15-122: Principles of Imperative Computation, Fall 2010
Assignment 3: Stacks, Queues, and Languages

William Lovas (wlovas@cs) Tom Cortina (tcortina@cs)

Out: Thursday, September 16, 2010
Due (Written): Thursday, September 23, 2010 (before lecture)
Due (Programming): Friday, September 24, 2010 (11:59 pm)

1 Written: (25 points)

The written portion of this week's homework will give you some practice working with stacks and queues and amortized analysis. You can either type up your solutions or write them *neatly* by hand, and you should submit your work in class on the due date just before lecture begins. Please remember to *staple* your written homework before submission.

1.1 Amortized Analysis

Exercise 1 (4 pts)**.** In class, you saw that for a k-bit counter with only an increment operation, the amortized cost of n increments is $O(n)$.

(a) Suppose you have a k-bit counter and you have two operations: increment and decrement. If you execute any sequence of n of these operations on your counter, show that the amortized cost is $O(nk)$.

(b) Suppose you have a k-bit counter and you have two operations: increment and reset (to 0). If you execute any sequence of n of these operations on an initially zero counter, show that the amortized cost is $O(n)$.

Exercise 2 (3 pts)**.** Show that the total number of bit flips required to increment an initially zero counter with k bits n times is less than $2n$. (**Hint:** Determine how many times each bit flips based on its bit position and add these values together.)

Exercise 3 (3 pts)**.** Suppose we define the operation `multi_enq` which adds one or more elements to a queue. If we execute any sequence of n operations from { `enq`, `deq`, `multi_enq` }, can we achieve $O(n)$ amortized cost? Why or why not?

1.2 Stacks and Queues

Exercise 4 (3 pts)**.** A `queue` and a `stack` of type `elem` are implemented with the following interfaces:

```
queue q_new();
bool q_empty(queue Q);
void enq(elem k, queue Q);
elem deq(queue Q);

stack s_new();
bool s_empty(stack S);
void push(elem k, stack S);
elem pop(stack S);
```

We do not know how these data structures are implemented. That is, we don't know if the programmer used linked lists, unbounded arrays, or something else—just that they somehow implemented the functions shown above, and that they have the same observable behavior as the ones we implemented in class.

Write a function `reverse` that takes a queue as its argument and reverses the order of the elements in the queue using a stack.

Exercise 5 (4 pts)**.** Consider the implementation of queues given in class. (See the course website for the code in C_0.)

(a) Write a function `size` that returns the number of elements stored in the queue without changing the queue `struct`. Include proper annotations in your solution. What is the run-time complexity of this function if there are n elements in the queue? Explain.

(b) Show how to change the queue `struct` so you can compute the size of the queue in constant time. What other functions need to change? Modify these functions and their annotations as necessary.

Exercise 6 (8 pts)**.** Consider the implementation of stacks given in class. (See the course website for the code in C_0.) Revise this implementation so that it includes another function `get_max` that returns the maximum stored in the stack, assuming it is not empty. **All stack operations should run in constant time**. (**Hint:** You will need to modify the original implementation in addition to adding the new function.)

For partial credit (4 points), write `get_max` so it runs in linear time. All other functions should still run in constant time.

2 Programming: Grammars (25 points)

For the programming portion of this week's homework, you'll learn a little about grammars and use a stack to check the validity of strings based on several grammars. Put your code for this assignment in a file called `parsing.c0`.

You should submit your code electronically by 11:59 pm on the due date. Detailed submission instructions can be found below.

Starter code. Download the file `hw3-starter.zip` from the course website. Inside, you'll find five files:

`elem.c0`	A type alias defining `elem` to be `char`
`lists.c0`	Linked lists containing `elem`s as developed in lecture
`stacks.c0`	Stacks containing `elem`s as developed in lecture
`readfile.c0`	Code for reading words from a file
`grammar.c0`	Grammars and code for reading a grammar from a file

For this homework, you are not provided any `main()` functions. Instead, you will write your own `main()` functions for testing your code, which you'll submit for credit. You should put this test code in files other than `parsing.c0`, but besides that, you may use any filenames you like.

Compiling and running. For this homework, use the `cc0` command as usual to compile your code. As before, you will need to use the libraries for file I/O (the `-lfile` switch), console I/O (the `-lconio` switch), and strings (the `-lstring` switch), in addition to the support code described above. When compiling multiple files, be sure to list them in dependency order—the starter code is shown in dependency order above. Don't forget to test your annotations by compiling with the `-d` switch to enable dynamic checking.

Submitting. Once you've completed some files, you can submit them by running the command

```
handin -a hw3 <file1>.c0 ... <fileN>.c0
```

The `handin` utility accepts a number of other switches you may find useful as well; try `handin -h` for more information.

You can submit files as many times as you like and in any order. When we grade your assignment, we will consider the most recent version of each file submitted before the due date. If you get any errors while trying to submit your code, you should contact the course staff immediately.

