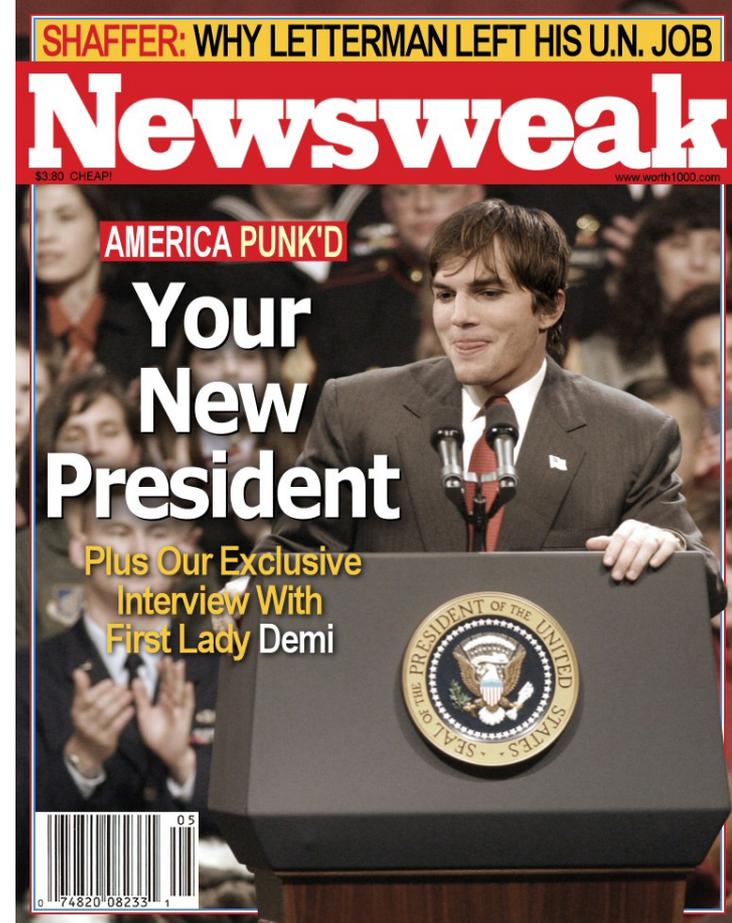


Top-Down Parsing



Extra Credit Question

- Given this grammar G:
 - $E \rightarrow E + T$
 - $E \rightarrow T$
 - $T \rightarrow T * \text{int}$
 - $T \rightarrow \text{int}$
 - $T \rightarrow (E)$
- Is the string $\text{int} * (\text{int} + \text{int})$ in $L(G)$?
 - Give a derivation or prove that it is not.



