CIT 595 Spring 2018
Homework #2

In this assignment, you'll continue the implementation of your C-to-LC-4 compiler.

This assignment will focus on these aspects of the compiler:

- Creating a symbol table that maps identifier names to their offset from the frame pointer on the stack
- Checking for semantic correctness, i.e. that identifiers are only used once and that variables are declared before they are used
- Generating LC-4 assembly language

Although you are free to discuss the intent and purpose 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, February 9, at 5:00pm.** Part 2 is due in Canvas on **Friday, February 16, at 5:00pm.** Don't wait until the first due date to start Part 2! This part of the assignment is 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 #2."

One of the files we are providing is a solution to Part 1 of Homework #1 (*find_symbols_soln.c*), which you will need for Part 1 of this assignment. You may use this implementation or your own.

For this assignment, we are providing a Makefile that you can use for compiling your code. You may modify it if you need to (you probably will), but you must include a Makefile with your submission and your Makefile must properly compile your code.

In order to standardize the grading of this assignment, **you are expected to use your VMware image from CIT 593 or the eniac computing cluster** for compiling and running your code; please do not use Windows environments such as Visual Studio, or even your Mac or another Linux platform. If you have trouble compiling the code we distributed, or do not know how to use the VMware image or eniac cluster, please notify a member of the instruction staff immediately.

Last, note that the TAs will also use Valgrind to check your code for memory leaks for **all parts** of the assignment. Be sure to free/deallocate any heap memory as necessary.

Part 1: Create symbol table

As we saw in class, when a compiler attempts to convert C code to assembly language instructions, it uses a data structure called a **symbol table** to keep track of where variables will be stored in memory. For local variables, it uses the offset (relative address) with respect to the frame pointer, which holds the address of the "bottom" of the stack frame for that function.

For instance, if we had the following C code:

```
int fun ( int a , int b ) {
  int x = 7 ;
  int y = a + b , z = 0 ;
  x = a + 4 ;
  int t = x + b ;
  return x + t ;
}
```

The LC-4 compiler would create a symbol table mapping variables to offsets as follows:

a	+4
b	+5
x	0
y	-1
z	-2
t	-3

The two parameters (`a` and `b`) are stored at addresses four and five greater than the frame pointer, respectively. And the four local variables are stored at addresses zero, one, two, and three less than the frame pointer.

If you are unsure of the purpose of the symbol table or how it is used by the compiler, please speak with a member of the instruction staff right away! The rest of the assignment won't make much sense if you're not sure how the symbol table is used.

Step 1. Implementing symbol table data structure

In this step, you'll implement the code that is needed for keeping track of the symbols and their offsets.

Implement the *add_symbol*, *get_offset*, and *clear* functions in symbol_table.c as follows:

add_symbol attempts to add the symbol and the offset to the symbol table as follows:

- if the *symbol* parameter is NULL or empty, the function should return -1 and not modify the table
- if the *symbol* parameter represents a symbol that is already in the symbol table (even if it is mapped to a different offset), the function should return 0 and not modify the table
- otherwise, the symbol table should be modified so that the *symbol* parameter is mapped to the *offset* parameter, and the function should return 1

get_offset attempts to find the offset in the symbol table for the specified *symbol* parameter as follows:

- if the *symbol* parameter is NULL or empty, or if the *offset* parameter is NULL, the function should return -1
- if the *symbol* parameter represents a symbol that is not found in the symbol table, the function should return 0
- otherwise, the value pointed to by *offset* (note that it is a pointer!) should be set to have the corresponding value for the *symbol* parameter in the symbol table, and the function should return 1

Last, the *clear* function should remove all entries/mappings from the symbol table and free the memory in which they are stored as necessary. This is required for testing purposes.

Obviously the three functions must share some common data structure, so that other functions can add symbols and then get their offsets. You may implement this data structure using either a hashtable or a linked list. A hashtable is most efficient, but a linked list would suffice in this case, too

However you choose to implement this data structure, **the implementation must be your own** and must not be code written by someone else. If it's code from CIT 593 that *you* wrote, that's fine, but if it comes from a friend or the Internet and you claim it as your own, well, that's pretty much the very definition of "plagiarism."

The data structure that you use should be a global variable in symbol_table.c so that the three functions can access it. You should not pass the data structure to the functions as we saw in class, i.e. you must **not** change the signatures of the functions; please speak to the instructor if you feel it is necessary to change these functions' signatures.

To check your implementation, modify symbol_table_test.c to write tests in which you call the different functions as described in the comments in the code. Some examples are provided but you should try other inputs as well.

Step 2: Populate symbol table

Next you'll write the code to populate the symbol table from a file containing the C code for a single function.

