Notes for Programming in C Lab Session #2

September 29, 2019

1 Introduction

The purpose of this lab session is primarily to ease you into programming in C, by writing a small example program.

2 Overview

In the first lecture, we saw how to reverse a whole string using a for loop. That is, given a variable `s` holding the string:

```
"University of Cambridge!"
```

we were able to modify the string so that it had the contents:

```
"!egdirbmaC fo ytisrevinU"
```

In this lab session, you will write a function that does not reverse the *whole* string, but instead reverses *each word* within the string. So instead, you will seek to modify the original string so that its contents are:

```
"ytisrevinU fo egdirbmaC!"
```

Moreover, in the second recorded lecture, we saw how to define C functions. So you will do this by implementing a small library of functions whose prototypes and specifications are given in `revwords.h`, and whose implementation should go in `revwords.c`.

3 Instructions

1. Download the `lab2.tar.gz` file from the class website.
2. Extract the file using the command `tar xvzf lab2.tar.gz`.
3. This will create a `lab2/` directory. Change into this directory using the `cd` command.
4. In this directory, there will be files `lab2.c`, `revwords.h`, and `revwords.c`.
5. There will also be a file `Makefile`, which is a build script which can be invoked with the command `make`. It will automatically invoke the compiler and build the `lab2` executable.
6. Run the `lab2` executable, and see if your program works.

4 The Functions to Implement

`void reverse_substring(char str[], int start, int end)`

`reverse_substring(s, start, end)` function takes a string s, and two integer indices start and end identifying the start and end of a substring of s. The function may assume that start and end are both valid indices nto the string.

`int find_next_start(char str[], int len, int i)`

`find_word_start(s, len, i)` takes a string s of length len, and an index i (which must be strictly less than len). It then returns the index k which is the starting position of the next word beginning at position i or later. If no such index exists, then it should return -1.

`int find_next_end(char str[], int len, int i)`

`find_word_end(s, len, i)` takes a string s of length len, and an index i (which must be strictly less than len). It returns the first index k past the end of the word starting at i.

`void reverse_words(char s[])`

`reverse_words(s)` takes a string s, and reverses all of the words in it. Here, a ”word” is defined as a contiguous sequence of alphabetic characters.

Notes for Programming in C Lab Session #3

September 29, 2019

1 Introduction

The purpose of this lab session is to write some small programs that do pointer and structure manipulations.

2 Overview

In the last couple of lectures, you have learned how to define structures, pointers, and functions. In this lab, you will learn how to define functions to manipulate pointer-based data structures, by working with the simplest pointer-based data structure of all — the singly-linked list.

In C, a datatype for linked lists can be declared with the following structure declaration:

```
struct List {
  int head;
  struct List *tail;
};

typedef struct List List;
```

This defines a type `struct List` which consists of a `head` field containing an integer, and a `tail` field which contains a pointer to another `struct List`. (The typedef defines a type abbrevation `List` standing for the structure type `struct List`. This lets us write `List` in function prototypes and variable declarations rather than repeating the keyword `struct` over and over again – this is a common idiom in C programming!)

A "linked list" is then just a pointer to this structure type. Next, you will implement a small library of functions whose prototypes and specifications are given in `list.h`, and whose implementation will go in `list.c`.

3 Instructions

1. Download the `lab3.tar.gz` file from the class website.
2. Extract the file using the command `tar xvzf lab3.tar.gz`.
3. This will extract the `lab3/` directory. Change into this directory using the `cd lab3/` command.
4. In this directory, there will be files `lab3.c`, `list.h`, and `list.c`.
5. There will also be a file `Makefile`, which is a build script which can be invoked by running the command `make` (without any arguments). It will automatically invoke the compiler and build the `lab3` executable.
6. Run the `lab3` executable, and see if your program works. The expected correct output is in a comment in the `lab3.c` file.

4 The Functions to Implement

4.1 Basic Exercises

The following functions should be relatively straightforward to implement. If you find yourself writing a lot of code for these functions, you should step back and rethink your approach.

- `int sum(List *list);`

  This function takes a linked list `list`, and returns the sum of all the elments of the list, taking the empty list to have a sum of 0.

- `void iterate(int (*f)(int), List *list);`

  The `iterate(f, list)` function takes two arguments. The first argument is a function pointer `f`, which takes an integer and returns an integer, and the second argument is a list `list`. This function then updates the head of each element of `list` by applying `f` to it.

- `void print_list(List *list);`

  `print(list)` takes a list `list` as an argument, and prints out the elements. Try to print out the elements as a comma-*separated* list.

4.2 Challenge Exercises

Once you have done the basic exercises, you can try the challenge exercises, which involve more subtle pointer manipulations. These two functions are the basic routines used to implement merge sort, which can sort a linked list in $O(n \log n)$ time.

- `List *merge(List *list1, List *list2);`

  Given two increasing lists `list1` and `list2` as arguments, `merge(list1, list2)` will return a linked list containing all of the elements of the two arguments in increasing order.

