Practice

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

G_1

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

G_2

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

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 and Kevin Lanctot

Cheat Sheet and Examples

Nullable:

- $A \to \epsilon$ 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)

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

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.

Just For Fun (Thanks to Troy Vasiga and Kevin Lanctot for this!)

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

Just For Fun (Thanks to Troy Vasiga and Kevin Lanctot for this!)

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

Consider $L = \{a^n b^m : n \geq m \geq 0\}$ (language with number of as is greater than or equal to the number of bs).

This is not $LL(k)$ for any k! (Consider $w = a^{k+1}b^k$ vs $x = a^{k+1}b$ which rules to use to reduce would need at least a $k+1$ look ahead). In fact there are ambiguous and unambiguous examples!

Create two CFGs that recognize this language; one ambiguous and one not.

Grammars for L

Ambiguous

$$\begin{array}{rcl} S & \rightarrow & \epsilon \\ S & \rightarrow & aS \\ S & \rightarrow & aSb \end{array}$$

Unambiguous

$$\begin{array}{rcl} S & \rightarrow & aS \\ S & \rightarrow & B \\ B & \rightarrow & aBb \\ B & \rightarrow & \epsilon \end{array}$$

A Classical 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 Classical 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 $LL(1)$?
- Sometimes. Let's see an example:

$$
\begin{array}{lcll}
S & \rightarrow & S{+}T & (1) \\
S & \rightarrow & T & (2) \\
T & \rightarrow & T*F & (3) \\
T & \rightarrow & F & (4) \\
F & \rightarrow & a \mid b \mid c \mid (S) & (5,6,7,8)
\end{array}
$$

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

Primary Issue

With this grammar (Recall: This respected BEDMAS):

$$
\begin{array}{lcll}
S & \to & S{+}T & (1) \\
S & \to & T & (2) \\
T & \to & T{*}F & (3) \\
T & \to & F & (4) \\
F & \to & a \mid b \mid c \mid (S) & (5,6,7,8)
\end{array}
$$

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 $\{1, 2\} \subseteq \text{Predict}(S, a)$. Our first step is to at least make this right recursive instead.

General Right Recursive Idea

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

$$A \;\rightarrow\; A\alpha \,|\, \beta$$

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

$$\begin{array}{lcl} A & \rightarrow & \beta A' \\ A' & \rightarrow & \alpha A' \,|\, \epsilon \end{array}$$

Right Recursive

The above solves our issue

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

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. Note in this example though, associativity is not an issue (why?)

Right Recursive Grammars are Still Mortal!

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

$$\begin{array}{lcll} S & \rightarrow & T{+}S & (1) \\ S & \rightarrow & T & (2) \\ T & \rightarrow & F{*}T & (3) \\ T & \rightarrow & F & (4) \\ F & \rightarrow & a \mid b \mid c \mid (S) & (5,6,7,8) \end{array}$$

we get a right recursive grammar. However this one is 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 $A \rightarrow \alpha\beta_1|...|\alpha\beta_n|\gamma$ where $\alpha \neq \epsilon$ 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**:

$$\begin{array}{lcl} A & \rightarrow & \alpha B \,|\, \gamma \\ B & \rightarrow & \beta_1 \,| \quad \ldots \,|\, \beta_n \end{array}$$

In this way, we remove the issues on the previous slide. We can factor to get the following grammar:

Right Recursive and Left Factored

Applying this technique to the previous example:

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

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-associative grammar and we want left-associative grammars to help with meaning.

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

Bottom Up Parsing

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

- Idea: Instead of going from S to w, let's try to go from w to S.
- 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.)

Example

Recall our grammar:

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

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

Notes On Bottom-Up Parsing

- Accept if and only if stack contains S' and input is ϵ.
- At each step, need to either shift a character from the stack OR reduce if the top of the stack is the right hand side of a grammar rule (then replace top of stack elements with the left hand side of the aforementioned rule)
- When should we shift vs reduce?
- In the previous example, we had a concern about which rule to use to reduce. How can we resolve this issue? (Turns out we don't need a look ahead in the above language but this is perhaps not intuitive yet)

Ideas

Could try to use the next character(s) of input to help decide when to shift or reduce like with LL(1) but the problem is still tricky. However Knuth in his paper *On the Translation of Languages from Left to Right* (1965) accomplished a few major milestones:

- Defined LR(k) grammars; grammars that can always be deterministically parsed left to right with right most derivations and a k symbol lookahead in the processed string.
- Argued that for such languages, parsing can be done efficiently (roughly linear with length of string).
- Argued that a language generated by a LR(k) grammar can also be generated by some LR(1) grammar
- and finally...

