

Top-Down Parsing and Intro to Bottom-Up Parsing

Lecture 7

Instructor: Fredrik Kjolstad

Slides based on slides designed by Prof. Alex Aiken

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
 - L means “left-to-right” scan of input
 - L means “leftmost derivation”
 - k means “predict based on k tokens of lookahead”
 - In practice, LL(1) is used

LL(1) vs. Recursive Descent

- In recursive-descent,
 - At each step, many choices of production to use
 - Backtracking used to undo bad choices
- In LL(1),
 - At each step, only one choice of production
 - That is
 - When a non-terminal A is leftmost in a derivation
 - The next input symbol is t
 - There is a unique production $A \rightarrow \alpha$ to use
 - Or no production to use (an error state)
- LL(1) is a recursive descent variant without backtracking

Predictive Parsing and Left Factoring

- Recall the grammar

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

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

- Hard to predict because
 - For T two productions start with int
 - For E it is not clear how to predict
- We need to left-factor the grammar

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

$$E \rightarrow T X$$

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

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

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

LL(1) Parsing Table Example

- Left-factored grammar

$$E \rightarrow TX$$

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

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

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

- The LL(1) parsing table:

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

next input token

leftmost non-terminal

rhs of production to use

LL(1) Parsing Table Example (Cont.)

- Consider the $[E, \text{int}]$ entry
 - “When current non-terminal is E and next input is int , use production $E \rightarrow TX$ ”
 - This can generate an int in the first position
- Consider the $[Y, +]$ entry
 - “When current non-terminal is Y and current token is $+$, get rid of Y ”
 - Y can be followed by $+$ only if $Y \rightarrow \varepsilon$

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 ”

Using Parsing Tables

- Method similar to recursive descent, except
 - For the leftmost non-terminal S
 - We look at the next input token a
 - And choose the production shown at $[S,a]$
- A stack records frontier of parse tree
 - Non-terminals that have yet to be expanded
 - Terminals that have yet to be matched against the input
 - Top of stack = leftmost pending terminal or non-terminal
- Reject on reaching error state
- Accept on end of input & empty stack

LL(1) Parsing Algorithm

```
initialize stack = <S $> and next
repeat
  case stack of
    <X, rest> : if T[X,*next] = Y1...Yn
                  then stack ← <Y1... Yn rest>;
                  else error ();
    <t, rest>  : if t == *next ++
                  then stack ← <rest>;
                  else error ();
until stack == < >
```

LL(1) Parsing Algorithm

\$ marks bottom of stack

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

repeat

case stack of

$\langle X, \text{rest} \rangle$: 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 ();

For terminal t on top of stack, check t matches next input token.

until stack == $\langle \rangle$

For non-terminal X on top of stack, lookup production

Pop X , push production rhs on stack. Note leftmost symbol of rhs is on top of the stack.

LL(1) Parsing Example

$$E \rightarrow T X$$

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

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

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

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

Constructing Parsing Tables: The Intuition

- Consider non-terminal A , production $A \rightarrow \alpha$, & token t
- $T[A, t] = \alpha$ in two cases:
 - If $\alpha \rightarrow^* t \beta$
 - α can derive a t in the first position
 - We say that $t \in \text{First}(\alpha)$
 - If $\alpha \rightarrow^* \varepsilon$ and $S \rightarrow^* \gamma A t \delta$
 - Useful if stack has A , input is t , and A cannot derive t
 - In this case only option is to get rid of A (by deriving ε)
 - Can work only if t can follow A in at least one derivation
 - We say $t \in \text{Follow}(A)$

Computing First Sets

Definition

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

Algorithm sketch:

1. $\text{First}(t) = \{ t \}$
2. $\varepsilon \in \text{First}(X)$
 - if $X \rightarrow \varepsilon$ or
 - if $X \rightarrow A_1 \dots A_n$ and $\varepsilon \in \text{First}(A_i)$ for all $1 \leq i \leq n$
3. $\text{First}(\alpha) \subseteq \text{First}(X)$
 - if $X \rightarrow A_1 \dots A_n \alpha$ and $\varepsilon \in \text{First}(A_i)$ for all $1 \leq i \leq n$

First Sets: Example

1. $\text{First}(t) = \{ t \}$
2. $\varepsilon \in \text{First}(X)$
 - if $X \rightarrow \varepsilon$ or
 - if $X \rightarrow A_1 \dots A_n$ and $\varepsilon \in \text{First}(A_i)$ for all $1 \leq i \leq n$
3. $\text{First}(\alpha) \subseteq \text{First}(X)$
 - if $X \rightarrow A_1 \dots A_n \alpha$ and $\varepsilon \in \text{First}(A_i)$ for all $1 \leq i \leq n$

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

$\text{First}(E) =$

$\text{First}(X) =$

$\text{First}(T) =$

$\text{First}(Y) =$

First Sets: Example

- Recall the grammar

$$E \rightarrow T X$$

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

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

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

- First sets

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

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

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

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

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

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

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

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

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

Computing Follow Sets

- Definition:

$$\text{Follow}(X) = \{ t \mid S \rightarrow^* \beta X t \delta \}$$

- Intuition

- If $X \rightarrow A B$ then $\text{First}(B) \subseteq \text{Follow}(A)$ and
 $\text{Follow}(X) \subseteq \text{Follow}(B)$
 - if $B \rightarrow^* \varepsilon$ then $\text{Follow}(X) \subseteq \text{Follow}(A)$
- If S is the start symbol then $\$ \in \text{Follow}(S)$

Computing Follow Sets (Cont.)