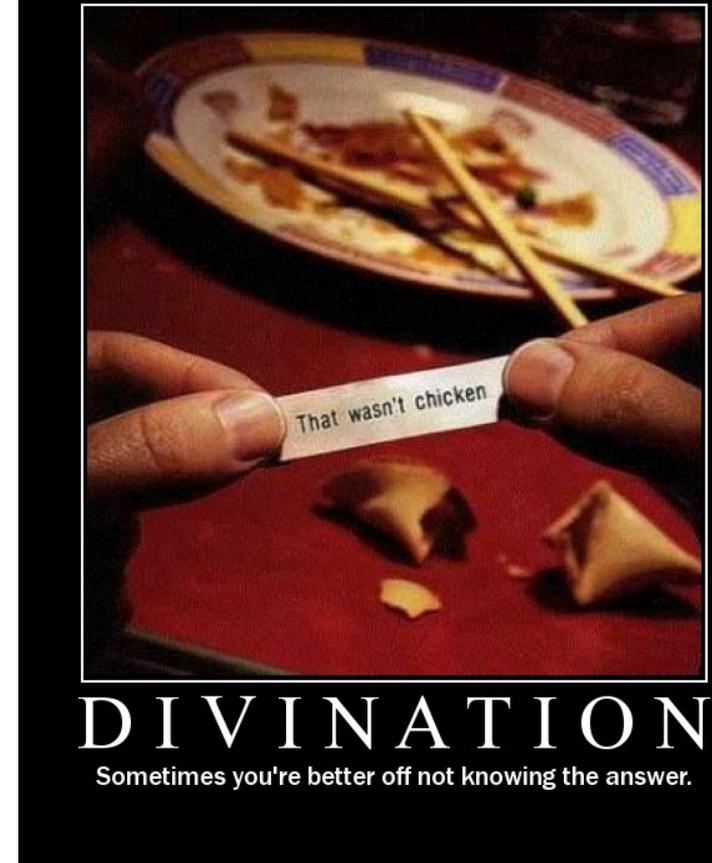
Revenge of Theory

- How do we tell if DFA P is equal to DFA Q ?
 - We can do: “is DFA P empty?”
 - How?
 - We can do: “ $P := \text{not } Q$ ”
 - How?
 - We can do: “ $P := Q \text{ intersect } R$ ”
 - How?
 - So do: “is $P \text{ intersect not } Q$ empty?”
- Does this work for CFG X and CFG Y ?
- Can we tell if s is in CFG X ?



Outline

- Recursive Descent Parsing
- Left Recursion
- LL(1) Parsing
 - LL(1) Parsing Tables
 - LP(1) Parsing Algorithm
- Constructing LL(1) Parsing Tables
 - First, Follow



In One Slide

- An **LL(1) parser** reads tokens from **left to right** and constructs a **top-down leftmost** derivation. LL(1) parsing is a special case of **recursive descent parsing** in which you can **predict** which single production to use from **one token of lookahead**. LL(1) parsing is **fast and easy**, but it does not work if the grammar is **ambiguous**, **left-recursive**, or **not left-factored** (i.e., it does not work for most programming languages).

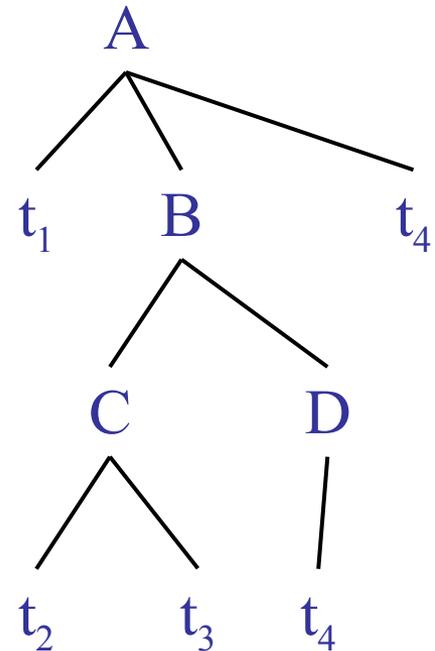
Intro to Top-Down Parsing

- Terminals are seen in order of appearance in the token stream:

t_1 t_2 t_3 t_4 t_5

The parse tree is constructed

- From the top
- From left to right



Recursive Descent Parsing

- We'll try **recursive descent** parsing first
 - “Try all productions exhaustively, backtrack”
- Consider the grammar

$$E \rightarrow T + E \mid T$$

$$T \rightarrow (E) \mid \text{int} \mid \text{int} * T$$

- Token stream is: **int * int**
- Start with top-level non-terminal **E**

- Try the rules for **E** in order

Recursive Descent Example

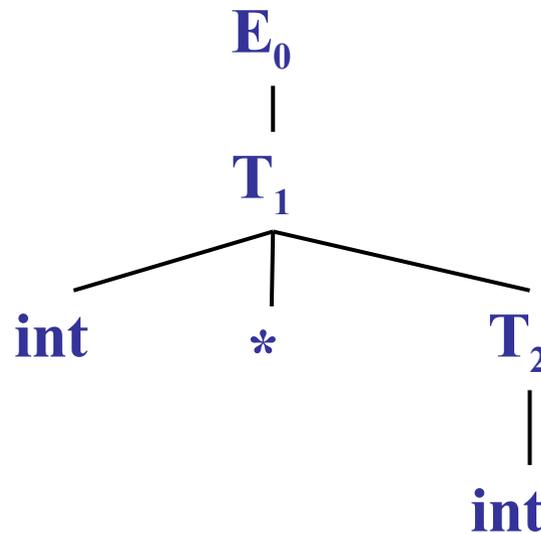
```
E → T + E | T
T → ( E ) | int | int * T
Input = int * int
```