  In this implementation, do not allocate any new list cells – it should be possible to merge the two lists purely through pointer manipulations on the two underlying lists.

- `void split(List *list, List **list1, List **list2);`

  Given a list `list` as an argument, update the two pointers to linked lists `list1` and `list2` with linked lists each containing roughly half of the elements of `list` each. Eg, if `list` is [0,1,2,3,4,5,6], you might set `list1` to be [0,2,4,6] and `list2` to be [1,3,5].

  (HINTS: taking alternating elements for `list1` and `list2` will make your life easier. Also, think about what to do when `list` has 0 or 1 elements.)

  In this implementation, do not allocate any new list cells – it should be possible to split the input list into two purely through pointer manipulations of the input.

Notes for Programming in C Lab Session #4

September 29, 2019

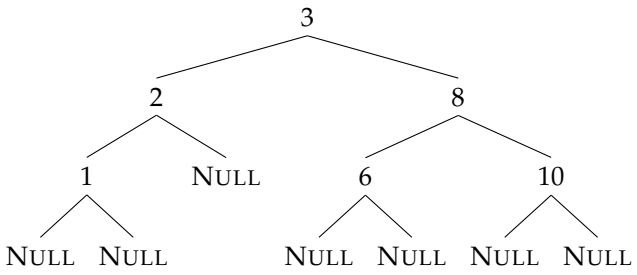
1 Introduction

The purpose of this lab session is to write some small programs that do pointer manipulations and dynamic memory allocation.

2 Overview

In this lab, you will define some functions to work with finite sets of integers, represented as (unbalanced) binary trees. Concretely, we will represent a set as a binary tree, in which each node contains an integer n, and the left subtree contains the elements which are smaller than n, and the right subtree contains the elements bigger than n.

For example, if we have the set $\{0, 1, 2, 3, 6, 8, 10\}$, we might represent it using the tree:



Note that each node of the tree contains an integer, and points to two subtrees. The subtrees can themselves be trees, or they can be the NULL value.

In C, a datatype for binary trees can be declared with the following structure declaration:

```
struct node {
  struct node *left;
  int value;
  struct node *right;
};
typedef struct node Tree;
```

This defines a type `struct node` which consists of a `left` field containing a pointer to the left subtree, and a `value` field containing an integer value, and a `right` field ointer to another `struct node`. (The typedef defines a type abbrevation `Tree` standing for the structure type `struct node`.)

A finite set is then just a pointer to this structure type. Next, you will implement a small library of functions whose prototypes and specifications are given in `list.h`, and whose implementation will go in `list.c`.

3 Instructions

1. Download the `lab4.tar.gz` file from the class website.
2. Extract the file using the command `tar xvzf lab4.tar.gz`.
3. This will extract the `lab4/` directory. Change into this directory using the `cd lab4/` command.
4. In this directory, there will be files `lab4.c`, `tree.h`, and `tree.c`.
5. There will also be a file `Makefile`, which is a build script which can be invoked by running the command `make` (without any arguments). It will automatically invoke the compiler and build the `lab4` executable.
6. Run the `lab4` executable, and see if your program works. The expected correct output is in a comment in the `lab4.c` file.

4 The Functions to Implement

4.1 Basic Exercises

The following functions should be relatively straightforward to implement. If you find yourself writing a lot of code for these functions, you should step back and rethink your approach.

- `int tree_member(int x, Tree *tree);`

  This function takes an integer `x` and a tree `tree`, and returns 0 if `x` does not occur in `tree`, and 1 if it does occur. This function should not allocate or deallocate any memory at all.

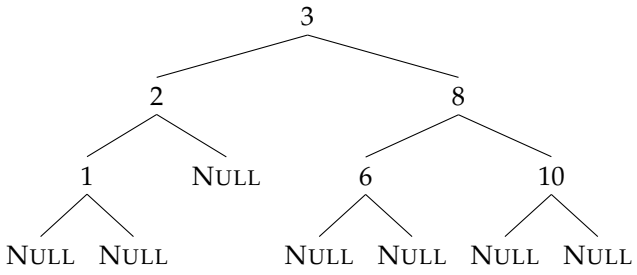
- `void tree_free(Tree *tree);`

  Given a tree `tree` as an argument, this function should free all of the memory associated with the tree. This function should recursively call `free` on each reachable node.

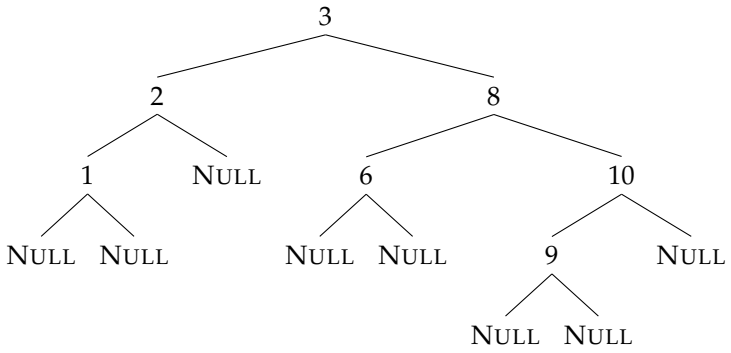
- `Tree *tree_insert(int x, Tree *tree);`

  This function should insert `x` into the tree `tree` if it is not present, and do nothing otherwise.

