CPSC 213 – Assignment 8

Dynamic Control Flow

Due: Saturday, November 4th, 2017 at 11:59pm
After a 12-hour, no-penalty, grace period, not late assignments will be accepted.

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. Racket-ish 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. Its like a list in Dr. Racket (or Java), and unlike arrays in C (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`, `testip.c`, and `Makefile`.

Part 1: Implement Map on a List of Integers

First note that we'll be working with lists of integers and so it will be convenient to use the trick we've talked about where we cast a pointer to an integer. Recall that for this to work correctly

the integer type we use must have the same size as the pointer (i.e., the same number of bytes). Virtually all machines these days have 8-byte pointers and so using the integer type `long` will typically work fine. But, the standard C library provides an special integer type called `intptr_t` whose size is adjusted so that it matches the native pointer size on the machine that compiles the program. So you can do things like this.

```
void*    v;
intptr_t l = (intptr_t) v;  // same as long on most systems
```

The first step is optional and not marked, but its recommended and will help you with Part 2. Create a small test program that implements `map1` and `foreach` (described in Part 2) to operate on list elements of type `intptr_t`. You are provided with the implementation of `foreach`, on `element_t`. Start by implementing `test_foreach` by replacing instances of `element_t` with `intprt_t`. Then use this code as a guide to implement `test_map1`.

```
void test_map1    (void (*f) (intptr_t*, intptr_t), struct list*, struct list*)
void test_foreach (void (*f) (intptr_t), struct list*)
```

Then use these operations in a simple test sequence to increment each element of a list of integers and then print the list.

The increment and print methods would thus look like this:

```
void inc (intptr_t* rp, intptr_t a) {
  *rp = a + 1;
}

void print (intptr_t v) {
  printf ("%d\n", v);
}
```

Create source and destination lists and then call `map` and `foreach` like this.

```
intptr_t a[]    = {1, 2, 3, 4, 5, 6, 7, 8, 9 , 10, 11};
struct list* l0 = list_create();
struct list* l1 = list_create();
list_append_array (l0, (element_t*) a, sizeof(a)/sizeof(a[0]));
test_map1    (inc, l1, l0)
test_foreach (print, l1);
```

Your program should print the numbers: `2`, `3`, …, `12`.

Get this to work and then get ready to move to Part 2, by converting your implementation of `test_map1` into `list_map1`, replacing `intptr_t` with `element_t`. And then change `inc` and `print` in the same way so that they now look like this.

```
void inc (element_t* rpv, element_t av) {
  intptr_t* rp = (intptr_t*) rpv;
  intptr_t  a  = (intptr_t)  av;
  *r = a + 1;
}
```

```
void print (element_t vv) {
  intptr_t v = (intptr_t) vv;
  printf ("%d\n", v);
}
```

That is all that you should need to change. See that it still works.

Part 2: 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 (google Dr. Racket for more):

- `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)
  ```

- `foreach` takes a function and a list of lists (in out case just one) and calls the function for each item in the list. It produces no value. (Note that the actual name of this function in Dr. Racket is `for-each`, but we are going to call it `foreach` since you can't have dashing is C names.

  ```
  (foreach 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. Note that if you did Part 1 you already have two of these completed.

Your code must pass `valgrind` with no memory leaks. Be sure to run it with a variety of inputs, include one with a long list of strings and integers (i.e., longer than 10 each).

Part 3: 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.,:

```
./a.out 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 also the maximum string length. So in this case the output would be.

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

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

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

Here is an outline of the program. You need to follow the steps listed. 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).

1. Read a list of strings from the command line and add them to a list. 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.
2. Iterate over the list to produce a new list of numbers (use `intptr_t` as in `test.c`) 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. See its `man` page for details.
3. Iterate 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

```
l0 = [4,3,5,2,3]
l1 = [“apple”, “peach”, “banana”, “grape”, “plum”]
```

The new list of strings would be

```
l3 = [“appl”, “pea”, “banan", “gr”, “plu"]
```

7. Print this revised list of strings, one string per line. Print nothing else on each line.
8. Compute the maximum value in the numbers list and print it. Again; just print this number.

Ensure that you program prints nothing other than what is specified in Steps 7 and 8.

Also provided is file `testip.c` does the same thing as `test.c` but uses `int*`'s instead of `intptr_t`'s for values. You can probably ignore this file, but you might find it helpful when considering how to manipulate lists of strings (which are `char*`'s).

Question 2: Switch Statements in Assembly [10%]

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 [45%]

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, its 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. The provided `code.zip` file contains a new version of `SimpleMachine213.jar` that you can use to convert assembly into machine code (the version you have does not do this). 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 SimpleMachine213.jar -d foo.s
```

The simulator places the resulting machine code in the file `foo.m`.

The provided `code.zip` file contains the following programs that have already been converted to machine code; i.e., you have "`.s`" and "`.m`" 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.
`a6-q3`	The Assignment 6 solution that finds the student with the median grade.

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.m and print out the registers and memory locations `0x2000-0x2007`, type the following at the command line.

```
./sm -r -m 2000:2 foo.m
```

Lets start by reading `sm.c`, seeing how cool this is, and then making a small change.

Read, Understand and Modify the C Simulator [10%]

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.

Now, you'll notice two `TODO` comments in `exec()`, for the two double-indirect jump instructions that you implemented in Java in Assignment 7. Implement them again, this time in

C. Use the implementations of the two load instructions and the indirect jump instruction as a guide.

Test your implementation by creating a small assembly test file. You might want to first test the test file in the Java version of the simulator. Make sure that your test program changes either memory or the register file in a recognizable way when the two instructions execute correctly. Then, convert this file to machine code (see above) and run it in your modified C simulator, having it print whatever memory or registers you need to confirm correctness (see above). You might need to add print statements or otherwise debug `sm.c`.

Replace Switch Statements with Jump Tables [35%]

Finally, copy `sm.c` 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.

Recall that label pointers (e.g., `&&L0`) and double-indirect goto's (e.g., `goto *jt[i]`) are a gnu-C extensions and so some compilers may not support them (all versions of `gcc` and `llvm` (the Mac compiler) do support these extension). So, you may need to move to a lab computer to test this your program.

Now, test your program. Start with `simple-test.m`. 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.m` 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. `README.txt` that contains the name and student number of you and your partner
2. `PARTNER.txt` containing your partner's CS login id and nothing else (i.e., the 4- or 5-digit id in the form a0z1). Your partner should not submit anything.
3. For Question 1: `list.c` and `trunc.c`.
4. For Question 2: `q2.s` and `q2.c`.
5. For Question 3: `sm.c` and `sm-jt.c`.