Top-Down Parsing

Extra Credit Question

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

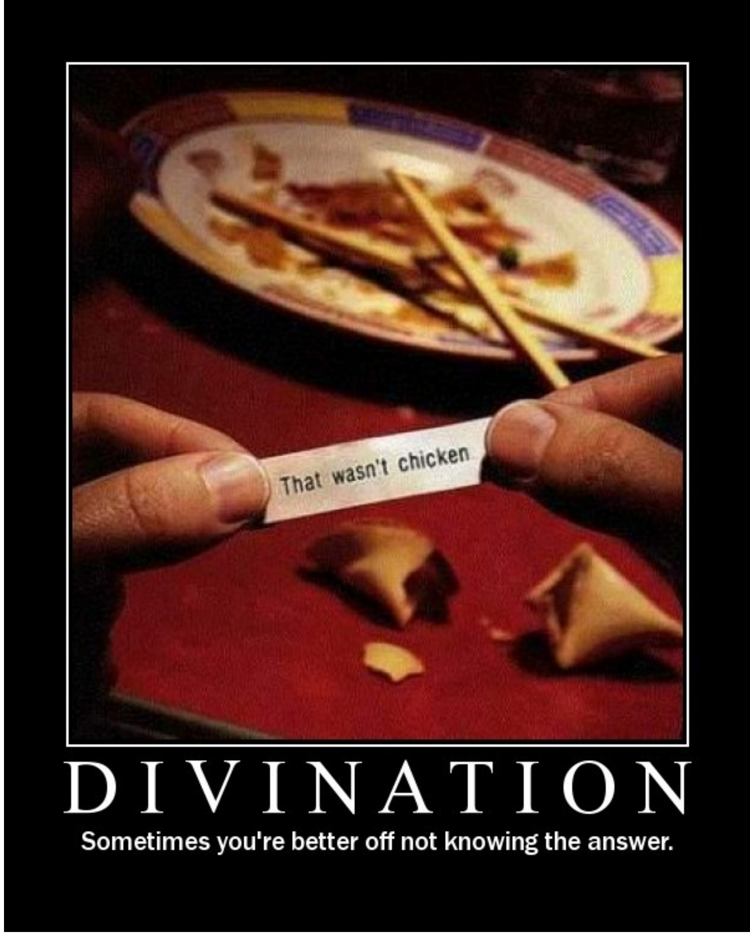
Revenge of Theory

- How do we tell if DFA **P** is equal to DFA **Q**?
  - We can do: "is DFA **P** empty?"
    - How?
  - We can do: "**P** := not **Q**"
    - How?
  - We can do: "**P** := **Q** intersect **R**"
    - How?
  - So do: "is **P** 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

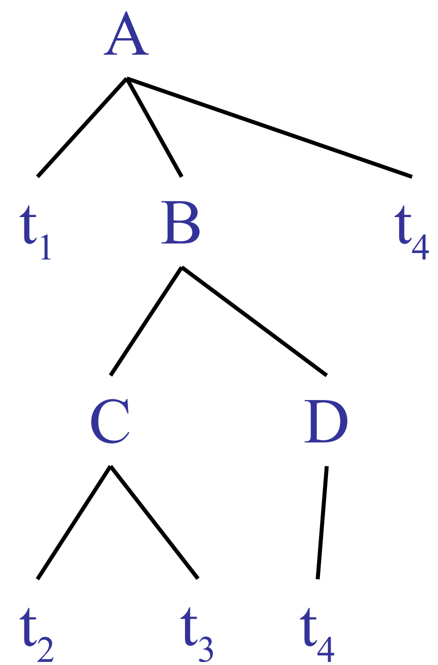
- An **LL(1) parser** reads tokens from left to right and constructs a top-down leftmost derivation. LL(1) parsing is a special case of recursive descent parsing in which you can predict which single production to use from one token of lookahead. LL(1) parsing is fast and easy, but it does not work if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

