CPSC 213 – Assignment 7

Dynamic Control Flow

Due: Wednesday, March 16, 2015 at 11:59pm
The grace period extends to Saturday, March 18 at 8am. No late assignments accepted after this time.

Goal

This week's assignment is all about dynamic control flow. You'll approach this from several perspectives. First, you will implement the last two instructions in the SM213 ISA. By now this should be quite straight forward.

Then, you will examine the code we covered in class that models Java polymorphism in C. You have Java, C and assembly versions of this file. You will use this code as a starting point to implement your own simple "Java program" in C, using our model of polymorphism to handle the polymorphic dispatch contained in the Java program.

Next we move to using function pointers as parameters. You will create a small portion of Dr. Racket in C. You will start with the implementation of a dynamically expandable list that knows its length. Its 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.

Finally, we move to switch statements. First, you will examine the snippet we looked at in class this gives you an example of a switch statement that uses a jump table. Next, you will examine a mystery assembly program to determine what it does. Finally, as a Bonus question, you will convert a C program that contains switch statements into one that uses `goto`'s and a jump table, and no switch statements. As a way to say adieu to the simulator, this C program is a C implementation of CPU.java.

Background

Hints: Manipulating Strings in C

You will also be manipulating strings in your C program in Question 2. Strings in C are more troublesome, by a long way, than strings in Java, due entirely to the dynamic-allocation issues we have been talking about; i.e., deciding what part of code is responsible for freeing something that is malloced from the heap.

A C string is stored in an array of characters. The string itself, which can be smaller than the array that contains it, is terminated by the first `null` (i.e., 0). You'll need to consider a couple of issues.

First, when a procedure receives a string as an input parameter, if it procedure stores that string, it should make its own copy of the string rather than storing the string pointer. Storing the pointer is dangerous because the caller could free the object following the call and thus turn this stored value into a dangling pointer. By copying, the procedure ensures that its copy of the string is safe from whatever its caller does with the string after the call returns.

You will want to use the standard method called `strdup` to do this (see its man page; e.g., via google or by typing "man strdup" at a unix command line). You will also need to add "`#include <string.h>`" to the beginning of your file.

For example your code should look like this:

```
void bar (struct X* anX, char* string) {
    anX->string = strdup (string);
}
```

And **not**, as it would in Java, like this:

```
void bar (struct X* anX, char* string) {
    anX->string = string;
}
```

Because if you did this, a caller that does the following creates a dangling pointer:

```
void foo (struct X* anX) {
    char string[] = "Hello World";
    bar (anX, string);
}
```

Similarly, as we have examined, it is often better to avoid writing a C procedure that returns a pointer to an object that it dynamically allocates. Instead, if it can, it should leave it to its caller to perform the dynamic allocation and simply copy its result to that location.

For example, Java code that looks like this:

```
String getString () {…}
```

Would in C look like this:

```
void getString (char* buf, int bufSize) {…}
```

Where `buf` is a pointer to memory provided by that caller into which `getString` copies its result up to the limit of `bufSize` bytes.

Finally, you will need to convert numbers to strings and to concatenate strings. The easiest way to do this is with the standard procedure called `snprintf` that uses `printf` formatting to write to a string. So, for example if you wanted to create the string "`Hello World 42`" from the string "`Hello World`" and the integer `42`, you would do something like this:

```
char buf [1000];
snprintf (buf, sizeof (buf), "%s %d", "Hello World", 42);
```

What to Do

Question 1: Implement and Test Double-Indirect Jumps [5%]

There are two remaining instructions to implement in the simulator, described below. Implement them.

Instruction	Assembly	Format	Semantics
dbl ind jmp b+d	`j *o(rt)`	`dtpp`	pc ← m[r[t] + (o == pp*4)]
dbl ind jmp indx	`j *(rb,ri,4)`	`ebi-`	pc ← m[r[b] + r[i]*4]

Then, use the simulator to examine the snippet `SA-dynamic-call` that you will find in this week's code file at *www.ugrad.cs.ubc.ca/~cs213/cur/assignments/a7/code.zip* in the subdirectory named `question1`. There you will also find the `CPU.java` solution from Assignment 6, which you can use as a starting point for this question, if you like.

Question 2: Modelling Polymorphism in C [35%]

In *www.ugrad.cs.ubc.ca/~cs213/cur/assignments/a7/code.zip* in the `question2` subdirectory you will find a file named `Poly.java`. This file contains a simple example of polymorphism using two classes: `Person` and `Student`, which extends `Person`. Write a new C program called `poly.c` that does the same thing in C. Start by copying `SA-dynamic-call.c`. Then make the necessary changes to replace `A` and `B` with `Person` and `Student`.