- Try $E_0 \rightarrow T_1 + E_2$
- Then try a rule for $T_1 \rightarrow (E_3)$
 - But $($ does not match input token int
- Try $T_1 \rightarrow int$. Token matches.
 - But $+$ after T_1 does not match input token $*$
- Try $T_1 \rightarrow int * T_2$
 - This will match but $+$ after T_1 will be unmatched
- Have exhausted the choices for T_1
 - **Backtrack** to choice for E_0

Recursive Descent Example (2)

$E \rightarrow T + E \mid T$
 $T \rightarrow (E) \mid \text{int} \mid \text{int} * T$
*Input = int * int*

- Try $E_0 \rightarrow T_1$
- Follow same steps as before for T_1
 - And succeed with $T_1 \rightarrow \text{int} * T_2$ and $T_2 \rightarrow \text{int}$
 - With the following parse tree



Recursive Descent Parsing

- Parsing: given a string of tokens $t_1 t_2 \dots t_n$, find its parse tree
- **Recursive descent parsing**: Try all the productions exhaustively
 - At a given moment the **fringe** of the parse tree is:
 $t_1 t_2 \dots t_k A \dots$
 - Try all the productions for A: if $A \rightarrow BC$ is a production, the new fringe is $t_1 t_2 \dots t_k B C \dots$
 - **Backtrack** when the fringe doesn't match the string
 - Stop when there are no more non-terminals

When Recursive Descent Does *Not* Work

- Consider a production $S \rightarrow S a$:
 - In the process of parsing S we try the above rule
 - What goes wrong?
- A left-recursive grammar has
$$S \rightarrow^+ S\alpha \quad \text{for some } \alpha$$

Recursive descent does not work in such cases

- It goes into an ∞ loop

What's Wrong With That Picture?



Elimination of Left Recursion

- Consider the left-recursive grammar

$$S \rightarrow S \alpha \mid \beta$$

- S generates all strings starting with a β and followed by a number of α

- Can rewrite using **right-recursion**

$$S \rightarrow \beta T$$

$$T \rightarrow \alpha T \mid \epsilon$$

Example of Eliminating Left Recursion

- Consider the grammar

$$S \rightarrow 1 \mid S 0$$

($\beta = 1$ and $\alpha = 0$)

It can be rewritten as

$$S \rightarrow 1 T$$

$$T \rightarrow 0 T \mid \epsilon$$

"Transported to a surreal landscape, a young girl kills the first person she meets and then teams up with three strangers to kill again."



ASSASSIN

They come in all shapes and sizes.

More Left Recursion Elimination

- In general

$$S \rightarrow S \alpha_1 \mid \dots \mid S \alpha_n \mid \beta_1 \mid \dots \mid \beta_m$$

- All strings derived from S start with one of β_1, \dots, β_m and continue with several instances of $\alpha_1, \dots, \alpha_n$

- Rewrite as

$$S \rightarrow \beta_1 T \mid \dots \mid \beta_m T$$

$$T \rightarrow \alpha_1 T \mid \dots \mid \alpha_n T \mid \epsilon$$

General Left Recursion

- The grammar

$$S \rightarrow A \alpha \mid \delta$$

$$A \rightarrow S \beta$$

is also left-recursive because

$$S \rightarrow^+ S \beta \alpha$$

- This left-recursive can also be eliminated
- See book, Section 2.3
- Detecting and eliminating left recursion are *popular test questions*



Signs

And some of them are not

Summary of Recursive Descent

- Simple and general parsing strategy
 - **Left-recursion** must be eliminated first
 - ... but that can be done automatically
- Unpopular because of **backtracking**
 - Thought to be too inefficient (repetition)
- We can avoid backtracking
 - Sometimes ...



