CPSC 213 – Assignment 8

Dynamic Control Flow

Due: Wednesday, July 31, 2019 at 11:59pm
You may use one of your two late days on this assignment to make it due Thursday.

Learning Objectives

Here are the learning objectives of this assignment, which will be examined in this week's quiz. They are a subset of the unit learning objectives listed in the slides.

After completing this assignment you should be able to:

1. write C programs that use function pointers;
2. explain why switch statements in C (and Java until version 1.7) restrict case labels to cardinal types (i.e, things that map to natural numbers);
3. convert C switch statement into equivalent C statement using goto's, and into an explicit jump table of label pointers (a gcc extension to C);
4. convert C switch statement into equivalent assembly language that uses a jump table; and
5. determine whether a given switch statement would be better implemented using if statements or a jump table and explain the tradeoffs involved.

Goal

This week you continue from where you left off last week with `poly.c` to look at the use of function pointers as parameters to the Dr. Racketish list primitives and as a way to implement `switch` statements.

First you will create a small portion of Dr. Racket in C. You will start with a provided implementation of a dynamically expandable list that knows its length. It's like a list in Dr. Racket (or Java), and unlike arrays in C that are fixed size and do not know how big they are intrinsically. You will implement several functions that operate on lists using *abstraction-functions* (i.e., C function pointers). You will test your program using a provided test file. Finally, you will implement your own program that uses this list code.

Then you will move to switch statements. First you will examine the snippet we looked at in class that gives you an example of a switch statement that is implemented by a jump table. Next, you will examine a mystery assembly program to determine what it does.

Finally you will convert a C program that contains switch statements into one that uses only `goto`'s and a jump table, instead of switch statements. Funny enough, this program is a C version of the Java Simple Machine simulator that you used in Assignments 1-7, perhaps a fitting way to bid a fond adieu to assembly language (for now).

Question 1: Using Function Pointers on Lists [45%]

In *www.ugrad.cs.ubc.ca/~cs213/cur/assignments/a8/code.zip* in the `q1` subdirectory you will find the following files: `list.[ch]`, `test.c`, and `Makefile`.

Part 1: Implement the List Iterator Functions

The file `list.c` contain the skeleton implementation of several list operators adapted from Dr. Racket: `map1`, `map2`, `foldl`, `filter`, and `foreach`. The two map functions are variations on Dr. Racket's map: `map1` operates on one list and `map2` on two.

Here's a brief summary of what these functions do (look up Dr. Racket documentation for more details):

- `map` takes a function and a set of lists (in our case one or two lists) and generates a new list by applying the function to each element of the list;

  ```
  (map + '(1 2 3) '(1 1 1)) => '(2 3 4)
  ```

- `foldl` takes a function, an initial value, and a set of lists (in our case just one list) and generates the value that results from iterating over the list, calling the function on an accumulated value, which starts at the specified initial value, and each element of the list in turn, updating the accumulated value as it goes;

  ```
  (fold + 0 '(1 2 3)) => 6
  ```

- `filter` takes a function and a set of lists (in our case just one) and generates a new list that contains the elements of the original list for which the function returns true;

  ```
  (filter positive? '(-2 3 -8 5)) => (3 5)
  ```

- `for-each` takes a function and a list of lists (in our case just one) and calls the function for each item in the list. It produces no value.

  ```
  (for-each print '(1 2 3)) => #<void>   printing the values 1, 2 and 3
  ```

Implement the `TODO` portions of `list.c` and use `test.c`, which uses these functions, to test your code. Your code must pass `valgrind` with no memory leaks. Be sure to run it with a

variety of inputs, and include one with a long list of strings and integers (*i.e.*, longer than 10 each).

Part 2: Implement a Program that Uses the List Iterator Functions

Finally, implement a program with no loops of any kind, other than an initial loop to read the values of `argv` in Step 1 below, that uses `list.[ch]` to do the following, placing the implementation in the file `trunc.c`.

