

CSE 4713/6713 – Programming Languages

Part 4 - TIPS Program Interpreter

Example Code

Examine the files in `Part_4_Example.zip`. Note that each of the node classes now has an `interpret` method that returns an integer:

```
class IntLitNode : public FactorNode {
public:
    int int_literal = 0;

    IntLitNode(int value);
    int interpret();
    ~IntLitNode();
    void printTo(ostream & os);
};
```

The purpose of the `interpret` method is to perform any operations needed by the node in order to compute and return its value. The `interpret` methods for the factors are the simplest. For example, if an `IntLitNode` is asked to interpret itself, the node returns the integer literal that it is storing.

```
int IntLitNode::interpret(){
    return int_literal;
}
```

The `IdNode` in the example code returns a 1 whenever it is called. That is because the Arithmetic Expression Grammar does not specify how to assign values to ids (variables).

The `ExprNode` and `TermNode` are more complicated. For example, a term is made up of 1 or more factors separated by the operations `*` (multiplication) or `/` (division). A `TermNode`'s `interpret` method must compute the value of its first `FactorNode`. Then the method must compute the values of any remaining factors and apply the appropriate math operations.

```
int TermNode::interpret(){
    // get the value of the first Factor
    int returnValue = firstFactor->interpret();

    int length = restFactorOps.size();
    for (int i = 0; i < length; ++i) {
        // get the value of the next Factor
        int nextValue = restFactors[i]->interpret();

        // perform the operation (* or /) that separates the Factors
        if (restFactorOps[i] == TOK_MULT_OP)
            returnValue = returnValue * nextValue;
        else
            returnValue = returnValue / nextValue;
    }

    return returnValue;
}
```

The only change to the driver code is to ask the root of the parse tree to interpret itself after an arithmetic expression is correctly parsed:

```
cout << root->interpret() << endl << endl;
```

Run the example interpreter several times with different inputs to see that interpreting the parse tree implements the expected order of operations.

TIPS Interpreter

Create a new project using your code from Part 3 as the starting point. Modify the makefile to create an executable named `tips.exe`.

Your interpreter is only expected to work with integers. Modify the symbol table of your program to hold the current value of each variable. The following is a suitable data type:

```
map<string, int> symbolTable;
```

Initialize each variable to 0 when it is declared in the VAR portion of a TIPS program.

Add an interpret method to each of your parse tree nodes that implements the functionality expected of that node. The return type of interpret should be `int`. For Boolean results, use 0 as false and any number over than 0 as true.

Modify the driver program to interpret the TIPS program after a successful parse and before the parse tree is deleted.

```
cout << endl << endl << "*** In order traversal of parse tree ***" << endl;
cout << *root << endl;

cout << endl << endl << "*** Interpreting the TIPS program ***" << endl;
root->interpret();

cout << endl << endl << "*** Delete the parse tree ***" << endl;
delete root;
```

NOTE: The behavior of the parser portion of your TIPS interpreter when it finds a syntax error should not change. Your program should print the line number, the error message, and exit.

The deliverable is a zip file of the source code files needed to build and execute your interpreter (`rules.l`, `productions.h`, `driver.cpp`, etc.). Create a zip file named `netid_part_4.zip` and upload that file to the assignment. Do not include any files generated by the makefile in your submission. For example, your submission should not include any `.o` or `.exe` files.

Grading

This is an individual assignment. Do not show / share your code with anyone else. You are responsible for debugging your own code. You are also responsible for testing your program so that the interpreter will function as expected with *any* input.

Good programming style will consist of

- File headers (see Part 1 for an example) for all source code files
- Reasonable variable names
- Consistent indenting to show code structure

Correct output for test files: 80 points

Correct submission of deliverable: 10 points

Good programming style 10 points

Submissions that are late will be penalized 10 points for each day past the due date/time.

Any code that does not compile / run in the Windows Subsystem for Linux will receive a grade of 0.

You will not have access to all of the test files before submitting your program. I recommend that you create both valid and invalid TIPS programs to test your interpreter before submitting your code.