



# Compiler

## Parsing: Top-Down

*© Copyright 2005, Matthew B. Dwyer and Robby. The syllabus and lectures for this course are copyrighted materials and may not be used in other course settings outside University of Nebraska-Lincoln and Kansas State University in their current form or modified form without express written permission of one of the copyright holders. During this course, students are prohibited from selling notes to or being paid for taking notes by any person or commercial firm without the express written permission of one of the copyright holders.*



# Implementing Parsers

- Two basic approaches
  - Top-down
  - Bottom-up
- Top-Down
  - Easier to understand and program manually
  - Supported by ANTLR
- Bottom-Up
  - Used by most other parser generators



# Parsing Algorithms

- Top-down parser
  - LL(k)
  - Left-to-right scan of input
  - Leftmost derivation
  - k symbols of lookahead
- Bottom-up parser
  - LR(k)
  - Left-to-right scan of input
  - Rightmost derivation (in reverse)
  - k symbols of lookahead



# Comparing Algorithms

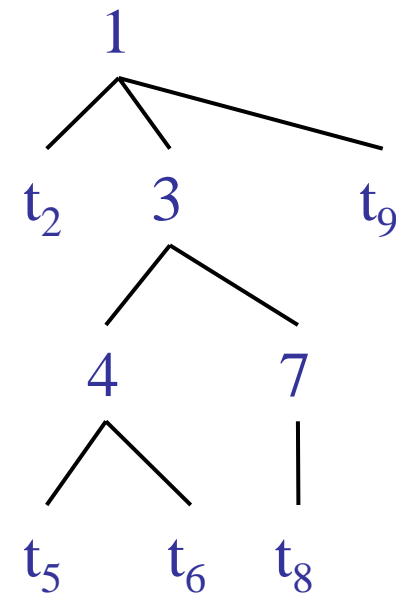
- Which approach to use?
  - $LR(k) > LL(k)$
- Usually one thinks of  $k=1$ 
  - $space(LL) = O(n^k)$
  - $LL(k+1) > LL(k)$
  - $LL(2) > LR(1)$
- LR tolerates ambiguity in grammar
  - LL requires transformation
- LL naturally incorporates actions

# Intro to Top-Down Parsing

- The parse tree is constructed
  - From the top
  - From left to right

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

$t_2$   $t_5$   $t_6$   $t_8$   $t_9$

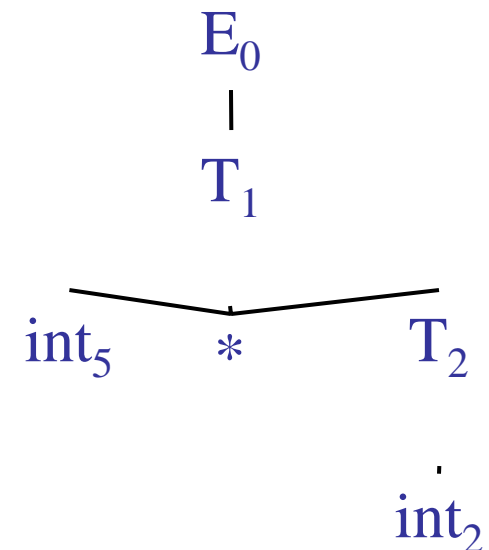


# Recursive Descent Parsing — An Example

- Consider the grammar
$$E \rightarrow T + E \mid T$$
$$T \rightarrow \text{int} \mid \text{int} * T \mid ( E )$$
- Token stream is:  
 $\text{int}_5 * \text{int}_2$
- Start with top-level non-terminal  $E$
- Try the rules for  $E$  in order
- Try  $E_0 \rightarrow T_1 + E_2$
- Then try a rule for  $T_1 \rightarrow ( E_3 )$ 
  - But ( does not match input token  $\text{int}_5$
- Try  $T_1 \rightarrow \text{int}$ .  
Token matches.
  - But + after  $T_1$  does not match input token \*
- Try  $T_1 \rightarrow \text{int} * T_2$ 
  - This will match but + after  $T_1$  will be unmatched

