screwed.
      fprintf(stderr,"exprReduce: unexpected variable ‘%s’\n", e→u.name);
    }
    break;
  default:
    fprintf(stderr,"exprReduce: illegal tag\n");
  }
  exit(1);
}                              // all error exit goes through this

// ------------------------------------
```

```
// printTrace(e)
//    - print the trace of an expression for each reduction until it reaches
//   full evaluation (i.e. a tagNum type).
//
//  e : The expression to trace
//  Result : N/A
// -------------------------------------
void printTrace(Expr e)
{
  Expr traceExpr = e;
  do
  {
    char* tracestr = toString(traceExpr);
    printf(" = %s\n",tracestr);
    free(tracestr);
    traceExpr = exprReduce(traceExpr);
  } while(traceExpr→tag ≠ tagNum);
}
```

Notes:

- Memory management is imperfect:
  - Overwriting symbol table entries does not free the overwritten expression.
  - Substitution does not copy literals and non-substituted variables, and therefore can introduce sharing. In the presence of possible sharing, *freeExpr* is unsafe.

    A "copy constructor" for *Expr* would be a natural solution.
- I would not put break after return in a switch, but you could call it a matter of taste…
- *destructivelySubstituteVariable* again has a reference to a reference to an object, i.e., a pointer to a pointer to a struct, as argument.
- Another calling convention for *exprReduce* could use the *NULL* variant of its return type to indicate that no reduction happened.

`simple_lexer.l`

```
%option noyywrap outfile="simple_lexer.c"
/* scanner for a toy calculator                    simple_lexer.l  */
%{
#include "Expr.h"                  /* required for the types in next line */
#include "simple_parser.tab.h"          /* token definitions and types */
%}
%%
```

```
[0–9]+            yylval.intval = atoi(yytext); return TOK_NUMBER;
if                return TOK_IF;
then              return TOK_THEN;
else              return TOK_ELSE;
let               return TOK_LET;
[=]               return TOK_EQ;
[a–z][a–z0–9]*    yylval.string = strdup(yytext); return TOK_ID;
[ \t]+            /* eat up whitespace */
[+\–*/()^\n]      { return yytext[0];}
.                 fprintf(stderr, "Unrecognized character: %s\n", yytext ); return –1;
```

---

simple_parser.y

```
%{
 #include <stdio.h>
 #include "Expr.h"
 int yylex(void);
 void yyerror (char const * s) { fprintf(stderr, "%s\n", s); }
%}
%union {
 long int intval;
 char *  string;
 Expr    expr;
 SymRec  rec;
};
%token <intval> TOK_NUMBER
%token <string> TOK_ID
%token TOK_IF TOK_THEN TOK_ELSE
%token TOK_LET TOK_EQ
%type <expr> expr term
%type <rec> vardec


%left '+' '-'
%left '*' '/'
%right '^'

%start input
%%

vardec : TOK_LET TOK_ID TOK_EQ expr  { $$ = mkSymrec($2,$4); }

term : TOK_NUMBER       { $$ = exprInt($1); }
     | TOK_ID           { $$ = exprVar($1); }
     | '(' expr ')'     { $$ = $2; }
```

```
expr : term
    | expr '^' expr { $$ = exprBin("^", $1, $3); }
    | expr '*' expr  { $$ = exprBin("*", $1, $3); }
    | expr '/' expr  { $$ = exprBin("/", $1, $3); }
    | expr '+' expr  { $$ = exprBin("+", $1, $3); }
    | expr '-' expr  { $$ = exprBin("-", $1, $3); }

input : /* empty */ | input line          /* line-by-line processing */

line : '\n'               {}                        /* empty lines allowed */
    | vardec '\n'        { addToTable($1); printSymbolTable(); }
    | expr '\n'          { printTrace($1); printf(" = %ld\n", exprEval($1)); freeExpr($1);}
%%
int main ( void ) { return yyparse(); }
```

Makefile

```
CC = gcc
simple_calc: simple_parser.tab.o simple_lexer.o Expr.o
	$(CC) $(CFLAGS) –lm –o $@ $^

simple_parser.tab.h simple_parser.tab.c: simple_parser.y
	bison –d $<

simple_lexer.o: simple_parser.tab.h Expr.h
simple_parser.tab.o: Expr.h
```
