CS 241, Lecture 13 - Bottom Up Parsing

1 Warm-up Problem

Given:

$$S' \to \vdash S \dashv \tag{1}$$
$$S \to c \tag{2}$$
$$S \to QRS \tag{3}$$
$$R \to \epsilon \tag{4}$$
$$R \to b \tag{5}$$
$$Q \to R \tag{6}$$
$$Q \to d \tag{7}$$

And:

	Nullable	First	Follow
S	F	$\{\vdash\}$	$\{\}$
S	F	$\{b, c, d\}$	$\{\dashv\}$
Q	T	$\{b, d\}$	$\{b, c, d\}$
R	T	$\{b\}$	$\{b, c, d\}$

Then our predict table is:

	$\vdash$	b	c	d	$\dashv$
S'	$\{0\}$				
S	$\{2\}$	$\{1, 2\}$	$\{2\}$		
Q		$\{5\}$	$\{5\}$	$\{5, 6\}$	
R		$\{3, 4\}$	$\{3\}$	$\{3\}$	

2 Top Down Parsing (cont.)

Recap of $LL(1)$:

- A grammar is $LL(1)$ iff:
  - No two distinct products with the same LHS can generate the same first terminal symbol
  - No nullable symbol A has the same terminal symbol a in both its first and follow sets

- There is only one way to send a nullable symbol to ϵ

- The warm-up problem shows that not all examples are $LL(1)$.
- But we can *convert* languages to $LL(1)$. For example, given:

$$S \to S + T \quad (1)$$
$$S \to T \quad (2)$$
$$T \to T * F \quad (3)$$
$$T \to F \quad (4)$$
$$F \to a|b|c|(S) \quad (5)$$

  This is NOT $LL(1)$. This is because it is left recursive, and left recursive grammars are **always** not $LL(1)$.

- For example, we can generate two derivations that have a as the predict character.
- We fix this by converting it to a *right* recursive grammar. For a left recursive grammar, say:

$$A \to A\alpha|\beta$$

  where β does not begin with the non-terminal A, we remove this rule from our grammar and replace it with:

$$A \to \beta A'$$
$$A' \to \alpha A'|\epsilon$$

  We can see that there are no left recursive rules!

- We apply this to our earlier example:

$$S \to TZ' \quad (1)$$
$$Z' \to +TZ'|\epsilon \quad (2,3)$$
$$T \to FT' \quad (4)$$
$$T' \to *FT'|\epsilon \quad (5,6)$$
$$F \to a|b|c|(S) \quad (7,8,9)$$

- But... right recursive grammars are not always $LL(1)$, either. Consider:

$$S \to T + S \quad (1)$$
$$S \to T \quad (2)$$
$$T \to F * T \quad (3)$$
$$T \to F \quad (4)$$
$$F \to a|b|c|(S) \quad (5)$$

  The above rule still suffers with $\{1, 2\} \subseteq Predict(S, a)$.

- This is as they both start with T.
- We apply a process known as factoring. Given a rule:

  $$A \to \alpha\beta_1 | \ldots | \alpha\beta_n | \gamma$$

  where $\alpha \neq \epsilon$ and γ is representative of other productions that do not begin with α, we can change this to the following equivalent grammar by **left factoring** (think literal factoring, we are "factoring" out α):

  $$A \to \alpha B | \gamma$$
  $$B \to \beta_1 | \ldots | \beta_n$$

- Applying this to our example:

  $$S \to TZ' \quad (1)$$
  $$Z' \to +S | \epsilon \quad (2)$$
  $$T \to FT' \quad (3)$$
  $$T' \to \epsilon | * T \quad (4)$$
  $$F \to a | b | c | (S) \quad (5)$$

  Now we don't have multiple situations of S with a.

- Note: Not all grammars can be converted to $LL(1)$, though if they are left recursive or can be factored, we now know how to make them $LL(1)$.

We can use the following cheat sheet for first, follow, nullable, and predict:

Nullable:
- $A \to \epsilon$ implies that Nullable(A) = true. Further Nullable(ϵ) = true.
- If $A \to B_1 ... B_n$ and each of Nullable(B_i) = true then Nullable(A) = true.

