CS203 Lab 4 – Pushdown Automata

Assigned: September 15th, 2025
Due: September 21st, 2025 by 11:59 pm to the Lab Moodle page as tar-zip archive.

Configuring the Project.

For this lab you are provided a tar-zip file on the lab Moodle page similar to the previous lab. The archive contains a variety of source and header files plus a Makefile to help you with the coding process.

Download the zipped file from the Moodle, un-archive the file, and examine what is provided. You will see the following files:

`main.c`	Contains the main function and is the starting point for the completed assignment.
`pda.h pda.c`	The files for implementing your Pushdown Automata functionality.
`stack.c stack.h`	These files provide you will a functioning stack implementation.
`test_stack.c`	A file containing a main for unit-testing of the stack implementation. This file also illustrates how the stack functionality works.
`list.c list.h`	These files are provided to all you to implement linked-list handling that should follow in a similar way how the stack is implemented.
`test_list.c`	A file containing a main for unit-testing the linked-list implementation. This file will also illustrate to me that your stack functions as it needs to.
`Makefile`	A make file that will help things compile in the correct way.

The Makefile provides the following options:

`test_stack`	Compiles only the stack functionality providing an executable called `test_stack`.
`test_list`	Compiles only the needed linked-list functionality providing an executable called `test_list`.
`lab4`	Compiles the full lab creating an executable called `lab4`.
`clean`	Removes all partially generated files, leaving only the source and Makefile.
`tar`	Will tar up your files, like what was provided in the previous lab.

This information will be reviewed during the start of lab. Please ask the professor or lab TA if you have any questions.

The Lab Assignment.

You will be required to construct a Push-Down Automata (PDA), which takes a state machine and adds a stack to representant large amounts of memory. Stacks are commonly found in programming languages to implement function calls, but also there is a type of Central Processing Unit (CPU) that is called a "stack machine". (https://en.wikipedia.org/wiki/Stack_machine). This type of machine does not use registers as most modern CPUs are constructed, instead there is a single stack for all memory processing. Java was conceived as a stack machine, with attempts to construct it in hardware (https://en.wikipedia.org/wiki/Java_processor).

What make PDA's more powerful, relative to a state machine, is that they can detect paired input alphabet values. A feature used in compliers to find paired braces, for example the string **"aa{bbbb}aa"** can be detected as valid by a PDA, while a state machine will lose track of the pairing for long well-formed strings with arbitrary numbers of pairs. Because of this feature, PDAs can detect well-formed palindromes, while state machines cannot.

As in the previous lab, you will be required to implement different machines using the PDA approach. Also, since you will be implementing a data structure, you will need to perform unit-testing. Plus, you will be working with multiple files, all this will start flexing your skills using your programing tools.

The goal here is to use the range of basic-C functionality, including: command-line input, pointers, and dynamic memory. But unspecified libraries are allowed.

What is a Push-Down Automata.

Again, this is one of the three primary theoretical machines that can remember multiple things, but the information is stored in a stack, so to access information that was stored earlier more recent information needs to be removed from the stack and forgotten.

For this lab you will be constructing three PDA's with each one conceptually building on the next. First you will be detecting the set of strings with paired symbols. The second machine will detect strings consisting of left and right paired-parenthesis. For paired-parenthesis, an example valid string is "(()(()))" while an invalid string would be "(()(())". The third machine will add additional functionality to detect variable initialization, and what scope the variable is in when it is detected. One example string is "**{ ix iy y { ix x { ix x y }}}**", where input "**ix**" indicates that the variable x is created in the current scope and input "**x**" indicates a variable access. Further details are provided below. You will not be allowed to use any string handling or algorithm libraries. You will need to code everything with plain-C. Hint – For the first and second machines, you will only need the stack. For the third case, you will also need a list with integrated stacks, requiring the creation of a linked-list implementation and additional unit-testing. Use the stack code and unit-test example as a starting point of the list.

Hint – Your stack and linked-list must be unit-tested and this coding should be performed before you have coded the PDA functionality. Building and testing these lower-level tools is essential to having a solid application, and will reduce the errors generated in the later process.