  As an example, inserting 9 into the following tree:



  should result in an updated tree:



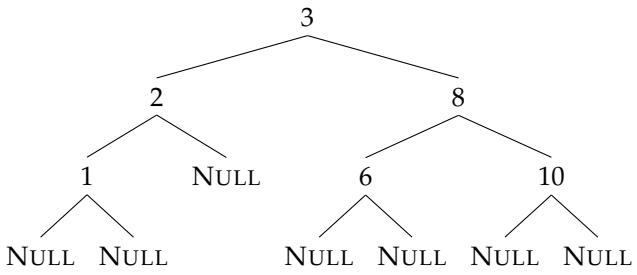
4.2 Challenge Exercises

Once you have done the basic exercises, you can try the challenge problem of *removing* an element from a set, which involves more complex pointer manipulations and pattern of memory deallocations.

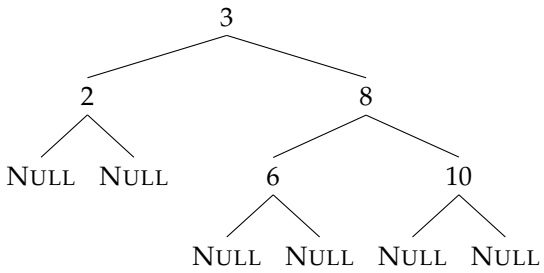
- `void pop_minimum(Tree *tree, int *min, Tree **new_tree);`

  This function should take a nonempty tree `tree` as its first argument, and then it should (a) return the minimum value held in the tree in the contents of `min`, and (b) modify `tree` so that it no longer contains `min`, returning an updated pointer in `new_tree`.

  As an example, calling `pop_minimum` on the following tree



  should return the value 1 in `min`, and modify the tree so it has the shape



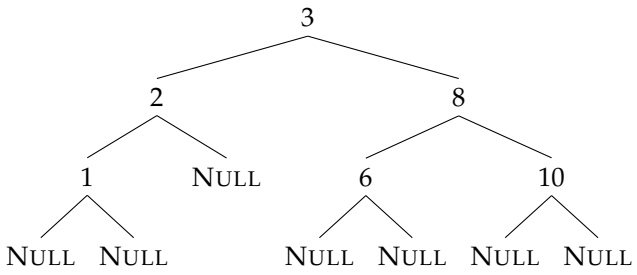
- `Tree *tree_remove(int x, Tree *tree);`

  This function should remove `x` from the tree `tree` if it is present, and do nothing otherwise.

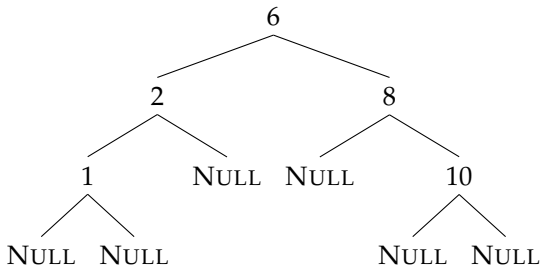
  Hints:

  1. The difficult case is when you have reached the node which needs to be removed.

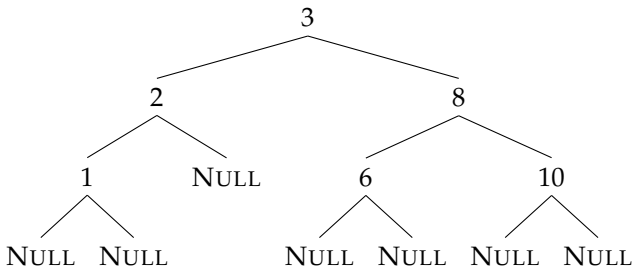
2. It will be very helpful to use `pop_minimum` as a subroutine – but remember you can only call it on nonempty trees!
3. If you remove 3 from the tree



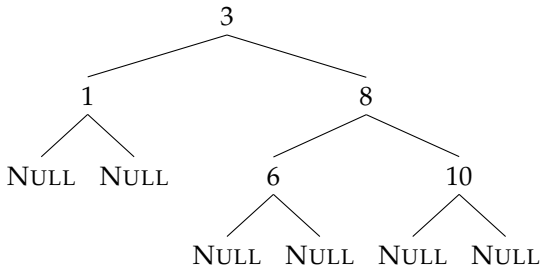
you should get



4. If you remove 2 from the tree



you should get



Notes for Programming in C Lab Session #5

September 29, 2019

1 Introduction

The purpose of this lab session is to practice debugging an existing program, using the ASan and UBSan tools built in to GCC and clang.

