

Top-Down Parsing

CS164 Lecture 6-7

Profs. Necula CS 164

1

Review

- A parser consumes a sequence of tokens s and produces a parse tree
- Issues:
 - How do we recognize that $s \in L(G)$?
 - A parse tree of s describes how $s \in L(G)$
 - Ambiguity: more than one parse tree (interpretation) for some string s
 - Error: no parse tree for some string s
 - How do we construct the parse tree?

Profs. Necula CS 164

2

Ambiguity

- Grammar

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

- Strings

$\text{int} + \text{int} + \text{int}$

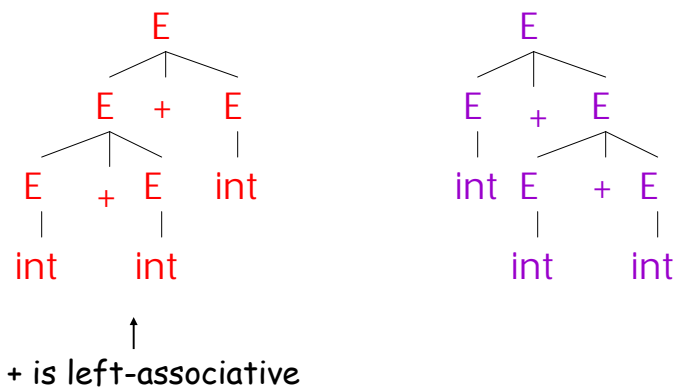
$\text{int} * \text{int} + \text{int}$

Profs. Necula CS 164

3

Ambiguity. Example

The string $\text{int} + \text{int} + \text{int}$ has two parse trees

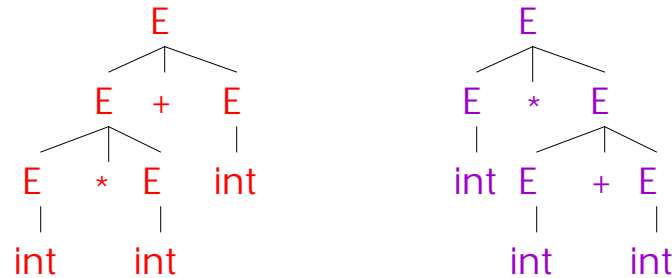


Profs. Necula CS 164

4

Ambiguity. Example

The string `int * int + int` has two parse trees



↑
* has higher precedence than +

Profs. Necula CS 164

5

Ambiguity (Cont.)

- A grammar is *ambiguous* if it has more than one parse tree for some string
 - Equivalently, there is more than one right-most or left-most derivation for some string
- Ambiguity is bad
 - Leaves meaning of some programs ill-defined
- Ambiguity is common in programming languages
 - Arithmetic expressions
 - IF-THEN-ELSE

Profs. Necula CS 164

6

Dealing with Ambiguity

- There are several ways to handle ambiguity
- Most direct method is to rewrite the grammar unambiguously

$E \rightarrow E + T \mid T$

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

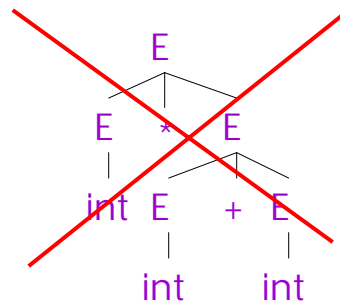
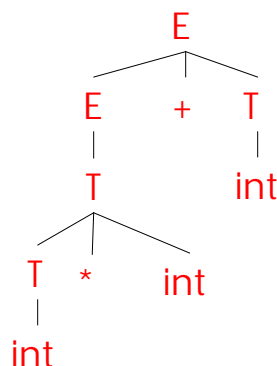
- Enforces precedence of $*$ over $+$
- Enforces left-associativity of $+$ and $*$

Profs. Necula CS 164

7

Ambiguity. Example

The $\text{int} * \text{int} + \text{int}$ has only one parse tree now



Profs. Necula CS 164

8

Ambiguity: The Dangling Else

- Consider the grammar

$E \rightarrow$ if E then E
| if E then E else E
| OTHER

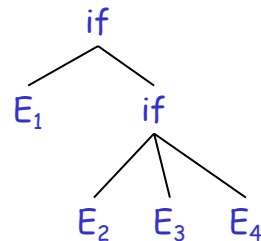
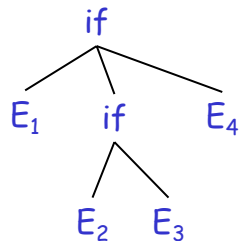
- This grammar is also ambiguous

The Dangling Else: Example

- The expression

if E_1 then if E_2 then E_3 else E_4

has two parse trees



- Typically we want the second form

The Dangling Else: A Fix

- `else` matches the closest unmatched `then`
- We can describe this in the grammar (distinguish between matched and unmatched "then")

```

E → MIF          /* all then are matched */
   | UIF          /* some then are unmatched */
MIF → if E then MIF else MIF
     | OTHER
UIF → if E then E
     | if E then MIF else UIF
  
