

Programming Project 8

EE312 Fall 2021

General: For our final project we will write our own little toy programming language. The language will have straight-line code consisting of assignment statements and output statements (i.e., `print`). Our language is designed to be very easy to parse (i.e., easy for a computer program to read and understand). It bears no resemblance to any real programming language. Details of the language are shown below.

You will be given relatively little starter code for this project. A large part of the assignment is for you to design your own approach, including data structures, algorithms, structs/classes, functions, etc. Part of your grade will be based on good choices of data structures and algorithms as well. You are prohibited from using most code from the standard C, standard C++, or any other 3rd-party library, but you may use `stack`, `string`, `map`, and `vector`. You can use any code developed in class, and you can naturally write any code you want yourself. You may use the functions C++ `new` and `delete`. Use `cout` << instead of `printf`. Everything else must be built from scratch. You are welcome to use any of the code from class, including previous projects and in-class examples (e.g., including code that I wrote for you). If you have questions about other specific STL structures or functions, ask the instructors.

All the code you produce in this project must be original. You may not seek outside help (other than the assistance of the instructor, TA and authorized tutors). Please keep in mind that we will be using a plagiarism checker with this project.

The Blip Language, `var`, `set`, `output` and `text` commands: We ask that you implement four language statements. Two of these (`var` and `set`) are for creating and assignment values to variables, and two of these (`output` and `text`) are for producing output (i.e., printing to the screen). A Blip program consists of zero or more statements. Each statement begins with a keyword (`var`, `set`, `output` or `text`).

- text statements result in a text message being displayed on the screen. The text message can be either a single word, or a quoted string. The following are examples of legal text statements
 `text Hello`
 `text "Hello World"`
- output statements result in a number being displayed on the screen. The number displayed is the result of evaluating any legal Blip expression. For now, Blip only supports integer variables, and so only integer values can be displayed. The following are examples of output statements. For more information on Blip expressions, see below

```

output 42
output + 1 1
output * x x
output + * x x * y y

```

- var statements initialize a new variable. If a var statement specifies a variable that already exists, then Blip must generate a warning (not an error), with the text, “variable <varName> incorrectly re-initialized”, where <VarName> is the variable’s name. Regardless of whether the warning message is created, the result of a var statement is to ensure that there is a variable in the Blip internal memory with the specified value. The variable must be set to the value of a legal Blip expression. The syntax is: “var <varName> <expr>”. The following are examples of legal var expressions

```

var x 42
var y + x 1

```

- set statements are just like var statements, except a set statement should be used to re-assign a new value to an existing variable. As such, if a set statement specifies a variable that does not already exist, then Blip must generate a warning (not an error) with the text, “variable <varName> not declared”. Regardless of whether the warning message is created, the result of a set statement is to ensure that there is a variable in the Blip internal memory with the specified value. The following are examples of legal set expressions

```

set x 42
set x + x 1

```

Blip programs do not have to have only a single statement on a line, in fact, line breaks don’t really matter to Blip (nor do tabs or spaces). Additionally, Blip programs can have comments between statements (but not in the middle of a statement). For example, the following is correct Blip

```

text “Hello, and welcome to Blip” // first line
var x 0
// set x to (5 + 3) * (x - 1)
set x
    *
        + 5 3
        - x 1

```

A comment is marked by // and continues to the end of the current line. Note that the following statement is illegal in Blip, since we can’t have comments in the middle of a statement

```

output + // two terms in this expression
    * 5 3 // 5 * 3 is the first term
    - 10 7 // 10 - 7 is the second term

```

Variables and the Blip memory: To implement variables, you will need to have a symbol table. A symbol table is like a little database that keeps track of the values of all the variables in your program. You should choose a reasonable data structure for your symbol table – one that will scale to potentially very large number of variables. It is not necessary to explicitly model memory (i.e., addresses and memory locations and things like that). However, you will need to implement some way of knowing that variable x was set to 42. How you implement this is up to you.

Blip Expressions: Blip has a fairly rich set of C-like integer operators, but has a distinctly non-C expression syntax. The reason for the odd syntax is so that Blip is easy to parse. Perhaps the easiest-to-parse languages are those that use prefix expression syntax, so that's what Blip uses. For example, instead of writing $2 + 2$ (infix expressions), we write "+ 2 2" in Blip. The order for the arguments is important for non-commutative operations, so, for example, the Blip expression / 10 5 will evaluate to 2 ($10 / 5$ is 2). The complete list of operators that you must support are:

Binary math operators: +, -, *, /, %

Binary logic operators: &&, ||

Comparison Operators: <, >, ==, !=, <=, >=

Unary Operators: !, ~

Note that for logic operations, we use the C conventions for integer values. Any value other than zero is "true", and zero is false. When evaluating an expression that produces a logical value (e.g., && or <=), you must evaluate true expressions to 1 and false expressions to zero. So...

