Practice

Construct the four tables (Nullable, First, Follow and Predict) for the following examples:

G_1

$$
\begin{array}{lcll}
S' & \to & \vdash S \dashv & (0) \\
S & \to & Bb & (1) \\
S & \to & Cd & (2) \\
B & \to & aB & (3) \\
B & \to & \varepsilon & (4) \\
C & \to & cC & (5) \\
C & \to & \varepsilon & (6)
\end{array}
$$

G_2

$$
\begin{array}{lcll}
S' & \to & \vdash S \dashv & (0) \\
S & \to & TZ' & (1) \\
Z' & \to & +TZ' \mid \varepsilon & (2,3) \\
T & \to & FT' & (4) \\
T' & \to & *FT' \mid \varepsilon & (5,6) \\
F & \to & a \mid b \mid c & (7,8,9)
\end{array}
$$

For G_1

	Nullable	First	Follow
S'	False	$\{\vdash\}$	$\{\}$
S	False	$\{a, b, c, d\}$	$\{\dashv\}$
B	True	$\{a\}$	$\{b\}$
C	True	$\{c\}$	$\{d\}$

Predict

	$\vdash$	a	b	c	d	$\dashv$
S'	0					
S		1	1	2	2	
B		3	4			
C				5	6	

For G_2

	Nullable	First	Follow
S′	False	$\{\vdash\}$	$\{\}$
S	False	$\{a, b, c\}$	$\{\dashv\}$
Z′	True	$\{+\}$	$\{\dashv\}$
T	False	$\{a, b, c\}$	$\{\dashv, +\}$
T′	True	$\{*\}$	$\{\dashv, +\}$
F	False	$\{a, b, c\}$	$\{\dashv, +, *\}$

Predict (Recall: Nullable(ε) is true).

	$\vdash$	a	b	c	$+$	$*$	$\dashv$
S′	0						
S		1	1	1			
Z′					2		3
T		4	4	4			
T′					6	5	6
F		7	8	9			

CS 241 Lecture 13

Bottom-Up Parsing

With thanks to Brad Lushman, Troy Vasiga, Kevin Lanctot, and Carmen Bruni

Cheat Sheet and Examples

Nullable:

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

First:

- $A \to a\alpha$ then $a \in$ First(A)
- $A \to B_1 \ldots B_n$ 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)

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

Practice With Realistic Grammars

- Arithmetic expressions(ish):
  - $S' \rightarrow \vdash S \dashv$ *(0)*
    $S \rightarrow a \mid b \mid c \mid (SRS)$ *(1,2,3,4)*
    $R \rightarrow + \mid - \mid * \mid /$ *(5,6,7,8)*
  - Based on your predict table, is this grammar LL(1)?

Practice With Realistic Grammars

- Arithmetic expressions (better):
  - $S' \to \vdash S \dashv$ *(0)*
    $S \to S+S \mid S-S \mid S*S \mid S/S$ *(1,2,3,4)*
    $S \to V$ *(5)*
    $V \to a \mid b \mid c \mid (S)$ *(6,7,8,9)*
  - Based on your predict table, is this grammar LL(1)?

Recap of LL(1)

A grammar is $LL(1)$ if and only if:

- no two distinct productions 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 for distinct production rules.
- there is only one way to send a nullable symbol to ε.

Grammars for L

Ambiguous

$$
\begin{aligned}
S &\rightarrow \varepsilon \\
S &\rightarrow a\,S\,S \\
&\rightarrow a\,S\,b
\end{aligned}
$$

Unambiguous

$$
\begin{aligned}
S &\rightarrow a\,S \\
S &\rightarrow B \\
B &\rightarrow a\,B\,b \\
B &\rightarrow \varepsilon
\end{aligned}
$$

A Classic Example

- The previous example has shown us that not all examples are *LL(1)*.
- Suppose we have a grammar that is not LL(1). Can we convert it to become *LL(1)*?

