CIT 595 Spring 2018
Homework #1

Over the course of the next two assignments, you will implement a simple compiler that converts code from C to LC-4 assembly language. Users of the compiler will provide a text file containing the C code, and your program will read the file and then generate LC-4 instructions.

This may sound like a fairly daunting task. And, well, it kind of is. But it's not so bad if you do it step-by-step, tackling the different parts of the program before finally putting it all together at the end. Which is why you'll start with a simple part of it in this assignment and then finish it in the next one.

This assignment will focus on these aspects of the compiler:

- Identifying the name of a function in its definition
- Identifying function parameter declarations
- Identifying local variable declarations
- Checking that the code conforms to legal C syntax

Although you are free to discuss the intent and instructions of this assignment with your classmates, the work you submit must be your own: *you should not be sharing or discussing code with anyone else, nor should you be sharing answers to the writeup questions*. Likewise, *you may not submit code you found online, in textbooks, etc.* See "Academic Honesty and Collaboration" below for more explanation. If you are uncertain of the policies, please see the course syllabus in Canvas or ask a member of the instruction staff.

Part 1 is due in Canvas on **Friday, January 26, at 5:00pm.** Parts 2 and 3 are due in Canvas on **Friday, February 2, at 5:00pm.** Don't wait until the first due date to start Parts 2 and 3! These parts of the assignment are considerably more complicated. Submission instructions are described below.

Getting Started

You can download all of the starter code and header files for this assignment in Canvas. Go to "Files", then "Homework Files", then "Homework #1."

For this assignment, we are providing a Makefile that you can use for compiling your code. You may modify it if you need to, but you must include a Makefile with your submission and your Makefile must properly compile your code on eniac. In subsequent assignments, you will be expected to write the Makefile yourself.

Before you begin, make sure you are able to compile the code we have provided using our Makefile. In the directory containing the Makefile and the two .c files, run the command:

```
make find_symbols
```

This will attempt to compile the find_symbols.c file that we have provided. Note that its output is find_symbols.o and not an executable, since find_symbols.c does not contain a "main" function.

The "main" function for the program is in find_symbols_test.c, which you can compile using the command
`make find_symbols_test`
This will compile find_symbols.c if it is not already compiled, and then compile find_symbols_test.c. If everything is successful, this will create an executable called test_find_symbols, which you can run in order to test your implementation. For simplicity, you can also compile find_symbols_test.c using the command:
`make`

If you are not familiar with Makefiles, there is a good tutorial at https://www.tutorialspoint.com/makefile/index.htm (many of the examples are in C++ but the concepts are the same for C). You'll be working a lot with Makefiles in subsequent assignments, so now is a good time to learn about them!

You are free to develop on any standard Linux platform, including recent versions of Mac OS, your VMware instance from CIT 593, or the eniac computing cluster; please do not use Windows environments such as Visual Studio. If you have trouble compiling the code we distributed, please notify a member of the instruction staff immediately.

Last, note that the TAs will also use Valgrind on eniac to check your code for memory leaks for **all parts** of the assignment. Be sure to free/deallocate any heap memory as necessary. See the section below on Grading for more information about what you need to do.

Part 1: Basic parsing (25 points)

In this part of the assignment, you will write two functions that read a line of code and find the names of variables that are being declared.

Step 1. Parsing a function header

The *parse_function_header* function in find_symbols.c is intended to identify the function name and parameters in the first line of the function (the "function header").

The input to the function is the string that contains the first line of a function definition in the C code that is being compiled. It should set the *function_name* global variable to the name of the function being defined and populate the appropriate entries in the global *parameter_names* array; it should then return 1.

For instance, if the input string were "`int fun ( int dog , int cat ) {` " then:

- *function_name* should be set to "fun"
- *parameter_names*[0] should be set to "dog"

- *parameter_names*[1] should be set to “cat”
- *parameter_names*[2] through *parameter_names*[9] should be NULL
- the function should return 1

Note that you are expected to make **copies** of the strings from the input, and not aliases. If you don’t know the difference, ask a member of the instruction staff!

