Implementation of a simple calculator using flex and bison

Somasundaram Meiyappan

Abstract:

A simple calculator program is created using the compiler tools – flex and bison - and using the C programming language. The calculator takes is simple numerical expressions as input and evaluate them to give the result of the evaluation to the user. Apart from providing some standard mathematical functions and capabilities, this implementation also lets the user to define functions that will extend the built-in capabilities of the calculator. It also generates *vcg* files for all trees that the program constructs giving the user a glimpse of the internals.

Description:

This assignment is about creating a simple calculator using flex and bison with C as the programming language. This implementation of the assignment can take in simple numerical expressions, evaluate them and give the result of the evaluation to the user. This calculator provides the user with some standard mathematical and logical operators and some standard built-in functions. The user can define his/her own functions that use these capabilities and can pass them normally just as the other expressions are entered or as files in the command line itself. These functions have to defined in the language of the calculator and will be described shortly. The program can also used to generate *vcg* files for all trees that it constructs giving the user a glimpse of the internals. These trees are used when evaluating an expression or function. These *vcg* files can be viewed using graph visualizing tools like *xvcg* or *aiSee* (after changing the extension to *gdl*).

Here is an overview of the features:

1. Standard mathematical operators like +, -, * and / that takes in two expressions with one on the left and the other on the right. eg: 4+2, 4 + 6 * 5
2. .mod. operator that gives the user the remainder of a division operation. Eg.: 17 .mod. 3 gives the result 2 and 17 .mod. 4 gives the result 1
3. Bitwise operators - .and., .or., .xor. and .not. representing and, or, xor and not operations respectively. The not operation computes the complement of the input integer and operates on a 8-byte long integer. Eg.: 15 .and. 3 gives 3 as the result.
4. .rshift. and .lshift. operators can be used to right shift or left shift the left-hand side operand by a fixed number of bits defined by the right-hand side operand.
5. Common logical operators like <, >, <=, >=, ==, != and ! are also supported with an integer value 1 representing true and 0 representing false. This can be used to execute some expressions in a function conditionally just like predicated instructions. Eg.: cond(x,y)=(x)*y*10+(!(x)*y*100) returns 100 times y when x is zero and 10 times y when x is non-zero.
6. sin, cos, tan and their inverse functions – asin, acos, atan- are supported as well. Sinusoidal functions assume that the inputs are in radians and the inverse sinusoidal functions give the output in radians as well.
7. Other functions like log, ln, ceil, floor are also supported. All built-in functions take only one input parameter; but this is not a limitation of the implementation and just that built-in functions that take in more than one parameter are not available.
8. x^y to represent x raised to the power y. eg.: 3^2 gives the result 9

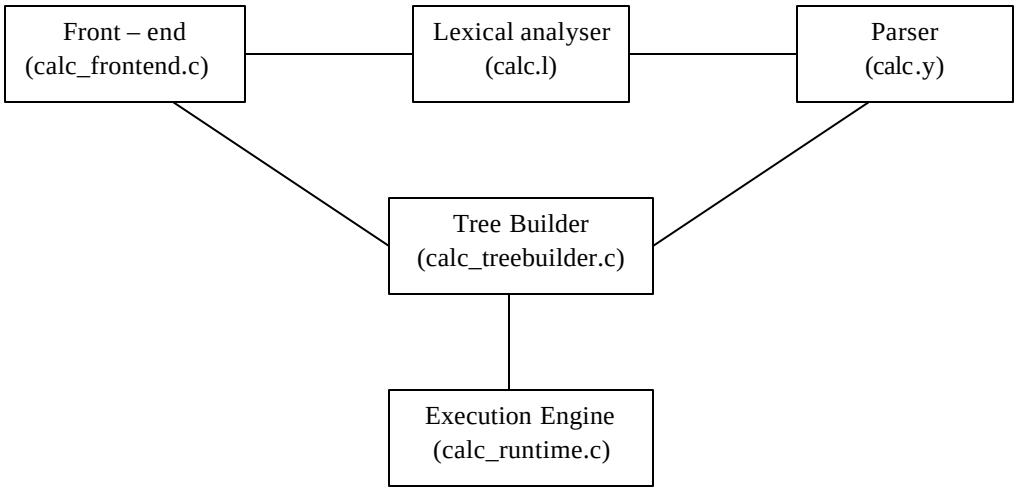
9. |x| gives the absolute value of x. eg.: |-3| gives the result 3.
10. All expressions are evaluated are evaluated from left to right with +, - having a lower precedence than the other operators. Eg.: 2 – 3 * 4 gives the result '–10.'
11. Parenthesis can be used to force the interpreter to evaluate that expression prior to executing the ones outside it. Eg.: (2-3)*4 gives the result '-4'.
12. User can define functions using the above operators and functions like in the following example.

```
f(x,y,z)=2*sin(x)*cos(y)+|x-PI/4|+tan(z+PI/4)
```