```

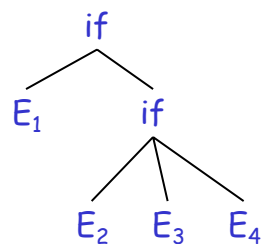
- Describes the same set of strings

Profs. Necla CS 164

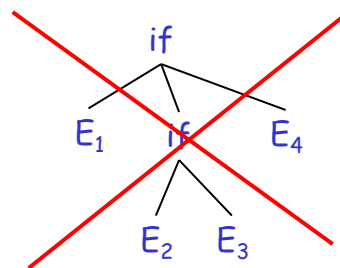
11

The Dangling Else: Example Revisited

- The expression `if E1 then if E2 then E3 else E4`



- A valid parse tree (for a `UIF`)



- Not valid because the `then` expression is not a `MIF`

Profs. Necla CS 164

12

Ambiguity

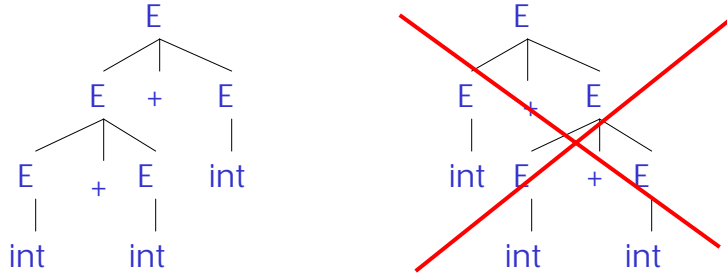
- No general techniques for handling ambiguity
- Impossible to convert automatically an ambiguous grammar to an unambiguous one
- Used with care, ambiguity can simplify the grammar
 - Sometimes allows more natural definitions
 - We need disambiguation mechanisms

Precedence and Associativity Declarations

- Instead of rewriting the grammar
 - Use the more natural (ambiguous) grammar
 - Along with disambiguating declarations
- Most tools allow precedence and associativity declarations to disambiguate grammars
- Examples ...

Associativity Declarations

- Consider the grammar $E \rightarrow E + E \mid \text{int}$
- Ambiguous: two parse trees of $\text{int} + \text{int} + \text{int}$



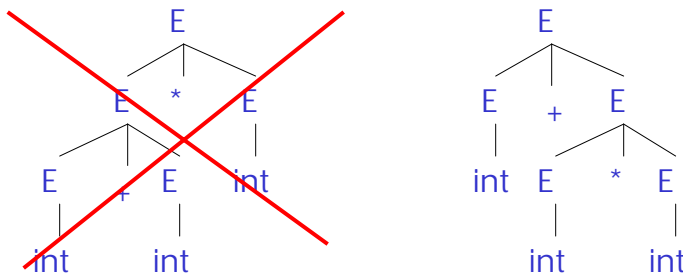
- Left-associativity declaration: `%left +`

Profs. Necula CS 164

15

Precedence Declarations

- Consider the grammar $E \rightarrow E + E \mid E * E \mid \text{int}$
- And the string $\text{int} + \text{int} * \text{int}$



- Precedence declarations: `%left +`
`%left *`

Profs. Necula CS 164

16

Review

- We can specify language syntax using *CFG*
- A parser will answer whether $s \in L(G)$
- ... and will build a parse tree
- ... and pass on to the rest of the compiler

- Next:
 - How do we answer $s \in L(G)$ and build a parse tree?

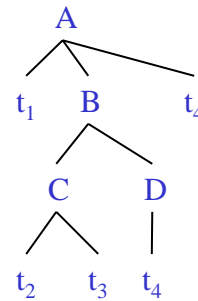
Top-Down Parsing

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



Profs. Necula CS 164

19

Recursive Descent Parsing

- Consider the grammar
$$E \rightarrow T + E \mid T$$
$$T \rightarrow (E) \mid \text{int} \mid \text{int} * T$$
- Token stream is: $\text{int} * \text{int}$
- Start with top-level non-terminal E

- Try the rules for E in order

Profs. Necula CS 164

20

Recursive Descent Parsing. Example (Cont.)

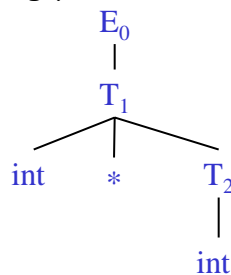
- Try $E_0 \rightarrow T_1 + E_2$
- Then try a rule for $T_1 \rightarrow (E_3)$
 - But $($ does not match input token int
- Try $T_1 \rightarrow int$. Token matches.
 - But $+$ after T_1 does not match input token $*$
- Try $T_1 \rightarrow int * T_2$
 - This will match but $+$ after T_1 will be unmatched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

Profs. Necula CS 164

21

Recursive Descent Parsing. Example (Cont.)