2 Overview

Once built, the `lab5` executable will read the arithmetic expressions passed to it as a command-line argument. Then, the program will parse the argument into a parse tree, evaluate the parsed expression, and then print both the parse tree and the result of the square of the evaluation.

```
$ ./lab5 "3"
3 squared = 9

$ ./lab5 "3+4"
(3+4) squared = 49

$ ./lab5 "3+4*5"
(3+(4*5)) squared = 529

$ ./lab5 "3+4*5+2"
((3+(4*5))+2) squared = 625

$ ./lab5 "3+4*(5+2)"
(3+(4*(5+2))) squared = 961

$ ./lab5 "(3+4)*(5+2)"
((3+4)*(5+2)) squared = 2401
```

The terms of the syntax this calculator accepts are:

- positive integer literals, such as `12` or `3124`.
- The sum of two terms, such as `2+3` or `(2*3)+4`.
- The product of two terms, such as `2*3` or `1*2*3`.
- A parenthesized term, such as `(1)` or `(2*3)` or `(((2*3+1)))`.
- Addition and multiplication associate to the left – i.e., `1+2+3` is the same as `(1+2)+3`.
- Addition is lower precedence than multiplication – i.e., `1+2*3` is the same as `1+(2*3)`.

For simplicity, no whitespace is permitted in arithmetic expressions, and neither is subtraction:

```
$ ./lab5 "1 + 2"
parse error

$ ./lab5 "1-2"
parse error
```

However, this is the theory! This program has been carefully salted with a few bugs, and it will crash on most inputs. Your task is to find and fix the bugs in this program. Hopefully, the use of UBSan and ASan will make it much easier to find these bugs than before!

3 Instructions

1. Download the `lab5.tar.gz` file from the class website.
2. Extract the file using the command `tar xvzf lab5.tar.gz`.
3. This will extract the `lab5/` directory. Change into this directory using the `cd lab5/` command.
4. In this directory, there will be files `lab5.c`, `expr.h`, `expr.c`, `parse.h`, and `parse.c`.
5. There will also be a file `Makefile`, which is a build script which can be invoked by running the command `make` (without any arguments). It will automatically invoke the compiler and build the `lab5` executable.
6. Once built, this file accepts command-line arguments to evaluate arithmetic expressions and square them.
7. To invoke the compiler with debugging instrumentation turned on, invoke make with the command `make sane`, which will turn on the address sanitizer and undefined behaviour sanitizers.

4 Documentation of the Types and Functions

4.1 The `expr.h` module

- The expression data type:

```
typedef enum type {LIT, PLUS, TIMES} expr_type;
typedef struct expr * expr_t;
struct expr {
  expr_type type;
  union {
    int literal;
    struct pair {
      expr_t fst;
      expr_t snd;
    } args;
  } data;
};
```

The `expr_t` type represents syntax trees of arithmetic expressions. It is a pointer to a struct, whose `type` field is an enumeration saying whether this expression is a literal `LIT`, an addition node `PLUS`, or a multiplication node `TIMES`. If the `type` field is `LIT`, the `data` field will be the `literal` branch

of the union, storing the literal integer this node represents. If `type` field is `PLUS` or `TIMES`, the `data` field will be in the `pair` branch of the union, with the `fst` and `snd` representing the left- and right-hand sides of the arithmetic operation.

- `expr_t mkLit(int n);`

  Construct a fresh `expr_t` representing the literal `n`.

- `expr_t mkPlus(expr_t e1, expr_t e2);`

  Construct a fresh `expr_t` representing the sum of `e1` and `e2`.

- `expr_t mkTimes(expr_t e1, expr_t e2);`

  Construct a fresh `expr_t` representing the product of `e1` and `e2`.

- `int eval_expr(expr_t e);`

  Return the integer which is the result of evaluating the expression `e`.

- `void print_expr(expr_t e);`

  Print the expression `e` to standard output.

- `void free_expr(expr_t e);`

  Free the memory associated with the expression `e`.

- `expr_t copy(expr_t e);`

  Construct a fresh copy of the expression tree `e`.

4.2 The `parse.h` module

- `int parse_int(char *s, int i, expr_t *result);`

  Parse an integer expression from the string `s`, beginning at position `i`. If the parse is successful, this function returns an integer index to the first character after the matched string, and writes the parse tree to the `result` pointer.

- `int parse_atom(char *s, int i, expr_t *result);`

  Parse an *atom* (i.e., either an integer or parenthesized expression) from the string `s`, beginning at position `i`. If the parse is successful, this function returns an integer index to the first character after the matched string, and writes the parse tree to the `result` pointer.

- `int parse_term(char *s, int i, expr_t *result);`

  Parse a term (i.e., a product of atoms, such as `1 * (2+3) * 4`) from the string `s`, beginning at position `i`. If the parse is successful, this function returns an integer index to the first character after the matched string, and writes the parse tree to the `result` pointer.

