Pushdown Automata

In the last section we found that restricting the computational power of computing devices produced solvable decision problems for the class of sets accepted by finite automata. But along with this ability to solve problems came a rather sharp decrease in computational power. We discovered that finite automata were far too weak to even tell if an input string was of the form a^nb^n. In this section we shall extend the power of finite automata a little so that we can decide membership in sets which cannot be accepted by finite automata.

Let's begin. In fact, let's provide a finite automaton with a data structure which will allow it to recognize strings of the form a^nb^n. To tell if a string is of the form a^nb^n we need to match the a's with the b's. We could use a counter for this, but thinking ahead a bit, there is a computer science way to do this. We shall allow the machine to build a pile of discs as it processes the a's in its input. Then it will unpile these disks as it passes over the b's. Consider the following algorithm for a machine of this kind.

```
place the input head on the leftmost
      input symbol

while symbol read = a
   advance head
   place disc on pile

while symbol read = b and pile contains discs
   advance head
   remove disc from pile

if input has been scanned
     and pile = empty then accept
```

Figure 1 - a^nb^n Recognition Algorithm

It is clear exactly what happens when the algorithm of figure 1 is used on the input aaabbb. The machine reads the a's and builds a pile of three discs. Then it reads the b's and removes the discs from the pile one by one as each b is read. At this point it has finished the input and its pile is empty so it accepts. If it was given aabbb, it would place two discs on the pile and then remove them

as it read the first two b's. Then it would leave the second while loop with one b left to read (since the pile was empty) and thus not accept. For aaabb it would end with one disk on the pile and not accept that input either. When given the input string aabbab, the machine would finish the second loop with ab yet to be read. Now, try the strings aaa and bbb as exercises. What happens?

We now have a new data structure (a pile) attached to our old friend, the finite automaton. During the last algorithm several conventions implicitly arose. They were:

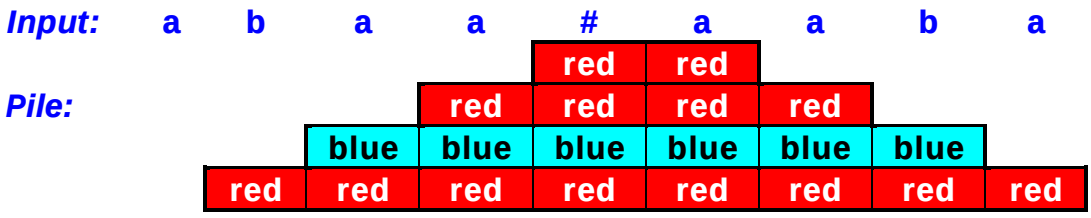
- The tape head advanced on each step,
- Discs were placed on *top* of the pile, and
- An empty pile means acceptance.

Let us now attempt something a bit more difficult. Here's where we shall use a structure more powerful than a counter. Why not try to recognize strings of the form $w\#w^R$ where w is a string over the alphabet {a, b} and w^R is the *reversal* of the string w? (Reversal is just turning the string around end for end. For example, $abaa^R = aaba$.) Now we need to do some comparing, not just counting. Examine the algorithm of figure 2.

```
place input head upon leftmost input symbol

while symbol being scanned ≠ #
   if symbol scanned = a, put red disk on pile
   if symbol scanned = b, put blue disk on pile
   advance input head to next symbol

advance input head past #

repeat
  if (symbol scanned = a and red disk on pile)
     or (symbol scanned = b and blue disk on pile)
        then remove top disk; advance input head
until (pile is empty) or (no input remains)
      or (no disk removed)

if input has been read and pile is empty then accept
```

Figure 2 - Accepting Strings of the Form $w\#w^R$

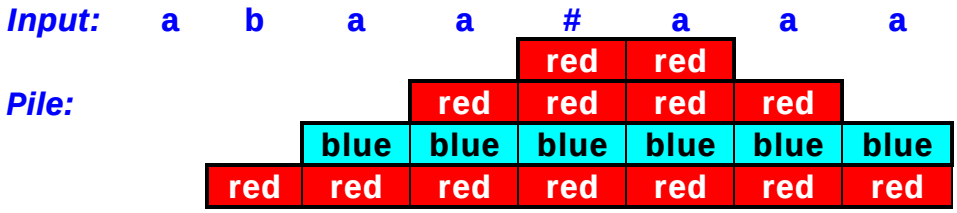
We will now look at what happens to the disc pile when this machine processes the input string abaa#aaba. Here is a picture:



At the right end of the picture, the machine reads the a and removes the red disk from the stack. Since the stack is empty, it accepts.