For this part of the assignment, **you can assume that the line contains legal syntax for a function header**, and that:

1. The return type of the function, i.e. the first token in the string, is always “int”
2. There is always a single space between the function name and the list of parameters
3. There is always a single space between the parentheses and other tokens, and between any commas and other tokens
4. All parameters are of type int
5. There is always a single space between the list of parameters and the open curly brace at the end
6. The line always ends with a curly brace
7. There are never more than 10 parameters
8. The function and parameter names are legal C identifiers

In this part of the assignment, you do not need consider any inputs that violate any of these assumptions; you will handle (some of) those cases in Part 2.

Thus, your function should correctly identify the function name and parameters for each of the following:

```
int fun0 ( ) {
int fun1 ( int a ) {
int fun2 ( int dog , int cat ) {
int fun3 ( int larry , int moe , int curly ) {
```

For this part of the assignment, we strongly suggest using the *strtok* and *strcmp* functions that are defined in string.h:

- https://www.tutorialspoint.com/c_standard_library/c_function_strtok.htm
- https://www.tutorialspoint.com/c_standard_library/c_function_strcmp.htm

Note that the *strtok* function attempts to modify the string that is passed to it, which will cause a segfault if the string is an immutable, interned string in the program (which it will be in our test program), so you need to make a copy of it before attempting to pass it to *strtok*.

For this and subsequent parts of the assignment, you must **not** change the signature (return type, name, and parameter types) of this function; if you do, the TAs will not be

able to grade it and you will receive a 0 for this part of the assignment. Please speak to the instructor if you feel that it is necessary to modify this function's signature.

To check your implementation, modify find_symbols_test.c to write tests in which you call the *test_parse_function_header* function as described in the comments in the code. An example is provided but you should try other inputs as well.

Step 2: Finding variable declarations

The *parse_line* function in find_symbols.c is intended to identify any variable declarations in a line of code.

The input to the function is the string that contains the line of code. The function should populate the appropriate entries in the global *variable_names* array and then return 1.

You can assume that all variables are ints, so that if the first token on a line is not the string "int", then there must not be any declarations on that line.

If the first token is "int", though, then you know there is at least one declaration there. However, there is a variety of ways in which you can declare/initialize a single variable or multiple variables on one line. So your function should be able to find the variables being initialized for each of the following, for example:

```
int x ;
int j , k ;
int y = 2 ;
int a = x + 5 ;
int b = a + y + 8 ;
int c = a + x , d , e = b ;
```

For this part of the assignment, **you can assume that the line contains legal syntax for declaring variables**, and that:

1. only the simple assignment operator ("=") is used
2. only the addition operator ("+") is used
3. there is only one statement per line (i.e., nothing like "int x = 4 ; a = b ;")
4. there are no assignment chains, e.g. "int x = y = 3;"
5. there are no function calls
6. there is a single space between each token, including the assignment operator, addition operators, commas, and the ending semicolon
7. there are never more than 10 variables being declared on a line

As in Step 1, you do not need to consider any inputs that violate any of these assumptions; you will handle (some of) those cases in Part 2.

Hint! In thinking about how to approach this part of the assignment, consider each token,

what could follow it, and what the following token tells you about the token you're considering and subsequent ones. For instance, you know that the first token will always be "int" and that the next one is a variable that's being declared. What can follow that is either a semicolon, the assignment operator, or a comma. If what follows is the assignment operator, you'll always have an identifier after it, and what can follow *that* is either the addition operator, a comma, or a semicolon. If you think about all the possible "states" or situations, and what to expect each time you see a particular token, you'll more easily be able to implement this and will set yourself up well for later parts of the assignment.

As in Step 1, you must **not** change the signature (return type, name, and parameter types) of this function; if you do, the TAs will not be able to grade it and you will receive a 0 for this part of the assignment. Please speak to the instructor if you feel that it is necessary to modify this function's signature.

To check your implementation, modify find_symbols_test.c to write tests in which you call the *test_parse_line* function as described in the comments in the code. An example is provided but you should try other inputs as well.

Don't forget, this part of the assignment is due **Friday, January 26**.

Part 2: Checking for syntactic correctness (60 points)

In this part of the assignment, you will modify your implementations from Part 1 so that some of the assumptions about the syntactic correctness of the code are removed.