- `int parse_expr(char *s, int i, expr_t *result);`

  Parse an expression (i.e., a sum of terms, such as `1 + 2*3 + (4*(5+6))`) of multiplied expressions from the string `s`, beginning at position `i`. If the parse is successful, this function returns an integer index to the first character after the matched string, and writes the parse tree to the `result` pointer.

- `int parse(char *s, int i, expr_t *result);`

  Parse an expression as with `parse_expr`, but return NULL if the parse doesn't consume the whole string.

Notes for Programming in C Lab Session #6

September 29, 2019

1 Introduction

The purpose of this lab session is to write some small programs that do a slightly more intense set of pointer manipulations and dynamic memory management than in the previous labs.

2 Overview

In this lab, you will define some functions to manipulate graphs. The graph programs themselves will not, alas, do much, but do illustrate many of the techniques you will need to when writing more interesting C programs.

As in lecture 6, the key data type is the *marked node*, which is a structure with a `value` field, possibly-null `left` and `right` subtree fields, and a boolean flag `marked`.

```
struct node {
  bool marked;
  int value;
  struct node *left;
  struct node *right;
};
typedef struct node Node;
```

A pointers to a `Node` can be used to represent arbitrary graphs in memory.[1]

As we saw in lecture, it is often useful to keep track of whether or not a node has been visited or not by updating the `marked` flag. In the instructions below, an "unmarked node" is (a) a non-null node whose `marked` field is false, and (b) for which every non-null node reachable from that node is also has a false `marked` field.

Conversely, a "marked node" is taken to mean a non-null node whose `marked` field is true, and (b) for which every non-null node reachable from that node is also has a true `marked` field.

3 Instructions

1. Download the `lab6.tar.gz` file from the class website.
2. Extract the file using the command `tar xvzf lab6.tar.gz`.
3. This will extract the `lab6/` directory. Change into this directory using the `cd lab6/` command.

[1] Technically, these nodes can represent graphs with a maximum branching factor of 2. More general graphs can be represented by replacing the left and right fields with an array of node pointers. However, this does not add any essential difficulty, so we won't consider it in this lab.

4. In this directory, there will be files `lab6.c`, `tree.h`, and `tree.c`.
5. There will also be a file `Makefile`, which is a build script which can be invoked by running the command `make` (without any arguments). It will automatically invoke the compiler and build the `lab6` executable.
6. You can (and should!) invoke `make sane` to build with the address and undefined behaviour sanitizers.
7. Run the `lab6` executable, and see if your program works. The expected correct output is in a comment in the `lab6.c` file.

4 The Functions to Implement

4.1 Basic problems

- `int size(Node *node);`

  Given a pointer to an unmarked node `node`, this function returns the total number of distinct, non-null nodes reachable from `node`, including itself.

  If passed a null pointer, it returns 0. It also marks all of the nodes reachable from `node`.

- `void unmark(Node *node);`

  Given a marked node `node`, this function sets the `marked` field of `node` and every node reachable from it to false.

- `bool path_from(Node *node1, Node *node2);`

  Given two nodes `node1` and `node2`, this function returns true if there is a path (via the `left` and `right` fields) of length 0 or more from `node1` to `node2`.

  If either `node1` or `node2` is NULL, then this function returns false.

- `bool cyclic(Node *node);`

  This function returns true if there is a path of length 1 or more from `node` to itself, and false otherwise.

4.2 Challenge problems

In lecture, we freed the memory associated with a graph by dynamically allocating a list storing all of the reachable nodes. In this lab exercise, you have implemented the `size` function, which tells you the number of reachable nodes.

This means it should be possible to deallocate a graph without using an array, rather than a linked list.

- `void get_nodes(Node *node, Node **dest);`

  This function receives a node pointer `node`, and a pointer into a buffer of node pointers `dest`, as arguments. The `get_node` function should then update the buffer with all of the unmarked nodes reachable from `node` via paths that only go through unmarked nodes.

- `void graph_free(Node *node);`

  This function should free a graph. It should find the nodes to deallocate by declaring an automatic array of the right size and passing a pointer into this array to `get_nodes`.

  Your implementation of `graph_free` should not be recursive, and should not allocate any memory beyond the stack allocation of the buffer storing the reachable nodes.

Notes for Programming in C Lab Session #7

September 29, 2019

1 Introduction

The purpose of this lab session is to write a small program that makes use of arena-style allocation.

2 Overview

In this lab, you will define some functions to do match strings against a subset of regular expressions. In the `re.h` header file, we define the following data type

```
enum re_tag { CHR, SEQ, ALT };
typedef struct re Regexp;
struct re {
  enum re_tag type;
  union data {
    struct { char c; } chr;
    struct { Regexp *fst; Regexp *snd; } pair;
  } data;
};
```

This is a data type for representing trees. We define an enumeration `enum re_tag`, which says that we have 3 possibilities, either a single-character `CHR`, an alternative `ALT`, and a sequential composition `SEQ`.