  $t_1 \quad t_2 \quad t_3 \quad t_4 \quad t_5$

  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

$$E \to T + E \;\mid\; T$$

$$T \to (\, E\,) \;\mid\; \text{int} \;\mid\; \text{int} * T$$

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

- Try the rules for E in order

Recursive Descent Example

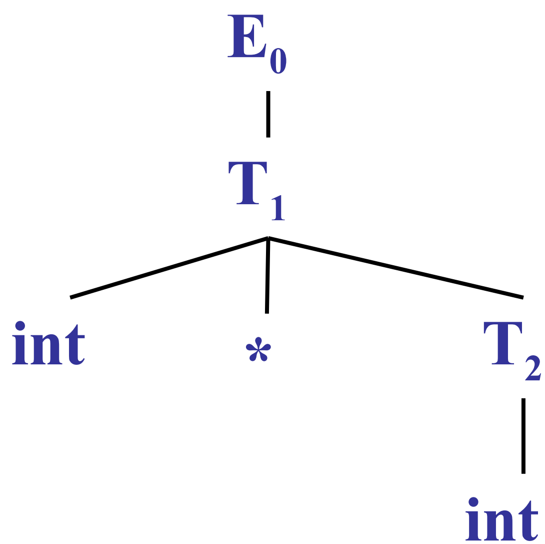
$E \rightarrow T + E \mid T$
$T \rightarrow (E) \mid int \mid int * T$
Input = int * int

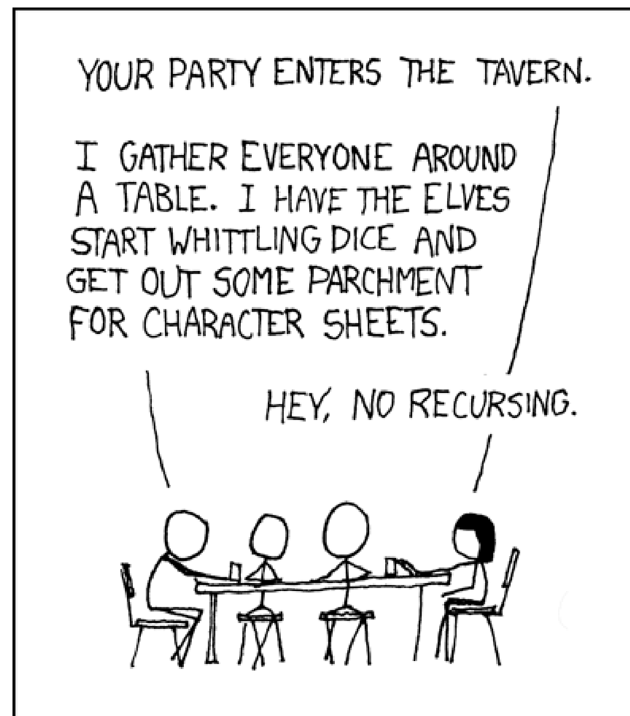
- Try $E_0 \rightarrow T_1 + E_2$
- Then try a rule for $T_1 \rightarrow (E_3)$
  - But (does not match input token int
- Try $T_1 \rightarrow int$. Token matches.
  - But + after T_1 does not match input token *
- Try $T_1 \rightarrow int * T_2$
  - This will match but + after T_1 will be unmatched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

Recursive Descent Example (2)

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

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





Recursive Descent Parsing

- Parsing: given a string of tokens $t_1\ t_2\ \ldots\ t_n$, find its parse tree
- **Recursive descent parsing**: Try all the productions exhaustively
  - At a given moment the **fringe** of the parse tree is: $t_1\ t_2\ \ldots\ t_k\ A\ \ldots$
  - Try all the productions for A: if $A \rightarrow BC$ is a production, the new fringe is $t_1\ t_2\ \ldots\ t_k\ B\ C\ \ldots$
  - **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 $S \to S\ a$:
  - In the process of parsing S we try the above rule
  - What goes wrong?