This part is significantly more difficult than Part 1 so be sure to give yourself plenty of time to complete it!

Step 1: Robust function header checking

Modify your *parse_function_header* function from Part 1 so that it removes the assumptions that were listed and, in addition to identifying the function name and parameters, determines whether the function header is syntactically legal.

The signature and behavior of the *parse_function_header* function should be the same as in Part 1 (in that it sets the global variables to the names of the function and any parameters), and **you can still assume that tokens are separated by a single space**, but you must consider the following:

1. The return type of the function is allowed to be "int", "void", or nothing, but can only be one of those three; otherwise, it is illegal
2. Although there will still be a space between the function name and the list of parameters, in order to be legal, the function name must not contain a space (i.e., it must only be one single token) and must not be "int" or "void", and the next token must be the left parenthesis
3. Also, although there will still be a space between the parentheses and other

tokens, and between any commas and other tokens, in order for the line to be legal, each parameter declaration must be separated by a comma and parameter names cannot contain spaces
4. Although all parameters are expected to be of type int, the line may contain invalid type names or none at all, either of which would be illegal
5. The line might not end with a curly brace, but this is legal
6. The identifiers must be legal names in C. For our purposes, we will say that identifiers must be a single token that starts with a letter and may not be the keywords "int", "void", or "return"; anything else is illegal.

Note that the key assumption is that all tokens are whitespace-separated, and for this part of the assignment **you do not need to consider any inputs that violate that assumption** (you'll have a chance to tackle that later).

If the line is legal, the *parse_function_header* function should have the same functionality as in Part 1; however, it if is illegal, the function should return 0 and the global variables should not be populated.

For example, the following lines are all **legal:**

`int fun ( )`
`void fun ( int a ) {`
`fun ( int dog , int cat ) {`

The following lines are all **illegal:**

`cat fun ( ) {`	Invalid return type
`int ( int a ) {`	No function name
`int fun ( a , int b ) {`	No type for first parameter
`int fun ( int dog cat ) {`	Parameter name can't contain space
`int fun ( int dog , )`	No second parameter
`int fun ( int 8cow )`	Illegal identifier

In these cases, this function does not need to provide the reason why the line is illegal; it simply needs to return 0.

Step 2: Robust syntax checking

Now modify your *parse_line* function from Part 1 so that its signature and behavior are the same but now it removes the assumptions that were listed and, in addition to finding variables that are being declared, determines whether the line of code is syntactically legal.

If the first token is "int", then you know that the line is attempting to declare one or more

variables. In this case, you need to check that the rest of the line contains syntactically legal variable declarations, though **you can still assume that all tokens are separated by a single whitespace**.

If the line is legal, the *parse_line* function should have the same functionality as in Part 1; however, it if is illegal, the function should return 0 and the global variables should not be populated.

The following lines are attempting to declare variables but are **illegal**:

`int x`	Doesn't end with semicolon
`int j k ;`	Variable names can't contain space
`int y = ;`	Need value after assignment operator
`int a = x 5 ;`	Need operator between values
`int void ;`	Illegal identifier name

If the first token is *not* "int", then either the line is attempting to assign a value to a variable, is a return statement, or is an illegal attempt to declare/assign one or more variables.

A legal return statement consists of:

- the keyword "return";
- followed by a semicolon, or:
    - followed by a token (an identifier or a literal), which is
    - followed by zero or more addition operations, i.e. "+" followed by a token, which are
    - followed by a semicolon

A statement that is attempting to assign a value to (but not declare) a variable consists of:

- the name of the variable;
- followed by the assignment operator ("=");
- followed by a token;
- followed by zero or more addition operations, i.e. "+" followed by a token;
- followed by a semicolon

Anything that does not adhere to one of these is illegal.

For example, the following lines are **illegal**:

`= 9 ;`	Missing variable name
`x = 8`	Doesn't end with semicolon
`a = b + ;`	Need value after addition operator
`y = + d ;`	Need value before addition operator
`m = a + b + ;`	Need value after addition operator

`cat x = 11 ;`	Illegal type
`return x`	Doesn't end with semicolon
`return 3 + ;`	Need value after addition operator

In the case where the line is not attempting to declare a variable, the *parse_line* function should return 1 if the line is valid and 0 if it is illegal, but the global variables should not be modified.