Annotations. Be sure to include appropriate `//@requires`, `//@ensures`, `//@assert`, and `//@loop_invariant` annotations in your program. You should write these as you are writing the code rather than after you're done: documenting your code as you go along will help you reason about what it should be doing, and thus help you write code that is both clearer and more correct. **Annotations are part of your score for the programming problems; you will not receive maximum credit if your annotations are weak or missing.**

Style. Strive to write code with *good style*: indent every line of a block to the same level, use descriptive variable names, keep lines to 80 characters or fewer, document your code with comments, etc. We will read your code when we grade it, and good style is sure to earn our good graces. Feel free to ask on the course bboard (`academic.cs.15-122`) if you're unsure of what constitutes good style.

2.1 Overview

A formal language is a set of strings that can be described by a grammar. Grammars are used to define the syntax of programming languages and used by compilers to parse code into a form suitable for translation to binary code. For example, the following production rule from the grammar in the C_0 reference describes the syntax of the `if` statement:

```
<stmt> ::= if ( <exp> ) <stmt> [ else <stmt> ]
```

This says that an `if` statement is a kind of statement that consists of the keyword `if` followed by an expression in parentheses, followed by another statement, and optionally followed by the keyword `else` and one more statement.

For this project, a grammar consists of a set of replacement rules such that each rule has a *non-terminal symbol* on the left side of the rule and a sequence of non-terminal and *terminal symbols* on the right side of the rule. Strings are *generated* by a grammar by starting from a distinguished non-terminal *start symbol* `S` and successively replacing non-terminals using the replacement rules until only terminal symbols remain. Terminals are represented by lowercase letters, and are fixed symbols that do not have replacement rules, whereas non-terminals are represented by uppercase letters and can be replaced by rules. In this problem, the right-hand side of every rule must start with a terminal symbol.

For example, consider the language described by the following grammar:

```
S --> xSz
S --> y
```

This grammar has two rules. `S` is a non-terminal that can either be replaced with `xSz` or `y`, where `x`, `y`, and `z` are terminals. Let's say we want to validate the string `xxyzz` using the grammar above. We start with the start symbol `S`, and then use the first

rule to replace S with xSz. We then replace the S in xSz with xSz using the first rule again to get xxSzz. Finally, we replace the S in xxSzz with y using the second rule to get xxyzz. We can visualize this as follows:

```
S --> xSz --> xxSzz --> xxyzz
```

Thus, xxyzz is a valid string according to the grammar, since it can be derived from the start symbol S by three rule applications. In general, the grammar represents all valid strings that can be derived from the given rules starting with the symbol S. Given a grammar and a string of terminals, we wish to determine if the string is valid according to the grammar. We will use an algorithm called LL(1) parsing to solve this problem, an algorithm which requires the use of a stack.

First, we push the start symbol S onto the stack. Then we iterate over the string, repeating the following steps until the string is declared valid or invalid:

1. If the top of the stack contains a non-terminal, pop it and push the elements of the right-hand side of a matching rule. A matching rule is a rule whose left-hand side matches the non-terminal and whose right-hand side starts with the same terminal as the next character of the input string. If there is no matching rule, the string is invalid. **Note:** The current character of the string is not passed by in this step.

2. If the top of the stack contains a terminal and it matches the current character in the string, pop the terminal from the stack and advance to the next character in the string. If the two characters do not match, the string is declared invalid.

3. If the stack is empty and the string is completely processed, declare the string valid. Otherwise, the string is invalid.

Example 1. Let's execute the algorithm using the grammar above with the string xxyzz. We start with a stack that contains the nonterminal S. The ^ under the string shows the current character to be processed.

```
Stack:  S     String:  xxyzz
                       ^
```

Using step 1, we see S on top of the stack and the current character in the string is x, so we replace the S on the stack with xSz using the first rule S --> xSz.

```
Stack:  x     String:  xxyzz
        S              ^
        z
```

Note that xSz is pushed onto the stack so that x is on top, not z. Also note that the current character in the string is still the first occurrence of x. Using step 2, the top of stack matches the current character in the string, so we pop the stack and advance to the next character in the string:

```
Stack:  S     String:  xxyzz
        z               ^
```

We see that the top of the stack is `S` (a non-terminal) and the current character in the string is `x`, so we use step 1 to replace `S` with `xSz`:

```
Stack:  x     String:  xxyzz
        S               ^
        z
        z
```

The top of the stack matches the current character in the string, so we pop the stack and advance to the next character in the string using step 2:

```
Stack:  S     String:  xxyzz
        z                ^
        z
```

We see that the top of the stack is `S` (a non-terminal) and the current character is `y`, so we use step 1 to replace `S` with `y` using the second rule `S --> y`.

```
Stack:  y     String:  xxyzz
        z                ^
        z
```

Now, step 2 is used, since the top of the stack matches the current character in the string:

```
Stack:  z     String:  xxyzz
        z                 ^
```

Step 2 is used again, since the top of the stack matches the current character in the string:

```
Stack:  z     String:  xxyzz
                          ^
```

Step 2 is once more, since the top of the stack matches the current character in the string:

```
Stack:  empty     String:  xxyzz
                                ^
```

At this point, the stack is empty and the string is completely processed. Therefore, according to step 3, the string `xxyzz` is valid according to this grammar.