First:
- $A \to a\alpha$ then $a \in$ First(A)
- $A \to B_1 ... B_n$ then First(A) = First(A) $\cup$ First(B_i) for each $i \in \{1, .., n\}$ until Nullable(B_i) is false.

Follow:
- $A \to \alpha B \beta$ then Follow(B) = First(β)
- $A \to \alpha B \beta$ and Nullable(β) = true, then Follow(B) = Follow(B) $\cup$ Follow(A)

$$\text{Predict}(A, a) = \{A \to \beta : a \in \text{First}(\beta)\} \cup \{A \to \beta : \beta \text{ is nullable and } a \in \text{Follow}(A)\}$$

=====End of midterm content=====

3 ““““Fun”””””

Is there a grammar that is not $LL(k)$ for any k?

Consider $L = \{a^n b^m : n \geq m \geq 0\}$. We note that this is **not** $LL(k)$ for any k. If we consider $w = a^{k+1}b^k$, then you would need a $k+1$ look ahead.

Let us try to create two CFGs that recognize this language, one that is ambiguous and one that isn't. Remember ambiguity is when you can form the same "word" multiple ways.

An ambiguous example is:

$$S \to \epsilon \tag{1}$$
$$S \to aS \tag{2}$$
$$S \to aSb \tag{3}$$
$$\tag{4}$$

A non-ambiguous example is:

$$S \to aS \tag{1}$$
$$S \to B \tag{2}$$
$$B \to aBb \tag{3}$$
$$B \to \epsilon \tag{4}$$

4 Bottom-up Parsing

Let us consider the following grammar:

$$S' \to \vdash S \dashv \quad (1)$$
$$S \to AcB \quad (2)$$
$$A \to ab \quad (3)$$
$$A \to ff \quad (4)$$
$$B \to def \quad (5)$$
$$B \to ef \quad (6)$$

We want to process the word $w = \vdash abcdef \dashv$ using bottom-up parsing:

Stack	Read	Processing	Action
	ϵ	$\vdash abcdef \dashv$	Shift $\vdash$
$\vdash$	$\vdash$	$abcdef \dashv$	Shift a
$\vdash a$	$\vdash a$	$bcdef \dashv$	Shift b
$\vdash ab$	$\vdash ab$	$cdef \dashv$	Note this is a rule, reduce w/ (2); pop b, a, push A
$\vdash A$	$\vdash ab$	$cdef \dashv$	Shift c
$\vdash Ac$	$\vdash abc$	$def \dashv$	Shift d
$\vdash Acd$	$\vdash abcd$	$ef \dashv$	Shift e
$\vdash Acde$	$\vdash abcde$	$f \dashv$	Shift f
$\vdash Acdef$	$\vdash abcdef$	$\dashv$	This is now rule 4. Reduce (4), pop f, d, e, push B
$\vdash AcB$	$\vdash abcdef$	$\dashv$	This is now rule 1, reduce; pop B, c, A, push S
$\vdash S$	$\vdash abcdef$	$\dashv$	Shift $\dashv$
$\vdash S \dashv$	$\vdash abcdef \dashv$	ϵ	This is now rule 0. Reduce, pop $\dashv, S, \vdash$; push S'
S'	$\vdash abcdef \dashv$	ϵ	Accept and terminate

But when do we shift or reduce? Knuth has a theorem: The set $\{wa : \exists xs.t.S \Rightarrow^* wax\}$ is a regular language. This means we can use DFAs on CFGs! For example, consider the following CFG:

$$S' \to \vdash S \dashv \quad (1)$$
$$S \to S + T \quad (2)$$
$$S \to T \quad (3)$$
$$T \to d \quad (4)$$

We construct a DFA based on this grammar. But first, we define an **item** is a production with a dot $\cdot$ somewhere on the right hand side of a rule. This indicates a partially completed rule, and we begin in the state labelled $S' \to \cdot \vdash S \dashv$.