Warm Up Problem

Draw a parse tree for $a + b * c$ in

$$
\begin{array}{lcl}
S & \rightarrow & S + T \,|\, T \\
T & \rightarrow & T * F \,|\, F \\
F & \rightarrow & a \,|\, b \,|\, c
\end{array}
$$

CS 241 Lecture 11

Top Down Parsing, First and Follow

With thanks to Brad Lushman, Troy Vasiga and Kevin Lanctot

Formally

Given: a CFG $G = (N, \Sigma, P, S)$ and a terminal string $w \in \Sigma^*$
Find: The derivation, that is, the steps such that $S \Rightarrow ... \Rightarrow w$ or prove that $w \notin L(G)$.

How Do We Find This Derivation?

Two ideas:

- Forwards (Top-Down Parsing). Start with S and then try to get to w (eg. LL(1))
- Backwards (Bottom-Up Parsing) Start with w and figure out how we could have gotten to w. (Eg. LR(0) and SLR(1))
- Neither seems like a great option...

Top-Down Parsing

- We will start with S and store intermediate derivations in a stack and then match characters to w.
- Every time we pop from the stack, we will have that consumed input plus reverse of stack is equal to a intermediate step in our derivation, that is, such a step is an α_i where $S \Rightarrow ... \Rightarrow \alpha_i \Rightarrow ... \Rightarrow w$

Top-Down Parsing

- In these procedures, we will actually augment our grammar to include $\vdash$ and $\dashv$ symbolizing the beginning and end of the file respectively. We also include a new start state S' to begin our parsing.
- Thus, our original CFG $G = (N, \Sigma, P, S)$ becomes $G' = (N', \Sigma', P', S')$ where

$$N' = N \cup \{S'\}$$
$$\Sigma' = \Sigma \cup \{\vdash, \dashv\}$$
$$P' = P \cup \{S' \rightarrow \vdash S \dashv\}$$

Algorithm (Note: Doesn't Always Work!)

Algorithm 1 Top-Down Parsing

1: Push S' onto the stack
2: **while** stack is non-empty **do**
3: α = pop from stack
4: **if** $\alpha \in N \cup \{S'\}$ **then**
5: Push symbols of β of a valid production rule $\alpha \rightarrow \beta$ in reverse order on the stack (note derivation)
6: **else**
7: c = read_char()
8: **if** $c \neq \alpha$ **then**
9: Reject
10: **end if**
11: **end if**
12: **end while**
13: **if** read_char() = EOF **then**
14: Accept
15: **else**
16: Reject
17: **end if**

Example

Let us determine whether or not $w = abcdef$ is inside $L(G)$ where $G = (\{S, A, B\}, \{a, b, c, d, e, f\}, P, S)$ is defined with P given by:

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

Example

Let us determine whether or not $w = abcdef$ is inside $L(G)$ where $G = (\{S, A, B\}, \{a, b, c, d, e, f\}, P, S)$ is defined with P given by:

$$
\begin{array}{lll}
S & \rightarrow & AcB \qquad (1) \\
A & \rightarrow & ab \qquad (2) \\
A & \rightarrow & ff \qquad (3) \\
B & \rightarrow & def \qquad (4) \\
B & \rightarrow & ef \qquad (5)
\end{array}
$$

First, augment the grammar with

$$S' \rightarrow \vdash S \dashv \qquad (0)$$

and look for $w = \ \vdash abcdef \dashv$ in this augmented grammar.