Follow the approach outlined in class. For example solution should include the following structs: `Person_class`, `Person`, `Student_class`, and `Student`. It should have the following static objects: `Person_class_obj` and `Student_class_obj`. And, it should have the following methods: `new_Person(char*)` and `new_Student(char*,int)`.

The class `Poly` contains two procedures that you need to implement as well, but there will be no `Poly` class or anything like it in your C program. Notice that these are both `static` procedures. One is, of course, `main`. And the other, `print`, is function that contains a polymorphic dispatch to the method `toString`.

The Java code uses the Java class `String`. You must use C strings instead. The Background section has a few pointers on using strings in C.

Question 3: Using Function Pointers on Lists [35%]

In *www.ugrad.cs.ubc.ca/~cs213/cur/assignments/a7/code.zip* in the `question3` subdirectory you will find the following files: `list.[ch]`, `test.c`, and `testip.c`.

Note that to compile a program such as `test.c` to use `list.c` you need to tell the compiler to combine them either like this, if you want to compile the two files separately:

```
gcc -std=gnu11 -c -o list list.c
gcc -std=gnu11 -o test test.c list.o
```

Or like this if you want to compile both at the same time:

```
gcc -std=gnu11 -o test test.c list.c
```

Step 1: Implement Map on a List of Integers

This step is optional, but recommended.

As a precursor to Step 2, implement the `map1` and `foreach` (described in Step 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 program 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 Step 3, 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.

Step 2: Implement the List Iterator Functions

The second part of the `list` files 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 Step 2 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).

Step 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 4: Reading Assembly Code [10%]

Examine the code for `SB-switch`, which you will find in the `question4` subdirectory of *www.ugrad.cs.ubc.ca/~cs213/cur/assignments/a7/code.zip*. Carefully read the three different implementations in the C file. Run the assembly version in the simulator and observe what happens.

The file `A7_4.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 `A7_4.c`.

Question 5: Switch Statements in Assembly [15%]

The file `sm.c`, found in the `question5` subdirectory of *www.ugrad.cs.ubc.ca/~cs213/cur/assignments/a7/code.zip*, is a C implementation of `CPU.java`; i.e., it is a sm213 virtual machine.

It has been setup to load and execute two assembly programs include in that subdirectory: `max.s` and `test.s`. It does this by loading the machine representation of the instructions since this program does not include an assembler. It is easy to get the translation from assembly to machine code from the Java simulator (its just to the left of the assembly code, line by line). Or you can run the command-line version of the simulator and print these values like this:

```
java -jar SimpleMachine213.jar -i cli
(sm) l max.s
(sm) i ins          prints the assembly code
(sm) e/x 0x50:62    prints the machine code
```

Take a look at this so you understand what is going on, but don't worry about the details of the instructions. They are stored in `Sm.c` as constants (i.e., `loadMax()` and `loadTest()`).

The difference between `max.s` and `test.s` is that `max` does something interesting (it computes the maximum value of an array of integers) and test simply executes each instruction in the ISA once (in order of their op code). For `max`, the virtual machine prints the value stored in memory at address 0 after `max` finishes; this is the maximum value of the list that starts at address 4. Notice that it works. Modify the integer array in `loadMax` so that it computes a different maximum.

Your task is to write an equivalent C program called `sm_jt.c` that replaces the two switch statements in `exec()` with `gotos` using a jump table and label pointers as shown in class. Note that label pointers are a gnu C extension and so some compilers may not support them (all versions of gcc and llvm (the Mac compiler) do support this extension). So, you may need to move to a lab computer to test this your program. Leave cases marked "NYI" empty; i.e., they are cases with an empty body so just implement them that way.

Note that there are really two switch statements in this program. You will want to treat each of the separately.

To test your program it is sufficient to show either that it computes the correct value for `max` or that `loadTest` runs okay. Start with `loadMax`. If it works; you are done. If not, then switch to `loadTest` and change every case arm to include a print statement that prints the value of the case label (e.g., "`case 2: printf ("2\n"); …`"). This will simplify debugging greatly. Then if you show that you print the case labels in the correct order, you are done.

What to Hand In

Use the `handin` program.

The assignment directory is `~/cs213/a7`, 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: `CPU.java`.
4. For Question 2: `poly.c`.
5. For Question 3: `list.c` and `trunc.c`.
6. For Question 4: `A7_4.s` and `A7_4.c`.
7. For Question 5: `sm_jt.c`.