&& 5 42 evaluates to 1

+ && 6 12 10 evaluates to 11

Note that we are not implementing the bit-wise logic at this time. Please implement ~ as arithmetic negation (i.e., ~ 5 is really -5).

PLEASE NOTE: in Blip there must be at least one space (or tab or newline) between each operator and operand. For example "~5" is an error. "~ 5" is the correct way to write -5 in Blip.

Input.cpp, Parse.h and Blip Syntax assistance: The only starter code you will receive for this project is the Input.cpp file and the corresponding Parse.h. You are free to modify these files in any way, and are ultimately responsible for any errors that might be present in these files. The purpose of Input.cpp is to help you read a Blip program from a file. Input.cpp contains functions that open and read characters from a file and bundle those characters together to create *tokens*. A Token is the atomic input unit for a programming language. In Blip the following tokens are defined

- NUMBER any number, note Blip numbers are always positive integers
- SYMBOL any of the standard Blip operators (e.g., +, *, ~) or the // comment marker

- NAME anything else, for example the keyword “set” in Blip is captured by Input.cpp as a NAME token.
- END is actually not a token, but represents the end of the input file. When you see the END token, you have read the entire input.

You read tokens from the input file by using the `read_next_token()` function. Note that `read_next_token()` does not return anything. Instead, the `read_next_token` sets global variables that describe the input token. These global variables are:

- `next_token_type` – an enumerated value that will be set to one of the symbolic constants, `NUMBER`, `SYMBOL`, `NAME` or `END` depending on the type of token last read
- `token_number_value` – an integer that represents the numeric value if (and only if) the token is type `NUMBER`. For example, if the next token on input is the characters “42”, then `read_next_token()` will set `next_token_type` to `NUMBER` and will set `token_number_value` to 42

The actual text of a token is always available to you if you call the function `next_token()`. This function returns a `const char*` pointer to a null-terminated string that contains the actual text for the token. You will need to use this text when the token type is `NAME`.

Input.cpp has a function to assist you with comments. When you are reading a Blip program, spaces, tabs and new lines are typically ignored (`read_next_token` takes care of this for you). However, if you encounter a `//` as the first token in a new Blip statement (recall that comments can only occur between Blip statements, not in the middle of a statements), then you need to ignore everything until the end of the current line. The function `skip_line()` will ignore all the remaining characters on whatever the current line is in the input file.

Finally, Input.cpp has a function to set/change the file that characters are being read from. During testing, you will want to open several small Blip programs and read their input. By invoking the function `set_input_file(<filename>)` you can instruct Input.cpp to start reading from that file.

RUNNING BLIP

To actually run blip, please write your own ‘`run()`’ function that will read and execute the blip program. Then include `run()` in your main. For example, if we want our main to run `test1.blip`, and `test2.blip`, then our `main.cpp` looks a little like this:

```
cout << "Test 1" << endl;
set_input("test1.blip");
run();

cout << endl << endl << "Test 2" << endl;
set_input("test2.blip");
run();
```

IMPORTANT SUBMISSION FORMATTING:

Before submitting, please change your main to the following:

```
int main(void) {  
    set_input("test_grader.blip");  
    run();  
}
```

`set_input` will use the 'test_grader.blip' file (you can rename test1.blip as test_grader.blip to check your main submission is correct), and `run()` will read and execute your blip program.

Design and Implementation Advice

Data Structures I recommend that you build your project using at least two major data structures. The first data structure will be the symbol table. The symbol table will be the “database” that stores all of the Blip variables and remembers what value those variables have been set to. While it is possible to implement the symbol table using an array (or two), that is really not what we expect from EE312 programming experts! Binary search trees or Maps make excellent data structures for implementing symbol tables.

The second major data structure is somewhat optional for this project. A parse tree is a data structure that represents within the computer’s memory the content and structure of a computer program. In other words, the parse tree is the internal representation of a Blip program that you have read. In a real compiler or interpreter, the input programs are read (parsed) in one pass. Then after the entire program has been read, the program will be executed (or translated to machine code). By separating parsing and executing into two distinct phases, we get two important advantages. First, our code is just more modular, and that’s a good thing. Second, if you have a loop or a function, then you can parse the code once (and store the internal representation of that code in your Parse Tree), and then execute that code multiple times without re-parsing it.

