1 Lab 11

Date: Apr 23, 2020

This document first describes the aims of this lab. It then provides necessary background. It then describes the exercises which need to be performed.

In the listings which follow, comments are any text extending from a `#` character to end-of-line.

1.1 Aims

The lab will give you some exposure to the following topics:

- Write a program from scratch with minimal help.
- Build your own ADT.
- Understand the use of a stack to recognize nested constructs.

1.2 Starting up

Follow the *provided directions* for starting up this lab in a new git `lab11` branch and a new `submit/lab11` directory. Start a `script` session to log your interaction into a `lab11.LOG` file.

You will be doing all your work in your `submit/lab11` directory:

```
$ cd ~/i240?/submit/lab11
```

The rest of this lab assumes that you are using `bash` as your shell.

1.3 Exercise

Write a program `balanced`, which will succeed silently with exit status 0 iff its command-line arguments are balanced delimiters. If they are not balanced delimiters, the program should terminate with an error message and non-zero exit status.

The delimiters which should be allowed are `(` and `)`, `<` and `>`, `[` and `]`, `{` and `}` (since many of these characters are special to the shell they should be quoted). Examples:

```
$ ./balanced '(' '[' '[' ']' ']' ')'
$ echo $?
0
$ ./balanced '(' '[' '[' ']' ']' ']'
```

```
unbalanced at argument 6
$ echo $?
1
$ ./balanced '(' '<' '[' '{' '}' ']' '>' ')'
$ echo $?
0
$ ./balanced '(' ')' '[' ']'
$ ./balanced '(' '<' '[' '{' '}' ']' '>'
unbalanced at argument 7
$ ./balanced '(' '@' ')'
$ invalid delimiter '@'
$
```

Checking whether the arguments are properly nested is very straight-forward when using a stack.

1. Initialize an empty stack.
2. For each argument:
   (a) If the current argument is not a delimiter, error out.
   (b) If the current argument is an opening delimiter like `(` or `<`, push it on to the stack.
   (c) If the current argument is a closing delimiter like `)` or `>`:
      - If the stack is empty, error out.
      - Pop the stack. If the popped element does not match the current delimiter, error out.
3. If the stack is empty after all arguments have been processed, then succeed; other error out.

You can proceed as follows:

1. Build a stack ADT. This can basically be a **class** which supports `push()`, `pop()` and `size()` operations. The stack can be maintained as a private `int stk[]` array within the `class`. You can assume a max stack size of say 16.

   You will need some kind of `stkIndex` member within your class to point to the current top-of-stack. You need to carefully maintain an invariant as to what `stkIndex` refers to:
   - Does it refer to the next free location?
   - Does it refer to the last occupied location?
2. Iterate through the command-line arguments, checking each argument as outlined above.

Note that each valid command-line argument should be a "string" of length 1. So validating that it is a delimiter should require checking the length of the argument and then validating the first character. Something like:

```
strlen(argv[i]) == 1 && chkDelim(argv[i][0])
```

You should not be hardcoding the delimiter chars all over your code. Instead use a table of delimiters specifying an encoding which makes it easy to determine which delimiters go together:

```
struct DelimMap {
  int delim;
  int encoding;
};
DelimMap delimEncodings[] = {
  { '(', 1 },
  { ')', -1 },
  { '[', 2 },
  { ']', -2 },
  ...
};
```

A look-up of this table can be encapsulated within a **getDelimEncoding()** function. If you set it up to return 0 on not finding a delimiter, then **chkDelim()** becomes trivial.

1.4 Winding Up

Follow the *provided directions* for winding up this lab. Terminate your `script` session producing the log file `lab11.LOG` in your `lab11` directory. Add all your files to git and commit. Then merge your `lab11` branch into the `master` branch and commit your changes.