- A **left-recursive grammar** has

$$S \to^{+} S\alpha \quad \text{for some } \alpha$$

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

$$S \to S\,\alpha \;|\; \beta$$

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

- Can rewrite using **right-recursion**

$$S \to \beta\;T$$

$$T \to \alpha\;T \;|\; \varepsilon$$

Example of Eliminating Left Recursion

- Consider the grammar

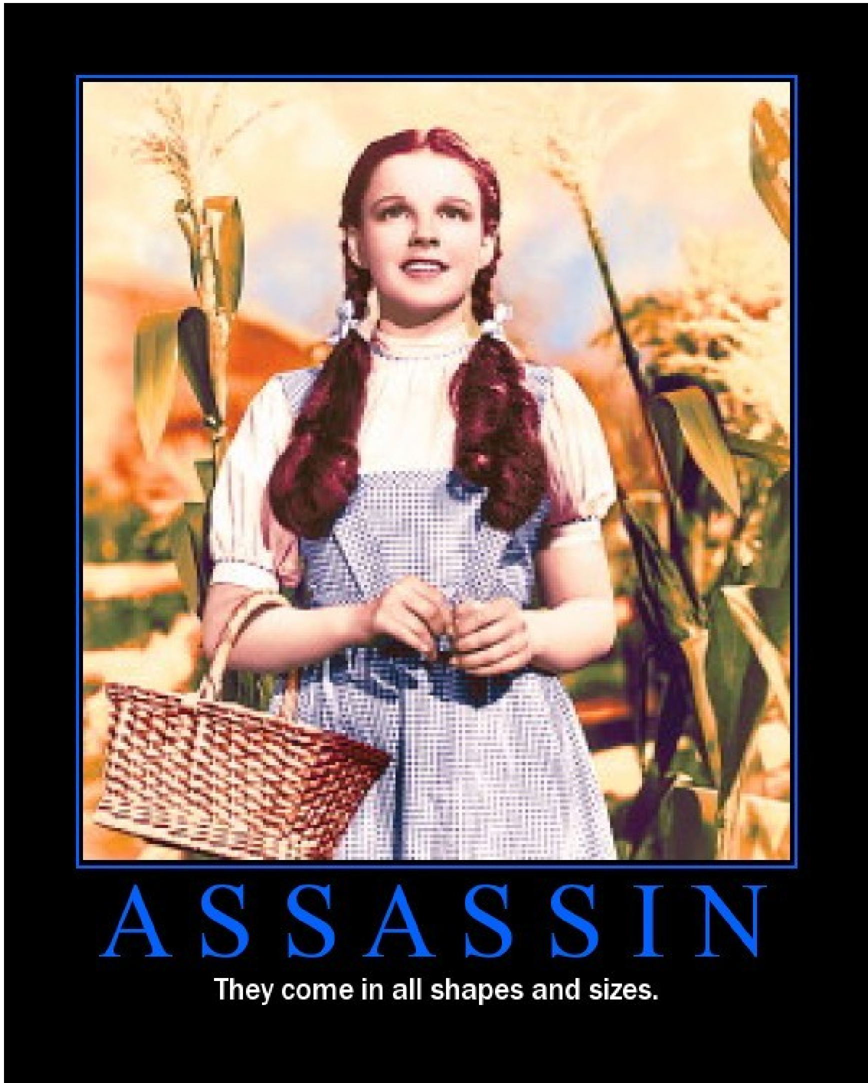
  $S \rightarrow 1 \mid S\ 0$

  (β = 1 and α = 0)

  It can be rewritten as

  $S \rightarrow 1\ T$

  $T \rightarrow 0\ T \mid \varepsilon$



More Left Recursion Elimination

- In general

$$S \to S\,\alpha_1 \mid \ldots \mid S\,\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$$

- All strings derived from S start with one of $\beta_1,\ldots,\beta_m$ and continue with several instances of $\alpha_1,\ldots,\alpha_n$
- Rewrite as