Algorithm sketch:

1. $\$ \in \text{Follow}(S)$
2. $\text{First}(\beta) - \{\varepsilon\} \subseteq \text{Follow}(X)$
 - For each production $A \rightarrow \alpha X \beta$
3. $\text{Follow}(A) \subseteq \text{Follow}(X)$
 - For each production $A \rightarrow \alpha X \beta$ where $\varepsilon \in \text{First}(\beta)$

Follow Sets: Example

- 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

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

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

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

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

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

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

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

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

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

Computing the Follow Sets (for the Non-Terminals)

- Recall the grammar

$$E \rightarrow TX$$

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

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

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

- $\$ \in \text{Follow}(E)$

Computing the Follow Sets (for the Non-Terminals)

- Recall the grammar

$$E \rightarrow TX$$

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

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

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

- $\$ \in \text{Follow}(E)$
- $\text{First}(X) \subseteq \text{Follow}(T)$
- $\text{Follow}(E) \subseteq \text{Follow}(X)$
- $\text{Follow}(E) \subseteq \text{Follow}(T)$ because $\varepsilon \in \text{First}(X)$

Computing the Follow Sets (for the Non-Terminals)

- Recall the grammar

$$E \rightarrow T X$$

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

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

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

- $\$ \in \text{Follow}(E)$
- $\text{First}(X) \subseteq \text{Follow}(T)$
- $\text{Follow}(E) \subseteq \text{Follow}(X)$
- $\text{Follow}(E) \subseteq \text{Follow}(T)$ because $\varepsilon \in \text{First}(X)$
- $) \in \text{Follow}(E)$

Computing the Follow Sets (for the Non-Terminals)

- Recall the grammar

$$E \rightarrow T X$$

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

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

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

- $\$ \in \text{Follow}(E)$
- $\text{First}(X) \subseteq \text{Follow}(T)$
- $\text{Follow}(E) \subseteq \text{Follow}(X)$
- $\text{Follow}(E) \subseteq \text{Follow}(T)$ because $\varepsilon \in \text{First}(X)$
- $) \in \text{Follow}(E)$
- $\text{Follow}(T) \subseteq \text{Follow}(Y)$

Computing the Follow Sets (for the Non-Terminals)

- Recall the grammar

$$E \rightarrow T X$$

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

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

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

- $\$ \in \text{Follow}(E)$
- $\text{First}(X) \subseteq \text{Follow}(T)$
- $\text{Follow}(E) \subseteq \text{Follow}(X)$
- $\text{Follow}(E) \subseteq \text{Follow}(T)$ because $\varepsilon \in \text{First}(X)$
- $) \in \text{Follow}(E)$
- $\text{Follow}(T) \subseteq \text{Follow}(Y)$
- $\text{Follow}(X) \subseteq \text{Follow}(E)$

Computing the Follow Sets (for the Non-Terminals)

- Recall the grammar

$$E \rightarrow T X$$

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

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

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

- $\$ \in \text{Follow}(E)$
- $\text{First}(X) \subseteq \text{Follow}(T)$
- $\text{Follow}(E) \subseteq \text{Follow}(X)$
- $\text{Follow}(E) \subseteq \text{Follow}(T)$ because $\varepsilon \in \text{First}(X)$
- $) \in \text{Follow}(E)$
- $\text{Follow}(T) \subseteq \text{Follow}(Y)$
- $\text{Follow}(X) \subseteq \text{Follow}(E)$
- $\text{Follow}(Y) \subseteq \text{Follow}(T)$

Computing the Follow Sets (for the Non-Terminals)

- Recall the grammar

$$E \rightarrow T X$$

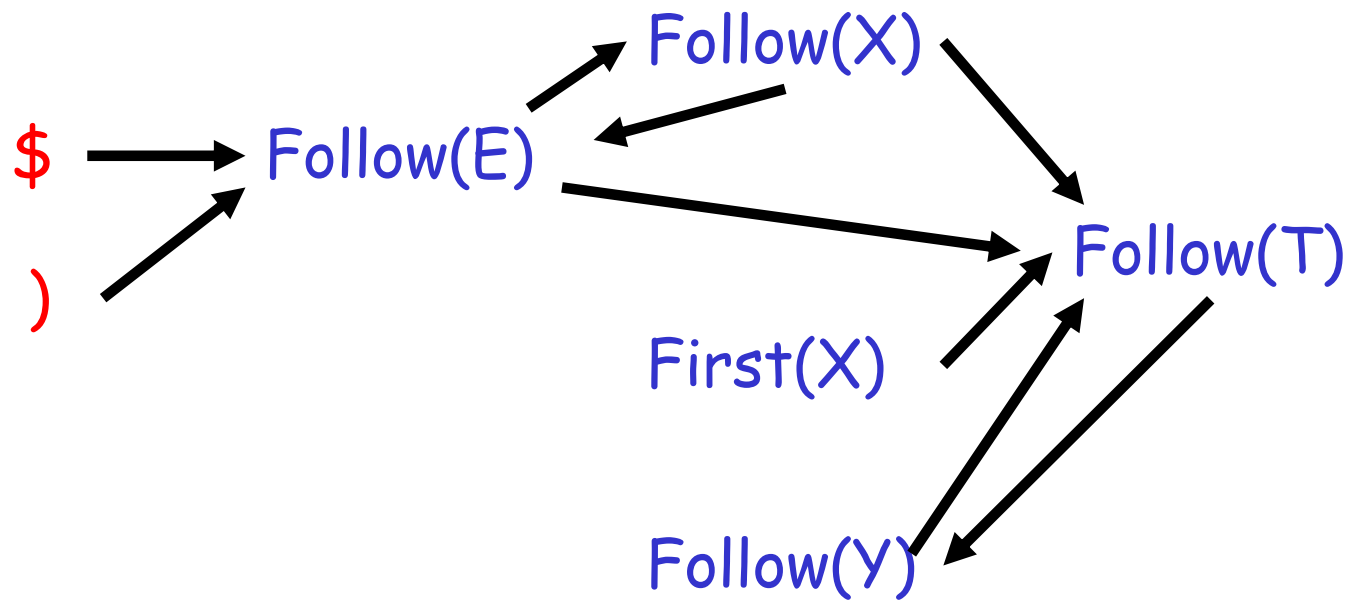
$$T \rightarrow (E) \mid \text{int } Y$$