Next, we define a structure type `Regexp`, with two fields. The first is the `type` field, which is one of the tags from the enumeration above. The second is a union type `data`, which is either a character `chr`, or a structure containing a pair of pointers to two regular expressions. The `type` fields determines which case of the union to use – a valid `Regexp` structure has a character if the type field is `CHR`, and a pair if the type field is `SEQ` or `ALT`.

The idea is that a CHR-regexp matches a single character string, the SEQ-regexp matches if the first part of the string matches the first element of the pair and the second part of the string, and the ALT-regexp matches if the string matches either the first or the second element of the pair.

Below, I give a table of example regexps, strings and whether or not there is a match. Here, `x`, `a`, `b`, `c`, `u`, and `v` are characters, juxtaposition represents SEQuential composition, and (+) denotes ALTernative.

Regexp	`"ab"`	`"xab"`	`"xba"`	`"axu"`
x(ab+ba)	✗	✓	✓	✗
(a+b+c)(x+y)(u+v+w)	✗	✗	✗	✓

3 Instructions

1. Download the `lab7.tar.gz` file from the class website.

2. Extract the file using the command `tar xvzf lab7.tar.gz`.
3. This will extract the `lab7/` directory. Change into this directory using the `cd lab7/` command.
4. In this directory, there will be files `lab7.c`, `re.h`, and `re.c`.
5. There will also be a file `Makefile`, which is a build script which can be invoked by running the command `make` (without any arguments). It will automatically invoke the compiler and build the `lab7` executable.
6. As usual, a sanitized version of the executable can be built with `make sane`.
7. Run the `lab7` executable, and see if your program works. The expected correct output is in a comment in the `lab7.c` file.

4 The Types and Functions to Implement

- **`struct arena`**

  The `re.h` file contains a declaration of the arena structure, but does not define it. Define an arena type for allocating pointers to `Regexp` structures, following the pattern of lecture 6.

- `Regexp *re_alloc(arena_t a);`

  Given an arena `a`, the `re_alloc` function should allocate a new `Regexp` and return a pointer to it. If the arena lacks room to allocate a new `Regexp`, it should return `NULL`.

- **`void`** `arena_free(arena_t a);`

  Given an arena `a`, the `arena_free` function should deallocate the arena and its associated storage. This should be a simple, non-recursive function!

- `Regexp *re_chr(arena_t a,` **`char`** `c);`

  Allocate a regexp matching the character `c`, allocating from the arena `a`. Return NULL if no memory is available.

- `Regexp *re_alt(arena_t a, Regexp *r1, Regexp *r2);`

  Allocate a regexp representing the alternative of `r1` and `r2` from the arena `a`. Return NULL if no memory is available.

- `Regexp *re_seq(arena_t a, Regexp *r1, Regexp *r2);`

  Allocate a regexp representing the sequencing of `r1` and `r2` from the arena `a`. Return NULL if no memory is available.

- **`int`** `re_match(Regexp *r,` **`char`** `*s,` **`int`** `i);`

  Given a regular expression `r`, a string `s`, and a valid index into the string `i`, this function should return an integer. If the function returns a nonnegative `j`, then the regular expression should match the substring running from i to j, including i but not j. (So if the `re_match(r, s, 5)` returns 8, then the subrange `s[5], s[6], s[7]` should match the regexp `r`.)

  It may help to look at the `re_print` function to see how to switch between the alternative branches of the regexp type.

Notes for Programming in C Lab Session #8

September 29, 2019

1 Introduction

The purpose of this lab session is to write matrix manipulation code to see how different memory access patterns can affect performance.

2 Overview

A *matrix* is a rectangular array of numbers, and also one of the fundamental concepts of mathematics. Matrices can represent linear transformations between vector spaces, extensive-form games in game theory, graph connectivity in graph theory, the systems of differential equations arising in control theory, just to list a few applications. As a result, high-performance implementations of matrices and operations on them are of great importance to a wide variety of scientific and engineering domains.

In this lab, we will work use the following datatype for matrices:

```
typedef struct matrix matrix_t;
struct matrix {
  int rows;
  int cols;
  double *elts;
};
```

Here, a matrix is represented by a structure containing a number of rows, a number of columns, and an array of doubles `elts` containing the elements of the array. As programmers, we immediately face a choice in how to represent arrays. An array is a two-dimensional object like:

$$A \equiv \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$$

However, a C array is *one-dimensional*. So we have to decide how to place the 12 elements of the 4×3 matrix A in memory. In C, it is typical to represent arrays in *row-major order*. This means that the `elts` array will have the following shape:

`elts` $\mapsto$

1	2	3	4	5	6	7	8	9	10	11	12

So the `elts` array stores the rows of A one after another in memory.[1]

As a result, if we have a matrix B of size $n \times m$, and we want to find $B(i, j)$ – the j-th column of the i-th row will be the $(n \times i) + j$-th element of the array.

[1] The choice of row-major order is purely conventional; historically Fortran has made the opposite choice!

