Parsing and PDAs

November 14, 2011
CS 81: Computability & Logic

Review

- ✓ Grammars
- ✓ Context-Free Grammars
- ✓ Ambiguous and Unambiguous Grammars
- ✓ A Pumping Lemma for CFLs ($uvxyz$)
  - $\{ a^n b^n c^n \mid n \geq 0 \}$ is not context free.
  - $\{ ww \mid w \in \{a, b\}^* \}$ is not context free
- ✓ Grammars can encode regular expressions/DFAs/NFAs.
  - Every regular language is also context free.

Chomsky Hierarchy

- ✓ Type 0: Unrestricted Grammars
- ✓ Type 1: Context-Sensitive Grammars
- ✓ Type 2: Context-Free Grammars
- ✓ Type 3: Regular Grammars

Context-Sensitive Example

What is the language of the following *context-sensitive* (*monotone*) grammar?

$$
\begin{array}{lcl}
S & \rightarrow & abc \\
S & \rightarrow & aSQ \\
bQc & \rightarrow & bbcc \\
cQ & \rightarrow & Qc
\end{array}
$$

Strings with Composite (Non-Prime) Lengths

$$
\begin{aligned}
S &\to HaC \\
C &\to aC \\
C &\to aT \\
T &\to ZU \\
AZ &\to ZA \\
aZ &\to ZAa \\
HZ &\to HY
\end{aligned}
\qquad
\begin{aligned}
YA &\to AY \\
Ya &\to aY \\
YU &\to T \\
T &\to X \\
aX &\to Xaa \\
AX &\to Xa \\
HX &\to \varepsilon
\end{aligned}
$$

The Parsing Problem

Given a grammar and a string,

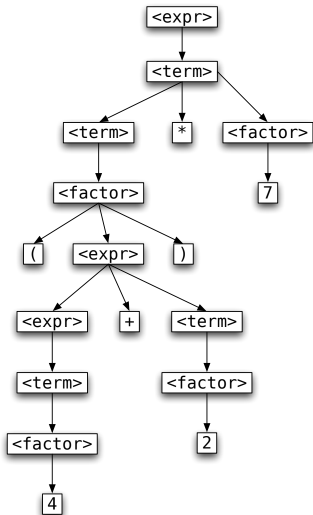
1. Is the string in the language?
2. Why? (Parse tree or other evidence)

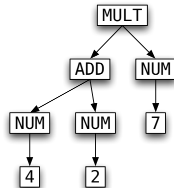
Evidence

```
<expr> ::= <term>
         | <expr> + <term>

<term> ::= <factor>
         | <term> * <factor>

<factor> ::= <int>
           | ( <expr> )
```





42

Naive Parsing

How about backtracking search? All ways to derive the string from S.

- ✓ Inefficient
- ✓ Harder to implement than you might think…

Bogus Backtracking

```
S -> Ac | Bc
A -> a | c | ab
B -> Bb | b
```

```
Consume_S():
    try Consume_A(), then consume c
    if fails, try Consume_B(), then consume c
Consume_A():
    try consume a
    if fails, try consume c
    if fails, try consume a then consume b
```

[What's wrong?]

```
Consume_B():
    try Consume_B(), then consume b
    if fails, try consume b
```

[What's wrong?]

LL(k) Grammars

```
S -> Aa | Ba
```

If *each* `Consume` function always "knew" which right-hand-side was correct, **we would never need to backtrack or and never get tangled in infinite loops**.

Then

- ✓ It would *be easy* to write correct `Consume` functions
- ✓ Our parser would run in linear time (in the length of the input).

We say that a grammar is $\mathrm{LL}(k)$ if, by "peeking ahead" no more than k tokens, *we can* guarantee *a decision* that is

1. correct
2. unique

When will Top-Down Parsing Work?

Are these grammars $\mathrm{LL}(k)$ for some k?

```
S ::= E $
E ::= n
    | plus E E
    | times E E
```

```
S ::= A
    | B
A ::= a
    | x A
B ::= b
    | y B
```

```
S ::= A
    | B
A ::= a
    | x A
B ::= b
    | x B
```

```
S ::= E $
E ::= n
    | n + E
```

```
S ::= E $
E ::= n
    | E + n
```

```
S ::= E $
E ::= E + E
    | E * E
    | n
```

Machines

Question: If FSMs recognize regular languages, what machines recognize CFLs?