Part 3: Parsing without whitespace (10 points)

Now modify your *parse_function_header* and *parse_line* functions so that they remove the assumption that all tokens are separated by a single whitespace.

Depending on how you implemented those functions, this may require **major** changes, so do this **very** carefully and keep in mind that you should not risk losing a lot of points in previous parts of the assignment in order to gain a small number here.

Keep in mind that in C, whitespace is almost never required (except for separating types from identifiers), so all of the following should be considered legal:

```
fun(int dog,int cat){
int   fun (int dog ,int        cat ) {
int c=a+x,d,e=b;
b=a+y +   8;
```

This part of the assignment is **much** more difficult than the other parts, and although you can still use the *strtok* function, it won't be as simple as tokenizing based on whitespace, which may or may not exist.

Help! I'm stuck!

If you are having trouble with the *algorithm* that you're supposed to use (i.e., the steps that you need to go through in order to solve the problem), then we suggest doing the following (in this order) to try to figure it out:

1. Walk away for 30 minutes or so. Sometimes it's best to let your brain rest for a bit, and then come back to the problem somewhat refreshed.
2. Look in your class notes or lecture notes. Most likely, you'll be able to figure out the algorithm you need to use by considering what was covered in class.
3. If you're still stuck, then try to go see a member of the instruction staff during office hours. Regular office hours are posted in Piazza.
4. If you can't make it to office hours, look on Piazza to see if anyone else has asked a similar question (and received a response). If not, post a question there. If you're

afraid that asking your question would reveal a solution, then make it a "private question" that only the instruction staff can see.

Note that "do a Google search" is **not** on this list!!! You should be coming up with the solution on your own (or with the help of the instruction staff) to whatever extent possible. Thinking algorithmically is a crucial skill and this assignment will help you develop that.

If you are having trouble with the *implementation* (e.g., a bug you can't fix, some code that won't compile, etc.), then we suggest doing the following (in this order) to try to figure it out:

1. Look in your notes from CIT 593 and the previous lectures in CIT 595. Pretty much all the C you need to know should have been covered in one of those.
2. If that's not helping, look at a C reference manual, such as https://www.tutorialspoint.com/c_standard_library/ **This is not the same as "go on stackoverflow"!!** You should be striving for a mastery of C, not just a solution to this one particular problem, and over-reliance on stackoverflow may cause more problems than it solves.
3. If you're still stuck, then try to go see a member of the instruction staff during office hours.
4. If you can't make it to office hours, look on Piazza and post your question if you don't see a similar one already there.

The moral of the story: try to figure it out by yourself first, then use reference material, then ask a member of the instruction staff or use Piazza. But do not go "into the wild" for help!

If you are really struggling with this assignment (e.g. it's taking more than 12-16 hours), particularly if you have not yet completed CIT 593 or do not feel comfortable programming in C, please speak with the instructor.

Getting help on Piazza

Although we encourage you to use Piazza to get help from the instruction staff, keep in mind that this is course has CIT 593 and/or reasonable proficiency in C as prerequisites.

As such, members of the instruction staff will **not** answer Piazza questions along the lines of "here's my code, can you tell me what's wrong with it?" Unless requested by a member of the instruction staff, you should never post large pieces of code (either as an attachment or inline) in Piazza and ask us to debug it for you.

Rather, if you need help debugging your code, you should describe the steps you have taken so far to debug it, along with your speculation as to what's wrong with it, and only include code snippets as part of your description.

If you truly have no idea what's wrong with your code and cannot do the above, then you may want to consider rewriting it (if even you don't understand it, how can you expect someone else to?) or go discuss it with a member of the instruction staff during office hours.

One last thing about Piazza...

Although you are encouraged to post questions on Piazza if you need help or if you need clarification, please be careful about accidentally revealing solutions.

If you think your question might accidentally reveal too much, please post it as a private question and we will redistribute it if appropriate to do so.

Which brings us to….

Academic Honesty and Collaboration

Put simply, you are expected to work on this assignment **alone**.