To understand PDAs, we need to define them mathematically. These machines do not use calculus in any way, instead, these machines rely on discrete math. The definition of a Push Down Automata is as follows:

$$PDA = \{S, I, L, g, s_0, A\}$$

S – is a set of states, given as s_0, s_1, s_2, s_3, s_4 …

I – input alphabet, letters of a char type.

L – stack alphabet, letters that will be stored on the stack as a char type. Stack alphabets will also include the "empty" symbol **#**, representing no modification of the stack.

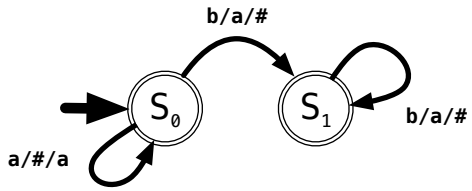
g – a transition function that accepts the current state, the current input, and determines the next state. Often this is provided as a transition table. PDA transitions still read in an input value, determining what transition is taken from one state to the next. But also, two more symbols are provided, one indicating what should be on the top of the stack (and will be removed) plus a second indicating what should be pushed onto the stack. An example could be `"a/d/f"`, indicating an input `'a'` when `'d'` is on the top of the stack, pop the top value and push `'f'`. A second example `"b/#/f"`, indicates if there is an input `'b'` remove nothing from the stack (ignoring the value that is there) and push a new value `'f'`. Finally, if no transition exists for the current situation, then the string is invalid.

s_0 – the initial state.

A – a set of acceptor states, where if the PDA runs through the entire sequence of inputs and it ends in one of these states, then the machine returns that the input string is valid. If the machine does not end in an acceptance state, then the input string will be considered invalid. In addition to the accepting state, for a string to be valid, the stack must be empty. If the stack is not empty when the last input letter is processed but the PDA is an accepting state, the string is still invalid.

Some versions of PDAs place a starting symbol on the stack. For this lab, your machine will start with an empty stack **and for a string to be valid it must end with an empty stack in an accepting state**. Thus: if all elements are removed from a stack with remaining input characters, then the string is invalid, and if the stack still has elements, but no input characters, then the string is invalid. So, the input string must be completely processed, and at the same time the stack has no more elements.

Because of the added memory and constraints, PDA's can be very simple. Below is an example of a machine that detects strings where there are **n** '**a**' values, and then **n** '**b**' values.



This will consume all the input values '**a**', pushing an '**a**' on the stack. Once the first '**b**'s are reached, then value '**a**' is repeatedly removed from the stack for each '**b**' encountered until all '**a**'s are removed from the stack. If a stack is emptied before all the letters are processed, the string is invalid. If all inputs are processed and the stack still has remaining elements, the string is invalid.

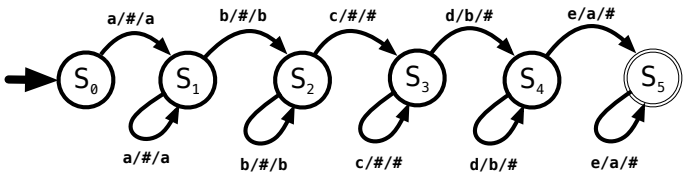
For this exercise, you can assume that all input strings will consist only of the valid input alphabet letters and that the string is well formed.

Implementing the lab.

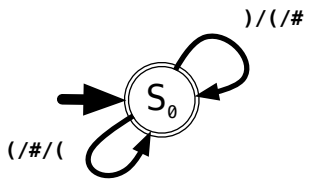
Again, you will create three PDAs.

The **first** is a PDA detecting the following set of strings:

$$\{a^m b^n c^o d^n e^m \mid \text{for all values } m > 0,\ n > 0, \text{ and } o > 0\}$$



The **second** PDA you will create will pair parenthesis, and ensure that all left-parenthesis match right-parenthesis. So the string "`( ( () () ) )`" is valid and "`()(()))`" is invalid. This PDA is very simple. In fact you could do this with a counter, but for this lab, use a stack.