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

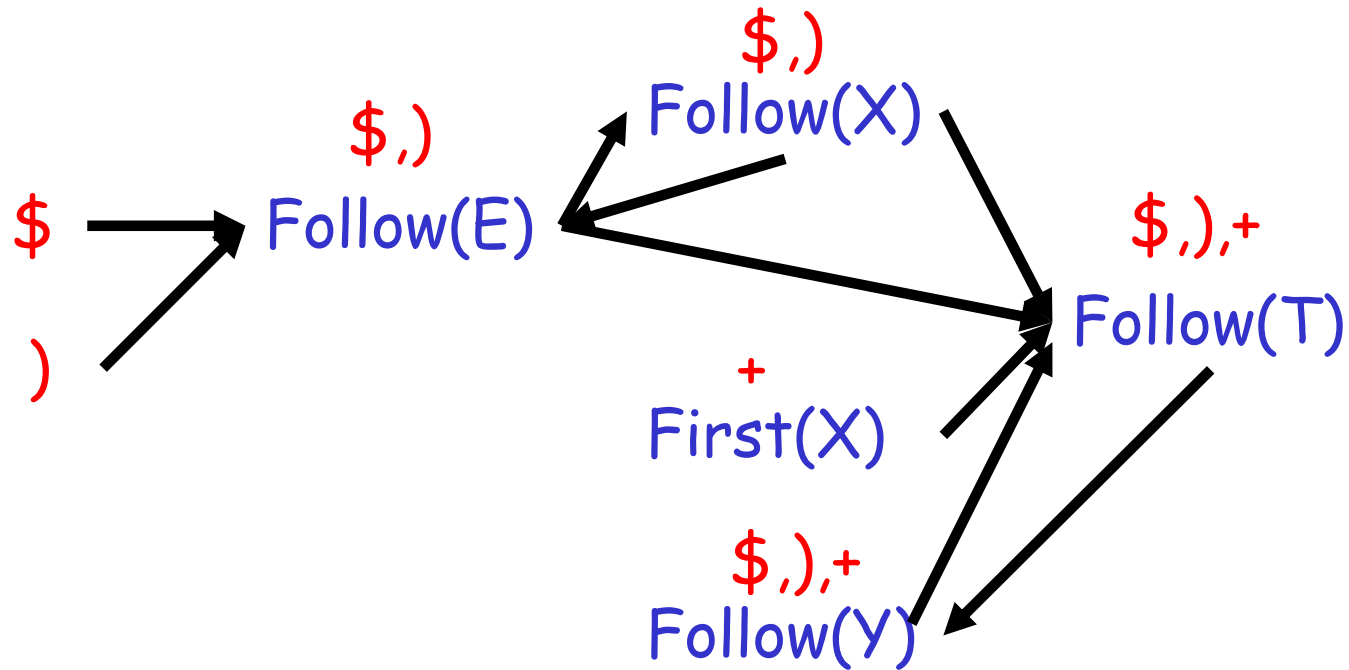
$$Y \rightarrow * T \mid \varepsilon$$

- $\$ \in \text{Follow}(E)$
- $\text{First}(X) \subseteq \text{Follow}(T)$
- $\text{Follow}(E) \subseteq \text{Follow}(X)$
- $\text{Follow}(E) \subseteq \text{Follow}(T)$
- $) \in \text{Follow}(E)$
- $\text{Follow}(T) \subseteq \text{Follow}(Y)$
- $\text{Follow}(X) \subseteq \text{Follow}(E)$
- $\text{Follow}(Y) \subseteq \text{Follow}(T)$

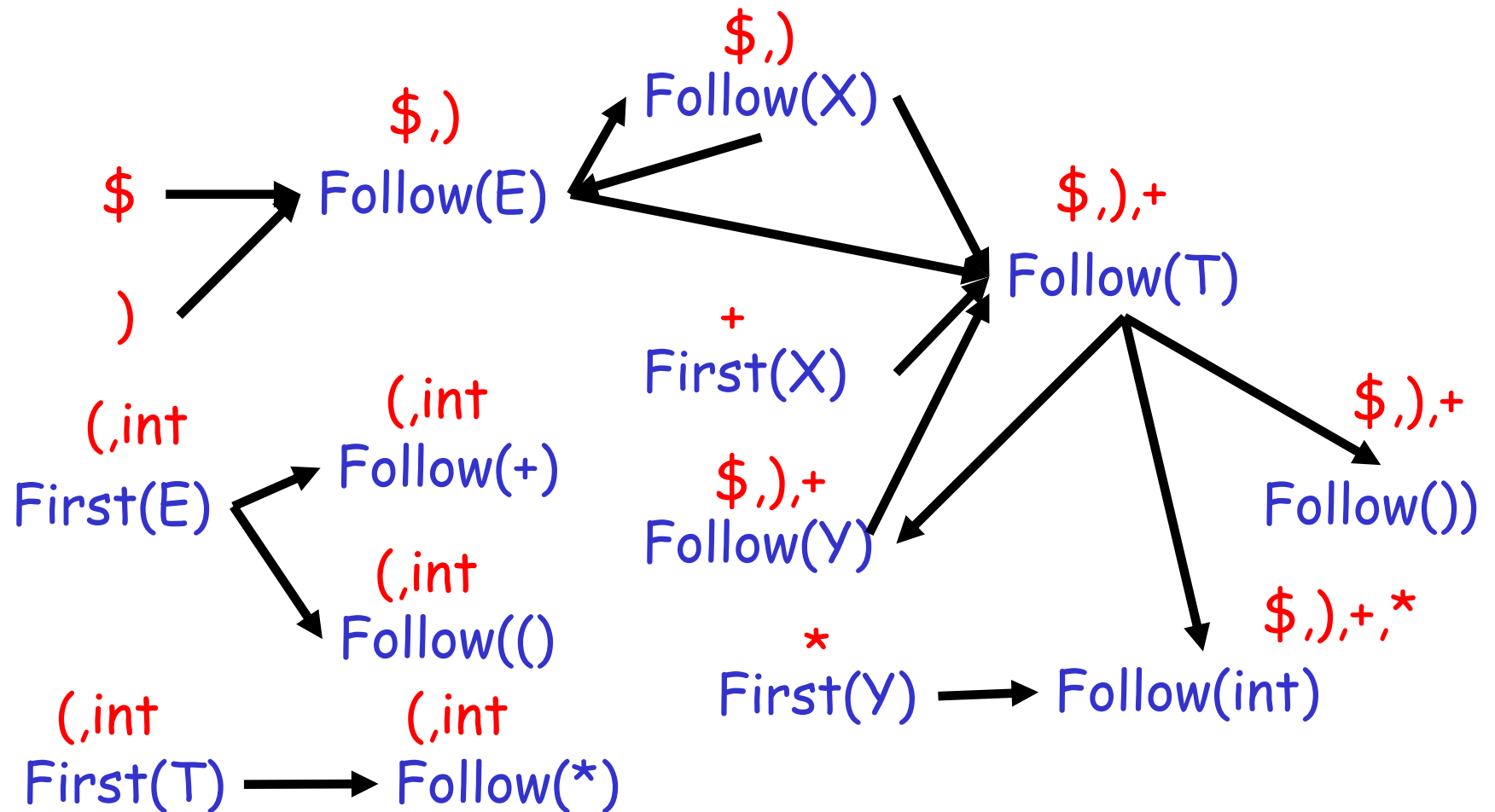
Computing the Follow Sets (for the Non-Terminals)



