CSE 4714/6714 – Programming Languages
Part 4 – TIPS Program Interpreter

The final step in the sequence of projects is to create the TIPS 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 a `float`:

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

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

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.

```
float 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 one 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.

```
float TermNode::interpret()
{
  // a Term is a Factor followed by 0 or more Factors separated by * or / operations

  // get the value of the first Factor
  float returnValue = firstFactor->interpret();

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

    // perform the operation (* or /) that separates the Factors
    switch(restFactorOps[i]) {
      case TOK_MULT_OP:
        returnValue = returnValue * nextValue;
      break;
      case TOK_DIV_OP:
        returnValue = returnValue / nextValue;
      break;
    }
  }
  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.

The driver now makes all of the printed parts of the interpreter dependent on command-line switches. Check out the actions of the switches `-p`, `-t`, `-s`, and `-d`.

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`.

Symbol Table. Through Part 3, the only role of the symbol table was to store variable names. For this purpose, a `set<string>` was appropriate. However, in Part 4, the value of each variable needs to be stored as well. For this, a `map` is most appropriate:

```
// Access the table that holds the symbols in the interpreted program
typedef map<string, float> symbolTableT;
extern symbolTableT symbolTable;
```

In the example code, study how the `symbolTable` is accessed. It illustrates all the `map` operations that are needed for Part 4. I recommend you start by switching your symbol table to a `map`. For each variable, `symbolTable` will store the variable's name and value.

Only Floats. For TIPS, a question arises: should both `float` and `integer` be supported? For simplicity, I recommend storing everything as a `float`. Note that the `map` data structure only provides for two elements, so the symbol table does not contain a place to store a variable's type. So, at a minimum, supporting both data types would require a more complex symbol table. In addition, substantial additional complexity would be required when parsing expressions (to determine when integers should be implicitly converted to floats).

What Is Truth? For Part 4, this question goes beyond the metaphysical, and must have a discrete answer. For simplicity, and like the C language, the default value of `true` is 1.0, and `false` is 0.0. However, only falsehood is really defined; `false == 0.0`, while `true` is any other numerical value.

However, how does one determine if a floating-point number is equal to 0.0? One **cannot** do this:

```
if(id == 0.0)
  // assume id is false
```

Why? Because any tiny numerical error will make `id != 0.0`. Instead, truth should be defined as near zero, to within an `EPSILON` tolerance. This is what I used:

```
// Define truth for a floating-point number:
// falsehood == F is within EPSILON of 0.0
//     truth == not falsehood
#define EPSILON 0.001
static bool truth(float F) {
  return !((EPSILON > F) && (F > -EPSILON));
}
```

What is Equality? Again, here we need more than a philosophical answer. Because we are comparing floating point numbers, equality cannot be found with the == operator. Instead, equality and non-equality also use the `EPSILON` tolerance:

```
// If there are two simple expressions, compare them according to
// the operation, and return the resulting truth value.
float first = firstSimpleExp->interpret();
float second = secondSimpleExp->interpret();
switch(simpleExpOp)
{
  case TOK_EQUALTO:
    // Equality means the values are within EPSILON of each other.
    if(abs(first - second) <= EPSILON)
      return 1.0; // true
    else
      return 0.0; // false
  break;

  case TOK_NOTEQUALTO:
    // Non-equality means the values differ by at least EPSILON.
    if(abs(first - second) > EPSILON)
      return 1.0; // true
    else
      return 0.0; // false
  break;

  ...
}
```

These are the definitions of truth and equality that I used to successfully implement Part 4's logical testing, as needed by the `<expression>`, `<while>`, and `<if>` statements.

Interpret Return Type. In the example code, the `interpret()` method returns a `float` for all nodes. For Part 4, `interpret()` should also return a `float` for all expression nodes: `<expression>`, `<simple_exp>`, `<term>`, and `<factor>`. In addition, all `<factor>` subclasses (`INTLIT, FLOATLIT, IDENTIFIER, ( <expr> ), NOT <factor>, - <factor>`) will also have an `interpret()` method that returns a `float`.

However, for Part 4, the `interpret()` methods for the non-expression nodes don't have to return anything. So, in my solution, the `interpret()` methods returned a void for the nodes `<program>`, `<block>`, `<statement>`, and all `<statement>` subclasses: `<assignment>`, `<compound>`, `<if>`, `<while>`, `<read>`, and `<write>`.