A Classic Example

- Sometimes. Let's see an example:

$S \rightarrow S + T$	*(1)*
$S \rightarrow T$	*(2)*
$T \rightarrow T * F$	*(3)*
$T \rightarrow F$	*(4)*
$F \rightarrow a \mid b \mid c \mid (S)$	*(5,6,7,8)*

This grammar is not *LL*(1). Why?

Primary Issue

With this grammar (Recall: This respected BEDMAS):

$S \rightarrow S + T$	*(1)*
$S \rightarrow T$	*(2)*
$T \rightarrow T * F$	*(3)*
$T \rightarrow F$	*(4)*
$F \rightarrow a \mid b \mid c \mid (S)$	*(5,6,7,8)*

The primary issue is that left recursion is at odds with *LL*(1). In fact, left recursive grammars are always **not** LL(1). For example, Examine the derivations for a and $a+b$ below:

$$S \Rightarrow S+T \Rightarrow T+T \Rightarrow F+T \Rightarrow a+T \Rightarrow a+b\; S \Rightarrow T$$
$$\Rightarrow F \Rightarrow a$$

Notice that they have the same first character but required different starting rules from S. That is $\{1, 2\} \subseteq$ Predict(S, a). Our first step is to at least make this right recursive instead.

General Right-Recursive Idea

g253wang

To make a [direct] left recursive grammar right recursive; say

$$A \rightarrow A\alpha \mid \beta$$

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

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \mid \varepsilon \end{aligned}$$

g253wang

g253wang g253wang g253wang

Right Recursive

The above solves our issue

$$
\begin{array}{lcll}
S & \to & T\,Z' & (1) \\
Z' & \to & +T\,Z' \mid \varepsilon & (2,3) \\
T & \to & F\,T' & (4) \\
T' & \to & *F\,T' \mid \varepsilon & (5,6) \\
F & \to & a \mid b \mid c \mid (S) & (7,8,9,10)
\end{array}
$$

we get a right-recursive grammar. This is $LL(1)$ (Exercise).

However: recall that we didn't want these grammars, because they were right associative! This is an issue we need to resolve with new techniques.

Right Recursive Grammars

However, not all right recursive grammars are *LL*(1)! Consider

$S \to T + S$	*(1)*
$S \to T$	*(2)*
$T \to F * T$	*(3)*
$T \to F$	*(4)*
$F \to a \mid b \mid c \mid (S)$	*(5,6,7,8)*

we get a right-recursive grammar. But not *LL*(1)!

$$S \Rightarrow T + S \Rightarrow F + S \Rightarrow a + S \Rightarrow a + T \Rightarrow a + b\,S \Rightarrow T$$
$$\Rightarrow F \Rightarrow a$$

Again, we have $\{1, 2\} \subseteq \text{Predict}(S, a)$. However, with this there is still hope. We can apply a process known as **factoring**.

Left Factoring

Idea: If $A \to \alpha\beta_1 | \ldots | \alpha\beta_n | \gamma$ where $\alpha \neq \varepsilon$ and γ is representative of other productions that do not begin with α, then we can change this to the following equivalent grammar by **left factoring**:

$$\begin{aligned} A. &\to \alpha B | \gamma \\ B. &\to \beta_1 | \; \ldots \; | \beta_n \end{aligned}$$

In this way, we remove the issues on the previous slide.

Right Recursive and Left Factored

Applying this technique to the previous example:

$$
\begin{array}{lcll}
S & \rightarrow & T\,Z' & (1) \\
Z' & \rightarrow & \varepsilon \,|\, +S & (2,3) \\
T & \rightarrow & FT' & (4) \\
T' & \rightarrow & \varepsilon \,|\, *\,T & (5,6) \\
F & \rightarrow & a \,|\, b \,|\, c \,|\, (S) & (7,8,9,10)
\end{array}
$$

we can get an $LL(1)$ grammar. The take-away from these last few slides is that $LL(1)$ is not compatible with a left-recursive grammar, or with grammars with left-ambiguous productions, but we want both of these for meaningful parse trees.

