Warm-Up Problem

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

$$
\begin{aligned}
S &\rightarrow S + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow a \mid b \mid c
\end{aligned}
$$

CS 241 Lecture 11

Top-Down Parsing, First and Follow

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

First Try (note: broken even with faerie magic!)

Algorithm Top-Down Parsing Algorithm (with faerie magic)

```
push S
for each 'a' in input do
    while top of stack is A ∈ N do
        pop A
        ask a magic faerie to tell you which production A →γ to use
        push the symbols in γ (right to left)
    end while
    // TOS is a terminal
    if TOS is not 'a' then
        Reject
    else
        pop 'a'
    end if
end for
Accept
```

Examples

Using this grammar:

- $S \rightarrow LRS \mid L$
- $L \rightarrow a \mid b \mid c$
- $R \rightarrow + \mid - \mid * \mid /$

Let's check these strings:

- a+b*c
- c-c/c
- b/c+
- lolwut

Problems

- Faeries aren't real[1]
- When we reached the end of the input, we had no way of realizing we weren't done with the stack
- The faerie should be able to tell us that no production matches at all

[1] *Allegedly*

Augmented Grammars

- When we reached the input, we had no way of realizing we weren't done with the stack
- We can make sure we recognize the end (and beginning) of the file by giving them symbols
  - S' $\rightarrow \vdash S \dashv$
  - S $\rightarrow$ $LRS \mid L$
  - L $\rightarrow$ $a \mid b \mid c$
  - R $\rightarrow$ $+ \mid - \mid * \mid /$

Augmented Grammars

- ⊢ and ⊣ are terminals
- When parsing, we have to imagine them into the beginning and ending of the file (there is no BOF or EOF ASCII character)
- We're stacking anyway, so we just need to create a BOF and EOF *token*

g253wang

g253wang g253wang g253wang

Examples

Using this grammar:

- $S' \quad \rightarrow \quad \vdash S \dashv$
- $S \quad \rightarrow \quad LRS \mid L$
- $L \quad \rightarrow \quad a \mid b \mid c$
- $R \quad \rightarrow \quad + \mid - \mid * \mid /$

Let’s check these strings:

- a+b*c
- c-c/c
- b/c+
- lolwut

Problems

- Faeries aren't real[1]
- When we reached the input, we had no way of realizing we weren't done with the stack
- The faerie should be able to tell us that no production matches at all

[1] *Allegedly*

Algorithm Top-Down Parsing Algorithm (with faerie magic)

```
push S
for each 'a' in input do
    while top of stack is A ∈ N do
        pop A
        ask a magic faerie to tell you which production A →γ to use
        if there is a valid production A →γ then
            push the symbols in γ (right to left)
        else
            reject
        end if
    end while
    // TOS is a terminal
    if TOS is not 'a' then
        Reject
    else
        pop 'a'
    end if
end for
Accept
```

Problems

- Faeries aren't real[1]
- When we reached the input, we had no way of realizing we weren't done with the stack
- The faerie should be able to tell us that no production matches at all

[1] *Allegedly*

The Oracle

- Could try all possible productions (way too expensive; want to be deterministic).
- Solution: Use a single symbol lookahead to determine where to go!
- We construct a predictor table to tell us where to go: given a non-terminal on the stack and a next input symbol, what rule should we use?
- Always looking at a *terminal* input symbol, so figuring out how they match is non-trivial.

Why This is Hard

- Consider this grammar again:
  - $0.\ S' \rightarrow \vdash S \dashv$
  - $1.\ S \rightarrow LRS$
  - $2.\ S \rightarrow L$
  - $3.\ L \rightarrow a \mid b \mid c$
  - $4.\ R \rightarrow + \mid - \mid * \mid /$
- How would I decide between (1) and (2) based on the next symbol?
- They both start with *L*, which is the same non-terminal, so either can start with a, b, or c!

Top-Down Parsing Sucks*

- This problem haunts top-down parsing with most grammars
- But, top-down parsing is easier to implement than most alternatives
- Most real parsers are hand written, and use the basic algorithm of top-down parsing, but then “cheat” to deal with these problems
- We will learn formal top-down parsing; just remember that these concepts are usually applied less formally.

Simpler Grammar

- 0. $S' \to \vdash S \dashv$
- 1. $S \to L\ M$
- 2. $L \to I$
- 3. $M \to O\ S$
- 4. $M \to \varepsilon$
- 5. $I \to a$
- 6. $I \to b$
- 7. $O \to +$
- 8. $O \to -$

Grammar Problems

- Our previous grammar makes useless parse trees
- Let’s just focus on getting something working for now; we’ll fix it later ☺

Predictor Table

0. S’ → ⊢ S ⊣
1. S → L M
2. L → I
3. M → O S
4. M → ε
5. I → a
6. I → b
7. O → +
8. O → -

	⊢	⊣	a	b	+	-
S’	0					
S			1	1		
L			2	2		
M		4			3	3
I			5	6		
O					7	8

The Good, The Bad, and The Ugly

- Good news: can have descriptive error messages (expected *x*, found *y*)
- Bad news: need to always know what to do based on one symbol
- Ugly news: needed to transform our grammar to make it fit, not all grammars can be transformed, and for those that can, we change the parse trees!

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, don't need sets in our predict table

Why is it called LL(1)?

- First L: Scan left to right (more precisely, beginning to end)
- Second L: Leftmost derivations
- Number of symbol lookahead: 1

LL(k>1) is possible but rare. LL(0) is basically meaningless. Technically, end-to-beginning version is RR(1), but end-to-beginning scanning is psychopath behavior.