Parsing

Stack	Read	Processing	Action
S'	ϵ	$\vdash abcdef \dashv$	Pop S', push $\dashv$, S, $\vdash$ Rule (0)
$\dashv S \vdash$	ϵ	$\vdash abcdef \dashv$	Match $\vdash$
$\dashv S$	$\vdash$	$abcdef \dashv$	Pop S push B, c, A Rule (1)
$\dashv BcA$	$\vdash$	$abcdef \dashv$	Pop A push b, a Rule(2)
$\dashv Bcba$	$\vdash$	$abcdef \dashv$	match a
$\dashv Bcb$	$\vdash a$	$bcdef \dashv$	match b
$\dashv Bc$	$\vdash ab$	$cdef \dashv$	match c
$\dashv B$	$\vdash abc$	$def \dashv$	Pop B, push f, e, d Rule (4)
$\dashv fed$	$\vdash abc$	$def \dashv$	match d
$\dashv fe$	$\vdash abcd$	$ef \dashv$	match e
$\dashv f$	$\vdash abcde$	$f \dashv$	match f
$\dashv$	$\vdash abcdef$	$\dashv$	match $\dashv$
ϵ	$\vdash abcdef \dashv$	ϵ	Accept (stack = input = ϵ)

Outstanding Questions

- When you popped A, you had multiple choices to pick - how do we know which one to take?

Outstanding Questions

- When you popped A, you had multiple choices to pick - how do we know which one to take?
- Could try all such combinations (way too expensive; want procedure to be deterministic).

Outstanding Questions

- When you popped A, you had multiple choices to pick - how do we know which one to take?
- Could try all such combinations (way too expensive; want procedure to be deterministic).
- Solution: Use a single character lookahead to determine where to go!
- We construct a *predictor table* to tell us where to go; given non-terminal on stack and input symbol, what rule should we use? (see next slide).
- Rows correspond to elements of N' and columns correspond to elements of Σ'.

Predictor Table for Grammar

	$\vdash$	a	b	c	d	e	f	$\dashv$
S'	{0}							
S		{1}					{1}	
A		{2}					{3}	
B					{4}	{5}		

- Numbers tell program what rule to follow.
- Empty cells are always ERROR states.

The Good, The Bad and The Ugly

- Good news is we can have extremely descriptive error messages:

  ERROR in row X column Y, no look ahead available. Expecting one of a, b, ...

- Bad news is this doesn't work well if an element of the aforementioned table contains more than one element (for example, if we added $A \rightarrow adf$ to our production rule set P)
- The ugly news is we can't always turn grammars into a suitable format where we can guarantee only one element in the table. Nonetheless let's try to anyway.

LL(1)

Definition

A grammar is called **LL(1)** if and only if each cell of the predictor table contains at most one entry.

For an $LL(1)$ grammar, we can drop the set notation from the predictor table.

Why is it called LL(1)?

- First L: Scan left to right
- Second L: Leftmost derivations
- Number of symbol lookahead: 1

Constructing the Lookahead Table

Our goal is the following function which is our predictor table (that is, the table is really a set of productions - note this will get updated!):

> *Predict*(A, a)*: production rule(s) that apply when* $A \in N'$ *is on the stack,* $a \in \Sigma'$ *is the next input character.*

To do this, we also introduce the following function, First$(\beta) \subseteq \Sigma'$:

> *First*(β)*: set of characters that can be the first letter of a derivation starting from* $\beta \in V^*$*.*

More formally:

$$\textit{Predict}(A, a) = \{A \to \beta : a \in \text{First}(\beta)\}$$
$$\textit{First}(\beta) = \{a \in \Sigma' : \beta \Rightarrow^* a\gamma, \text{ for some } \gamma \in V^*\}$$

Note Predict as defined above is incorrect! Why? More later - first an example!

Example of First

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

- $\text{First}(S') = \{\vdash\}$
- $\text{First}(S) = \{a, f\}$
- $\text{First}(A) = \{a, f\}$
- $\text{First}(B) = \{d, e\}$

Notice that to compute some of the first values, you have to follow up on subsequent non-terminal symbols (as was the case for $\text{First}(S)$).

Also $\text{First}(a\gamma) = a$ for all $a \in \Sigma$ and all $\gamma \in V^*$.