The input to this program is a list of numbers and strings presented on the command line; *e.g.*,:

```
./trunc 4 apple 3 peach 5 banana 2 3 grape plum
```

The program uses the numbers to truncate the strings and prints the resulting strings, one per line and, then the string concatenated (and separated by spaces), and finally the maximum string length. So in this case the output would be.

```
appl
pea
banan
gr
plu
appl pea banan gr plu
5
```

Notice that the numbers are paired with strings based on their order in the string and not their proximity to each other and so the following would produce the same output.

```
./trunc 4 3 5 2 3 apple peach banana grape plum
```

Here is an outline of the program. You need to follow the steps listed. This will involve creating several functions to pass to the higher-order functions in `list.h`. Take each step one at a time. Implement a step and then test it by using `list_foreach` to print the list you create (be sure to remove these extra prints, however, before you `handin` your solution; the only prints allowed in the final version are those specified in Steps 7 and 8, otherwise you will confuse the autograder). For string processing, you may use any standard library function that starts with `str`, for instance, `strcpy`, `strncpy`, `strcmp`, `strdup`, `strcat`, `strtol`, etc., so you should not require any explicit loops to handle strings.

1. Read a list of strings from the command line and add them to a list (using your one and only regular loop). Note that `argv` is an array of the command line arguments and `argc` is the length of that array. Ignore `argv[0]` since that is the path to the program itself. Read over past assignment code to see how the command line is processed.
2. Map over the list to produce a new list of numbers (the list must actually be a list of pointers to integers) that processes each string to determine if it is a number. If it is, the corresponding value in this new list should be that number, otherwise it should be

a -1. Use the standard-C library procedure `strtol` to convert strings to numbers. *Hint: In A7, we already provided a function that does the number/string conversion, which you may adapt.*

3. Map over the new list of numbers and the original list of strings together to produce a new list of strings with a NULL value for every string that is a number (*i.e.*, where the number list is not -1).
4. Filter the number list to produce a new list with all negative values removed. The list may thus be shorter than the original list.
5. Filter the string list to produce a new list with all NULLs removed.
6. Produce yet another list of strings that uses these two new lists of numbers and strings to truncate strings, on a pairwise basis, taking the values in the number list to be the maximum length of the entry in the string list. Recall that strings end with a `null` (i.e., 0) character. For example if you had these two lists

   ```
   list0 = [4, 3, 5, 2, 3]
   list1 = [“apple”, “peach”, “banana”, “grape”, “plum”]
   ```

   The new list of strings would be

   ```
   list2 = [“appl”, “pea”, “banan", “gr”, “plu"]
   ```

7. Print this revised list of strings, one string per line. Print nothing else on each line.
8. Then join this list into a single array, separated by spaces and print it on the next line. If there are no input strings (or if all of the lengths are zero) then print the empty string (i.e., a blank line).

   ```
   s = “appl pea banan gr plu”;
   ```

9. Compute the maximum value in the numbers list and print it. Again; just print this number.
10. Now rid your program of memory leaks by ensuring that everything that was malloced in previous steps is freed. Again, use list functions to carry this out.

Ensure that you program prints nothing other than what is specified in Steps 7-9.

Question 2: Switch Statements in Assembly [15%]

Examine the code for `SB-switch`, which you will find in the `q2` subdirectory of *www.ugrad.cs.ubc.ca/~cs213/cur/assignments/a8/code.zip*. Carefully read the three different implementations in the C file. Run the assembly version in the simulator and observe what happens. This part is not for marks, but it will help you with the next step.

The file `q2.s` contains uncommented assembly code that corresponds to a single C procedure starting at address `0x300` and another procedure that sets up the stack and calls the first

procedure. Read this code and run it in the simulator. If you see a jump table, you might want to add labels for the addresses it stores to make the code easier to read. Comment every line and write an equivalent C program that is a likely candidate for the program the compiler used to generate this assembly file (with the exception of the stack-setup code, which is only for the .s file). Call this program `q2.c`.