In order to complete this part of the assignment, you will need a solution to Part 1 from Homework #1, i.e. the basic *parse_function_header* and *parse_line* functions. You may use your own, or use the find_symbols_soln.c implementation available in Canvas.

Implement the *populate_symbol_table* function in symbol_table.c so that it opens the specified file using the *filename* parameter and then reads it one line at a time.

You can assume that the first line of the file will contain the function header. Pass that line to the *parse_function_header* function from Homework #1, and use the *add_symbol* function from Step 1 of this assignment to populate the symbol table with the names and offsets of the parameters. We will say that in LC-4, the offset for the first parameter is +4, the second is +5, and so on.

Then, pass all subsequent lines to the *parse_line* function so that, if there are any declarations in the line, the variable names and offsets are put into the symbol table. For local variables, the offset for the first local variable will be 0, for the second it's -1, and so on. Keep reading lines until you have reached the end of the file.

For simplicity, you may assume that:

- the file contains the definition of only one C function
- there are no blank lines in the file
- all tokens in the file are separated by a single whitespace
- each line contains syntactically valid code
- each identifier/variable name is syntactically legal
- each line of code contains no more than 100 characters
- each line of code (including the function header) declares no more than 10 variables

You may not, however, make any assumptions about the total number of variables being declared or the number of characters in each variable's name.

If you are unsure whether it is safe to assume something, please ask a member of the instruction staff.

Last, modify the implementation of *populate_symbol_table* so that it returns 1 to indicate that the file was read and that the symbol table was populated correctly, and returns -1 if the argument to the function (the name of the file) is null, if the file cannot be opened, or if an error occurred in populating the symbol table, e.g. if an attempt was made to add the same symbol twice.

For this and all subsequent parts of the assignment, you may modify the *parse_function_header* and *parse_line* functions from Homework #1 in any manner you choose, but you must **not** change the signature of the *populate_symbol_table* function in symbol_table.c.

To test your code, use the Makefile to compile populate_symbol_table_test.c and then run the test_populate_symbol_table program; be sure that the file “sample.c” (which you can download from Canvas) is in the same directory. This will run your *populate_symbol_table* function on that simple file, and then use your *get_offset* function from Part 1 to check that the correct values are in the table. You can, of course, write your own test input files but your *populate_symbol_table* function should work correctly with this one.

Note that this part of the assignment is due **Friday, February 9.**

Part 2: Converting code from C to LC-4

Now the fun part! In this part of the assignment, you will implement a compiler that will make two passes through the C code:

- the first pass will use the *populate_symbol_table* function from Part 1 to build the symbol table
- the second pass will convert each line of C code to LC-4 instructions, using the symbol table to determine the offset for the assembly language instructions

Step 1: Generating assembly language

The *main* function in compiler.c that we have provided makes the first pass through the code by attempting to build the symbol table using the *populate_symbol_table* function from Part 1.

Modify the *main* function so that, if *populate_symbol_table* returns 1, the program then re-reads the input file and generates LC-4 assembly code for each line of C code.

You do not have to generate the prologue or epilogue for the function being defined in the file, but you should be able to handle these four types of statements:

1. Simple declaration.
This would be something like "int x ;" or "int a , b ;" That is, a line which is a declaration (you'll know because it will start with "int") and no initialization (you'll know because there are no "=" operators). This is the easiest one to compile because there's nothing to do here!

2. Assignment.
This would be something like "x = a ;" or "a = h + b ;" or "y = x + 4 + t ;" or "dog = cat + mouse + elephant + zebra ;". In this case, you'll have to evaluate the right-hand side first.

Recall that, in order to read a variable from memory, you'd use its offset from the symbol table. So the assembly language code would be `LDR R#, FP, #<offset>`, where "R#" is the register to which you want to load (using numbers 0-3) and “<offset>” corresponds to the value’s entry in the symbol table.

If the symbol you're reading is an immediate (numeric) value, though, then the assembly language for using R0 would be:

```
AND R0, R0, #0
ADD R0, R0, #<value>
```

That is, you would first need to clear the value in the register, i.e. set it to 0, and then add the numeric value.

Because the ADD instruction can only hold +15 as its largest immediate value, you can assume that all numeric literals are between 0 and 15, inclusive.

Once you're done evaluating the right-hand side (for simplicity, only addition and simple assignment operators are allowed), then you would need to write the result to the variable on the left-hand side, like this: `STR R0, FP, #<offset>`.

3. Declaration and initialization.
This is probably the trickiest part. It would be something like "int x = 5, y = a + b;" It is similar to the assignment statements, except that there are conceivably multiple assignments on a single line. Think about how you found the variables being initialized in Homework #1, and then reuse as much as possible from how you solved the regular assignment statements above.