There is one other situation we can attempt to resolve, and that is the situation where a rule of the form $A \rightarrow \varepsilon$ causes the ambiguity. I will leave this as an example to consider.

Recursive-Descent Parsing

- Fixing the parse trees from right-recursive and left-factored grammars is the #1 thing that recursive-descent ad hoc solutions fix
- The actual sequence of steps is LL(1), but then they generate a different parse tree by changing it on the fly
- In reality, most parsers generate *abstract* syntax trees, which is why this is ad hoc, instead of using a formalized solution

Bottom-Up Parsing

Recall: Determining the α_i in $S \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow w$

- Idea: Instead of going from *S* to *w* , let's try to go from *w* to *S*.
- Overall idea: look for the RHS of a production, replace it with the LHS. When you don't have enough for a RHS, read more input. Keep grouping until you reach the start state.

Bottom-Up Parsing

- Our stack this time will store the α_i in reverse order (Contrast to top-down which stores the α_i in order!)
- Our invariant here will be Stack + Unread Input = α_i . (Contrast to top-down where invariant was consumed input + reversed Stack contents = α_i .)

Bottom-Up Parsing with Faerie Magic

Algorithm Bottom-up parsing with faerie magic

for each symbol a in the input from left to right **do**
    ask a magical faerie to tell us whether to shift, reduce, or reject, and with which production to reduce if we should reduce
    while the faerie tells us to reduce with some $B \to \gamma$ **do**
        stack.pop symbols in γ
        stack.push B
    end while
    if the faerie told us to reject **then**
        reject
    end if
    stack.push a
end for
accept

Example

Recall our grammar:

$$
\begin{array}{lcll}
S' & \rightarrow & \vdash S \dashv & (0) \\
S & \rightarrow & AcB & (1) \\
A & \rightarrow & ab & (2) \\
A & \rightarrow & ff & (3) \\
B & \rightarrow & def & (4) \\
B & \rightarrow & ef & (5)
\end{array}
$$

We wish to process $w = \ \vdash abcdef \dashv$ using this bottom-up technique.

Parsing Bottom-Up

Stack	Read	Processing	Action
	ε	$\vdash abcdef \dashv$	Shift $\vdash$
$\vdash$	$\vdash$	$abcdef \dashv$	Shift a
$\vdash a$	$\vdash a$	$bcdef \dashv$	Shift b
$\vdash ab$	$\vdash ab$	$cdef \dashv$	Reduce (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
$\vdash Acdef$	$\vdash abcdef$	$\dashv$	Reduce (4); pop f, d, e push B
$\vdash AcB$	$\vdash abcdef$	$\dashv$	Reduce (1); pop B, c, A push S
$\vdash S$	$\vdash abcdef$	$\dashv$	Shift $\dashv$
$\vdash S \dashv$	$\vdash abcdef \dashv$	ε	Reduce (0); pop $\dashv$, S, $\vdash$ push S'
S'	$\vdash abcdef \dashv$	ε	Accept

Major Theorem

Theorem (Knuth 1965)

For any grammar G, the set of viable prefixes (stack configurations), namely

$$\{\alpha a : \alpha \in V^* \text{ is a stack} \\ a \in \Sigma \text{ is the next character} \\ \exists\, x \in \Sigma^* \text{ such that } S \Rightarrow^* \alpha a x\}$$

is a regular language, and the NFA accepting it corresponds to items of G! (Recall $V = N \cup \Sigma$). Converting this NFA to a DFA gives a machine with states that are the set of valid items for a viable prefix!

We will show how to use this theorem to create a LR(0), SLR(1) and LR(1) automata to help us accept the words generated by a grammar.

An Example

Consider the following context-free grammar:

$$
\begin{array}{llll}
S' & \rightarrow & \vdash S \dashv & (0) \\
S & \rightarrow & S + T & (1) \\
S. & \rightarrow & T & (2) \\
T. & \rightarrow & d & (3)
\end{array}
$$

We'll construct the LR(0) automaton associated with this grammar.