$$S \to \beta_1\,T \mid \ldots \mid \beta_m\,T$$

$$T \to \alpha_1\,T \mid \ldots \mid \alpha_n\,T \mid \varepsilon$$

General Left Recursion

- The grammar

$$S \to A\,\alpha \mid \delta$$

$$A \to S\,\beta$$

is also left-recursive because

$$S \to^{+} S\,\beta\,\alpha$$

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

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 → T + E | T

  T → int | int * T | (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

Left-Factoring Example

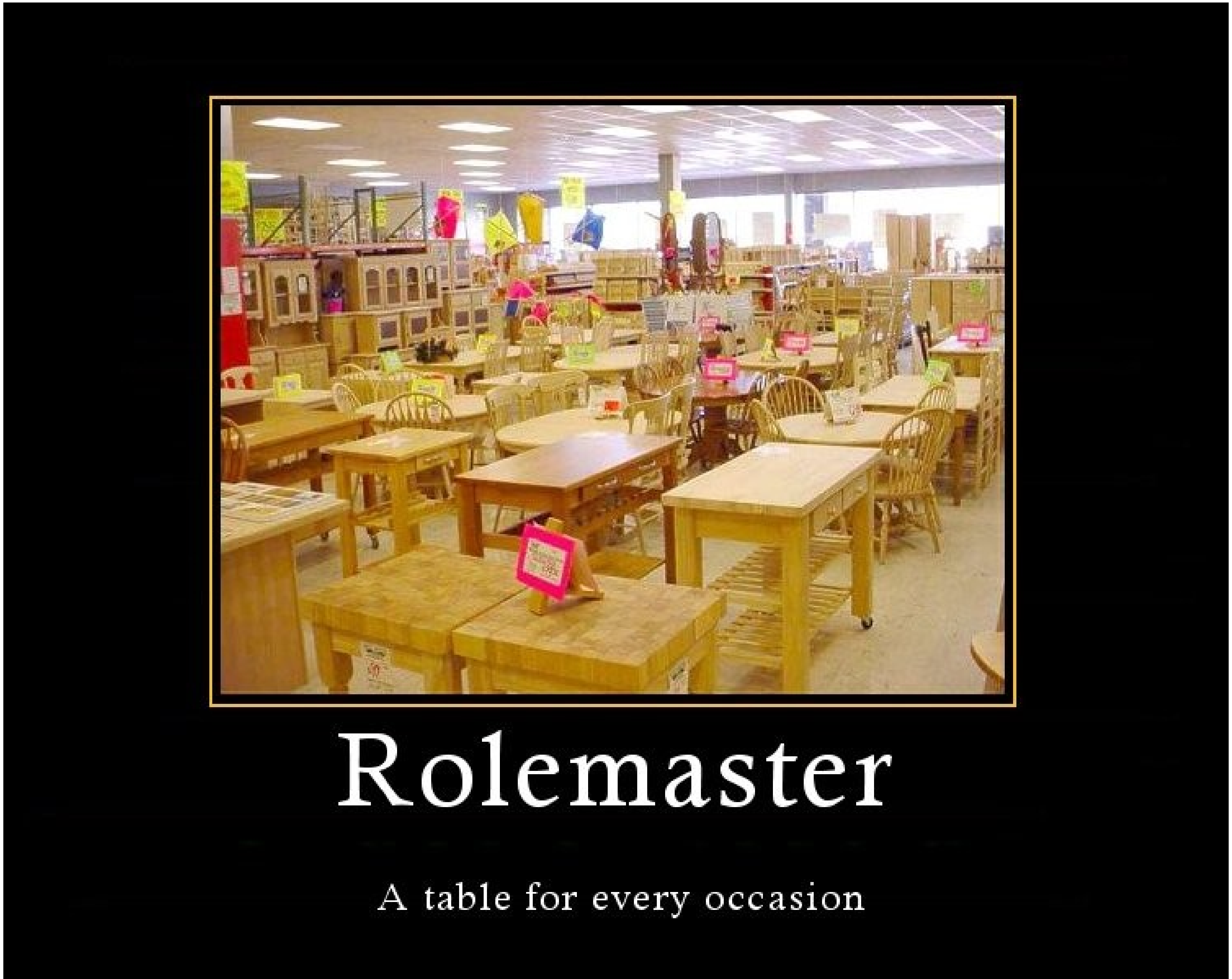
- Recall the grammar

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

- *Factor out* common prefixes of productions

$$E \to T\,X$$
$$X \to + E \mid \varepsilon$$
$$T \to (E) \mid \text{int}\,Y$$
$$Y \to * T \mid \varepsilon$$

Introducing: Parse Tables



LL(1) Parsing Table Example

- Left-factored grammar

E → T X   X → + E | ε

T → (E) | int Y   Y → * T | ε

- 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 → T X"
  - This production can generate an int in the first position

