Lecture 22

HW #22: 3.7.5 a

When we discussed the conversion of a context free grammar G into a PDA that would accept L(G), I mentioned that the operation of the PDA was like the operation of a parser – but not exactly.

Why not? Because a PDA is non-deterministic, and we don't want our parsers to be non-deterministic.

So what really goes on in parsing?

Want to build deterministic pushdown automata. We need to realize that not all CFLs can be accepted by deterministic pushdown automata. But (fortunately) programming language constructs generally can be.

A PDA is deterministic iff there is at most one transition applicable to each configuration.

Formal Def:
Two strings w1 and w2 are **consistent** if the first is a prefix of the second or vice versa.

Two transitions ((p1, w1, β1),(q1, γ1)) and ((p2, w2, β2),(q2, γ2)) are **compatible** iff p1 = p2, w1 and w2 are consistent, β 1 and β 2 are consistent.

i.e., there is a situation in which both transitions are applicable.

A PDA M is **deterministic** if it has no two distinct compatible transitions.

Example. Recall our PDA to accept the language $\{a^n c b^n, n \geq 0\}$

Let M = (K, Σ, Γ, Δ, s, F), where

Δ contains ((s,a,e),(s,a))
((s,c,e),(f,e))
((f,b,a),(f,e))

This PDA is deterministic. Those PDAs in which we "guessed" the middle of the string were not deterministic.

Not only do DPDAs know which transitions to make, but they also know when they are at the end of their input.

$L \subseteq \Sigma^*$ is a deterministic CFL iff $L\$ = L(M)$ for some DPDA M.

[Note that we need this – otherwise we would have a hard time recognizing $a^* \cup \{a^n b^n : n \geq 0\}$

Now…

We want to parse CFLs with a DPDA, but we can't always do it.
And here's some more bad news. Even if we have a deterministic CFL, if we apply the construction that gives us a PDA from a CFG, the PDA constructed will likely be non-deterministic.

Example.

Consider the following grammar that generates arithmetic expressions:

E -> [E B E]
E -> 2
B -> +
B -> *

E is the start symbol
$\Sigma = \{2, +, *, [,]\}$
$V = \{2, +, *, [,], E, B\}$

An example of a string generated by this grammar is:
[2 * [2 + 2]]

Using the algorithm for conversion of a grammar to a PDA, we get the following transitions:

((p, e, e), (q, E))
((q, e, E), (q, [EBE]))
((q, e, E), (q, 2))
((q, e, B), (q, +))
((q, e, B), (q, *))
((q, 2, 2), (q, e))
((q, *, *)), (q, e))

((q, +, +), (q, e))
((q, [, [), (q, e))
((q,],]), (q, e))

This PDA is non-deterministic: note transitions 2 and 3; also note transitions 4 and 5.

Fortunately, there are rules we can apply to help us fix this PDA. We can replace 2 and 3 by:

((q, [, e), ($q_{[}$, e))
(($q_{[}$, e, E), ($q_{[}$, [EBE]))
(($q_{[}$, e, [), (q, e))

and

((q, 2, e), (q_2, e))
((q_2, e, E), (q_2, 2))
((q_2, e, 2), (q, e))

We can similarly replace 4 and 5.

Now consider what happens when we parse [2 * [2 + 2]]

State	Input	Stack
p	[2 * [2 + 2]]	e
q	[2 * [2 + 2]]	E
$q_{[}$	2 * [2 + 2]]	E
$q_{[}$	2 * [2 + 2]]	[EBE]
q	2 * [2 + 2]]	EBE]
q_2	* [2 + 2]]	EBE]
q_2	* [2 + 2]]	2BE]

Etc.

The stack consists of pointers to nodes in a parse tree.

Each expansion on the stack is a leftmost expansion in the derivation.

Called a **top-down parser**. More specifically, called LL(1).
LL(k) stands for Left-to-right scan; leftmost derivation; k-symbol lookahead.

Note that the parser accepts/rejects the input string, but it also constructs a parse tree while it does so. The parse tree is useful for later steps in compilation.

But other problems can still arise. We used one character to disambiguate. What if it's still ambiguous? Consider the following rules:

<CondStmt> -> If <BoolExpr> Then <Stmt> Else <Stmt>
<CondStmt> -> If <BoolExpr> Then <Stmt>

Both have the same initial terminal symbol.

Need to do left-factoring:

<CondStmt> -> If <BoolExpr> Then <Stmt> <OptElse>
<OptElse> -> Else <Stmt>
<OptElse> -> e

But there are still more troubles to handle. Consider the following set of rules:

E -> E + T
E -> T
T -> T * F
T -> F
F -> 2

With corresponding transitions

((p, e, e), (e, E))
((q, e, E), (q, E + T))
((q, e, E), (q, T))
((q, e, T), (q, T * F))
((q, e, T), (q, F))
((q, e, F), (q, 2))

If you have input 2 + 2 + 2, what do you do? How do you choose which transition to apply? Can't do arbitrarily long lookahead.

The solution is to remove the left-recursion:

E -> T E'
E' -> + T E'

E' -> e

T -> F T'
T' ->* F T'
T' -> e

F -> 2

Now one-symbol lookahead will work.

Bottom-up parsing

Bottom-up parsing is a bit more complex, but it can be faster.

First, we need a new algorithm for converting grammars to PDAs:

((p, σ, e), (p, σ)), for each $\sigma \in \Sigma$. **Shift**
((p, e, α^R), (p, A)), if A -> α is a rule **Reduce**
((p, e, S), (q, e))

Example.

E -> E + T
E -> T
T -> a
T -> b

((p, a, e), (p, a))
((p, b, e), (p, b))
((p, +, e), (p, +))

((p, e, T+E), (p, E))
((p, e, T), (p, E))
((p, e, a), (p, T))
((p, e, b), (p, T))

((p, e, E), (q, e))

For b+b+a, consider the following (rightmost) derivation:

E => E+T => E+a => E+T+a => E+b+a => T+b+a => b+b+a

Our parser will simulate the rightmost derivation (backward) on the stack:

State	Input	Stack	
p	b+b+a	e	
p	+b+a	b	
p	+b+a	T	T->b
p	+b+a	E	E->T
p	b+a	+E	
p	+a	b+E	
p	+a	T+E	T->b

And so on.

But this is non-deterministic! How do you decide between shift and reduce?

(1) Need precedence relation to tell you which is appropriate based on next input and top of stack.
(2) When reducing, always reduce longest possible string on stack.

Yacc is based on LR(1) grammars.

LR(k) = left-to-right scan; rightmost derivation; k-symbol lookahead in precedence relation.