Computing the Follow Sets (for the Non-Terminals)



Computing the Follow Sets (for all symbols)



Constructing LL(1) Parsing Tables

- Construct a parsing table T for CFG G
- For each production $A \rightarrow \alpha$ in G do:
 - For each terminal $t \in \text{First}(\alpha)$ do
 - $T[A, t] = \alpha$
 - If $\varepsilon \in \text{First}(\alpha)$, then for each $t \in \text{Follow}(A)$ do
 - $T[A, t] = \alpha$
 - If $\varepsilon \in \text{First}(\alpha)$ and $\$ \in \text{Follow}(A)$ do
 - $T[A, \$] = \alpha$

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 CFGs are not LL(1)

Bottom-Up Parsing

- Bottom-up parsing is more general than top-down parsing
 - And just as efficient
 - Builds on ideas in top-down parsing
- Bottom-up is the preferred method
- Concepts today, algorithms next time

An Introductory Example

- Bottom-up parsers don't need left-factored grammars

- Revert to the “natural” grammar for our example:

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

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

- Consider the string: $\text{int} * \text{int} + \text{int}$

The Idea

Bottom-up parsing *reduces* a string to the start symbol by inverting productions:

int * int + int	$T \rightarrow \text{int}$
int * T + int	$T \rightarrow \text{int} * T$
T + int	$T \rightarrow \text{int}$
T + T	$E \rightarrow T$
T + E	$E \rightarrow T + E$
E	

Observation

- Read the productions in reverse (from bottom to top)
- This is a reverse rightmost derivation!

int * int + int

$T \rightarrow \text{int}$

int * T + int

$T \rightarrow \text{int} * T$

T + int

$T \rightarrow \text{int}$

T + T

$E \rightarrow T$

T + E

$E \rightarrow T + E$

E

Important Fact #1

Important Fact #1 about bottom-up parsing:

A bottom-up parser traces a rightmost derivation in reverse

A Bottom-up Parse

int * int + int

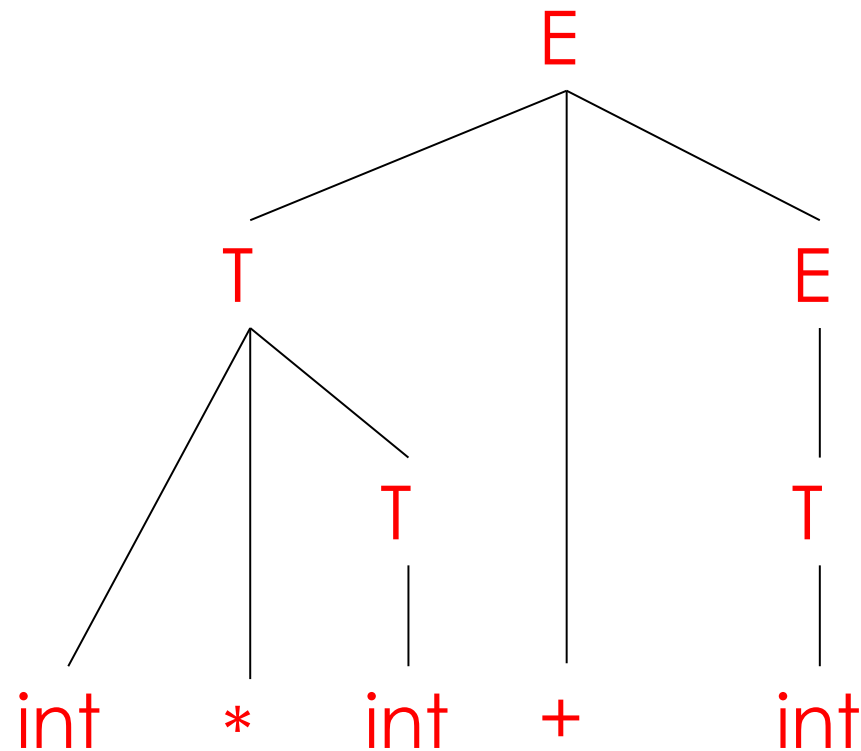
int * T + int

T + int

T + T

T + E

E



A Bottom-up Parse in Detail (1)