Parse trees are linked data structures (trees, naturally). In my implementation I plan on having the Expression Tree. Note that Blip expressions (like C expressions) are naturally recursive things. For example, an add expression “+ A B” is defined as an operator (+) followed by two expressions – A and B can be any expression, even add expressions. So, for example “+ + 1 3 + 2 4” is a legal Blip expression which means $(1 + 3) + (2 + 4)$. If you build a parse tree to represent this expression, your tree will have at least three nodes (mine has seven nodes). You’ll have one node for each + operator in the expression. At the root of the tree, you have the last + that gets done – this happens to be the + that corresponds to the first + in the original Blip text. The root of the tree has two children, each child is itself a + expression. The left child is “+ 1 3” and the right child is “+ 2 4”. In my parse tree, I choose to represent the numbers as nodes (leaves) in the tree. So I end up with three + nodes and four NUMBER nodes in my parse tree.

Please note that whether you choose to explicitly build a parse tree or not, you will almost certainly have to write your parsing and executing function(s) using recursion.

Fortunately, the recursion required to do this is super easy (whether you build a parse tree or not, the recursion is very natural).

Use of Structs, memory leaks and other finer details.

We strongly encourage you to use structs and to create your own abstract data types as you see fit. Some obvious types you might consider creating are

- struct Expression – something that represents an expression, for me, an Expression struct is a node in my Expression Tree parse tree
- struct Variable – something that represents the name and current value of a variable

We won't check for memory leaks at all. What this means in practice is, you can leave your symbol table and parse trees on the heap and not deallocate them – you can leak the memory in your major data structures, since the amount of memory required to implement those data structures is dependent on the number of statements in the Blip input program file.

What to turn in

Zip all your .h and .cpp files (please make sure you include Input.cpp and Parse.h if you edit them) and submit the zip file. Your .h and .cpp files should be at the root directory of the zip file. Zip on the ECE-LRC servers and not your local computers to make sure that the zip file is formatted correctly.

We are not providing you with CLion startup files, and we are not telling you what .cpp files you need to create in order to build your final project. Those aspects of your design are left up to you. We are not providing you with a main function. When we grade your program, we will compile all the .cpp files that you commit to Canvas and assemble those files into a single executable program. You, therefore, must provide a single main function. You are **STRONGLY** discouraged from adding any code to Input.cpp. Instead, create your own .cpp files (and .h files) for your solution. If you fail to include the files, then we can't get them, your project will fail to compile and you'll fail all of the tests (and you should expect a grade of zero in that case).

FAQ

Q1. What is Polish notation?

A: A Polish Notation expression has one of two forms when parsed from Left to Right.

1. Value
2. Operator (Expression) (Expression)

where Expression is of the type of one of the two above.

So to evaluate the expression, you can use recursion:

The base case is #1 above. There is no operator here.

The recursive case is #2 above, where the first character is an operator. Keep evaluating each of the two expressions recursively, and combine the two return values with the operator.

The special case of unary operators is a simple extension where there is only one expression instead of two in the recursive case.

Instead of recursion, you can use a Stack on the heap. You have two stacks, in fact -- one to hold operators, and one to hold values. You parse the given expression from Right to Left. When you see an operator, pop two values from the value stack, use the operator to combine them, and push the result on to the value stack. When you see a value, push it on the value stack.

Search the web for more details if you like.

Q2: What should the main file look like?

A: Please make your main file look like the following i.e. create a run function! This will make grading possible. Then for testing, you just need to make a temporary copy of the target blip file.

```
int main(void) {  
    set_input("test_grader.blip");  
    run();  
}
```

Q3: Are we allowed to use stack, map and vector from the stdlib?

A: Yes.

Q4: Do we need to worry about dividing by zero?

A: No.

Q5: Can we assume arithmetic expressions are in the correct format?

A: Yes.

Q6: Should we output a new line after the warning message?

A: Yes.

Q7: What files do I need to submit?

A: All of your .cpp and .h files must be submitted for this project. Do not submit blip files.

Q8: `var x 10` and then `var x 5`: what is x?

A: Update and override, so x is now 5.

Q9: `next_token` vs. `next_token_peek`: what is the difference?

A: `next_token_peek` does not consume the token.

Q10: What if the program tries to access an uninitialized variable?

A: We will not test that.

Q11: How do we handle escape characters (ex. `\n`)?

A: `cout` does that for you already. Try `cout << "test\n";`

This will output a newline after the word `test`.

Q12: What are we allowed to use from standard libraries?

A: You can use already these existing data structures: stacks, maps, and vectors. Do not use anything else that you have not written, that was not done by us in class or supplied by the instructors.

CHECKLIST – Did you remember to:

- ☐ Re-read the requirements after you finished your program to ensure that you meet all of them?
- ☐ Make sure that your main.cpp is in the required form?
- ☐ Make sure that your program passes all the testcases?
- ☐ Make up your own testcases?
- ☐ Upload your solution (all file) to Canvas in .zip or gzip format?
- ☐ Download your uploaded solution into a fresh directory and re-run all testcases?