For the third PDA, you will perform additional processing. The input alphabet consists of: '`{`', '`}`', '`x`', '`y`', '`z`', '`ix`', '`iy`', '`iz`'. The two-character symbols indicate the creation of a variable, so **`ix`** indicates that variable **x** is being created (and is analogous to "int x"). The single character symbol indicates a variable access, so just **x** means **x** is being accessed from a particular scope. But what scope is it? This is where the stack is essential. Your third PDA should print the lines below for the given input string "**`{ix iy {iy x y} y}`**". **For this machine, you can ignore blanks.**

```
init x level 0
init y level 0
init y level 1
read x level 0
read y level 1
read y level 0
string is valid
```

Thus, as a variable is created print that it was initialized in a particular scope level. Then when a variable is read this value will be from a particular scope, so **y** will read from the local scope in one case and a higher scoping in a different case. Scope change will follow the changing stack. Also, to track the **shadowed variables**, create a linked list with each element representing variable and a stack to track how where the different initializations have occurred. If a variable read occurs before the variable is created, this should be identified as an invalid string. You can assume that the string will only consist of input symbols and blanks that can be ignored. To implement this third machine, you might need to modify the stack implementation a little bit.

C knowledge.

Unit-testing is required for this lab and will be done in a very primitive way. Using a make file, you can have different targeted executables, ones for each-unit test and one for the main program. In the unit-test mains, you can use the C assert library that provided the assert command.

```
NAME
     assert – expression verification macro

SYNOPSIS
     #include <assert.h>

     assert(expression);

DESCRIPTION
     The assert() macro tests the given expression and if it is false, the
     calling process is terminated.  A diagnostic message is written to stderr
     and the abort(3) function is called, effectively terminating the program.

     If expression is true, the assert() macro does nothing.
```

Using assert you can ask questions using normal relational expressions. For example, in the stack unit-test there is the following:

```
node = generate_stack_node('2');
assert(node->item == '2');
assert(node->next == NULL);
```

This verifies that the generated node contains the item value 2 and next value NULL. If any assert is false, the program will fail and you will be informed where that failure occurs. If the assert succeeds, the nothing happens, and the program continues. If your unit-tests do not fail the program runs normally. It is a very simple approach that works because C is a very simple language with few constraints. For more about using the assert function see the stack unit-test.

When memory is generated using a malloc, the software developer must check to see if there is an error. A failed malloc will generate a NULL pointer and checking it when it happens makes a software developer's life easier.

```
struct stack_node *node = malloc(sizeof(struct stack_node) );

// This is how you do error checking in C.
if(node == NULL) {
    printf("ERROR: %s\n", __FUNCTION__);
    printf("     : %s\n", strerror(errno));

    // Ends the program.
    exit(-1);
}
```

Code for detecting and reporting the error is given above. The malloc occurs and the returned value is verified. If the returned value is NULL, then two lines are printed. The first-line consists of the name of the function provided by the **__FUNCTION__** macro providing the enclosing function name. The second-line determines what the error is using the function **strerror** provided by the system and the variable **errno** stored in the system. This error handling is provided by **errno.h** library and uses **string.h**. The last line contains **exit**, which will terminate the program. Thus, the combination of statements, indicates: where the error is, what the error is and then halts the program.

Much of the knowledge for this program was developed in the first lab. The new things are described above. There is also the book and topics we have covered in lecture. Still, any questions you have, please ask, these questions might generate answers that will be useful to everyone.

Requirements.

- Several files have been provided as part of the project. For the most part these will be the files you will use to implement the lab.
- The main program will use command-line input only and not ask for input. The format of this handling is discussed in the provided code.
    - Using the command-line, you will receive the string to be evaluated. For strings that contain spaces, you can use double-quotes to identify the entire string as a single item.
- The **`string.h`** has been included in the files. **But you are not allowed to use the string library**, instead this is provided only for the error handling.
- The stack functionality has been provided for you, use this as an example for your list implementation.
- There might be a point where extending the stack implementation makes sense, but it still needs to support the functionality need for all forms of pda. Of course, any modifications need to also be unit-tested.
- Both the stack and the list must be unit-tested, the stack has already been unit-tested, use this as example for your list unit-testing. You might also consider unit-testing your PDA implementations. (**hint, hint**)

You might find a debugger could be handy… however, it seems fully unit-tested code often does not really need a debugger.