	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 [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 = <S $>
           next = (pointer to tokens)
repeat
  match stack with
  | <X, rest>: if T[X,*next] = Y_1…Y_n
                 then stack ← <Y_1… Y_n rest>
                 else error ()
  | <t, rest>: if t == *next ++
                 then stack ← <rest>
                 else error ()
until stack == < >
```

Stack | Input | Action

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

Stack	Input	Action
E $	int * int $	T X

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

Stack	Input	Action
E $	int * int $	T X
T X $	int * int $	int Y

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

Stack	Input	Action
E $	int * int $	T X
T X $	int * int $	int Y
int Y X $	int * int $	**terminal**

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

Stack	Input	Action
E $	int * int $	T X
T X $	int * int $	int Y
int Y X $	int * int $	**terminal**
Y X $	* int $	* T

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

Stack	Input	Action
E $	int * int $	T X
T X $	int * int $	int Y
int Y X $	int * int $	**terminal**
Y X $	* int $	* T
* T X $	* int $	**terminal**

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

Stack	Input	Action
E $	int * int $	T X
T X $	int * int $	int Y
int Y X $	int * int $	**terminal**
Y X $	* int $	* T
* T X $	* int $	**terminal**
T X $	int $	int Y

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

Stack	Input	Action
E $	int * int $	T X
T X $	int * int $	int Y
int Y X $	int * int $	**terminal**
Y X $	* int $	* T
* T X $	* int $	**terminal**
T X $	int $	int Y
int Y X $	int $	**terminal**

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

Stack	Input	Action
E $	int * int $	T X
T X $	int * int $	int Y
int Y X $	int * int $	**terminal**
Y X $	* int $	* T
* T X $	* int $	**terminal**
T X $	int $	int Y
int Y X $	int $	**terminal**
Y X $	$	ε

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

Stack	Input	Action
E $	int * int $	T X
T X $	int * int $	int Y
int Y X $	int * int $	**terminal**
Y X $	* int $	* T
* T X $	* int $	**terminal**
T X $	int $	int Y
int Y X $	int $	**terminal**
Y X $	$	ε
X $	$	ε

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

Stack	Input	Action
E $	int * int $	T X
T X $	int * int $	int Y
int Y X $	int * int $	terminal
Y X $	* int $	* T
* T X $	* int $	terminal
T X $	int $	int Y
int Y X $	int $	terminal
Y X $	$	ε
X $	$	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)

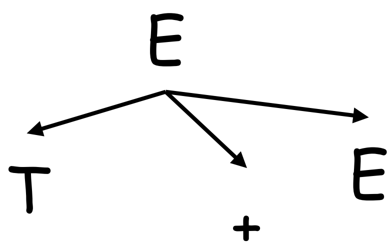
- For two of the following four lines from the 1976 Eagles song **Hotel California**, give enough words to complete the rhyme.
  - *So I called up the captain / "please bring me my wine"*
  - *Mirrors on the ceiling / pink champagne on ice*
  - *And in the master's chambers / they gathered for the feast*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

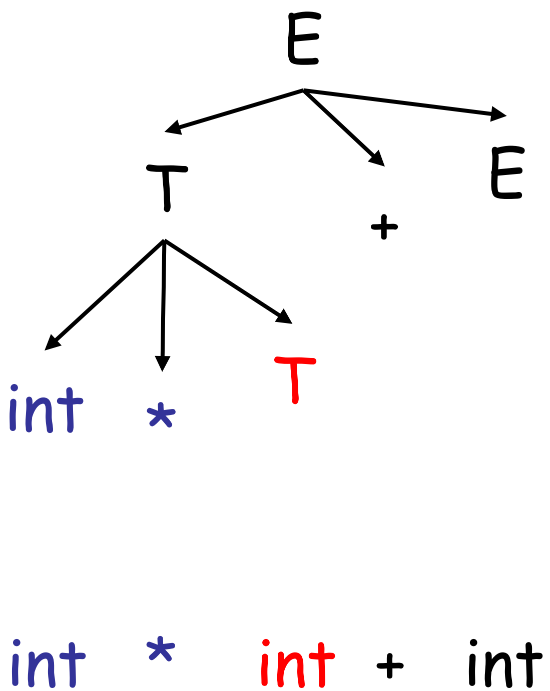
- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal



int * int + int

Top-Down Parsing. Review

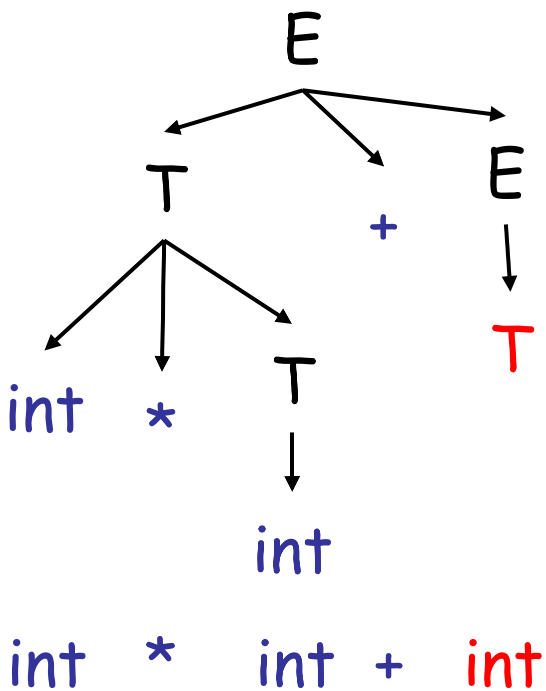
- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal



- 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

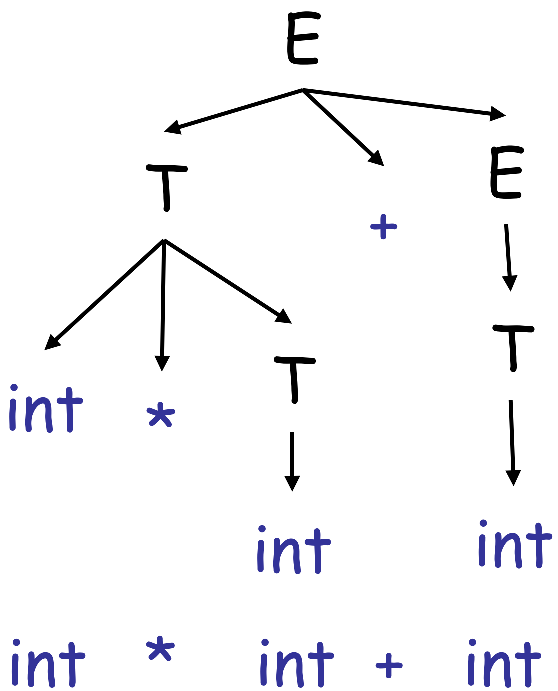
- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal



- 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 $b \in$ **First**(α)

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., bω)
  - Means that b can appear after A in a derivation of the form $S \rightarrow^* \beta A b \omega$
  - We say that $b \in$ <u>**Follow**</u>(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 $\varepsilon \in$ **First**(A) in this case

Computing First Sets

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

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

- For all productions $X \rightarrow A_1 \ldots A_n$
  - 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

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

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

- First sets

First(( ) = { (}

First( )) = {) }

First(int) = { int }

First(+) = { + }

First(*) = { * }

First(T) = {int, (}

First(E) = {int, (}

First(X) = {+, ε }

First(Y) = {*, ε }

Computing Follow Sets

Definition $\text{Follow}(X) = \{ b \mid S \to^{*} \beta\ X\ b\ \omega \}$

- Compute the First sets for all non-terminals first
- Add $ to Follow(S) (if S is the start non-terminal)

- For all productions $Y \to \ldots X\ A_1 \ldots A_n$
  - Add $\text{First}(A_1) - \{\varepsilon\}$ to Follow(X). Stop if $\varepsilon \notin \text{First}(A_1)$
  - Add $\text{First}(A_2) - \{\varepsilon\}$ to Follow(X). Stop if $\varepsilon \notin \text{First}(A_2)$
  - …
  - Add $\text{First}(A_n) - \{\varepsilon\}$ to Follow(X). Stop if $\varepsilon \notin \text{First}(A_n)$
  - Add Follow(Y) to Follow(X)

Example Follow Set Computation

- Recall the grammar

$E \rightarrow T\ X$ $\qquad X \rightarrow +\ E\ |\ \varepsilon$

$T \rightarrow (\ E\ )\ |\ \text{int}\ Y$ $\qquad Y \rightarrow *\ T\ |\ \varepsilon$

- Follow sets

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

Follow(( ) = { int, (} $\qquad$ Follow(E) = {), $}

Follow(X) = {$,) } $\qquad$ Follow(T) = {+,) , $}

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

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

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 $b \in$ First(α) do
    - T[A, b] = α
  - If $\alpha \rightarrow^* \varepsilon$, for each $b \in$ Follow(A) do
    - T[A, b] = α

LL(1) Table Construction Example

- Recall the grammar

$E \rightarrow T\ X \qquad X \rightarrow +\ E\ |\ \varepsilon$

$T \rightarrow (\ E\ )\ |\ \text{int}\ Y \qquad Y \rightarrow *\ T\ |\ \varepsilon$

- Where in the row of Y do we put $Y \rightarrow *\ T$?
  - In the columns of 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

  $E \to T\ X \qquad\qquad X \to +\ E \mid \varepsilon$

  $T \to (\ E\ ) \mid \text{int}\ Y \qquad Y \to *\ T \mid \varepsilon$

- Where in the row of Y we put $Y \to \varepsilon$?
  - In the columns of 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

- WA1 (written homework) due
  - Turn in to drop-box.
- PA2 (Lexer) due
  - You may work in pairs.
- Keep up with the reading ...