ENSC 251 – Summer 2017 – Course Project Part 5

Due: August 8, 2017, 8:35 AM (morning!)

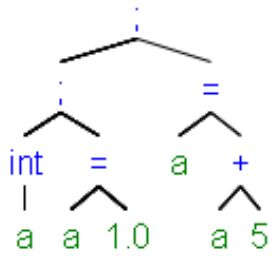
For our course project so far, we have developed code for tokenization of c-language source and classification of those tokens into their respective classes. We also developed a generic TreeNode class for building trees for different applications and then we used the same for creating abstract syntax trees using recursive descent parsing based on our provided BNF grammar rules.

The next phase for our course project is to interpret or evaluate an Abstract Syntax Tree (AST) generated in part 4. Note that we have only worked on a limited set of operators for this project and you will be working on evaluation for the same the same operators.

The flow of evaluation is somewhat similar to what we have been doing in parts 3 and 4 of the project. In order to evaluate an AST we need to traverse it, or at least parts of it (one of the operands of any conditional expressions doesn't need to be evaluated). Because of the custom traversal needs for evaluation, each class of Token has a virtual 'eval' function that implicitly does any needed traversal of subtrees for any instance of the class.

You will be using *evalMap* (an instance of a class based on the map class template from the Standard Template Library) for storing the *type_spec* of each identifier and also the last value assigned to each identifier (i.e. variable) during evaluation.

Consider that we want to evaluate the below mentioned AST:



(1) We come across the type_spec ‘int’. We create a new entry in *evalMap*, storing a copy of the type_spec for the variable with name “a”.

(2) We come across an assignment operator.

(3) The assignment operator asks child[1], a constant, to evaluate itself, and a shared pointer to a copy of the constant, a float in this case, is created.

(4) Then the assignment operator looks up in the map the *type_spec* for the variable to be assigned to and asks the *type_spec* object to create another copy of the constant, but of the needed type. In this case, from a float constant 1.0 an int constant 1 is created and then stored in the map for the variable with name “a”.

(5) Now, we come across a second assignment operator.

(6) The assignment operator asks child[1], an additive operator, to evaluate itself.

(7) The additive operator asks its children to evaluate themselves.

(8) First, an identifier, “a”, is evaluated, by looking up the appropriate entry in the map. Then the constant “5” is evaluated which results in a shared pointer to a copy of the constant being created.

(9) The additive operator then adds the two evaluation results and this creates a shared pointer to a constant created for the sum. This is returned to the assignment operator.

(10) The assignment operator completes its processing similar to what was described in point 4 above.

The objective here is to make sure that correct values for each variable end up in *evalMap* after the AST has been evaluated. Read carefully the provided template to understand the flow of execution. The code as given to you compiles, but might crash when you try to run it. Look for ***** in the code for hints regarding where to make changes to the code.

For considerable bonus marks, extend and redesign the code as necessary to suitably be able to work with pointers, including being able to assign strings. Strings would be assigned to pointers to the char type_spec. Also, consider being able to support the '&' unary operator, which obtains the address of things like variables (you can decide the meaning of "address" for the evaluation code).

For example, consider the following code:

```
int i;
char *cp1;
char *cp2;
int *ip;
i = 1;
cp1 = "Hello World";
cp2 = cp1;
ip = &i;
```

Happy coding!