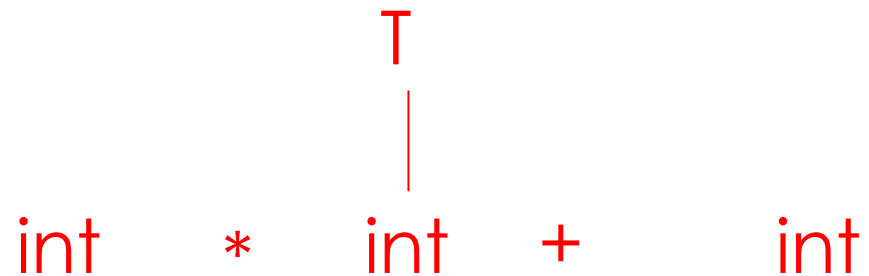
int * int + int

int * int + int

A Bottom-up Parse in Detail (2)

int * int + int

int * T + int

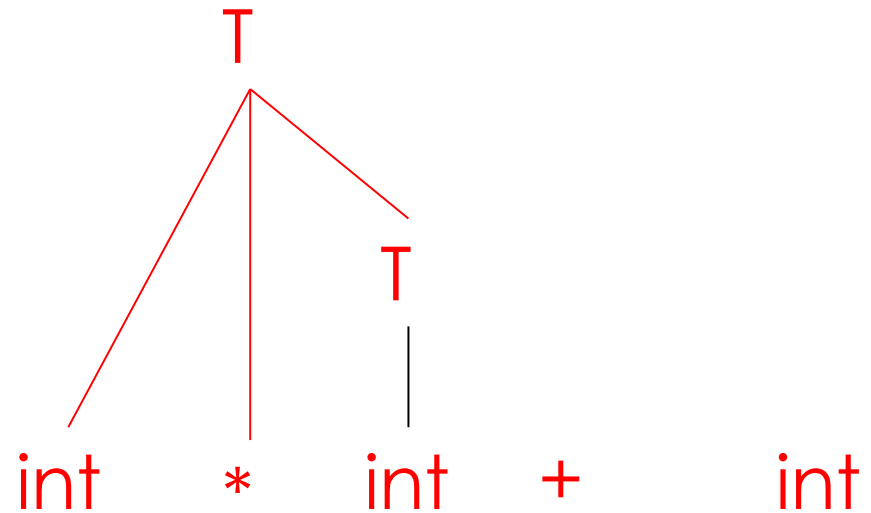


A Bottom-up Parse in Detail (3)

int * int + int

int * T + int

T + int



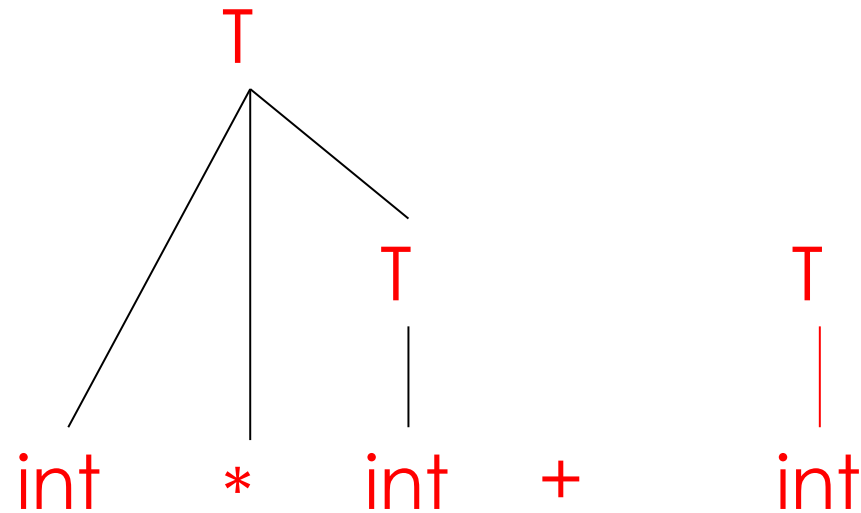
A Bottom-up Parse in Detail (4)

int * int + int

int * T + int

T + int

T + T



A Bottom-up Parse in Detail (5)

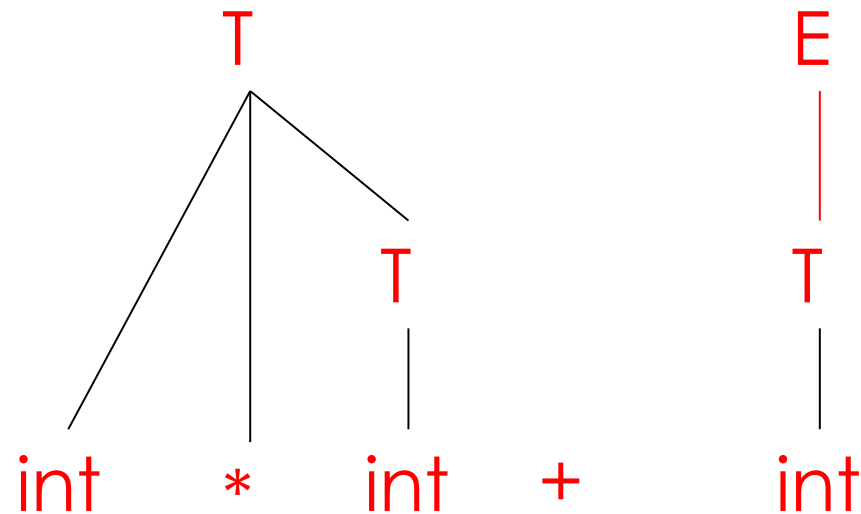
int * int + int

int * T + int

T + int

T + T

T + E



A Bottom-up Parse in Detail (6)

