# CISS445: Programming Languages
# Assignment 1

OBJECTIVES:
1. Write a simple pure interpreter
2. Understand the linebyline interpretation carried out by an interpreter
3. Understand the importance of string parsing

The `p` Interpreter Project

The following is a bruteforce implementation of an interpreter. Yes. No tools. Bare hands. You develop everything using C++. The idea is that you must write a C++ program that will read from the keyboard or from a file and execute it. You will begin to appreciate the compiler/language tools better and you will understand the theory of automata and languages in a much deeper way. Again you must use only the core C/C++ language (that includes STL) and not tools outside the language. Furthermore you are not allowed to refer to any algorithms from compiler or automata/language theory books. I do encourage discussion. However coding must be done alone. For instance it is fine to have a discussion on the organization of your program or algorithms you come up with. Enough rules. On with the fun.

SUBMISSION.
- I suggest you create a directory `ciss445` somewhere in linux environment to hold your assignments.
- In directory `ciss445`, create directory `a` and in a created directory `a01`.
- In directory `ciss445/a/a01`, create directory `a01q01`.
- Put all your files (`*.cpp, *.h`) in `ciss445/a/a01/a01q01`.
- In directory `ciss445/a/`, execute the following command: `tar cvf a01.tar a01`
- In directory `ciss445/a/`, execute the following command: `gzip a01.tar`
- You now have `a01.tar.gz` in your `ciss445/a/` directory.
- (I only care that in `a01.tar.gz`, you have directory `a01q01` containing your work for Q1 of this assignment.)
- Email `a01.tar.gz` to yliow.submit@gmail.com. You must use the following subject line:

  `ciss445 a01`

- You must email using your college email account.

Here's a simple makefile that you can put into `ciss445/a/a01/a01q01/`:

```
# Makefile for main
# Y. Liow
#-----------------------------------------------------------------------
# Macros
#-----------------------------------------------------------------------
CXX       = g++
CXXFLAGS  = g Wall
LINK      = g++
LINKFLAGS =
OBJS      = main.o
EXE       = main.exe

#-----------------------------------------------------------------------
# Executable
#-----------------------------------------------------------------------
$(EXE): $(OBJS)
        $(LINK) $(LINKFLAGS) $(OBJS) o $(EXE)

#-----------------------------------------------------------------------
# Object Files
#-----------------------------------------------------------------------
main.o: main.cpp
        $(CXX) $(CXXFLAGS) main.cpp c o main.o

#-----------------------------------------------------------------------
# Utilities
#-----------------------------------------------------------------------
clean:
        rm $(OBJS) $(EXE)

c:
        rm $(OBJS) $(EXE)

run:
        ./$(EXE)
r:
        ./$(EXE)
```

Read my make tutorial if you want to know more. The above assumes you have one cpp file named `main.cpp` and the executable created is `main.exe`. Modify the makefile is you have more cpp files. (The spacing above before a command is one tab, not spaces)

Using the makefile, build the executable by doing

```
make
```

To run the executable you do this:

```
make r
```

or

```
make run
```

To clean up (i.e. remove object file and executable), you do this:

```
make c
```

or

```
make clean
```

Q1.

The name of the program is p. Here we go...

Name the main cpp file `main.cpp`.

**Step 1.** p simply echoes the input. If you run p without an argument, p will read from the keyboard and display what you type. The prompt is `">>> "`. If you enter ctrl-d and press the enter key, the program terminates.

**Step 2.** Here comes the first program statement. If a line of code is an integer, p simply echoes it. Trailing spaces (left or right) are allowed. For instance here's an execution

```
>>> 3
3
>>>      5
5
>>> 23456
23456
>>> +123
123
>>> 0
0
>>> 042
ERROR: I don't understand octals yet
>>>
```

(p does not understand octals). Anything else will result in an error:

```
>>> x
ERROR: "x" is neither a value nor found in the symtable
>>>
```

What is the format of an integer? (Just use words.) Such a description will help you write the code needed to verify if the input is an integer or not.

`symtable` (symbol table) is just a structure for variable names and their values. See later examples. You will see a more complex structure for a symbol table in, for instance, a compilers course. In the context of CISS445, working on this project will give you a better understanding of the concept of an "environment" that is frequently used in writing interpreters and especially important in functional languages.

(We are now entering into the theory of automata and languages ... But you don't need any deep theory for our case since it's so simple... )

**Step 3.** p has grown up! It (he? she?) understands variables! (We will use the same rules as C/C++ for declaring identifiers, i.e. alphanumerics and underscore where the first character is nonnumeric.) If the statement is just a declared variable, p prints the value of that variable if p is running in the interactive mode; otherwise nothing is printed. For the time being all variables are integers.

```
>>> x
ERROR: "x" is neither a value nor found in the symtable
>>> x = 1
>>> x
1
>>> x = 3
>>> x
3
>>>
>>>
>>> y = z
ERROR: "z" is neither a value nor found in the symtable
>>> 1 = x
ERROR: "1" is not a variable name
>>>
>>>
>>> number_of_arms = 3
>>> number_of_arms
3
>>> number of arms = 3
ERROR: "number of arms" is not a variable name
>>>
```

Right now, the righthand side of `=` is an integer value.

