CS 341 : Programming Languages
Spring 2022

# Project 05 : Executing simple C (option #2)

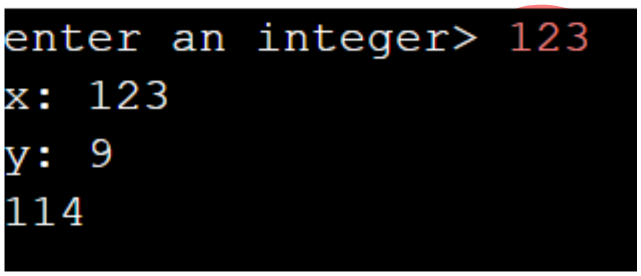
(doc v1.0)

**Assignment:** F# program to execute simple C programs
**Evaluation:** Gradescope followed by manual execution & review
**Policy:** Individual work only
**Complete By:** Friday April 29th @ 11:59pm CDT for full credit

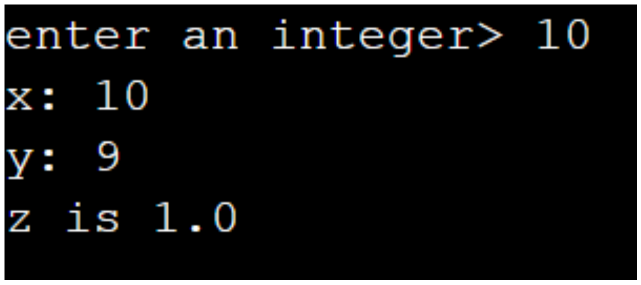
Early submission: no early bonus available
Late submissions: 10% penalty if submitted up to 24 hrs late (by Saturday 4/30 @ 11:59pm)
No submissions accepted after Saturday 4/30

## Background

This is part 3 of a multi-part programming project in F#. Here in part 3 we're going to execute **simple C** programs that have passed both syntax and semantic checks. The syntax will remain unchanged from project #04, which introduced the notion of real variables and literals. A sample simple C program is shown on the right, with a couple different execution results shown below (the output depends on the user's input):



```
enter an integer> 123
x: 123
y: 9
114
```



```
enter an integer> 10
x: 10
y: 9
z is 1.0
```

```
void main()
{
  cout << "enter an integer> ";
  int x;
  cin >> x;

  int  y;
  real z;
  y = 3^2;   // 3 squared:
  z = x-y;

  cout << "x: ";
  cout << x;
  cout << endl;

  cout << "y: ";
  cout << y;
  cout << endl;

  if (z == 1.0)
    cout << "z is 1.0";
  else
    cout << z;

  cout << endl;
}
```

y= y+1

## BNF definition of simple C

For reference purposes, here is the complete BNF for simple C. Nothing has changed since project 04, which added real variables and literals.

```
<simpleC>   -> void main ( ) { <stmts> } $

<stmts>     -> <stmt> <morestmts>
<morestmts> -> <stmt> <morestmts>
             | EMPTY

<stmt> -> <empty>
        | <vardecl>
        | <input>
        | <output>
        | <assignment>
        | <ifstmt>

<empty>        -> ;
<vardecl>      -> int identifier ;
                | real identifier ;
<input>        -> cin >> identifier ;
<output>       -> cout << <output-value> ;
<output-value> -> <expr-value>
                | endl
<assignment>   -> identifier = <expr> ;
<ifstmt>       -> if ( <condition> ) <then-part> <else-part>
<condition>    -> <expr>
<then-part>    -> <stmt>
<else-part>    -> else <stmt>
                | EMPTY

<expr>       -> <expr-value> <expr-op> <expr-value>
              | <expr-value>

<expr-value> -> identifier
              | int_literal
              | real_literal
              | str_literal
              | true
              | false

<expr-op>    -> +
              | -
              | *
              | /
              | ^
              | <
              | <=
              | >
              | >=
              | ==
              | !=
```

## Getting Started

We are providing all necessary files to build the compiler, including our solution from project #04. There are a total of six F# files involved in this project. The file “runner.fs” is the new component here in project 05:

1. analyzer.fs
2. checker.fs
3. lexer.fs
4. parser.fs
5. **runner.fs**
6. main.fs

The files are available on replit.com in the project “**Project05-02 – executing simple C in F#**”. If you prefer to work outside of replit.com, you can download the files from dropbox.

Feel free to delete any of the provided files and substitute your own. However, don’t change the API for how the various modules interact, as we will be testing only your “runner.fs” file with our compiler implementation.