int * int + int

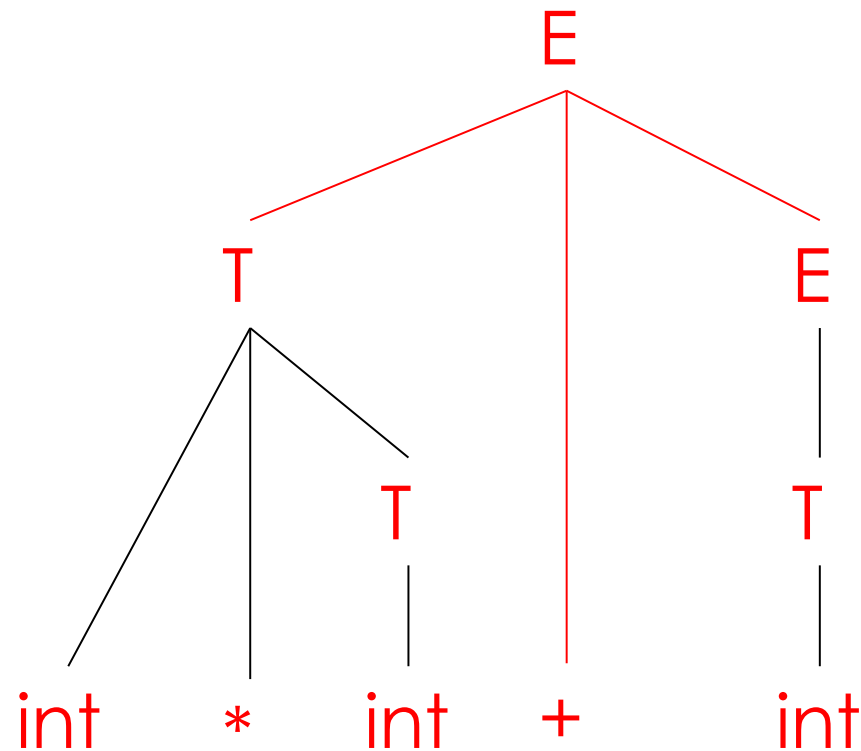
int * T + int

T + int

T + T

T + E

E



A Trivial Bottom-Up Parsing Algorithm

Let I = input string

repeat

 pick a non-empty substring β of I

 where $X \rightarrow \beta$ is a production

 if no such β , backtrack

 replace one β by X in I

until $I = "S"$ (the start symbol) or all possibilities are exhausted

Questions

- Does this algorithm terminate?
- Does the algorithm handle all cases?
- How do we choose the substring to reduce at each step?

Where Do Reductions Happen?

Important Fact #1 has an interesting consequence:

- Let $\alpha\beta\omega$ be a step of a bottom-up parse
- Assume the next reduction is by $X \rightarrow \beta$
- Then ω is a string of terminals

Why? Because $\alpha X \omega \rightarrow \alpha\beta\omega$ is a step in a right-most derivation

Notation

- Idea: Split string into two substrings
 - Right substring is as yet unexamined by parsing (a string of terminals)
 - Left substring has terminals and non-terminals
- The dividing point is marked by a |
 - The | is not part of the string
- Initially, all input is unexamined | $x_1 x_2 \dots x_n$

Shift-Reduce Parsing

Bottom-up parsing uses only two kinds of actions:

Shift

Reduce

Shift

- *Shift*: Move | one place to the right
 - Shifts a terminal to the left string

$ABC|xyz \Rightarrow ABCx|yz$

Reduce

- Apply an inverse production at the right end of the left string
 - If $A \rightarrow xy$ is a production, then

$$Cbxy|ijk \Rightarrow CbA|ijk$$

The Example with Reductions Only

int * int | + int

int * T | + int

reduce $T \rightarrow \text{int}$

reduce $T \rightarrow \text{int} * T$

T + int |

T + T |

T + E |

E |

reduce $T \rightarrow \text{int}$

reduce $E \rightarrow T$

reduce $E \rightarrow T + E$

The Example with Shift-Reduce Parsing

int * int + int	shift
int * int + int	shift
int * int + int	shift
int * int + int	reduce $T \rightarrow \text{int}$
int * T + int	reduce $T \rightarrow \text{int} * T$
T + int	shift
T + int	shift
T + int	reduce $T \rightarrow \text{int}$
T + T	reduce $E \rightarrow T$
T + E	reduce $E \rightarrow T + E$
E	

A Shift-Reduce Parse in Detail (1)

|int * int + int

int * int + int
m

A Shift-Reduce Parse in Detail (2)

|int * int + int

int | * int + int

int * int + int
m

A Shift-Reduce Parse in Detail (3)

| int * int + int

int | * int + int

int * | int + int

int * int + int
m

A Shift-Reduce Parse in Detail (4)

| int * int + int

int | * int + int

int * | int + int

int * int | + int

int * int + int
m

A Shift-Reduce Parse in Detail (5)

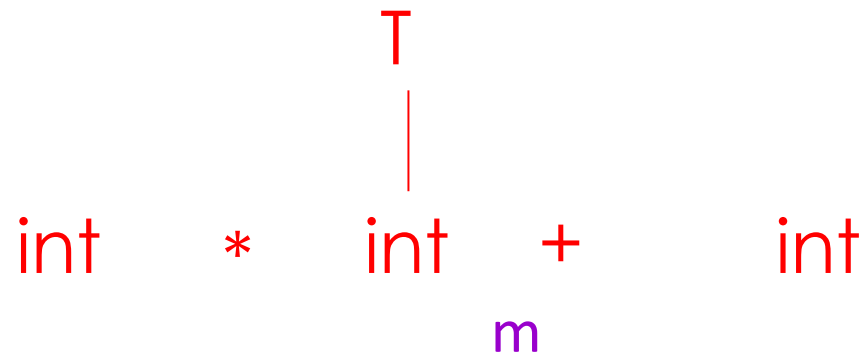
| int * int + int

int | * int + int

int * | int + int

int * int | + int

int * T | + int



A Shift-Reduce Parse in Detail (6)