**Step 4.** p has acquired mathematical skills! p can now do simple arithmetic of the form `[value1] [op] [value2]` where `[op]` is `+`.

```
>>> 1 + 2
3
>>> 1+2
3
>>> 1         + 2
3
```

**Step 5.** Now p is smart enough to do addition on the right of = !!!! WOW !!!!

```
>>> x = 1 + 2
>>> x
3
```

**Step 6.** Now p can even do addition with variables! Unbelievable!

```
>>> x = 1
>>> y = 2
>>> z = x + y
>>> z
3
```

Are we having fun yet?

The following are optional and not graded.

**Step 7.** p can even do multiplication:

```
>>> x = 2
>>> y = 3
>>> z = x * y
>>> z
6
```

**Step 8.** Now here's the "interesting" one: p can do expressions and understand precedence:

```
>>> x = 2
>>> y = 3
>>> z = x + y * 2 + 1
>>> z
9
```

GRADING SCHEME For this assignment, I will *not* look at the way your code works. This is a higher-level class. You should understand the value of writing clean and well documented code. If you don't, the complexity and messiness of your spaghetti will either make the project unmanageable or you will waste a lot of time scratching your head over what you have written.

Some test cases are included below. Your error messages must match mine exactly. Each test case is a new execution of the your program. Furthermore, I might make minor modifications to some values in the actual test case. For instance, instead of executing `1 + 2`, I might use `1 + 3`. This is to prevent students from writing programs that do nothing more than match inputs to outputs.

For operators, you only need to implements unary + and -, binary `+,-,*,/,%`. Your interpreter should understand precedent of operators. (Of course you should use the standard precedence rules in, for instance, C/C++ and Python.) Don't forget that you must also implement the assignment operator.

This assignment is worth 10 points. Each test case that I will use is worth the same number of points. Therefore if I use 5 test cases, then each test case is worth 2 points. If I use 10 test cases, then each test case is worth 1 point. If I use 20 test cases, then each is worth 0.5 points. Etc.

**Test 1. Echoing integer values**

```
>>> 1
1
>>> 123
123
>>> 0002
ERROR: I don't understand octals yet
```

**Test 2. Unary operators**

```
>>> +1
1
>>> ++1
1
>>> +++1
1
>>> -1
1
>>> --1
1
>>> ---1
1
>>> +-1
1
>>> -+1
1
>>> +-+1
1
>>> -+-1
1
>>> +--1
1
>>> --+1
1
```

**Test 3. Addition of values**

```
>>> 1 + 2
3
>>> 3 + 7
10
```

**Test 4. Addition and unary operators**

```
>>> +3 + +7
10
>>> +3 + -7
4
>>> 3+-7
4
>>>
```

**Test 5. Variables and declaration**

```
>>> x
ERROR: "x" is neither a value nor found in the symtable
>>> x = 5
>>> x
5
>>>
```

**Test 6. Changing the value of a variable**

```
>>> x = 5
>>> x
5
>>> x = 7
>>> x
7
>>>
```

**Test 7. Addition involving variables**

```
>>> a = 5
>>> b = 1 + a
>>> b
6
>>> b = a + 5
>>> b
10
>>> c = a + b
>>> c
15
>>> c = b + a
>>> c
15
>>> a + b
15
>>> b + a
15
```

**Test 8. Addition with missing operand**

```
>>> 1 +
ERROR: right operand for + is missing
>>> 1+
ERROR: right operand for + is missing
>>>         1+
ERROR: right operand for + is missing
>>> 1            +
ERROR: right operand for + is missing
>>> x = 1
>>> x +
ERROR: right operand for + is missing
```

**Test 9. Undeclared variable in expression**

```
>>> 1+x
ERROR: "x" is neither a value nor found in the symtable
>>> 1+    x
ERROR: "x" is neither a value nor found in the symtable
>>> 1    +      x
ERROR: "x" is neither a value nor found in the symtable
>>> x + 1
ERROR: "x" is neither a value nor found in the symtable
>>> x = y
ERROR: "y" is neither a value nor found in the symtable
```

**Test 10. Assignment to non variables**

```
>>> 1 = 5
ERROR: "1" is not a variable name
>>> 3abc = 5
ERROR: "3abc" is not a variable name
>>> hello world = 5
ERROR: "hello world" is not a variable name
>>> hello_world + forty_two = 5
ERROR: "hello_world + forty_two" is not a variable name
```

### STREAM I/O

The following is an example of how you should perform input. Run the program:

```
#include <iostream>
#include <limits>

const int MAX_BUF = 1024;

int main()
{
    while (1)
    {
        char s[MAX_BUF];
        std::cin.getline(s, MAX_BUF);
        if (std::cin.eof()) break;
        if (std::cin.fail() || std::cin.bad())
        {
            std::cin.clear();
            std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
        }
        std::cout << '[' << s << "]\n";
    }

    return 0;
}
```

Note in particular that your program must terminate when an EOF is reached:

```
if (std::cin.eof()) break;
```

Make sure you try sending Ctrl-D as input to the program.

For output, you must output to `std::cout`.

To make your program robust and platform independent, you have to know that the string read from a stream might have non-visible characters at the end of the string. For instance: on some systems, when a user presses the enter key, the character `'\r'` is placed in the stream. This means that a string read from the stream (up to `'\n'`) might end with `'\r'`. You are strongly advised to remove all such right-trailing characters from your strings which are read from a stream.

### THE `std::string` CLASS

I don't really care if you have C-style strings or C++ string class. You pick. There are lots of C++ resources online if you need to read up on the C++ string class (or other classes.) Here are some:

- http://en.cppreference.com/w/
- http://www.cplusplus.com/