## Assignment details

Modify the execution engine in “runner.fs” to execute the simple C program. A template file is provided for you:

```
//
// Execution engine for simple C programs.  This component
// executes a simple C program that has passed all syntax
// and semantic checks. Returns a string denoting successful
// completion/error, along with a list of tuples (name,value)
// representing the final state of memory.
//
// << YOUR NAME? >>
//
// Original author:
//   Prof. Joe Hummel
//   U. of Illinois, Chicago
//   CS 341, Spring 2022
//

namespace compiler

module runner =
  //
  // NOTE: all functions in the module must be indented.
  //
```

```
let simpleC tokens symboltable =
  []


//
// execute tokens symboltable
//
// Given a list of tokens and a symbol table, executes
// the given simple C program. Returns a tuple
// (result, memory), where result is a string denoting
// "done" if the execution was successful, otherwise
// a string of the form "execution_error:..." is returned.
//
// On success, the final state of memory is returned as a
// list of tuples of the form (name, value), e.g.
// [("x","123"); ("y","3.14159")]; the order of the names
// in memory should be in the same order as they appear
// in the symbol table. On an error, the returned list
// is empty [].
//
let execute tokens symboltable =
  try
    let memory = simpleC tokens symboltable
    ("done", memory)
  with
    | ex -> ("execution_error: " + ex.Message, [])
```

You are required to represent the state of memory using a list of tuples, in the same order as the symbol table. For example, if the symbol table is [("z", "real"); ("y", "int"); ("x", "int")], then the initial state of memory should be initialized to [("z", "?"); ("y", "?"); ("x", "?")]. As the program runs, you'll use this list to keep track of each variable's value, and then return this list upon successful completion of the execution run.

In F#, use **float** to represent real variables in simple C. When you need to convert values, use the F# functions int, float and string --- i.e. **(int value)**, **(float value)**, and **(string value)**. Finally, note that F# has an operator ** to denote raising an exponent to a power; use this for executing simple C's ^ operator.

To input from the keyboard, use **System.Console.ReadLine( )**, which returns a string you'll immediately convert to int or float; don't worry about error checking, much like C++ just throw an exception if the user does not input a value of the correct type. For output, use **printf** and **printfn**. If the simple C program outputs endl, execute **printfn ""** from F#. If the simple C program is outputting an integer, real, or boolean value, execute using **printf "%A" value** from F#. The only exception are string literals, which should be output using **printf "%s" string_literal** to avoid the output of the double-quote characters "…".

For consistency, integer and real literals should be converted to their target type --- int or float --- before being operated upon. For example, if the simple C code is

```
cout << 3.5000;
```

Then the execution of this via F# will output **3.5** given the conversion to float followed by its output. Similarly, if the variable z is of type real, and its current value in memory is “1”, the simple C code

```
cout << z;
```

will output **1.0** if you properly convert to float before the output. [ *Why would the value in memory be “1” and not “1.0”? When you convert the value to a string for storage in the memory, the string function will drop the “.0” and return “1”. As a result, real values may appear as integers in the memory list. The only way we know they are real numbers is the type information in the symbol table.* ]

$y = 1$

$y = x+5$

## Requirements

No imperative programming. In particular this means no mutable variables, no loops, and no data structures other than F# lists. No file I/O other than what is performed by the lexer; this implies no calling of system functions like “g++” to compile the code and then another system call to execute it.

Also, no global/module level variables/values other than what is provided in the lexer. As in project 04, the tokens and symbol table should be passed around from function to function. Likewise, you’ll need to pass around the memory list to support execution and build your final representation of memory --- do not try to store the memory list into some sort of global or module-level variable to avoid parameter-passing. One of the goals of this assignment (and this class) is to learn functional programming, and parameter-passing is an important aspect of functional programming. Taking shortcuts like global / module-level variables will lead to a project score of 0.

## Electronic Submission and Grading

Grading will be based on the correctness of your compiler. We are not concerned with efficiency at this point, only correctness. Note that we will test your compiler against a suite of simple C input files, some that compiler and some with syntax and semantic errors. Make sure you name appears in the header comments of any file you modify. You also need to comment the body of any files you modify, as appropriate.

When you are ready to submit your program for grading, login to Gradescope and upload your program files. We will grade “runner.fs”, and ignore all other submitted files.

You have unlimited submissions, and Gradescope keeps a complete history of all submissions you have made. By default, Gradescope records the score of your last submission, but if that score is lower, you can click on “Submission history”, select an earlier score, and click “Activate” to select it. The activated submission will be the score that gets recorded, and the submission we grade. If you submit on-time and late, we’ll grade the last submission (the late one) unless you activate an earlier submission.

The grade reported by Gradescope will be a tentative one. After the due date, submissions will be manually reviewed to ensure project requirements have been met. Failure to meet a requirement --- e.g. use of mutable variables or loops --- will generally fail the submission with a project score of 0.

## Policy

Late work *is* accepted. You may submit as late as 24 hours after the deadline for a penalty of 10%. Given it's the end of the semester, submissions are not accepted after 24 hours.

Unless stated otherwise, all work submitted for grading *must* be done individually. While we encourage you to talk to your peers and learn from them, this interaction must be superficial with regards to all work submitted for grading. This means you *cannot* work in teams, you cannot work side-by-side, you cannot submit someone else's work (partial or complete) as your own. The University's policy is available here:

https://dos.uic.edu/conductforstudents.shtml .

In particular, note that you are guilty of academic dishonesty if you extend or receive any kind of unauthorized assistance. Absolutely no transfer of program code between students is permitted (paper or electronic), and you may not solicit code from family, friends, or online forums. Other examples of academic dishonesty include emailing your program to another student, copying-pasting code from the internet, working in a group on a homework assignment, and allowing a tutor, TA, or another individual to write an answer for you. Academic dishonesty is unacceptable, and penalties range from a letter grade drop to expulsion from the university; cases are handled via the official student conduct process described at https://dos.uic.edu/conductforstudents.shtml .

If ( true )

If ( false . )

[illegible] bool