# Recursive Descent Parsing – An Example (Cont.)

- Has exhausted the choices for  $T_1$ 
  - Backtrack to choice for  $E_0$
- 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  $T_2 \rightarrow \text{int}$





# Recursive Descent Parsing – Preliminaries

- Let Token be the type of tokens
  - Special tokens INT, LPAREN, RPAREN, PLUS, TIMES
- Let the field **next** point to the next token



# Recursive Descent Parser (2)

- Define boolean functions that check the token string for a match of
  - A given token terminal

```
boolean term(Token tok) {  
    boolean result = next.equals(tok);  
    next = nextToken(next);  
    return result;  
}
```
  - A given production of S (the  $n^{\text{th}}$ )

```
boolean Sn() { ... }
```
  - Any production of S:

```
boolean S() { ... }
```
- These functions advance **next**



# Recursive Descent Parser (3)

- For production  $E \rightarrow T$

```
boolean E1() { return T(); }
```

- For production  $E \rightarrow T + E$

```
boolean E2() { return T() && term(PLUS) && E(); }
```

- For all productions of E (with backtracking)

```
boolean E() {  
    Token save;  
    save = next;  
    if (E1()) return true;  
    save = next;  
    if (E2()) return true;  
    return false;  
}
```



# Recursive Descent Parser (4)

- Functions for non-terminal T

```
boolean T1() {
    return term(LPAREN) && E() && term(RPAREN);
}
boolean T2() { return term(INT) && term(TIMES) && T(); }
boolean T3() { return term(INT); }

boolean T() {
    Token save;
    save = next;
    if (T1()) return true;
    save = next;
    if (T2()) return true;
    save = next;
    if (T3()) return true;
    return false;
}
```



# Recursive Descent Parsing – Notes

- To start the parser
  - Initialize next to point to first token
  - Invoke E()
- Notice how this simulates our previous example
- Easy to implement by hand
- But does not always work ...

# When Recursive Descent Does Not Work

- Consider a production  $S \rightarrow S a$

```
boolean S1() { return S() && term(a); }  
boolean S() { return S1(); }
```

- $S()$  will get into an infinite loop
- A *left-recursive grammar* has a non-terminal  $S$   
 $S \rightarrow^+ S\alpha$  for some  $\alpha$
- Recursive descent does not work in such cases

# Elimination of Left Recursion

- Consider the left-recursive grammar

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

- S generates all strings starting with a  $\beta$  and followed by a number of  $\alpha$
- Can rewrite using right-recursion

$$S \rightarrow \beta S'$$

$$S' \rightarrow \alpha S' \mid \varepsilon$$

# More Elimination of Left-Recursion

- 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  $\beta_1, \dots, \beta_m$  and continue with several instances of  $\alpha_1, \dots, \alpha_n$

- Rewrite as

$$\begin{aligned} S &\rightarrow \beta_1 S' \mid \dots \mid \beta_m S' \\ S' &\rightarrow \alpha_1 S' \mid \dots \mid \alpha_n S' \mid \varepsilon \end{aligned}$$



# 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 *indirect* left-recursion can also be eliminated
- See book for general algorithm



# Summary of Recursive Descent Parsing

- 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
- In practice, backtracking is eliminated by restricting the grammar



# 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(2) is used



# LL(1) Languages

- In recursive-descent, for each non-terminal and input token there may be a choice of production
- LL(1) means that for each non-terminal and token there is only one production
- Can be specified via 2D tables
  - One dimension for current non-terminal to expand
  - One dimension for next token
  - A table entry contains one production



# Predictive Parsing and Left Factoring

- Recall the grammar

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

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

- Hard 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

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

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

- Factor out *common prefixes* of productions

$$E \rightarrow T X$$

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

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

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

# LL(1) Parsing Table Example

- Left-factored grammar

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

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

- The LL(1) parsing table:

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

# LL(1) Parsing Table Example (Cont.)

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

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

# LL(1) Parsing Table Example (Cont.)

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

- Consider the [Y,+] entry
  - “When current non-terminal is Y and current token is +, get rid of Y”
  - Y can be followed by + only in a derivation in which  $Y \rightarrow \epsilon$



# 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 each non-terminal  $S$
  - look at the next token  $a$
  - choose the production shown at  $[S,a]$
- use a stack to keep track of pending non-terminals
- reject when error state
- accept when end-of-input ( $\$$ )



# 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.equals(next)
      then { next = nextToken(next);
            stack = <rest>; }
      else error();
until stack == < >
```

# LL(1) Parsing Example

<u>Stack</u>	<u>Input</u>	<u>Action</u>
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 \$	\$	$\epsilon$
X \$	\$	$\epsilon$
\$	\$	ACCEPT



# Constructing Parsing Tables

- LL(1) languages are those defined by a parsing table for the LL(1) algorithm
- No table entry can be multiply defined
  
- We want to generate parsing tables from CFG

# Constructing Parsing Tables (Cont.)

- If  $A \rightarrow \alpha$ , where in the column of  $A$  does  $\alpha$  go?
- In the column of  $t$  where  $t$  can start a string derived from  $\alpha$

$$\alpha \rightarrow^* t \beta$$

□ We say that  $t \in \text{First}(\alpha)$

- In the column of  $t$  if  $\alpha$  is  $\epsilon$  and  $t$  can follow an  $A$

$$S \rightarrow^* \beta A t \delta$$

□ 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 (see book for details):

1. for all terminals  $t$  do  $\text{First}(t) \leftarrow \{ t \}$
2. for each production  $X \rightarrow \varepsilon$  do  $\text{First}(X) \leftarrow \{ \varepsilon \}$
3. if  $X \rightarrow A_1 \dots A_n \alpha$  and  $\varepsilon \in \text{First}(A_i)$ ,  $1 \leq i \leq n$  do
  - add  $\text{First}(\alpha)$  to  $\text{First}(X)$
4. for each  $X \rightarrow A_1 \dots A_n$  s.t.  $\varepsilon \in \text{First}(A_i)$ ,  $1 \leq i \leq n$  do
  - add  $\varepsilon$  to  $\text{First}(X)$
5. repeat steps 4 & 5 until no **First** set can be grown

# 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  $S$  is the start symbol then  $\$ \in \text{Follow}(S)$
- 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  $\text{Follow}(X) \subseteq \text{Follow}(A)$



# Computing Follow Sets (Cont.)

Algorithm sketch:

1.  $\text{Follow}(S) \leftarrow \{ \$ \}$
2. For each production  $A \rightarrow \alpha X \beta$ 
  - add  $\text{First}(\beta) - \{ \epsilon \}$  to  $\text{Follow}(X)$
3. For each  $A \rightarrow \alpha X \beta$  where  $\epsilon \in \text{First}(\beta)$ 
  - add  $\text{Follow}(A)$  to  $\text{Follow}(X)$
  - repeat step(s) until no  $\text{Follow}$  set grows

# 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

$$\text{Follow}( + ) = \{ \text{int}, ( \}$$
$$\text{Follow}( ( ) = \{ \text{int}, ( \}$$
$$\text{Follow}( X ) = \{ \$, ) \}$$
$$\text{Follow}( ) ) = \{ +, ) , \$ \}$$
$$\text{Follow}( \text{int} ) = \{ *, +, ) , \$ \}$$
$$\text{Follow}( * ) = \{ \text{int}, ( \}$$
$$\text{Follow}( E ) = \{ ), \$ \}$$
$$\text{Follow}( T ) = \{ +, ) , \$ \}$$
$$\text{Follow}( Y ) = \{ +, ) , \$ \}$$

# Constructing LL(1) Parsing Tables

- 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  $\varepsilon \in \text{First}(\alpha)$ , 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, left-recursive, not left-factored, ...
- Most programming language grammars are not LL(1)
- There are tools that build LL(1) tables