Issue

What is the problem with

$$\text{Predict}(A, a) = \{A \to \beta : \beta \Rightarrow^* a\gamma, \text{ for some } \beta, \gamma \in V^*\}$$

Issue

What is the problem with

$$\text{Predict}(A, a) = \{A \to \beta : \beta \Rightarrow^* a\gamma, \text{ for some } \beta, \gamma \in V^*\}$$

The problem with

$$\text{Predict}(A, a) = \{A \to \beta : \beta \Rightarrow^* a\gamma, \text{ for some } \beta, \gamma \in V^*\}$$

is that it is entirely possible that $A \Rightarrow^* \epsilon$! This would mean that the a didn't come from A but rather some symbol *after* A!

Example $S' \Rightarrow\vdash abc \dashv$

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)
B	$\rightarrow \epsilon$	(6)

Notice now that

$$
\begin{aligned}
S' &\Rightarrow\vdash S \dashv \\
&\Rightarrow\vdash AcB \dashv \\
&\Rightarrow\vdash abcB \dashv \\
&\Rightarrow\vdash abc \dashv
\end{aligned}
$$

- In the top-down parsing algorithm, when we reach $\vdash abcB \dashv$ the stack is $\dashv B$ and remaining input is $\dashv$.
- We look at Predict$(B, \dashv)$ which is empty since $\dashv \notin$ First(B)! Thus we reach an error!

Augment Predict Table

We augment our predict table to include the elements that can *follow* a non-terminal symbol.

In this case, we need to include that $\dashv \in$ Predict$(B, \dashv)$:

	$\vdash$	a	b	c	d	e	f	$\dashv$
S'	{0}							
S		{1}					{1}	
A		{2}					{3}	
B					{4}	{5}		{**6**}

Note also, this is the only new entry.

Correction

The issue in the previous slide is entirely centred around the fact that $B \Rightarrow^* \epsilon$ (in fact $B \Rightarrow \epsilon$).

To correct this, we introduce two new functions.

> *Nullable(β): boolean function; for $\beta \in V^*$ is true if and only if $\beta \Rightarrow^* \epsilon$.*
> *Follow(A): for any $A \in N'$, this is the set of elements of Σ' that can come immediately after A in a derivation starting from S'.*

More formally:

> *Nullable(β)* = true iff $\beta \Rightarrow^* \epsilon$ and false otherwise
> *Follow(A)* $= \{b \in \Sigma' : S' \Rightarrow^* \alpha A b \beta$ for some $\alpha, \beta \in V^*\}$

Definition

We say that that a $\beta \in V^*$ is **nullable** if and only if Nullable(β) = true.

Example of First

$$
\begin{array}{lll}
S' \rightarrow \vdash S \dashv & (0) \\
S \rightarrow AcB & (1) \\
A \rightarrow ab & (2) \\
A \rightarrow ff & (3) \\
B \rightarrow def & (4) \\
B \rightarrow ef & (5) \\
B \rightarrow \epsilon & (6)
\end{array}
$$

- Follow(S') = $\{\}$ (Always!)
- Follow(S) = $\{\dashv\}$
- Follow(A) = $\{c\}$
- Follow(B) = $\{\dashv\}$

Note

What happens with Predict(A, a) if Nullable$(A) = \mathit{False}$?

Note

What happens with Predict(A, a) if Nullable(A) = *False*?

- Follow(A) is still some set of terminals but it won't be relevant since we would need to consider what happens to First(A) first.

Note

What happens with Predict(A, a) if Nullable(A) = *False*?

- Follow(A) is still some set of terminals but it won't be relevant since we would need to consider what happens to First(A) first.
- Hence, the follow function really only matters with regard to the prediction table if we have that nullable is true.

This motivates the following correct definition of our predictor table:

Updated Predictor Table Definition

Definition

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

This definition is correct and is the one we want to use. Notice that this still requires that the table only have one member of the set per entry to be useful as a deterministic algorithm.