Predictive Parsers

- Like recursive descent but parser can “predict” which production to use
 - By looking at the next few tokens
 - **No backtracking**
- **Predictive parsers** accept **LL(k)** grammars
 - First **L** means “left-to-right” scan of input
 - Second **L** means “leftmost derivation”
 - The **k** means “predict based on k tokens of lookahead”
- In practice, **LL(1)** is used

Sometimes Things Are Perfect

- The “.ml-lex” format you emit in PA2
- Will be the input for PA3
 - actually the *reference* “.ml-lex” will be used
- It can be “parsed” with *no* lookahead
 - You always know just what to do next
- Ditto with the “.ml-ast” output of PA3
- Just write a few mutually-recursive functions
- They read in the input, one line at a time

LL(1)

- In recursive descent, for each non-terminal and input token there may be a choice of which production to use
- **LL(1)** means that for each non-terminal and token there is *only one* production that could lead to success
- Can be specified as a 2D table
 - One dimension for **current non-terminal** to expand
 - One dimension for **next token**
 - Each table entry contains **one production**

Predictive Parsing and Left Factoring

- Recall the grammar

$$E \rightarrow T + E \mid T$$

$$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$$

- Impossible to **predict** because
 - For **T** two productions start with **int**
 - For **E** it is not clear how to predict
- A grammar must be **left-factored** before use for predictive parsing



DEFEAT

Sometimes you just should
have seen it coming.

Left-Factoring Example

- Recall the grammar

$$E \rightarrow T + E \mid T$$

$$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$$

- Factor out* common prefixes of productions

$$E \rightarrow T X$$

$$X \rightarrow + E \mid \varepsilon$$

$$T \rightarrow (E) \mid \text{int} Y$$

$$Y \rightarrow * T \mid \varepsilon$$

Introducing: Parse Tables



Rolemaster

A table for every occasion

LL(1) Parsing Table Example

- Left-factored grammar

$$E \rightarrow T X$$

$$X \rightarrow + E \mid \epsilon$$

$$T \rightarrow (E) \mid \text{int } Y$$

$$Y \rightarrow * T \mid \epsilon$$

- The LL(1) parsing table ($\$$ is a special end marker):

	int	*	+	()	\$
T	int Y			(E)		
E	T X			T X		
X			+ E		ϵ	ϵ
Y		* T	ϵ		ϵ	ϵ

LL(1) Parsing Table

Example Analysis

- Consider the [E, int] entry
 - “When current non-terminal is **E** and next input is **int**, use production **$E \rightarrow TX$** ”
 - This production can generate an **int** in the first position

	int	*	+	()	\$
T	int Y			(E)		
E	TX			TX		
X			+ E		ϵ	ϵ
Y		* T	ϵ		ϵ	ϵ

LL(1) Parsing Table

Example Analysis

- Consider the [Y,+] entry
 - “When current non-terminal is **Y** and current token is **+**, *get rid of Y*”
 - We’ll see later why this is so

	int	*	+	()	\$
T	int Y			(E)		
E	T X			T X		
X			+ E		ϵ	ϵ
Y		* T	ϵ		ϵ	ϵ

LL(1) Parsing Tables: Errors

- Blank entries indicate **error** situations
 - Consider the [E,*] entry
 - “There is *no way* to derive a string starting with * from non-terminal E”

	int	*	+	()	\$
T	int Y			(E)		
E	T X			T X		
X			+ E		ϵ	ϵ
Y		* T	ϵ		ϵ	ϵ

Using Parsing Tables

- Method similar to recursive descent, except
 - For each non-terminal S
 - We look at the next token a
 - And choose the production shown at $[S,a]$
- We use a **stack** to keep track of pending non-terminals
- We **reject** when we encounter an error state
- We **accept** when we encounter end-of-input

LL(1) Parsing Algorithm

initialize stack = $\langle S \$ \rangle$

next = *(pointer to tokens)*

repeat

match stack with

| $\langle X, \text{rest} \rangle$: **if** $T[X, *next] = Y_1 \dots Y_n$
 then stack $\leftarrow \langle Y_1 \dots Y_n \text{rest} \rangle$

else error ()

| $\langle t, \text{rest} \rangle$: **if** $t == *next ++$

then stack $\leftarrow \langle \text{rest} \rangle$

else error ()