One of the most important matrix operations is *matrix multiplication*. Given an $n \times m$ matrix A, and an $m \times o$ matrix B, we define the following $n \times o$ matrix $A \times B$ as the product:

$$(A \times B)(i,j) = \sum_{k \in \{0 \ldots n\}} A(i,k) \times B(k,j)$$

In the calculation of $A(i,j)$, we will touch the following entries:

$$\begin{pmatrix} A_{(0,0)} & \cdots & \cdots & A_{(0,m-1)} \\ \vdots & & & \vdots \\ A_{(i,0)} & \cdots & \cdots & A_{(i,m-1)} \\ \vdots & & & \vdots \\ A_{(n-1,0)} & \cdots & \cdots & A_{(n-1,m-1)} \end{pmatrix} \times \begin{pmatrix} B_{(0,0)} & \cdots & B_{(0,j)} & \cdots & B_{(0,o-1)} \\ \vdots & & & & \vdots \\ \vdots & & & & \vdots \\ B_{(m-1,0)} & \cdots & B_{(m-1,j)} & \cdots & B_{(m-1,o-1)} \end{pmatrix}$$

Note that we are accessing the elements of $A_{(i,k)}$ in a row-wise order, but accessing the elements of $B_{(k,j)}$ in a column-wise order. As a result, we risk a *cache miss* on each access to B!

However, if B were *transposed* – i.e., if rows and columns were interchanged – then we would be accessing the elements of B in a row-wise order as well. In equational form, we can make the following observation (writing B^T for the transpose of B):

$$\begin{aligned} (A \times B^T)(i,j) &= \sum_{k \in \{0 \ldots n\}} A(i,k) \times B^T(k,j) \\ &= \sum_{k \in \{0 \ldots n\}} A(i,k) \times B(j,k) \end{aligned}$$

By making use of the observation that $B^T(k,j) = B(j,k)$, we can replace a column-wise traversal with a row-wise traversal.

So in this exercise, you will implement naive multiplication, transpose, and transposed multiplication, and compare the performance of naive multiplication to building a transpose and then doing a transposed multiplication.

3 Instructions

1. Download the `lab8.tar.gz` file from the class website.
2. Extract the file using the command `tar xvzf lab8.tar.gz`.
3. This will extract the `lab8/` directory. Change into this directory using the `cd lab8/` command.
4. In this directory, there will be files `lab8.c`, `matrix.h`, and `matrix.c`.
5. There will also be a file `Makefile`, which is a build script which can be invoked by running the command `make` (without any arguments). It will automatically invoke the compiler and build the `lab8` executable.
6. There is a test routine to check if you have implemented matrix multiplication probably works, together with expected correct output in the `lab8.c` file.
7. Once it works, run the timing functions on your two matrix multiplication routines to see which one is faster.

4 The Types and Functions to Implement

- `matrix_t matrix_create(int rows, int cols);`

  Given integer arguments `rows` and `cols`, return a new matrix of size `rows` $\times$ `cols`. Initializing the elements of the array is optional, but may help you debug.

- `void matrix_free(matrix_t m);`

  Deallocate the storage associated with the matrix `m`.

- `void matrix_print(matrix_t m);`

  You don't have to implement this – it comes for free to help you test your code.

- `double matrix_get(matrix_t m, int r, int c);`

  Return the value in the r-th row and c-th column of m.

- `void matrix_set(matrix_t m, int r, int c, double d);`

  Modify the value in the r-th row and c-th column of m to d.

- `matrix_t matrix_multiply(matrix_t m1, matrix_t m2);`

  Given an $n \times m$ matrix `m1` and an $m \times k$ matrix `m2`, return the $n \times k$ matrix that is the matrix product of `m1` and `m2`.

  You should be able to implement this with a simple triply-nested for-loop.

- `matrix_t matrix_transpose(matrix_t m);`

  Given an $n \times m$ matrix `m` as an argument, return the $m \times n$ transposed matrix. (That is, if A is the argument and B is the return value, then $A(i, j) = B(j, i)$.)

- `matrix_t matrix_multiply_transposed(matrix_t m1, matrix_t m2);`

  Given an $n \times m$ matrix `m1` and an $k \times m$ matrix `m2`, return the $n \times k$ matrix that corresponds to `m1` times the transpose of `m2`.

- `matrix_t matrix_multiply_fast(matrix_t m1, matrix_t m2);`

  This function should also implement matrix multiplication, but do it by constructing the transpose of `m2`, and then passing that to `matrix_multiply_fast`. Don't forget to free the transposed matrix when you are done!

Notes for Programming in C Lab Session #9

September 29, 2019

1 Introduction

The purpose of this lab session is to practice debugging an existing program.

2 Overview

The program in this lab file is an implementation of a command-line calculator program. In fact, it is the same calculator as in lab 5, only with a new and different set of bugs.