| int * int + int

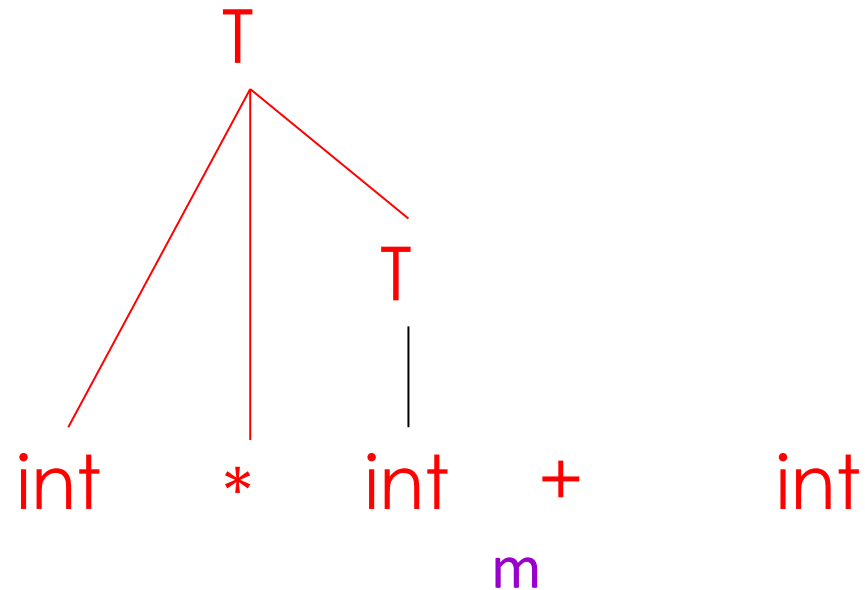
int | * int + int

int * | int + int

int * int | + int

int * T | + int

T | + int



A Shift-Reduce Parse in Detail (7)

| int * int + int

int | * int + int

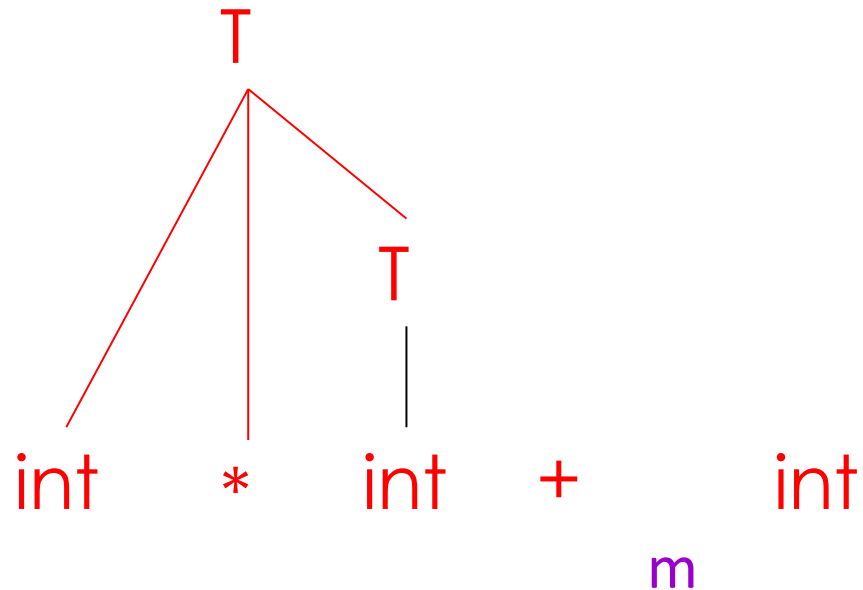
int * | int + int

int * int | + int

int * T | + int

T | + int

T + | int



A Shift-Reduce Parse in Detail (8)