4. Return statement.
The return statement always starts with the keyword "return" and is followed by some expression that you may need to evaluate. It's very similar to the regular assignment statement, except that rather than writing to some variable, you always write to Frame Pointer offset 3, which is where the Return Value goes. So it's just `STR R0, FP, #3`, assuming the value to return is in R0.

For simplicity, you may assume that:

- the file contains the definition of only one C function
- there are no blank lines in the file
- all tokens in the file are separated by a single whitespace
- each line contains syntactically valid code
- each identifier/variable name is syntactically legal
- all variables are declared before they are used
- only simple assignment statements (using the "=" operator) and addition operators are used.

In order to help the TA grade your solution, your compiler must automatically generate comments in the LC-4 code for each line of C code, using a semicolon to start the comment. For instance, if your compiler encounters "y = a + b ;" in the C code, it should generate something like this:

```
; y = a + b ;
LDR R0, FP, #4
LDR R1, FP, #5
ADD R0, R0, R1
STR R0, FP, #-1
```

Note that there are a handful of different ways in which the same C code can be compiled into LC4, depending on the order in which you do various instructions and where you store values. But if you are unsure about your own solution, please ask a member of the instruction staff for help.

Rather than writing the LC-4 instructions to the screen using printf, your compiler should write them to a text file. See https://www.tutorialspoint.com/cprogramming/c_file_io.htm if you do not know how to do this in C.

The name of the file you create should be similar to the name of the input file, but with a ".lc4" extension instead of a ".c" extension. For instance, if the name of the input file is "test.c", then the file you create should be called "test.lc4"; doing this conversion is a little tricky, so ask for help (don't look online!) if you get stuck.

For this part of the assignment, you can write your own simple .c files and see if your compiler produces the correct LC-4 code. You might want to start with sample.c from Part 1 and compare the output of your program to sample.lc4 (available in Canvas); they should include the same instructions, though you may use different registers.

Step 2: Checking for errors

After completing Step 1, modify your program so that the *populate_symbol_table* function returns 0 if the program in the input file attempts to declare two variables with the same name (this includes parameters, too). *Hint:* use the return value of the *add_symbol* function from Part 1!

In addition to returning 0, the function should print (using printf) the name of the variable that is being declared again and the line number on which this occurs, and the program should terminate **with no output file being created**. The program only needs to do this for the first duplicate variable declaration it encounters, and not all of them.

Then, modify your program so that the *populate_symbol_table* function returns 0 if the program in the input file attempts to use (as part of an assignment expression) a variable that is not in the symbol table, i.e. has not been declared. In addition to returning 0, the function should print (using printf) the name of the variable that was not in the symbol table and the line number on which the variable was found, and then it should terminate with no output file being written. The program only needs to do this for the first undeclared variable it encounters, and not all of them.

Last, make sure that your program does not accidentally think that numeric literals (e.g.

8, 4, etc.) are variable names! Same with the various operators (such as "+" and "="), punctuation (like curly braces) and the keywords “int”, “void”, and “return”.

<u>Getting help on Piazza</u>

Although we encourage you to use Piazza to get help from the instruction staff, 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.

Additionally, 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.

<u>Academic Honesty and Collaboration</u>

As with Homework #1, 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.* 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 together on paper, a whiteboard, etc., even if you type it up 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 things like that, 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.

Grading

This assignment is worth a total of 100 points:

- Part 1 is worth 30 points total: Step 1 is worth 20 points and Step 2 is worth 10 points.
- Part 2 is worth 70 points total: Step 1 is worth 50 points and Step 2 is worth 20 points.

For all parts of the assignment, your score will be determined by an automatic grading utility that will check that your functions have the correct functionality for different inputs and different test files. Generally speaking, you will receive:

- 100% credit if all behavior is correct
- 75% credit if there are minor mistakes
- 50% credit if there are major mistakes
- 25% credit if a reasonable effort was made but the behavior is almost never correct
- 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, February 9, at 5pm.** To submit this part of the assignment, put the following into a *single* zip file:

- *symbol_table.c*
- *symbol_table_test.c* and *populate_symbol_table_test.c*
- the .c file containing the implementations of *parse_function_header* and *parse_line* that you used in this assignment
- *compiler.c*
- any other .c and .h files that you created for this assignment
- Makefile
- A PDF file describing the platform you used to compile and run your code: it must either be the CIT 593 VMware instance or the eniac cluster

Then submit the zip file into the "Homework #2 – 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.

Part 2 is due **Friday, February 16, at 5pm.** To submit these parts of the assignment, put the updated files into a *single* zip file and submit the zip file into the “Homework #2 – Part 2” assignment in Canvas. You may submit as many times as you'd like; only the last version will be graded.

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

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 your PDF file. 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: 2 Feb 2018, 2:40pm