13. Generate VCG files representing all the user-defined functions and expression defined/evaluated. This is used to show the user the internal representation of the various expressions. This is accomplished using the '-g' option when starting the program in the command line.
14. Comments can be enclosed within /* …. */ following C style comments
15. If an expression or a function definition takes more than a line, the user can optionally use '\' to signal the interpreter to use the next line as a part of this line.
16. Usage: mycalc.exe [-Lfilename -g]
    -Lfilename: Used to add new functions to the function library
    -g: Writes .vcg files of all functions and expressions

Code organization and system diagram:

The following diagram gives the overall organization of the program.



This section describes the code organization.

- lex_src : contains the *.l files that are inputs to flex, the lexical analyzer generator
  - calc.l
- yacc_src: contains the *.y files that are input to bison, the parser generator
  - calc.y
- src: contains the *.c source files
  - calc_runtime.c

    - calc_treebuilder.c
    - calc_frontend.c
- inc: contains the *.h header files
    - more_types.h
    - calc_struct.h
    - calc_operators.h
    - calc_treebuilder.h

Lexical analyser and Parser:

The lexical analyzer, that is generated using flex, emits tokens to the parser for use in its grammar. The lexical analyzer uses some services from the *calc_frontend.c* file to get some tokens. For example: the function *isThisUserDefinedFunction* searches the function table, that has the list of all functions defined by the user, and tells the parser that this is a user-defined function and also passes the unique index of the function. Another example is the *isThisAnOperator* function, which not only tells the lexer that the passed in string is an operator, but also gives the token of that corresponds to that operator. Some unrecognized stream of input are reported as error directly by the lexical analyzer whereas some other unrecognized characters are passed to the parser which signals that we have an error. This file also hosts the main function.

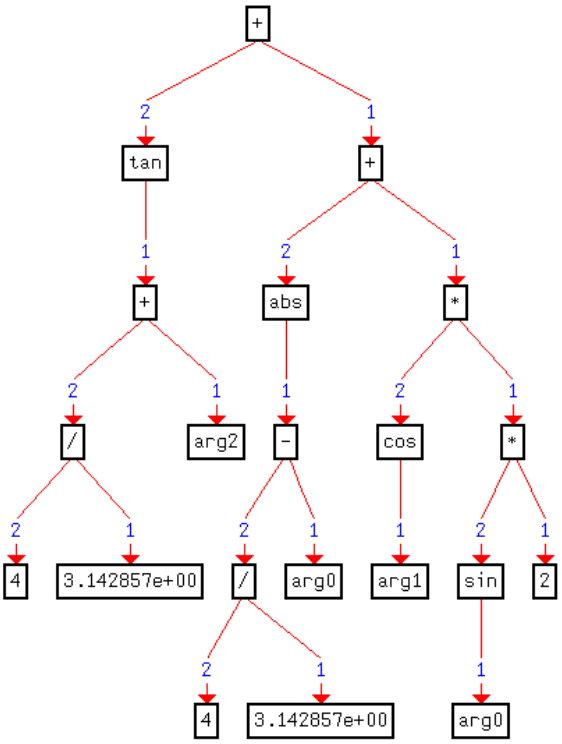
The parser, that is generated using bison, uses the tokens from the lexical analyzer in reducing the various productions that are in grammar of the language accepted by the program. The parser uses the services provided by the *calc_treebuilder.c* file to build its parse-trees that are later used in execution. All allocations are kept track of locally and the memory is released either on successful execution or on error conditions. In the case of user-defined functions, the memory is not released as the functions are intended to be visible for this entire instance of the program.