Answer: Pushdown Automata (PDAs)

- ✓ Finite state machine + stack
- ✓ Transitions
  - Depend on the state and/or input symbol
  - Change state + add/remove/replace top-of-stack
- ✓ In general, can be nondeterministic.
- ✓ Accept if in accept state *and* [Rich] stack is empty

Example

Using a state machine and a stack, how could we recognize

$$\{ 0^n 1^n | n \geqslant 0 \}$$

Official Definition

A PDA consists of

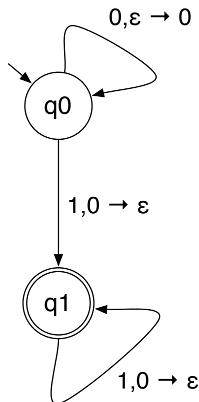
- ✓ A finite set K of states
- ✓ A finite alphabet Σ
- ✓ A finite "stack alphabet" Γ
- ✓ A start state $q_0 \in Q$
- ✓ Accepting states $A \subseteq Q$
- ✓ Transitions, each from the set

$$\left(K \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\})\right) \times \left(K \times (\Gamma \cup \{\varepsilon\})\right)$$

PDA Example

Using a state machine and a stack, how could we recognize

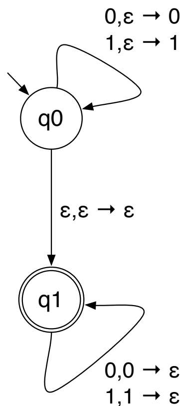
$$\{ 0^n 1^n \mid n > 0 \}$$



PDA Example

Using a state machine and a stack, how could we recognize

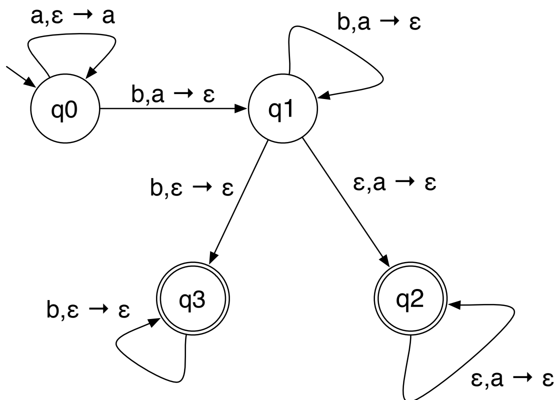
$$\{ ww^R \mid w \in \{0, 1\}^* \}$$



PDA Example

Using a state machine and a stack, how could we recognize

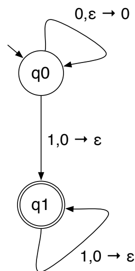
$$\{ a^i b^j \mid i, j > 0 \ \wedge \ i \neq j \}$$

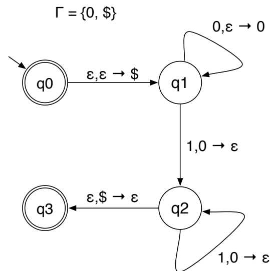


PDA Variations

There are many equivalent ways of defining PDAs.

Some authors say a PDA accepts *whenever* it's in an accept state. How would fix the previous PDAs?





Other authors (e.g., Rich) let transitions push or pop any finite string from the stack.

PDAs vs CFGs: Summary

1. PDAs can recognize any CFL
   - Given a grammar, construct a PDA for top-down parsing
   - Use nondeterminism to always "guess" the correct rule (No backtracking required!)

2. PDAs recognize only CFLs
   - Turn a PDA into a grammar that simulates it
   - See the book: for every two pair of states p, q and each nonterminal x, define a nonterminal $\langle p, x, q \rangle$, strings that get you from state p to q in the PDA with the same stack except for popping x.