| int * int + int

int | * int + int

int * | int + int

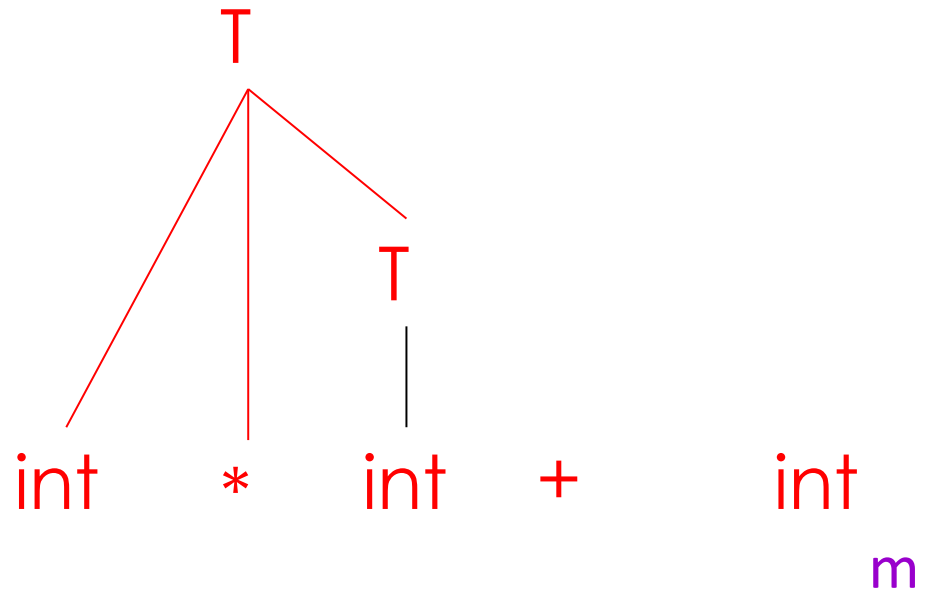
int * int | + int

int * T | + int

T | + int

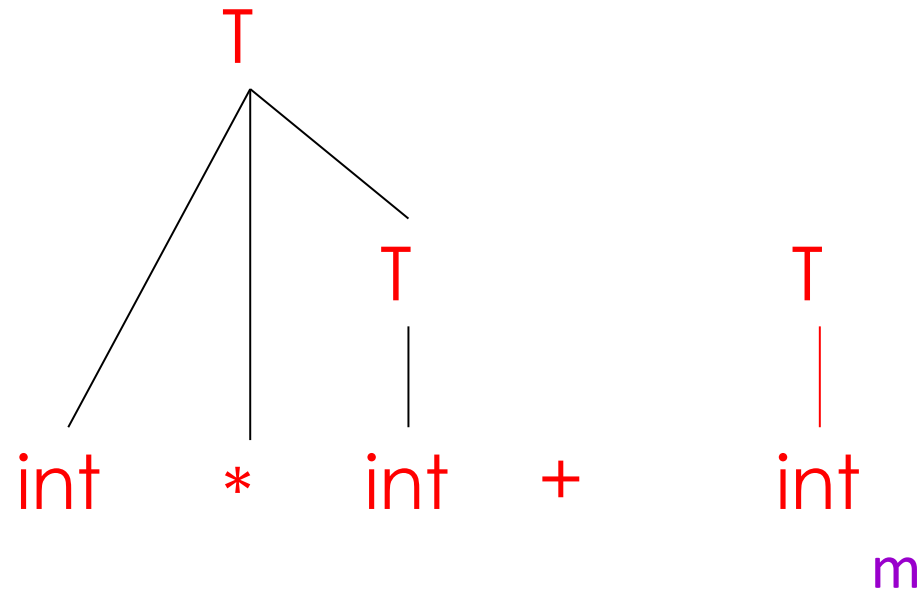
T + | int

T + int |



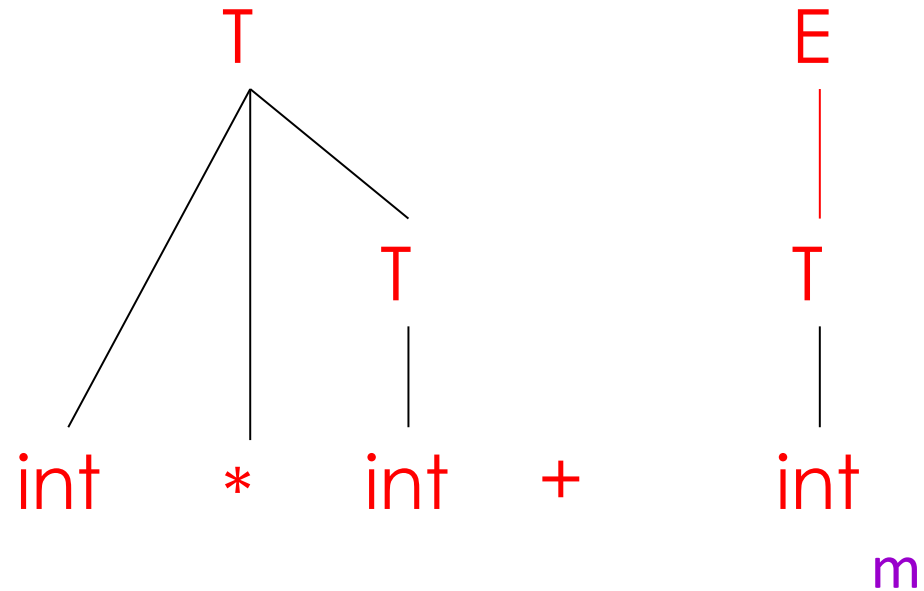
A Shift-Reduce Parse in Detail (9)

| int * int + int
int | * int + int
int * | int + int
int * int | + int
int * T | + int
T | + int
T + | int
T + int |
T + T |



A Shift-Reduce Parse in Detail (10)

| int * int + int
int | * int + int
int * | int + int
int * int | + int
int * T | + int
T | + int
T + | int
T + int |
T + T |
T + E |



A Shift-Reduce Parse in Detail (11)

| int * int + int

int | * int + int

int * | int + int

int * int | + int

int * T | + int

T | + int

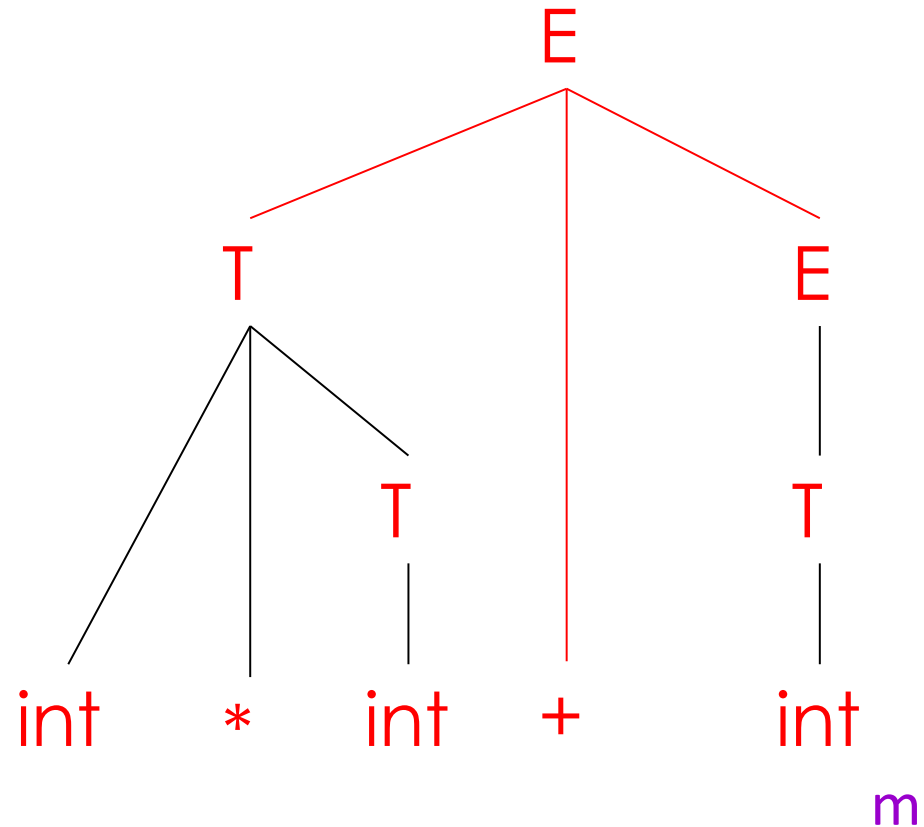
T + | int

T + int |

T + T |

T + E |

E |



The Stack

- Left string can be implemented by a stack
 - Top of the stack is the |
- Shift pushes a terminal on the stack
- Reduce pops 0 or more symbols off of the stack (production rhs) and pushes a non-terminal on the stack (production lhs)

Conflicts

- In a given state, more than one action (shift or reduce) may lead to a valid parse
- If it is legal to shift or reduce, there is a *shift-reduce* conflict
- If it is legal to reduce by two different productions, there is a *reduce-reduce* conflict
- You will see such conflicts in your project!
 - More next time . . .