For your solution to pass the *handin* tests it must declare a procedure named `q2` that encapsulates the code starting at address `0x300`, that has the appropriate number of arguments (in the correct order), and that returns the appropriate value. To test your code, you can include a `main` procedure and any other procedures you like.

Question 3: Jump Tables in C [40%]

The file `sm.c`, found in the `q3` subdirectory of *www.ugrad.cs.ubc.ca/~cs213/cur/assignments/a8/code.zip*, is a C implementation of `CPU.java`; i.e., it is a *sm213* virtual machine. It executes *sm213* machine code, but it doesn't include an assembler and it doesn't have a gui. As a result, it's a small C program that you should be able to read and understand.

Since the C version doesn't contain an assembler, we'll need to use the Java version to convert assembly into machine code. To do this, run the simulator from the command line using the "`-d`" option. For example, to convert `foo.s` into machine code type the following at the UNIX command line.

```
java -jar SimpleMachine213WithSyscalls.jar -d foo.s
```

The simulator places the resulting machine code in the file `foo.machine`. This machine code can be loaded into either the C SM213 implementation or the existing Java simulator.

The provided `code.zip` file contains the following programs that have already been converted to machine code; i.e., you have "`.s`" and "`.machine`" versions of each of these programs.

`simple-test`	A very simple test that includes only two instructions.
`test`	A test that uses every instruction (except the double-indirect jumps).
`a2-test`	The `test.s` file from Assignment 2 that tests ALU and memory instructions.
`max`	Compute maximum value of a list of integers.

Feel free to convert your own files to serve as test cases. When you run the C simulator you specify which file to execute and provide zero or more of the following command line options.

`-p` *a*	Set the starting `pc` to address *a* (assumed to be in hex); default is first instruction.
`-r`	Print the ending values of the register file.
`-m` ***a*:*c***	Print the ending value of memory at address ***a***, continuing for ***c*** lines (4 bytes per line). This option can be repeated to print multiple non-contiguous memory regions.

For example, to run foo.machine and print out the registers and memory locations `0x2000-0x2007`, type the following at the command line.

```
./sm –r –m 2000:2 foo.machine
```

Let's start by reading `sm.c` and figuring out how it all works.

Read and Understand the C Simulator

Examine the `fetch()` and `exec()` procedures in `sm.c`. These are equivalent to the two functions in `CPU.java`. They execute in sequence each clock tick to fetch an instruction from memory and execute it. Read `exec()` carefully. Note what `mem[]` and `reg[]` are. Pause to reflect on how simple and cool this all is. This right here is all that a computer needs to do.

There is nothing to hand in for this part.

Replace Switch Statements with Jump Tables [40%]

To see how the compiler implements switch statements like those found in `sm.c`, copy this file to the file `sm-jt.c` and then modify `sm-jt.c` to remove all switch statements in the `exec()` procedure and replace them with jump tables and goto's as described in class. Note that there are two switch statements and so you will need two jump tables (or, a single large one covering every possible value for the first byte).

Label pointers (e.g., `&&L0`) and double-indirect goto's (e.g., `goto *jt[i]`) are gnu-C extensions and so some compilers may not support them (all versions of `gcc` and `clang`/`llvm` do support these extensions). So, you may need to move to a lab computer to develop this program.

Now, test your new jump-table-driven program. Start with `simple-test.machine`. This test uses one instruction from each of the switch statements and so if it works you'll likely have got this mostly right. You'll also want to use `test.machine` to ensure that you are dispatching to the correct case block for every instruction. The easiest way to do this is to place a print statement in every case block and then run the test to ensure that every print statement executes, and the execute in the correct order.

What to Hand In

Use the `handin` program.

The assignment directory is `~/cs213/a8`, it should contain the following ***plain-text*** files.

1. (optional) `PARTNER.txt` containing your partner's CWL login id and nothing else. Your partner should not submit anything.
2. For Question 1: `list.c` and `trunc.c`.

3. For Question 2: `q2.s` and `q2.c`.
4. For Question 3: `sm-jt.c`.