until stack == $\langle \rangle$

Stack

Input

Action



	int	*	+	()	\$
T	int Y			(E)		
E	T X			T X		
X			+ E		ϵ	ϵ
Y		* T	ϵ		ϵ	ϵ

Stack
E \$

Input
int * int \$

Action
T X

	int	*	+	()	\$
T	int Y			(E)		
E	T X			T X		
X			+ E		ϵ	ϵ
Y		* T	ϵ		ϵ	ϵ

Stack

E \$

T X \$

Input

int * int \$

int * int \$

Action

T X

int Y

	int	*	+	()	\$
T	int Y			(E)		
E	T X			T X		
X			+ E		ϵ	ϵ
Y		* T	ϵ		ϵ	ϵ

Stack

E \$

T X \$

int Y X \$

Input

int * int \$

int * int \$

int * int \$

Action

T X

int Y

terminal

	int	*	+	()	\$
T	int Y			(E)		
E	T X			T X		
X			+ E		ϵ	ϵ
Y		* T	ϵ		ϵ	ϵ

Stack

E \$

T X \$

int Y X \$

Y X \$

Input

int * int \$

int * int \$

int * int \$

* int \$

Action

T X

int Y

terminal

* T

	int	*	+	()	\$
T	int Y			(E)		
E	T X			T X		
X			+ E		ϵ	ϵ
Y		* T	ϵ		ϵ	ϵ

Stack

E \$

T X \$

int Y X \$

Y X \$

* T X \$

Input

int * int \$

int * int \$

int * int \$

* int \$

* int \$

Action

T X

int Y

terminal

* T

terminal

	int	*	+	()	\$
T	int Y			(E)		
E	T X			T X		
X			+ E		ϵ	ϵ
Y		* T	ϵ		ϵ	ϵ

Stack

E \$

T X \$

int Y X \$

Y X \$

* T X \$

T X \$

Input

int * int \$

int * int \$

int * int \$

* int \$

* int \$

int \$

Action

T X

int Y

terminal

* T

terminal

int Y

	int	*	+	()	\$
T	int Y			(E)		
E	T X			T X		
X			+ E		ϵ	ϵ
Y		* T	ϵ		ϵ	ϵ

Stack

E \$

T X \$

int Y X \$

Y X \$

* T X \$

T X \$

int Y X \$

Input

int * int \$

int * int \$

int * int \$

* int \$

* int \$

int \$

int \$

Action

T X

int Y

terminal

* T

terminal

int Y

terminal

	int	*	+	()	\$
T	int Y			(E)		
E	T X			T X		
X			+ E		ϵ	ϵ
Y		* T	ϵ		ϵ	ϵ

Stack

E \$

T X \$

int Y X \$

Y X \$

* T X \$

T X \$

int Y X \$

Y X \$

Input

int * int \$

int * int \$

int * int \$

* int \$

* int \$

int \$

int \$

\$

Action

T X

int Y

terminal

* T

terminal

int Y

terminal

ϵ

	int	*	+	()	\$
T	int Y			(E)		
E	T X			T X		
X			+ E		ϵ	ϵ
Y		* T	ϵ		ϵ	ϵ

Stack

E \$

T X \$

int Y X \$

Y X \$

* T X \$

T X \$

int Y X \$

Y X \$

X \$

Input

int * int \$

int * int \$

int * int \$

* int \$

* int \$

int \$

int \$

\$

\$

Action

T X

int Y

terminal

* T

terminal

int Y

terminal

ϵ

ϵ

	int	*	+	()	\$
T	int Y			(E)		
E	T X			T X		
X			+ E		ϵ	ϵ
Y		* T	ϵ		ϵ	ϵ

Stack

E \$

T X \$

int Y X \$

Y X \$

* T X \$

T X \$

int Y X \$

Y X \$

X \$

\$

Input

int * int \$

int * int \$

int * int \$

* int \$

* int \$

int \$

int \$

\$

\$

\$

Action

T X

int Y

terminal

* T

terminal

int Y

terminal

ϵ

ϵ

ACCEPT

	int	*	+	()	\$
T	int Y			(E)		
E	T X			T X		
X			+ E		ϵ	ϵ
Y		* T	ϵ		ϵ	ϵ

LL(1) Languages

- **LL(1) languages** can be LL(1) parsed
 - A language Q is LL(1) if there exists an LL(1) table such the LL(1) parsing algorithm using that table accepts exactly the strings in Q
- No table entry can be **multiply defined**
- Once we have the table
 - The parsing algorithm is **simple and fast**
 - **No backtracking** is necessary
- Want to generate parsing tables from CFG!