Interpret Double Dispatch. In Part 3, double dispatch is needed when using the output operator << for printing either of the two abstract base classes `<statement>` and `<factor>`. The same thing is true for the `interpret()` method. For example, here is how I routed `interpret()` from an abstract `StatementNode` to a subclassed `AssignmentStmtNode`:

```
// Use double dispatch to route to proper interpret method
void StatementNode::interpret()
{
  this->interpret();
}
...
void AssignmentStmtNode::interpret()
{
  // Look up the variable that will store expression result
  symbolTableT::iterator variable = symbolTable.find(ident);
  // Put the expression in the variable
  variable->second = expression->interpret();
}
```

In addition, for the abstract base classes, `interpret()` must be a pure virtual method, like `printTo()`:

```
// <statement> -> <assignment> || <compound> || <if> || <while> || <read> || <write>
// Abstract class; base class for statement types
class StatementNode {
public:
  int level = 0;                    // recursion level of this node

  virtual void printTo(ostream &os) = 0; // pure virtual method
  virtual void interpret() = 0;     // pure virtual method, makes the class Abstract
  virtual ~StatementNode();         // labeling the destructor as virtual allows
                                    // the subclass destructors to be called
};
ostream& operator<<(ostream&, StatementNode&); // StatementNode print operator
```

Development Order. For Part 4, I worked top-down. I first implemented an `interpret()` method for `ProgramNode`; this `interpret()` was a stub that just printed out when it was called. Then, I did the same thing with the `interpret()` method for `BlockNode`, and went back and modified `ProgramNode::interpret()` to call `BlockNode::interpret()`:

```
void ProgramNode::interpret()
{
  block->interpret();
}
```

I continued to work top-down in this manner, through the complexity of `CompoundStmtNode` and `StatementNode`. The first `StatementNode` I implemented was `WriteStmtNode`:

```
void WriteStmtNode::interpret()
{
  // Implement the write statement, depending on what is being written
  switch(type) {
    case TOK_STRINGLIT:
      // If we are writing a string, write it. First, strip
      // leading and trailing quote characters.
      cout << content.substr(1, content.length()-2) << endl;
    break;
    case TOK_IDENT:
      // If we are writing a variable, look up and print value
      symbolTableT::iterator variable = symbolTable.find(content);
      cout << variable->second << endl;
    break;
  }
}
```

Just like when developing in C / C++, while debugging it was *very helpful* to be able to print things out from within my TIPS code. I maintained a file, `test.pas`, in my development directory, which I used to unit test every node as it was implemented.

As with any complex code development, it is very important to work carefully and systematically. This means unit testing every bit of code, before moving on to the next thing.

Written Report

Write a report about your experience. The report should cover the following items:

1. Describe, in a few sentences, the development process you used during this assignment. In what order did you develop the `interpret()` methods? How did you test your code?
2. Which `interpret()` methods where the most difficult to write? Why were they difficult? More generally, which aspects of this assignment were the most difficult, and which were relatively easy?

3. Thinking over all the project assignments, Parts 1 through 4, what were the most difficult aspects of the project? What do you feel that you learned?
4. Describe, in a few sentences, any sources you consulted during your development of this assignment.
5. In addition to the requirements specified above, the report should be neat, well structured, grammatically correct, and use words that are spelled correctly.

List your *name*, *netID*, and the *date* at the top of your report. Otherwise, the report may use any format and fonts. The report does not have any specific length requirement. Completeness is valued more than length, and excessive length could result in a lower grade.

Written Report Grading Rubric

Criteria	Ratings				Points
Development Process	**rank 1: Mastery** Engaging sentences. Correct and clear writing.	**rank 2: Good** Adequate sentences. Presentation covers material but has minor writing or structural errors.	**rank 3: Developing** Sentences leave out some minor material or are poorly structured. Recurring writing issues. Details not as in depth as required. Poor formatting.	**rank 4: Lacking** No sentences or significant issues. Incomplete, too short or too long, or repeated, significant writing issues. An obvious slap-dash effort.	25
Writing Methods	**rank 1: Mastery** Engaging sentences. Correct and clear writing.	**rank 2: Good** Adequate sentences. Presentation covers material but has minor writing or structural errors.	**rank 3: Developing** Sentences leave out some minor material or are poorly structured. Recurring writing issues. Details not as in depth as required. Poor formatting.	**rank 4: Lacking** No sentences or significant issues. Incomplete, too short or too long, or repeated, significant writing issues. An obvious slap-dash effort.	25
Project Discussion	**rank 1: Mastery** Engaging answer. Correct and clear writing. Complete answers.	**rank 2: Good** Adequate sentences. Presentation covers material but has minor writing or structural errors.	**rank 3: Developing** Sentences leave out some minor material or are poorly structured. Recurring writing issues. Details not as in depth as required. Poor formatting.	**rank 4: Lacking** No sentences or significant issues. Incomplete, too short or too long, or repeated, significant writing issues. An obvious slap-dash effort.	25
Overall Quality	**rank 1: Mastery** Clearly formatted, contains all required elements, attractively laid out.	**rank 2: Good** Some minor formatting issues or missing elements, but report is still easy to follow.	**rank 3: Developing** Significant issues with formatting, or many missing elements. Report is not so easy to follow.	**rank 4: Lacking** No demonstration of mastery. Significant formatting issues, or many missing elements. An obvious slap-dash effort.	25

Deliverable

The deliverable is a zip file of the source code files needed to build and execute your parser (`rules.l`, `productions.h`, `driver.cpp`, etc.). Do NOT include any of your own test cases (`.pas` files). 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 executable files.

Feel free to assist other students on this lab assignment. If you need additional help, please contact the TA or attend their office hours.

The sample input / output files provided with the assignment are a good starting point for testing your program. You are responsible for testing your program so that the lexical analyzer will function as expected with ***any*** input.