Example 2. If we tried to process the string `xyx` with this algorithm, we would find it is invalid:

```
Stack:  S     String:  xyx
                       ^

Stack:  x     String:  xyx          (using step 1)
        S              ^
        z

Stack:  S     String:  xyx          (using step 2)
        z               ^

Stack:  y     String:  xyx          (using step 1)
        z               ^

Stack:  z     String:  xyx          (using step 2)
                         ^

Stack:  z     String:  xyx          (INVALID using step 2,
                         ^           x and z do not match)
```

2.2 Auxiliary and Specification Functions

You will start this assignment by writing some specification functions that you can use later in your annotations.

We will represent a rule as a `struct` with a left-hand side (`lhs`) and a right-hand side (`rhs`):

```
struct rule {
  char lhs;
  string rhs;
};

typedef struct rule* rule;
```

A grammar is simply an array of rules (along with the length of the array):

```
struct grammar {
    rule[] rules;
    int length;
};

typedef struct grammar* grammar;
```

Task 1 (2 pts)**.** Implement two auxiliary functions matching the following prototypes:

```
bool nonterminal(char ch);
bool terminal(char ch);
```

The first function returns `true` if the given character represents a non-terminal symbol, i.e., is uppercase, and the second function returns `true` if the given character represents a terminal symbol, i.e., is lowercase. Both functions return `false` otherwise.

Task 2 (2 pts)**.** Implement a specification function matching the following prototype:

```
bool is_rule(rule r);
```

This function should return `true` if the supplied rule is valid and `false` if it is not. For this assignment, a valid rule is one in which the left-hand side is a single non-terminal and the right-hand side is a non-empty string of terminals and non-terminals whose first character is a terminal.

Task 3 (4 pts)**.** Implement a specification function `is_grammar` matching the following prototype:

```
bool is_grammar(grammar G);
```

This function returns `true` if all of the rules in the grammar are valid, at least one rule has a left-hand side that contains the start symbol `S`, and rules with the same left-hand side non-terminals have right-hand sides that start with different terminals. It returns `false` otherwise.

Task 4 (2 pts)**.** Implement a specification function `all_terminals` matching the following prototype:

```
bool all_terminals(string s);
```

This function returns `true` if the given string contains only terminals; otherwise, it returns `false`.

2.3 The Validation Algorithm

In this part of the assignment, you will implement the parsing algorithm described in the example in section 2.1.

Task 5 (10 pts)**.** Implement a function matching the following prototype:

```
bool validate(grammar G, string s);
```

where the grammar `G` holds a valid set of rules for a grammar and `s` represents the string that is to be validated against the grammar `G`. This function should return `true` if the string can be generated by the rules of the grammar and `false` otherwise. You should use the algorithm described above to determine its answer.

You must include annotations for the precondition(s), postcondition(s) and loop invariant(s) for this function and any other auxiliary functions you write. You may include additional annotations for assertions as necessary.

2.4 Testing Your Validator

In this exercise, you will create text files that contain rules for five different grammars that you will analyze. You should create `main()` functions that use the support code to read in a grammar from a file and call your `validate` function on various strings of your choosing in order to try to determine what language each grammar describes.

To store the rules of a grammar in a file, list one rule per line, where each rule is simply its left-hand side followed by a space followed by its right-hand. For example, the grammar

```
S --> xSz
S --> y
```

would be stored in a text file as follows:

```
S xSz
S y
```

Task 6 (5 pts)**.** Create text files for each of the following 5 grammars.

```
GRAMMAR 1:
S --> xSy
S --> y

GRAMMAR 2:
S --> cMNc
M --> aMa
M --> b
N --> bNb
N --> a
```

```
GRAMMAR 3:
S --> xSz
S --> y

GRAMMAR 4:
S --> xPy
S --> yPx
P --> xy
P --> yx
```

```
GRAMMAR 5:
A --> aA
A --> bB
B --> bC
B --> aB
C --> aC
C --> c
S --> cA
```

Then create `main()` functions that call your `validate` function repeatedly with different input strings, printing out the results of all of your tests. Based on your results, create a text file named `answers.txt` that lists each grammar and the language that the grammar is describing. Your answers should be as precise and concise as possible.

2.5 Optional: King's Prize

Many more interesting grammars can be written if we also allow special productions of the form

```
N --> .
```

where `.` represents the empty string (usually written ϵ), meaning that the non-terminal `N` can be removed at any time and replaced by nothing. For example, the following grammar uses an empty transition to describe a language of balanced `l`-`r` pairs:

```
S --> lSrS
S --> .
```

Modify your `validate` function so that input strings can be validated against grammars that include these special kinds of rules. You will also need to modify the support code to deal with reading in rules that contain the empty string.

The winner of the King's Prize is the student who implements and tests this feature in the most elegant and well-annotated manner.