Q: Movies (263 / 842)

- This 1982 Star Trek film features Spock nerve-pinching McCoy, Kirstie Alley "losing" the *Kobayashi Maru* , and Chekov being mind-controlled by a slug-like alien. Ricardo Montalban is "*is intelligent, but not experienced. His pattern indicates two-dimensional thinking.*"

Q: Music (238 / 842)

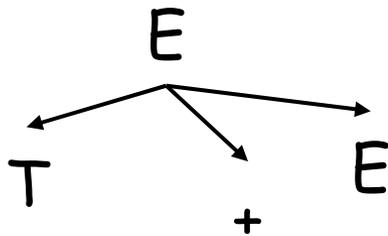
- For two of the following four lines from the 1976 Eagles song **Hotel California**, give enough words to complete the rhyme.
 - *So I called up the captain / "please bring me my wine"*
 - *Mirrors on the ceiling / pink champagne on ice*
 - *And in the master's chambers / they gathered for the feast*
 - *We are programmed to receive / you can checkout any time you like,*

Q: Books (727 / 842)

- Name 5 of the 9 major characters in A. A. Milne's 1926 books about a "*bear of very little brain*" who composes poetry and eats honey.

Top-Down Parsing. Review

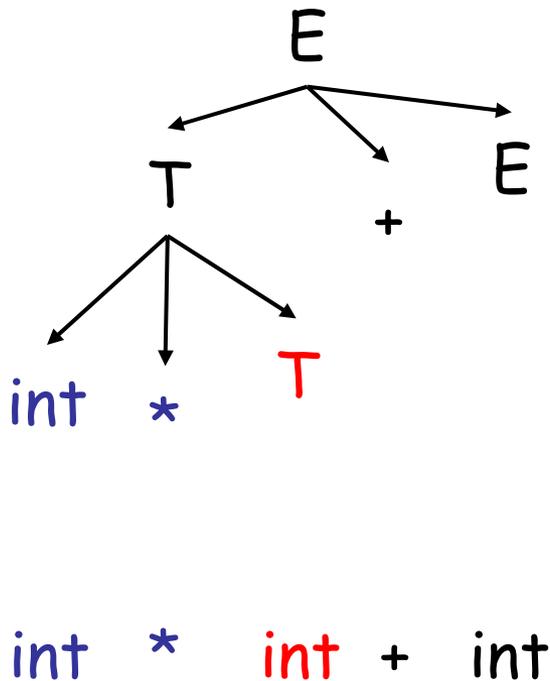
- Top-down parsing expands a parse tree from the start symbol to the leaves
 - Always expand the leftmost non-terminal



int * int + int

Top-Down Parsing. Review

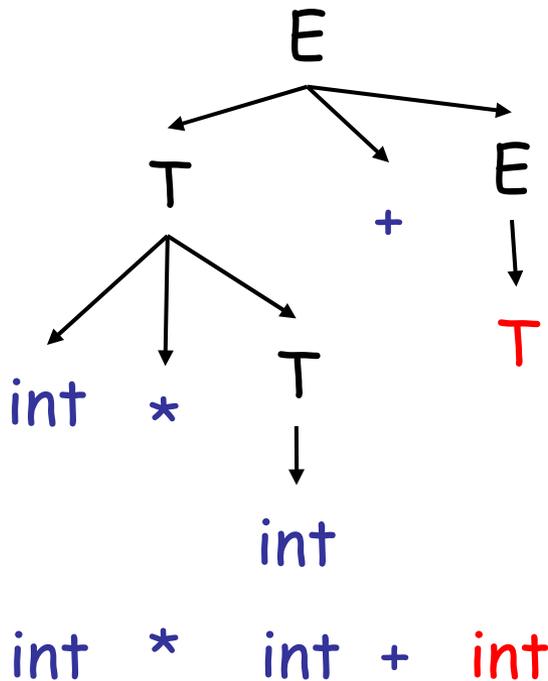
- Top-down parsing expands a parse tree from the start symbol to the leaves
 - Always expand the leftmost non-terminal



- The leaves at any point form a string $\beta A \gamma$
 - β contains only terminals
 - The input string is $\beta b \delta$
 - The prefix β matches
 - The next token is b