Our first machine (figure 1) used its discs to count the a's and match them against the b's. The second machine (figure 2) used the pile of discs to full advantage in that it compared actual symbols, not just the number of them. Note how this machine recorded the symbols before the marker (#) with discs and then matched them against the symbols following the marker. Since the input was completely read and the pile was empty, the machine accepted.

Now, here is what happens when the string abaa#aaab is processed:



In this case, the machine stopped with ab yet to read and discs on the pile since it could not match an a with the blue disc. So, it rejected the input string. Try some more examples as exercises.

The machines we designed algorithms for above in figures 1 and 2 are usually called *pushdown automata*. All they are is finite automata with auxiliary storage devices called *stacks*. (A *stack* is merely a pile. And symbols are normally placed on stacks rather than various colored discs.) The rules involving stacks and their contents are:

a) Symbols must always be placed upon the top of the stack.
b) Only the top symbol of a stack can be read.
c) No symbol other than the top one can be removed.

We call placing a symbol upon the stack a *push* operation and removing one from the top of the stack a *pop* operation.

Figure 3 provides a picture of one of these machines.

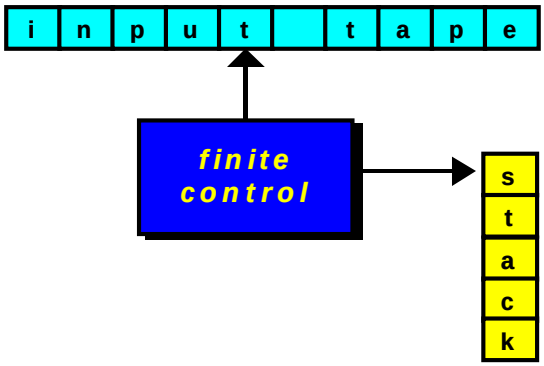


Figure 3 - A Pushdown Automaton

Pushdown automata can be presented as state tables in very much the same way as finite automata. All we need to add is the ability to place (or *push*) symbols on top of the stack and to remove (or *pop*) symbols from the top of the stack. Here is a state table for our machine of figure 1 which accepts strings of the form $a^n b^n$.

state	read	pop	push	goto
1	a		A	1
	b	A		2
2	b	A		2

Note that this machine operates exactly the same as that of the algorithm in figure 1. During operation, it:

a) reads a's and pushes A's on the stack in state 1,
b) reads b's and pops A's from the stack in state 2, and
c) accepts if the stack is empty at the end of the input string.

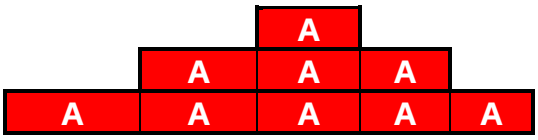
Thus, the states of the pushdown machine perform exactly the same as the while loops in the algorithm presented in figure 1.

If a pushdown machine encounters a configuration which is not defined (such as the above machine being in state 2 reading an a, or any machine trying to pop a symbol from an empty stack) then computation is terminated and the machine rejects. This is very similar to Turing machine conventions.

A trace of the computation for the above pushdown automaton on the input aaabbb is provided in the following picture:

Input Remaining:	aaabbb	aabbb	abbb	bbb	bb	b	
State:	s_1	s_1	s_1	s_1	s_2	s_2	s_2

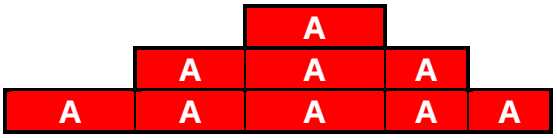
Stack:



and a trace for the input aaabb is:

Input Remaining:	aaabb	aabb	abb	bb	b	
State:	s_1	s_1	s_1	s_1	s_2	s_2

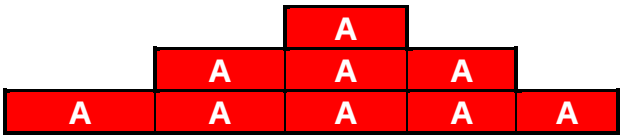
Stack:



In the first computation (for input aaabbb), the machine ended up in state s_2 with an empty stack and accepted. The second example ended with an A on the stack and thus the input aaabb was rejected. If the input was aaabba then the following would take place:

Input Remaining:	aaabba	aabba	abba	bba	ba	a
State:	s_1	s_1	s_1	s_1	s_2	s_2

Stack:



In this case, the machine terminates computation since it does not know what to do in s_2 with an a to be read on the input tape. Thus aaabba also is rejected.

Our second machine example (figure 2) has the following state table.

state	*read*	*pop*	*push*	*goto*
1	a		A	1
	b		B	1
	#		2	
2	a	A		2
	b	B		2

Note that it merely records a's and b's on the stack until it reaches the marker (#) and then checks them off against the remainder of the input.

Now we are prepared to precisely define our new class of machines.

Definition. *A **pushdown automaton** (pda) is a quintuple* $M = (S, \Sigma, \Gamma, \delta, s_0)$, *where:*

S *is a finite set (of states),*
Σ *is a finite (input) alphabet,*
Γ *is a finite (stack) alphabet,*
δ: $S \times \Sigma \times \Gamma \cup \{\varepsilon\} \rightarrow \Gamma^* \times S$ *(transition function),*
and $s_0 \in S$ *(the initial state).*

In order to define computation we shall revert to the conventions used with Turing machines. A *configuration* is a triple <s, x, α> where s is a state, x a string over the input alphabet, and α a string over the stack alphabet. The string x is interpreted as the input yet to be read and α is of course the content of the stack. One configuration *yields* another (written $C_i \rightarrow C_k$) when applying the transition function to it results in the other. Some examples from our first machine example are:

$$\begin{aligned} \langle s_1, aaabbb, \varepsilon\rangle &\rightarrow \langle s_1, aabbb, A\rangle \\ \langle s_1, aabbb, A\rangle &\rightarrow \langle s_1, abbb, AA\rangle \\ \langle s_1, abbb, AA\rangle &\rightarrow \langle s_1, bbb, AAA\rangle \\ \langle s_1, bbb, AAA\rangle &\rightarrow \langle s_2, bb, AA\rangle \\ \langle s_2, bb, AA\rangle &\rightarrow \langle s_2, b, A\rangle \\ \langle s_2, b, A\rangle &\rightarrow \langle s_2, \varepsilon, \varepsilon\rangle \end{aligned}$$

Note that the input string decreases in length by one each time a configuration yields another. This is because the pushdown machine reads an input symbol every time it goes through a step.

We can now define *acceptance* to take place when there is a sequence of configurations beginning with one of the form $\langle s_0, x, \varepsilon\rangle$ for the input string x and ending with a configuration $\langle s_i, \varepsilon, \varepsilon\rangle$. Thus a pushdown automaton accepts when it finishes its input string with an empty stack.

There are other conventions for defining pushdown automata which are equivalent to that proposed above. Often machines are provided with an initial stack symbol Z_0 and are said to terminate their computation whenever the stack is empty. The machine of figure 2 might have been defined as:

state	read	pop	push	goto
0	a	Z_0	A	1
	b	Z_0	B	1
	#	Z_0		2
1	a		A	1
	b		B	1
	#			2
2	a	A		2
	b	B		2

if the symbol Z_0 appeared upon the stack at the beginning of computation. Including it in our original definition makes a pushdown automaton a *sextuple* such as:

$$M = (S, \Sigma, \Gamma, \delta, s_0, Z_0).$$

Some definitions of pushdown automata require the popping of a stack symbol on every move of the machine. Our example might now become:

state	read	pop	push	goto
0	a	Z_0	A	1
	b	Z_0	B	1
	#	Z_0		2
1	a	A	AA	1
	a	B	AB	1
	b	A	BA	1
	b	B	BB	1
	#	A	A	2
	#	B	B	2
2	a	A		2
	b	B		2

where in state s_1 the symbols which were popped are placed back upon the stack.