Trees involved and evaluation of expressions:

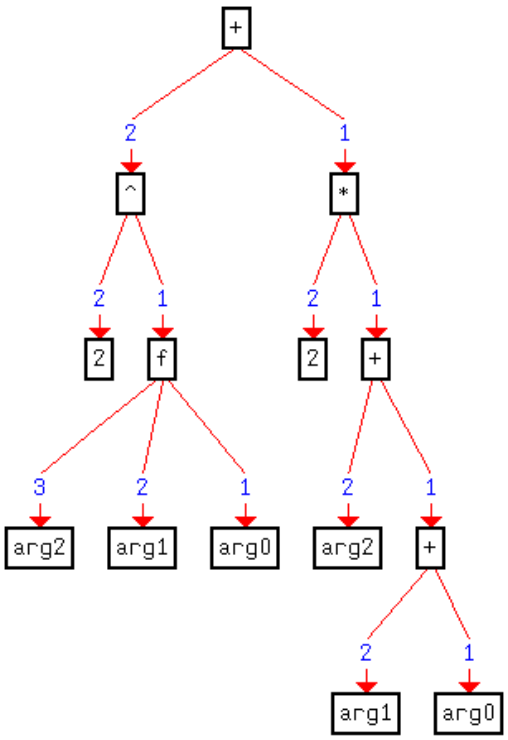
The parser generates trees for all expressions and then reduces them following certain rules to evaluate them. Parse trees are also generated for user-defined functions and parameters are passed into these trees at appropriate places for evaluating the function. Examples of the internal trees and expressions are given below.

The trees are evaluated by post-order traversal of the tree. In the last tree below, g(0,0,0) is represented with ‘g’ as the parent of three children, one corresponding to each parameter. The edge, with label 1, points to the first parameter 0 and so on. This also means that the value of g(0,0,0) will be computed first and then 45*3 will be computed and then (g(0,0,0)) + (45*3) will be computed.

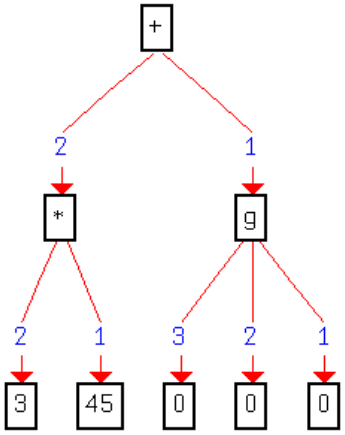
`f(x,y,z)=2*sin(x)*cos(y)+|x-PI/4|+tan(z+PI/4)` gets translated into



`g(x,y,z)=(x+y+z)*2+(f(x,y,z)^2)` gets translated into



The user defined function `g(0,0,0)+ 45 * 3` gets translated into



For operators, please note that the left hand side of the operand actually is the edge labeled 1 and not the one that is labeled 2 as above. This could be slightly confusing for operations like / and -. Please take note of this.

An entry in the table of functions has the following form

```
struct TblFunctions{
  char *FuncName;               /* Null-terminated string that gives the name
of the function */
  short nArgs;                  /* Number of arguments that this function takes
*/
  enum EnumDataType *pArgType;  /* List of the data type of the parameters:
from left to right */
  union NodeValue *pArgs;       /* List of pointers to the arguments' value
holder */
                               /* Note that these place-holders should be used
in TreeNode */
  enum EnumDataType RetType;    /* Return type of the function */
  struct TreeNode *RootNode;    /* Root Node of the function: Only used for
user-defined ones */
};
```

Parameters that go into to a function are actually placed in the respective locations in pArgs member. And the arg0, arg1, arg2 that in the trees of f(x,y,z) and g(x,y,z) point to these locations in the function table and hence, they have the values directly.

Traversing the parse-tree in pre-order generates the *vcg* files. Please note that the label name of the edges and they can be useful in cases like above when 45 which is edge 1 for * operator comes to the right of 3, which is edge 2.

Future enhancements:

With this infrastructure, variables can be allowed very easily. These variables can be used to store results of some expressions for use later in the session. This was not included in this implementation because of lack of time. User definable functions can be made to have more than 1 statement. Also, support for loops can also be added. After all, we can try to bring all the features that C has.