Language Acceptors

- **Push down automatons (PDAs)** are the language acceptors for grammar
  - Accept words in the language
  - Correspond to finite state machines
Language Acceptor

• For “regular” languages, we specified a finite state machine.

Finite set of states limiting factor preventing $a^n b^n$ recognition

Finite set of states
• Input word
• Transition between states based on characters in word

Could specify FMS with set of triples describing transitions: 
$(currentState, input, nextState)$

$T = \{(0,a,1),(0,b,2),(2,a,1)\}$

and a list of accepting states

$A = \{1\}$
Language Acceptors

• Let’s augment the FSM, to include a stack of symbols

We’ll store extra information by pushing it down on the stack (hence, “push down” automata)
Stack = primitive memory unit

Often characters we’ll push on, but not limited to characters;

Can also pop off of stack
PDA Example

• PDA for recognizing $a^n b^n$
• Informally, push all a’s on, see a b, start popping a’s off. Hope that there aren’t any a’s left when finish with b’s and hope that there are enough a’s to finish with b’s.

• More Formally:
  – FSM component: Need two states to enforce the a-b ordering
    First b is a trigger to start popping items off stack
  – Use the stack to count the a’s. Match b count to that of a’s by popping off a symbols like we did with [] matcher.
Let # be end of string and bottom of stack marker (on by default).

- 'read' outputs are letters consumed from inputs (input alphabet).
- 'pop' outputs are letters from stack alphabet (doesn’t have to match input alphabet).
a^n b^n recognizer PDA

- Red highlights “states” of system
- Working in a state involves a state operation and a stack operation

FLOW CHART VERSION
a^n b^n recognizer PDA

Use a 5-tuple representation
(currentState, input, toPop, nextState, toPush)

\{(0,a,NULL,0,a), (0,b,a,1,NULL), (1,b,a,1,NULL), (1,#,#,2,NULL) \} (accepting is 2, reject is implicit)
PDA Example

Execution for aabb#

Transitions: {((0,a,NULL,0,a), (0,b,a,1,NULL), (1,b,a,1,NULL), (1,#,#,2,NULL) } (accepting is 2, reject is implicit)
PDA Example

• PDA for recognizing balanced brackets:
  – [], [[]], [[[]], ... (ending with # symbol)

• Intuitively,
  – Push every opening bracket [ onto the stack
  – When see a closing bracket ], pop the corresponding opening [ off.
  – At end of input word, should have an empty stack if all pairs matched up.

  – What is different here than a^nb^n – we have arbitrary timed pushes and pops – not a strict ordering of all left brackets before all right brackets (change up our states a little)
Paired Bracket Recognizer PDA

- Let # be end of string and bottom of stack marker (on by default)
- Read outputs are letters consumed from inputs (input alphabet)
- Pop outputs are letters from stack alphabet

FLOW CHART VERSION

What error is this?
Not enough [ ]

What error is this?
Not enough [ ]
PDA Example

– Three transition definitions:
  • (0,[],null,0,[]) – If you’re in state 0 and read a [ on the input, pop nothing off the stack, stay in state 0, and push the [ onto the stack
  • (0,],[],0,null) – If you’re in state 0 and read a ] on the input, pop the [ off the top of the stack, stay in state 0, and push nothing onto the stack
  • (0,#,#,1,null) Accept if out of letters and out of stack
PDA Example

Execution for [[[[]]]#

state 0 until --------------------------------------------→ state1

Transitions: {(0,[,null,0,]), (0,][,0,null), (0,#,#,1,null)}

At end, empty input, empty stack, still in state 0 which is accepting, so balanced

If we had an extra [, that would leave the stack non-empty. If we had an extra ], we would have an empty stack and a char left in the input string for which we couldn’t remove a [ from the stack. Both of these REJECT.
A more rigorous definition

• Acceptance can be defined as:
  – Consuming all the input and in an accepting final state and the stack is empty
  – If it works for your domain, you can “say – accept on empty stack OR accept on accepting final state”, because there are transformations which allow you to take that acceptance and automatically also get the alternative mechanism
    • For example, in our earlier machine, ‘matched brackets’, we really just needed ‘empty stack’ (that’s all we were looking at)
• Machine can detect an empty stack and enter an accepting final state by detecting a unique symbol pushed by the initial state.
  – We are basically already doing this

OR

• An accepting final state can perform a pop loop to get to an empty stack
PDA Example

• Accept the language “almost-palindrome”:
  – Words over the alphabet (a,b,c) such that $w = (a \mid b)^*$, $w^R = \text{reverse}(w)$, and $c = c$, and input = $wcw^R$
    • Example words: abcba, bacab, ...

• Any ideas?
PDA Example

- How to handle:
  - Push onto stack any (a|b)* combo in order of seeing a’s and b’s
  - When hit c, need to make sure that all a’s and b’s after that are matched with the same letter on the top of the stack
    - Sounds like a state transition needs to be triggered on c, moving to a new rule set
    - Once in new state, need to check stack characters are the same
- $T = \{(0,a,\text{null},0,a),(0,b,\text{null},0,b),(0,c,\text{null},1,\text{null}), (1,a,a,1,\text{null}),(1,b,b,1,\text{null}),(1,#,#,2,\text{null})\}$
  - Note that C never gets pushed onto stack – it just is a state changer
- $A = \{2\}$