You may discuss the assignment specification with other students, as well as your general implementation strategy and observations, but *you absolutely must **not** discuss or share code or the algorithm you are using to solve the problems in this assignment.*

As described in the course syllabus in Canvas, this includes but is not limited to:

- co-authoring code, either through pair programming or distributing the work
- sharing and distributing code, i.e., one student writes it and allows other students to see it (even if the other students further modify it)
- sketching out code or pseudocode together on paper, a whiteboard, etc., even if you type up the code separately
- reviewing another student's code "just to see how they did it"
- helping another student debug/troubleshoot his or her code

Although you can, of course, look online for help with the C standard library and syntax, and for explanations of cryptic error messages, you should not be receiving any help from outside sources, including students not taking this course, and you *definitely* should not be copy/pasting code from the Internet. You are expected to implement *all* of this on your own.

Also, on this assignment you may **not** use third-party libraries other than the standard C library, i.e. the ones listed at https://www.tutorialspoint.com/c_standard_library/index.htm. You may not use functions from other libraries without permission from the instructor.

Suspected violation of these policies will be referred to the Office of Student Conduct. Students who are found to have violated the policies will receive a score of 0 on this

assignment, in addition to other sanctions as deemed appropriate by the OSC.

If you run into problems, please ask a member of the teaching staff for help before trying to find help online or asking your friends! I'd rather give you an extension on the assignment than have to send you to OSC!

Grading

This assignment is worth a total of 100 points:

- Part 1 is worth 25 points total: Step 1 is worth 10 points and Step 2 is worth 15 points.
- Part 2 is worth 65 points total: Step 1 is worth 30 points and Step 2 is worth 35 points. Note that the test cases for Part 1 will not be used in evaluating Part 2, but you should ensure that your Part 2 implementations still work correctly given the assumptions from Part 1 regarding whitespace and syntactic correctness.
- Part 3 is worth 10 points.

For all parts of the assignment, your score will be determined by an automatic grading utility that will check that your functions correctly modify the *function_name*, *parameter_names*, and/or *variable_names* global variables for different inputs. Generally speaking, you will receive

- 100% credit if all global variables are correctly set
- 75% credit if there are minor mistakes in setting the variables
- 50% credit if there are major mistakes
- 25% credit if a reasonable effort was made but the variables are almost never set correctly
- 0% credit if no reasonable effort was made

The grader may also deduct up to 5 points at her discretion for things like not submitting a Makefile, submitting a Makefile that does not compile the code, egregious violations of C coding conventions (https://www.gnu.org/prep/standards/html_node/Writing-C.html is a good place to start), lack of comments, problems with readability, etc.

Additionally, we will use Valgrind to check for memory leaks for both parts of this assignment. It is okay if Valgrind reports that there is memory "in use at exit" or that memory is "possibly lost" or "still reachable," but you will lose points up to 5 points if Valgrind reports that memory is "definitely lost."

Submission

Part 1 is due **Friday, January 26, at 5pm.** To submit this part of the assignment, put the following into a *single* zip file:

- *find_symbols.c*
- *find_symbols_test.c*
- any other .c files that you created for this assignment
- Makefile

Then submit the zip file into the "Homework #1 – Part 1" assignment in Canvas. You may submit as many times as you'd like; only the last version will be graded. However, be sure your zip file contains *all* files whenever you resubmit.

Parts 2 and 3 are due **Friday, February 2, at 5pm.** To submit these parts of the assignment, put the updated .c files and your Makefile into a *single* zip file and submit the zip file into the "Homework #1 – Parts 2 & 3" assignment in Canvas. You may submit as many times as you'd like; only the last version will be graded.

Note that *find_symbols_test.c* will not be graded but must be submitted anyway and is subject to the same academic honesty policy as all other code.

Late submissions will be penalized by 10% if submitted up to 24 hours late, 20% for 24-48 hours late, and so on, up to one week, after which they will no longer be accepted.

If you know that your program does not work correctly, e.g. you didn't finish all the functionality, potentially incorrect outputs/behavior, things that make the code crash, memory leaks, or other known bugs, please describe these in a plain-text or PDF file and include it in your submission. This will greatly help the TAs grade your assignment and make sure that they give you credit for the things that you *did* get working.

Updated: 19 Jan 2018, 7:40am