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):

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

More formally:

$Predict(A, a) = \{A \to \beta \ : a \in \text{First}(\beta)\}$

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

Note: Predict as defined above is incorrect! Why? More on that later.

Example of First

0. S’ → ⊢ S ⊣	First(⊢ S ⊣) = {⊢}
1. S → L M	First(L M) = {a, b}
2. L → I	First(I) = {a, b}
3. M → O S	First(O S) = {+, -}
4. M → ε	First(ε) = {}
5. I → a	First(a) = {a}
6. I → b	First(b) = {b}
7. O → +	First(+) = {+}
8. O → -	First(-) = {-}

Example of First

0. S’ → ⊢ S ⊣	First(⊢ S ⊣) = {⊢}
1. S → L M	First(L M) = {a, b}
2. L → I	First(I) = {a, b}
3. M → O S	First(O S) = {+, -}
4. M → ε	First(ε) = {}
5. I → a	First(a) = {a}
6. I → b	First(b) = {b}
7. O → +	First(+) = {+}
8. O → -	First(-) = {-}

Note (1): To compute First(L M), first we needed First(I)!

Issue

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 $A \Rightarrow^* \varepsilon$! 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	$\rightarrow$	ε	(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, *if* it can reduce to ε.

In this case, we need to include that ⊣ ∈ Predict(*B*, ⊣):

	⊢	a	b	c	d	e	f	⊣
S′	{0}							
S		{1}					{1}	
A		{2}					{3}	
B					{4}	{5}		{**6**}

Correction

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

To correct this, we introduce two new functions.

Nullable(β): boolean function; for $\beta \in V^*$ is true if and only if $\beta \Rightarrow^* \varepsilon$.

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 $\beta \Rightarrow^* \varepsilon$ 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 Follow

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

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

Note

What happens with Predict(A, a) if Nullable(A) = 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.
- Thus, the Follow function only matters if Nullable is true.

This motivates the following correct definition of our predictor table:

Updated Predictor Table Definition

Definition

$$\text{Predict}(A, a) = \{A \rightarrow \beta \;:\; a \in \text{First}(\beta)\}$$
$$\cup\, \{A \rightarrow \beta \;:\; \text{Nullable}(\beta) \text{ and } a \in \text{Follow}(A)\}$$

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

Notes on Nullable

- Note that Nullable(β) = false whenever β contains a terminal symbol.
- Further, Nullable(AB) = Nullable(A) ∧ Nullable(B)
- Thus, it suffices to compute Nullable(A) for all A ∈ N.

g253wang

g253wang g253wang g253wang

Computing Nullable

Algorithm 2 Nullable(A) for all $A \in N'$

Initialize Nullable(A) = false for all $A \in N'$.
repeat
    for each production in P **do**
        if (P is $A \to \epsilon$) or (P is $A \to B_1 \cdots B_k$ and $\bigwedge_{i=1}^{k}$ Nullable(B_i) = true) **then**
            Nullable(A) = true
        end if
    end for
until nothing changes

Example of Nullable

$$
\begin{array}{lcll}
S' & \rightarrow & \vdash S \dashv & (0) \\
S & \rightarrow & c & (1) \\
S & \rightarrow & QRS & (2) \\
Q & \rightarrow & R & (3) \\
Q & \rightarrow & d & (4) \\
R & \rightarrow & \varepsilon & (5) \\
R & \rightarrow & b & (6)
\end{array}
$$

Nullability Table

Iter	0	1	2	3
S′	F	F	F	F
S	F	F	F	F
Q	F	F	T	T
R	F	T	T	T

Thus, Nullable(S') = Nullable(S) = F and Nullable(Q) = Nullable(R) = T

Notes About First

- Main idea: Keep processing $B_1B_2\ldots B_k$ from a production rule until you encounter a terminal or a symbol that is not nullable. Then go to the next rule. Repeat until no changes are made during the processing.
- Remember, ε isn't a real symbol, and can't be in a First set!
- For First, we will ignore **trivial productions** of the form $A \to \varepsilon$ based on the above observation.
- Further, First(S') = {⊢} always.
- We first compute First(A) for all $A \in N$ and then we compute First(β) for all relevant $\beta \in V^*$

Computing First

Algorithm 3 First(A) for all $A \in N'$

Initialize First$(A) = \{\}$ for all $A \in N'$.
repeat
for each rule $A \to B_1 B_2 \cdots B_k$ in P **do**
for $i \in \{1, .., k\}$ **do**
if $B_i \in T'$ **then**
First$(A) =$ First$(A) \cup \{B_i\}$; **break**
else
First$(A) =$ First$(A) \cup$ First(B_i)
if Nullable(B_i) $==$ False **then break**
end if
end for
end for
until nothing changes

Example of First

$$
\begin{aligned}
S' &\rightarrow \vdash S \dashv \\
S &\rightarrow c \\
S &\rightarrow QRS \\
Q &\rightarrow R \\
Q &\rightarrow d \\
R &\rightarrow \varepsilon \\
R &\rightarrow b
\end{aligned}
$$

First Table:

Iter	0	1	2	3
S′	$\{\}$	$\{\vdash\}$	$\{\vdash\}$	$\{\vdash\}$
S	$\{\}$	$\{c\}$	$\{b, c, d\}$	$\{b, c, d\}$
Q	$\{\}$	$\{d\}$	$\{b, d\}$	$\{b, d\}$
R	$\{\}$	$\{b\}$	$\{b\}$	$\{b\}$

Recall, Nullable(S') = Nullable(S) = F and Nullable(Q) = Nullable(R) = T

Thus, First(S') = $\{\vdash\}$, First(S) = $\{b, c, d\}$, First(Q) = $\{b, d\}$, First(R) = $\{b\}$