Stack ADT

You are familiar with stacks in your everyday life. You can have a stack of books on a table. You can have a stack of plates on a shelf. In computer science, a *stack* is an abstract data type (ADT) with two operations: `push` and `pop`. You either push something onto the top of the stack or you pop the element that is on the top of the stack. A more elaborate ADT for the stack might allow you to check if the stack has any items in it (`isEmpty`) or to examine the top element without popping it (`top`, also known as `peek`). But these operations not necessary for us to call something a stack.

Example 1

Here we make a stack of numbers. We assume the stack is empty initially, and then we have a sequence of pushes and pops.

```
push(3)
push(6)
push(4)
push(1)
pop()
push(5)
pop()
pop()
```

The elements that are popped will be `hello`, `green`, `blue` in that order, and afterwards the stack will have two elements in it (`fred` and `3`, with `fred` being on “top” of the stack).

```
                  1            5
            4     4      4     4      4
      6     6     6      6     6      6     6
3     3     3     3      3     3      3     3
--    --    --    --     --    --     --    --
```

Example 2: Balancing parentheses

It often occurs that you have a string of symbols which include left and right parentheses that must be properly nested or balanced. (In this discussion, I will use the term “nested” and “balanced” interchangeably.) One checks for proper nesting using a stack.

Example 2a

Consider the opening and closing parentheses “`(`” and “`)`” and an expression:

```
3 + (4 - x)* 7 + (y - 2 * (2 + x)).
```

in which the parentheses *are* balanced. But how do we determine that it is balanced?

Let’s scan through this expression and ignore the variables (`x`,`y`), numbers, and operators `*`, `+`, `-`. Whenever we see a left parenthesis, we push it on the stack, and when we see a right parenthesis, we pop the (matching) left parenthesis from the stack. The sequence of stack states is:

```
                    (
      (         (   (   (
---  ---  ---  ---  ---  ---  ---
```

If were to try to pop an empty stack (i.e. we reach a right parenthesis and the stack is empty), or we were to finish scanning the string and the stack were non-empty, then we would have an error – the parentheses would not be balanced.

Example 2b

Here is an example in which the parentheses are *not* balanced:

```
3 + (4 - x)* 7  + (y))) - ((2 * (2 + x)) )
```

If we try to use the stack to check for balanced parentheses, we get a sequence of stack states:

```
      (         (
---  ---  ---  ---  ---  X
```

where the `X` indicates that an error has occured because we reach a right parenthesis (the second one after the `y` when the stack is empty).

For our problem here, you don't really need a stack. You just need a counter. You start the counter at 0 and increment the counter when you see a left parenthesis and decrement the counter when you see a right parenthesis. The parentheses are balanced if the counter never becomes negative, and it is 0 when you are finished scanning.

Example 2c

A more interesting example of balancing parentheses in which you *do* need a stack is if there are multiple types of left and right parentheses, for example, `(`, `)`, `{`, `}`, `[`, `]`, and you require that the parentheses are properly *nested*. Consider the string:

```
( ( [ ] ) ) [ ] { [ ] }
```

You can check for balanced parentheses using a stack. You scan the string left to right. When you read a left parenthesis you push it onto the stack. When you read a right parenthesis, you pop the stack (which contains only left parentheses) and check if the popped left parenthesis matches the right parenthesis that you just read. For the above example, the sequence of stack states would be as follows.

```
         [
      (  (  (                          [
   (  (  (  (  (        [        {     {   {
-  -  -  -  -  -   -    -    -   -     -   -   -
```

The basic algorithm for matching parentheses is shown below. We assume the input has been already partitioned ("parsed") into disjoint *tokens*. A token can be one of the following:

- a left parenthesis (there may be various kinds)
- a right parenthesis (there may be various kinds)
- a string not containing a left or right parenthesis (operators, variables, numbers, etc)

Any token other than a left or right parenthesis is ignored by the algorithm.