Major Theorem

Theorem (Knuth 1965)

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

$$\begin{aligned}\{\alpha a : &\alpha \in V^* \text{ is a stack}\\ &a \in \Sigma' \text{ is the next character}\\ &\exists x \in (\Sigma')^* \text{ such that } S \Rightarrow^* \alpha a x\}\end{aligned}$$

is a regular language and the NFA accepting it corresponds to items of G! (Recall $V = (N' \cup \Sigma')$, $N' = N \cup \{S'\}$ and $\Sigma' = \Sigma \cup \{\vdash, \dashv\}$). 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.

Summary

- Creating the associated DFA can help us to make shift or reduce decisions.
- The resulting method is called an LR scan
  - Left to right scanning
  - Do rightmost derivations.
- We will start off with an LR(0) automaton.

An Example (Thanks to Brad and Kevin for this example)

Consider the following context-free grammar:

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

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

Notation and Procedure

Definition

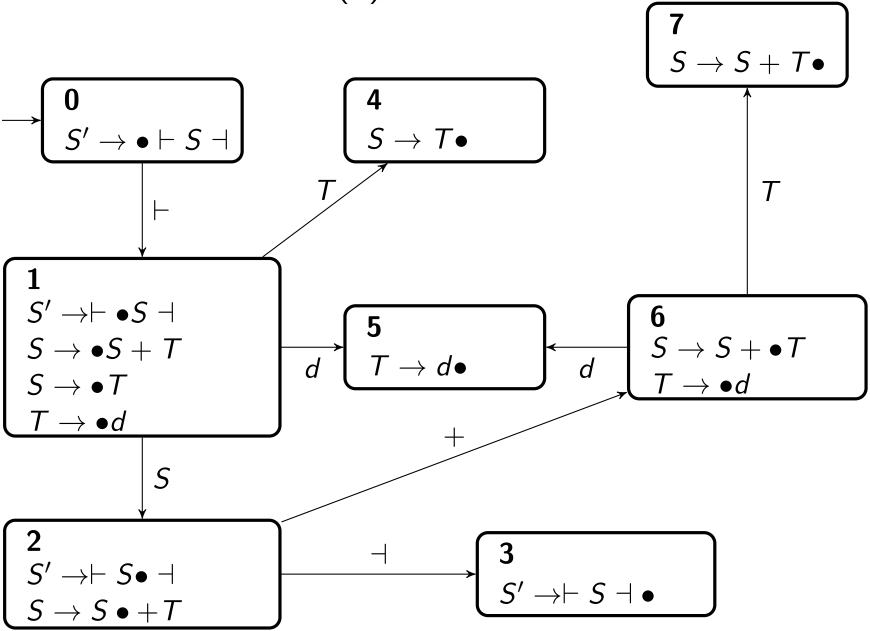
An **item** is a production with a dot $\bullet$ somewhere on the right hand side of a rule.

- Items indicate a partially completed rule.
- We will begin in a state labelled by the rule $S' \to \bullet \vdash S \dashv$.

LR(0) Construction

- From a state, for each rule in the state, move the dot forward by one character. The transition function is given by the symbol you jumped over.
- For example, with $S' \to \bullet \vdash S \dashv$, we move the $\bullet$ over $\vdash$. Thus, the transition function will consume the symbol $\vdash$.
- The state we end up in will contain the transition $S' \to \vdash \bullet S \dashv$. It also contains more!
- In the new state, if in the set of rules we have $\bullet A$ for some non-terminal A, we then add all rules with A in the left hand side of a production with a dot preceding the right hand side!
- In this case, this state will include the rules $S \to \bullet S + T$ and $S \to \bullet T$.
- Notice now we also have $\bullet T$ and so we also need to include the rules where T is the left hand side adding the rule $T \to \bullet d$.
- We continue on with these rules to get the final DFA.

New LR(0) Automaton



Using the Automaton

- Start in the start state with an empty stack.
- Shift:
  - Shift a character from input to the stack.
  - Follow any transition that follows with the character as a label
  - If none, reduce or give an error
- Reduce:
  - Reduce states have only one item and the $\bullet$ is in the rightmost position.
  - Reduce the rule in the state: Pop the RHS off the stack and backtrack in your DFA the number of states corresponding to the number of elements in the RHS. Then follow the transition for the LHS and push the LHS on your stack.
- Accept if S' is on the stack and the input is empty.

NOTE: Because we need to backtrack, we also need to push the DFA states on the stack as well! (so really accept when stack has only the start state).

Example

Consider processing $\vdash d + d + d \dashv$