Closure Properties

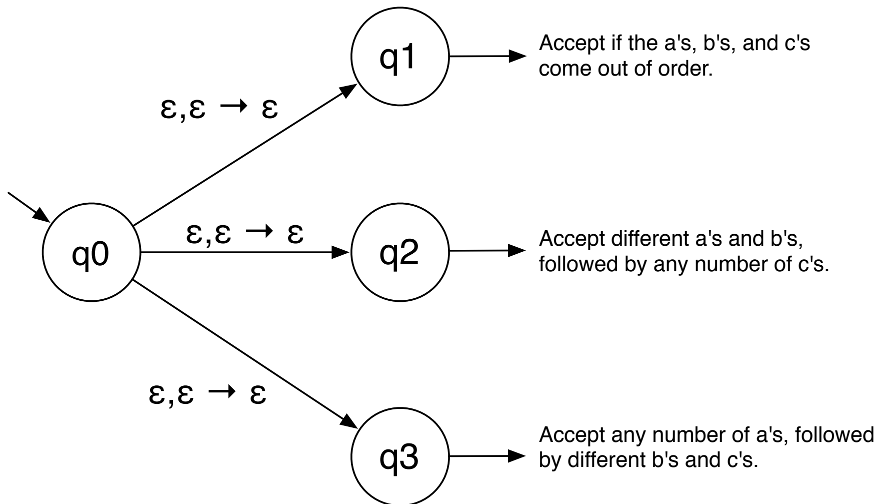
Context-Free Languages are:

- ✓ Closed under star, union, concatenation.
- ✓ *Not* closed under intersection, complement.

Key issue: nondeterministic PDAs are strictly more powerful than deterministic PDAs!

(No analogy to the subset construction)

$\Sigma^* \setminus \{ a^n b^n c^n \mid n \geq 0 \}$



Practical Parsing

Recursive Descent

- ✓ Form of code follows the grammar.
- ✓ Efficient and correct for LL grammars.

Another very practical method: Shift-Reduce Parsing

- ✓ Efficient and correct for LR grammars
- ✓ Parser code is usually computer-generated!

But neither of these handle ALL CFGs...

CYK (CKY) Algorithm

- ✓ Works for any CFG.
- ✓ Parses inputs in $O(n^3)$ ($n =$ length of input)
- ✓ Requires a grammar in "Chomsky Normal Form"
  - All rules of the form $A \to a$ or $A \to BC$.
- ✓ Key ideas:
  - For every substring, what nonterminals produce it?
  - Dynamic programming for efficiency

Dynamic Programming

Recursive expressions, such as

$$
\begin{aligned}
f(n) &:= 1 && \text{if } n < 3 \\
f(n) &:= f(n-1) + f(n-3) && \text{otherwise}
\end{aligned}
$$

are very clear, but inefficient if taken literally. Two roughly-equivalent

solutions:

- ✓ Memoization: keep track of what f values have been computed
- ✓ Dynamic Programming: Compute all f values in a good order

CYK Algorithm

Let

$$x = x_1 x_2 \cdots x_n$$

be the string to be parsed.

Define

$$a(i,j) := \{ B \mid B \Rightarrow^* x_i x_{i+1} \cdots x_j \}$$

Then $x \in L(G)$ iff $S \in a(1,n)$

$$\begin{aligned} a(i,i) &= \{ C \mid C \to x_i \} \\ a(i,k) &= \{ C \mid C \to AB \wedge A \in a(i,j) \wedge B \in a(j+1,k) \} \end{aligned}$$

CYK "Wavefront" Matrix

a(1, 1)	a(1, 2)	a(1, 3)	a(1, 4)		a(1, n-1)	a(1, n)
	a(2, 2)	a(2, 3)	a(2, 4)		a(2, n-1)	a(2, n)
		a(3, 3)	a(3, 4)		a(3, n-1)	a(3, n)
			a(4, 4)			
					a(n-1, n-1)	a(n-1, n)
						a(n, n)

Each entry is computed from entries in its same row and column, e.g.
a(1, 4) from
a(1,1) and a(2, 4),
a(1, 2) and a(3, 4),
a(1, 3) and a(4, 4).

CYK Example

Let's parse `(()())` using the grammar

$$
\begin{aligned}
S &\rightarrow LT \\
T &\rightarrow SR \\
S &\rightarrow LR \\
S &\rightarrow SS \\
L &\rightarrow (\\
R &\rightarrow)
\end{aligned}
$$