Top-Down Parsing. Review

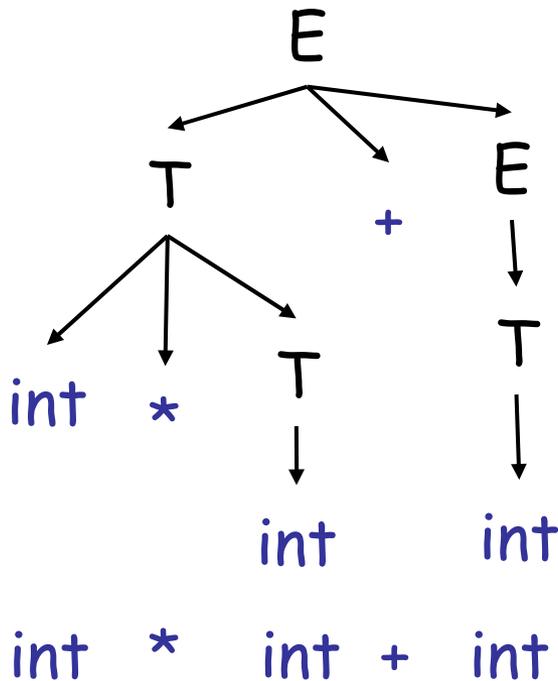
- Top-down parsing expands a parse tree from the start symbol to the leaves
 - Always expand the leftmost non-terminal



- The leaves at any point form a string $\beta A \gamma$
 - β contains only terminals
 - The input string is $\beta b \delta$
 - The prefix β matches
 - The next token is b

Top-Down Parsing. Review

- Top-down parsing expands a parse tree from the start symbol to the leaves
 - Always expand the leftmost non-terminal



- The leaves at any point form a string $\beta A \gamma$
 - β contains only terminals
 - The input string is $\beta b \delta$
 - The prefix β matches
 - The next token is b

Constructing Predictive Parsing Tables

- Consider the state $S \rightarrow^* \beta A \gamma$
 - With b the next token
 - Trying to match $\beta b \delta$

There are two possibilities:

- b belongs to an expansion of A
 - Any $A \rightarrow \alpha$ can be used if b can start a string derived from α

In this case we say that $b \in \underline{\text{First}}(\alpha)$

Or...

Constructing Predictive Parsing Tables

- **b** does not belong to an expansion of **A**
 - The expansion of **A** is empty and **b** belongs to an expansion of γ (e.g., $b\omega$)
 - Means that **b** can appear after **A** in a derivation of the form $S \rightarrow^* \beta A b \omega$
 - We say that $b \in \text{Follow}(A)$ in this case
 - What productions can we use in this case?
 - Any $A \rightarrow \alpha$ can be used if α can expand to ϵ
 - We say that $\epsilon \in \text{First}(A)$ in this case

Computing First Sets

Definition **First**(X) = $\{ b \mid X \rightarrow^* b\alpha \} \cup \{ \varepsilon \mid X \rightarrow^* \varepsilon \}$

- $\text{First}(b) = \{ b \}$
- For all productions $X \rightarrow A_1 \dots A_n$
 - Add $\text{First}(A_1) - \{ \varepsilon \}$ to $\text{First}(X)$. Stop if $\varepsilon \notin \text{First}(A_1)$
 - Add $\text{First}(A_2) - \{ \varepsilon \}$ to $\text{First}(X)$. Stop if $\varepsilon \notin \text{First}(A_2)$
 - ...
 - Add $\text{First}(A_n) - \{ \varepsilon \}$ to $\text{First}(X)$. Stop if $\varepsilon \notin \text{First}(A_n)$
 - Add ε to $\text{First}(X)$
(ignore A_i if it is X)