Once built, the `lab9` executable will read the arithmetic expressions passed to it as a command-line argument. Then, the program will parse the argument into a parse tree, evaluate the parsed expression, and then print both the parse tree and the result of the evaluation.

```
$ ./lab9 "3"
3 = 3

$ ./lab9 "3+4"
(3+4) = 7

$ ./lab9 "12+13"
(12+13) = 25

$ ./lab9 "3+4*5"
(3+(4*5)) = 23

$ ./lab9 "3+4*5+2"
((3+(4*5))+2) = 25

$ ./lab9 "3+4*5+2"
((3+(4*5))+2) = 25

$ ./lab9 "3+4*(5+2)"
(3+(4*(5+2))) = 31

$ ./lab9 "(3+4)*(5+2)"
((3+4)*(5+2)) = 49
```

The terms of the syntax this calculator accepts are:

- positive integer literals, such as `12` or `3124`.
- The sum of two terms, such as `2+3` or `(2*3)+4`.

- The product of two terms, such as `2*3` or `1*2*3`.
- A parenthesized term, such as `(1)` or `(2*3)` or `(((2*3+1)))`.
- Addition and multiplication associate to the left – i.e., `1+2+3` is the same as `(1+2)+3`.
- Addition is lower precedence than multiplication – i.e., `1+2*3` is the same as `1+(2*3)`.

For simplicity, no whitespace is permitted in arithmetic expressions, and neither is subtraction:

```
$ ./lab9 "1 + 2"
parse error

$ ./lab9 "1-2"
parse error
```

However, this is the theory! This program has been carefully salted with bugs, and it will crash on most inputs. Your task is to find and fix the bugs in this program.

3 Instructions

1. Download the `lab9.tar.gz` file from the class website.
2. Extract the file using the command `tar xvzf lab9.tar.gz`.
3. This will extract the `lab9/` directory. Change into this directory using the `cd lab9/` command.
4. In this directory, there will be files `lab9.c`, `expr.h`, `expr.c`, `parse.h`, and `parse.c`.
5. There will also be a file `Makefile`, which is a build script which can be invoked by running the command `make` (without any arguments). It will automatically invoke the compiler and build the `lab9` executable.
6. Once built, this file accepts command-line arguments to evaluate arithmetic expressions.
7. Find and fix the bugs!

4 Documentation of the Types and Functions

4.1 The `expr.h` module

- The expression data type:

```
typedef enum type {LIT, PLUS, TIMES} expr_type;
typedef struct expr * expr_t;
struct expr {
  expr_type type;
  union {
    int literal;
    struct pair {
      expr_t fst;
      expr_t snd;
    } args;
  } data;
};
```

The `expr_t` type represents syntax trees of arithmetic expressions. It is a pointer to a struct, whose `type` field is an enumeration saying whether this expression is a literal `LIT`, an addition node `PLUS`, or a multiplication node `TIMES`. If the `type` field is `LIT`, the `data` field will be the `literal` branch of the union, storing the literal integer this node represents. If `type` field is `PLUS` or `TIMES`, the `data` field will be in the `pair` branch of the union, with the `fst` and `snd` representing the left- and right-hand sides of the arithmetic operation.

- `expr_t mkLit(int n);`

  Construct a fresh `expr_t` representing the literal `n`.

- `expr_t mkPlus(expr_t e1, expr_t e2);`

  Construct a fresh `expr_t` representing the sum of `e1` and `e2`.

- `expr_t mkTimes(expr_t e1, expr_t e2);`

  Construct a fresh `expr_t` representing the product of `e1` and `e2`.

- `int eval_expr(expr_t e);`

  Return the integer which is the result of evaluating the expression `e`.

- `void print_expr(expr_t e);`

  Print the expression `e` to standard output.

- `void free_expr(expr_t e);`

  Free the memory associated with the expression `e`.

4.2 The `parse.h` module

- `int parse_int(char *s, int i, expr_t *result);`

  Parse an integer expression from the string `s`, beginning at position `i`. If the parse is successful, this function returns an integer index to the first character after the matched string, and writes the parse tree to the `result` pointer.

- `int parse_atom(char *s, int i, expr_t *result);`

  Parse an *atom* (i.e., either an integer or parenthesized expression) from the string `s`, beginning at position `i`. If the parse is successful, this function returns an integer index to the first character after the matched string, and writes the parse tree to the `result` pointer.

- `int parse_term(char *s, int i, expr_t *result);`

  Parse a term (i.e., a product of atoms, such as `1 * (2+3) * 4`) from the string `s`, beginning at position `i`. If the parse is successful, this function returns an integer index to the first character after the matched string, and writes the parse tree to the `result` pointer.

- `int parse_expr(char *s, int i, expr_t *result);`

  Parse an expression (i.e., a sum of terms, such as `1 + 2*3 + (4*(5+6))`) of multiplied expressions from the string `s`, beginning at position `i`. If the parse is successful, this function returns an integer index to the first character after the matched string, and writes the parse tree to the `result` pointer.

- `int parse(char *s, int i, expr_t *result);`

  Parse an expression as with `parse_expr`, but return NULL if the parse doesn't consume the whole string.