Another very well known convention is to have pushdown machines *accept by final state*. This means that the automaton must pass the end of the input in an accepting or final state. Just like finite automata. Now we have a final state subset and our machine becomes:

$$M = (S, \Sigma, \Gamma, \delta, s_0, Z_0, F)$$

and our tuples get larger and larger.

Converting this example to this format merely involves detecting when the stack has only one symbol upon it and changing to an accepting state if things are satisfactory at this point. We do this by placing special sentinels (X and Y) on the bottom of the stack at the beginning of the computation. Here is our example with s_3 as an accepting state. (Note that the machine accepts by empty stack also!).

state	read	pop	push	goto
0	a		X	1
	b		Y	1
	#			3
1	a		A	1
	b		B	1
	#			2
2	a	A		2
	a	X		3
	b	B		2
	b	Y		3

All of these conventions are equivalent (the proofs of this are left as exercises) and we shall use any convention which seems to suit the current application.

Now to get on with our examination of the exciting new class of machines we have defined. Our first results compare them to other classes of automata we have studied.

Theorem 1. *The class of sets accepted by pushdown automata properly includes the regular sets.*

Proof. This is very easy indeed. Since finite automata are just pushdown machines which do not ever use their stacks, all of the regular sets can be accepted by pushdown machines which accept by final state. Since strings of the form $a^n b^n$ can be accepted by a pushdown machine (but not by any finite automaton), the inclusion is proper.

Theorem 2. *All of the sets accepted by pushdown automata are recursive.*

Proof. For the same reason that the regular sets are recursive. Pushdown machines are required to process an input symbol at each step of their computation. Thus we can simulate them and see if they accept.

Corollary. *The class of recursively enumerable sets properly contains the class of sets accepted by pushdown automata.*

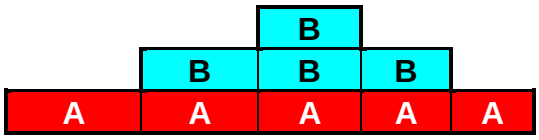
As you may have noticed (and probably were quite pleased about), the pushdown machines designed thus far have been deterministic. These are usually called *dpda*'s and the *nondeterministic* variety are known as *npda*'s. As with nondeterministic finite automata, npda's may have several possible moves for a configuration. The following npda accepts strings of the form ww^R by nondeterministicly deciding where the center of the input string lies.

state	read	pop	push	goto
0	a		A	0
	a	A		1
	b		B	0
	b	B		1
1	a	A		1
	b	B		1

Note that in s_0 there are choices of either to remain in s_0 and process the w part of ww^R or to go to s_1 and process the w^R portion of the input. Here is an example computation of the machine on the string abbbba.

Input Remaining:	**abbbba**	**bbbba**	**bbba**	**bba**	**ba**	**a**	
State:	**s_0**	**s_0**	**s_0**	**s_0**	**s_1**	**s_1**	**s_1**

Stack:



One should note that the machine does in fact change from s_0 to s_1 when bba remains to be read and a B is on top of the stack. We shall not prove it here, but *nondeterminism does indeed add power to pushdown machines*. In fact, the set of strings of the form ww^R cannot be accepted by a deterministic pushdown automaton because it would have no way to detect where the center of the input occurs.

Our last example of a set which can be accepted by a pushdown automaton is a set of simple arithmetic expressions. Let's take v as a variable, + as an operator, and put them together (in the proper manner) with parentheses. We get expressions such as:

v+v+v, or v+(v+v), or (v+v)+(v+v)

but not expressions like:

v+v+v+, or (v+v)(v+v), or (v+(v+(v+v)).

The strategy for the machine design is based on the method in which these simple expressions are generated. Since an expression can be:

a) a variable,
b) a variable, followed by a plus, followed by an expression, or
c) an expression enclosed in parentheses.

Here is a simple but elegant, one state, nondeterministic machine which decides which of the above three cases is being used and then verifies it. The machine begins computation with the symbol E upon its stack.

read	*pop*	*push*
v	E	OE
v	E	
(	E	EP
+	O	
)	P	

With a little effort this nondeterministic machine can be turned into a deterministic machine. Then, more arithmetic operators (such as subtraction, multiplication, etc.) can be added. At this point we have a major part of a parser for assignment statements in programming languages. And, with some output we could generate code exactly as compilers do. This is discussed in the treatment on formal languages.