Example First Set Computation

- Recall the grammar

$$E \rightarrow T X$$

$$X \rightarrow + E \mid \varepsilon$$

$$T \rightarrow (E) \mid \text{int } Y$$

$$Y \rightarrow * T \mid \varepsilon$$

- First sets

$$\text{First}(() = \{ (\}$$

$$\text{First}(T) = \{ \text{int}, (\}$$

$$\text{First}()) = \{) \}$$

$$\text{First}(E) = \{ \text{int}, (\}$$

$$\text{First}(\text{int}) = \{ \text{int} \}$$

$$\text{First}(X) = \{ +, \varepsilon \}$$

$$\text{First}(+) = \{ + \}$$

$$\text{First}(Y) = \{ *, \varepsilon \}$$

$$\text{First}(*) = \{ * \}$$

Computing Follow Sets

Definition **Follow**(X) = { b | S \rightarrow^* β X b ω }

- Compute the **First** sets for all non-terminals first
- Add **\$** to **Follow**(S) (if S is the start non-terminal)
- For all productions $Y \rightarrow \dots X A_1 \dots A_n$
 - Add **First**(A₁) - { ϵ } to **Follow**(X). Stop if $\epsilon \notin \text{First}(A_1)$
 - Add **First**(A₂) - { ϵ } to **Follow**(X). Stop if $\epsilon \notin \text{First}(A_2)$
 - ...
 - Add **First**(A_n) - { ϵ } to **Follow**(X). Stop if $\epsilon \notin \text{First}(A_n)$
 - Add **Follow**(Y) to **Follow**(X)

Example Follow Set Computation

- Recall the grammar

$$E \rightarrow T X$$

$$T \rightarrow (E) \mid \text{int } Y$$

$$X \rightarrow + E \mid \varepsilon$$

$$Y \rightarrow * T \mid \varepsilon$$

- Follow sets

$$\text{Follow}(+) = \{ \text{int}, (\}$$

$$\text{Follow}(() = \{ \text{int}, (\}$$

$$\text{Follow}(X) = \{ \$,) \}$$

$$\text{Follow}()) = \{ +,) , \$ \}$$

$$\text{Follow}(\text{int}) = \{ *, +,) , \$ \}$$

$$\text{Follow}(*) = \{ \text{int}, (\}$$

$$\text{Follow}(E) = \{), \$ \}$$

$$\text{Follow}(T) = \{ +,) , \$ \}$$

$$\text{Follow}(Y) = \{ +,) , \$ \}$$

Constructing LL(1) Parsing Tables

- Here is how to construct a parsing table T for context-free grammar G
- For each production $A \rightarrow \alpha$ in G do:
 - For each terminal $b \in \text{First}(\alpha)$ do
 - $T[A, b] = \alpha$
 - If $\alpha \rightarrow^* \epsilon$, for each $b \in \text{Follow}(A)$ do
 - $T[A, b] = \alpha$

LL(1) Table Construction Example

- Recall the grammar

$$E \rightarrow T X$$

$$X \rightarrow + E \mid \epsilon$$

$$T \rightarrow (E) \mid \text{int } Y$$

$$Y \rightarrow * T \mid \epsilon$$

- Where in the row of Y do we put $Y \rightarrow * T$?
 - In the columns of $\text{First}(*T) = \{ * \}$

	int	*	+	()	\$
T	int Y			(E)		
E	T X			T X		
X			+ E		ϵ	ϵ
Y		* T	ϵ		ϵ	ϵ

LL(1) Table Construction Example

- Recall the grammar

$$E \rightarrow T X \qquad X \rightarrow + E \mid \epsilon$$

$$T \rightarrow (E) \mid \text{int } Y \qquad Y \rightarrow * T \mid \epsilon$$

- Where in the row of Y we put $Y \rightarrow \epsilon$?
 - In the columns of $\text{Follow}(Y) = \{ \$, +,) \}$

	int	*	+	()	\$
T	int Y			(E)		
E	T X			T X		
X			+ E		ϵ	ϵ
Y		* T	ϵ		ϵ	ϵ

Avoid Multiple Definitions!



Notes on LL(1) Parsing Tables

- If any entry is **multiply defined** then **G is not LL(1)**
 - If G is ambiguous
 - If G is left recursive
 - If G is not left-factored
 - *And in other cases as well*
- Most programming language grammars are **not LL(1)** (e.g., Java, Ruby, C++, OCaml, Cool, Perl, ...)
- There are tools that build LL(1) tables

Simple Parsing Strategies

- Recursive Descent Parsing
 - But backtracking is too annoying, etc.
- Predictive Parsing, aka. LL(k)
 - Predict production from k tokens of lookahead
 - Build LL(1) table
 - Parsing using the table is fast and easy
 - But many grammars are not LL(1) (or even LL(k))
- Next: a more powerful parsing strategy for grammars that are not LL(1)

Homework

- Today: WA1 (written homework) due
- Soon: PA2 (Lexer) due
 - You may work in pairs.
- Read: Chapters 2.3.3
 - Optional Wikipedia article