Algorithm: check for balanced left and right parentheses
Input: sequence of tokens
Output: true or false (i.e. balanced or not)

```
while (not at end of token sequence) do
   t ← get next token
   if t is a left parenthesis then
      push(t)
   else
      if t is a right parenthesis then
         if stack is empty then
            return false
         else
            left ← pop()
            if !(match(left, t)) then
               return false
            end if
         end if
      end if
   end if
end while
if stack is empty then
   return true
else
   return false
end if
```

Example 2c

Here is an example where each type of parenthesis on its own is balanced, but overall the parentheses are not balanced.

```
    ( ( [ ) ] ) [ [ ] ] { [ } ]



              [
          (   (
      (   (   (
--- --- --- ---  X  since next symbol is ) which doesn't match top
```

While this example might seem obscure at first glance, in fact it arises in practice, for example, with HTML *tags*.[1] They are of the form `<tag>` and `</tag>` and these correspond to left and right parentheses, respectively. For example, `<b>` and `</b>` are "begin boldface" and "end boldface".

HTML tags are *supposed to be* properly nested, but some web browsers will allow improper nesting. For example, consider

```
<b> I am boldface, <i> I am boldface and italic, </i>
</b><i> I am just italic </i>.
```

whose tag sequence is `<b><i></i></b><i></i>` which is balanced. Compare that too

```
<b> I am boldface, <i> I am boldface and italic </b>
  I am just italic </i>
```

whose tags sequence is `<b><i></b></i>` which is not properly balanced. Most browsers will correctly interpret the latter, even though it is (strictly speaking) incorrect. Browsers need to be smart enough to correctly interpret these errors, otherwise people will switch to browsers that do understand.

Example 5: stacks in graphics

The next example is a simple version of how stacks are used in computer graphics. Consider a drawing program which can draw unit line segments (say 1 cm). Suppose the pen tip has a *state* (x, y, θ) that specifies its (x, y) position on the page and an angular direction θ. This is the direction in which it will draw the next line segment (see below). The pen state is initialized to be (0,0,0), where $\theta = 0$ is in the direction of the x axis.

Let's say there are five commands:

- `D` - draws a unit line segment from the current position and in the direction of θ, that it, it draws it from (x_0, y_0) to $(x_0 + \cos\theta, y_0 + \sin\theta)$.
- `L` - turns left (counter-clockwise) by 45 degrees
- `R` - turns right (clockwise) by 45 degrees

[1]If you have never looked at HTML source code before, then open a web browser right NOW and look at "view → page source" and observe the tags. They are the things with the angular brackets.

- `[` - pushes the current state onto the stack
- `]` - pops the stack, and current state ← popped state

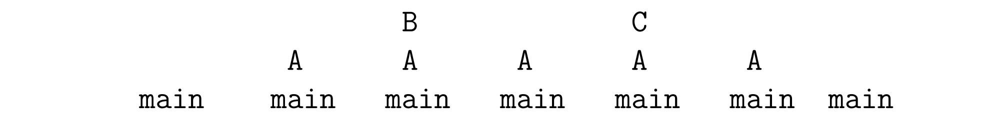
See the slides for a few examples of how to draw using such a drawing program.

Call Stack

We have been discussing stacks of things. One can also have a stacks of tasks.[2] Imagine you are sitting at your desk getting some work done (main task). Someone knocks on your door and you let them in and chat (task A). While chatting, the phone rings and you answer it (task B). You finish the phone conversation and go back to the person in your office (A). Then maybe there is another interruption (C) which you take care of, return to A and return to main.

A similar stack of tasks occurs when a computer program runs. Say we have a `main` method (as in Java or C) where the program starts. The main method typically has instructions that cause other methods to be called. The program "jumps" to these methods, executes them and returns to the main method. Sometimes these methods themselves call other methods, and so the program jumps to these other methods, executes them, returns to the calling method, which finishes, and then returns to main, etc.

A natural way to think of jumping from method to method is in terms of a stack. Suppose `main` calls method `mA` which calls method `mB`, and then when `mB` returns, `mA` calls `mC`, which eventually returns to `mA`, which eventually returns to `main` which then finishes. (See slides for this lecture.)



Why do we need to use a stack? Can't you just jump from method to method as the program runs? No, that won't work. The problem is that when a method finishes and returns, it needs to remember where to return to. This information (or "return address") is thrown on a stack.

How *exactly* the call stack and the "return address" work is a more advanced topic which you will learn about properly in COMP 273 or in ECSE 221. For now, let me mention that the call stack contains other data in addition to the return address that the method needs to know:

- any parameters passed to this method
- local variables of the method

These data and the return address together define a *stack frame*. The *call stack* just consists of a stack of these frames. When a method is called, a new frame is pushed on the call stack. When a method returns, its frame is popped from the call stack.

[2]though typically we don't call them "stacks", that is what they are.