- 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



Profs. Necula CS 164

22

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

Profs. Necula CS 164

23

When Recursive Descent Does Not Work

- Consider a production $S \rightarrow S \alpha$:
 - In the process of parsing S we try the above rule
 - What goes wrong?
- A left-recursive grammar has a non-terminal S
 $S \rightarrow^+ S \alpha$ for some α
- Recursive descent does not work in such cases
 - It goes into an infinite loop

Profs. Necula CS 164

24

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

$$S \rightarrow \beta S'$$

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

Profs. Necula CS 164

25

Elimination of Left-Recursion. Example

- Consider the grammar

$$S \rightarrow 1 \mid S 0 \quad (\beta = 1 \text{ and } \alpha = 0)$$

can be rewritten as

$$S \rightarrow 1 S'$$

$$S' \rightarrow 0 S' \mid \varepsilon$$

Profs. Necula CS 164

26

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 β_1, \dots, β_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 left-recursion can also be eliminated
- See book, Section 4.3 for general algorithm

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
- Often, we can avoid backtracking ...

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) 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 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
 - A table entry contains one production

Profs. Necula CS 164

31

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

Profs. Necula CS 164

32

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

$$E \rightarrow TX$$

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

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

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

Profs. Necula CS 164

33

LL(1) Parsing Table Example

- Left-factored grammar

$$E \rightarrow TX$$

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

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

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

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

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

Profs. Necula CS 164

34

LL(1) Parsing Table Example (Cont.)

- 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 place
- 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

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
 - 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,$> and next (pointer to tokens)
repeat
  case stack of
    <X, rest> : if T[X,*next] =  $Y_1 \dots Y_n$ 
                then stack  $\leftarrow$  < $Y_1 \dots Y_n$  rest>;
                else error ();
    <t, rest> : if t == *next ++
                then stack  $\leftarrow$  <rest>;
                else error ();
until stack == < >
```

LL(1) Parsing Example

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

Profs. Necula CS 164

39

Constructing Parsing Tables

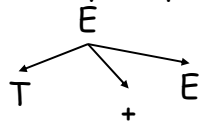
- LL(1) languages are those defined by a parsing table for the LL(1) algorithm
- No table entry can be multiply defined
- Once we have the table
 - The parsing algorithm is simple and fast
 - No backtracking is necessary
- We want to generate parsing tables from CFG

Profs. Necula CS 164

40

Top-Down Parsing. Review

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



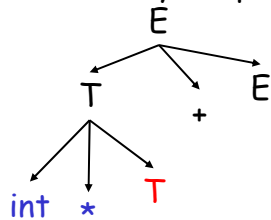
int * int + int

Profs. Necula CS 164

41

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

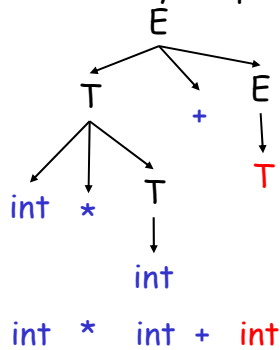
int * int + int

Profs. Necula CS 164

42

Top-Down Parsing. Review

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



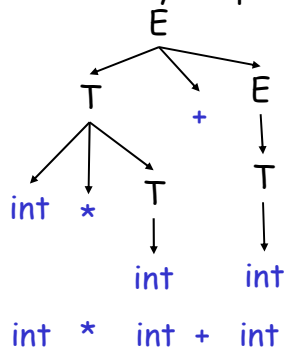
- The leaves at any point form a string $\beta A \gamma$
 - β contains only terminals
 - The input string is $\beta b \delta$
 - The prefix β matches
 - The next token is b

Profs. Necula CS 164

43

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

Profs. Necula CS 164

44

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:

1. 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 $b \in \text{First}(\alpha)$

Or...

Profs. Necula CS 164

45

Constructing Predictive Parsing Tables (Cont.)

2. 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

Profs. Necula CS 164

46

Computing First Sets

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

1. $\text{First}(b) = \{ b \}$

2. For all productions $X \rightarrow A_1 \dots A_n$

- Add $\text{First}(A_1) - \{ \epsilon \}$ to $\text{First}(X)$. Stop if $\epsilon \notin \text{First}(A_1)$
- Add $\text{First}(A_2) - \{ \epsilon \}$ to $\text{First}(X)$. Stop if $\epsilon \notin \text{First}(A_2)$
- ...
- Add $\text{First}(A_n) - \{ \epsilon \}$ to $\text{First}(X)$. Stop if $\epsilon \notin \text{First}(A_n)$
- Add ϵ to $\text{First}(X)$

First Sets. Example

- Recall the grammar

$E \rightarrow T X$

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

$X \rightarrow + E \mid \epsilon$

$Y \rightarrow * T \mid \epsilon$

- First sets

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

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

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

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

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

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

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

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

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

Computing Follow Sets

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

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

Follow Sets. Example

- Recall the grammar

$$\begin{array}{ll} E \rightarrow TX & X \rightarrow + E \mid \epsilon \\ T \rightarrow (E) \mid \text{int } Y & Y \rightarrow * T \mid \epsilon \end{array}$$

- Follow sets

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

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

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

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

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 $b \in \text{First}(\alpha)$ do
 - $T[A, b] = \alpha$
 - If $\alpha \rightarrow^* \epsilon$, for each $b \in \text{Follow}(A)$ do
 - $T[A, b] = \alpha$

Constructing LL(1) Tables. Example

- Recall the grammar
$$\begin{array}{ll} E \rightarrow TX & X \rightarrow + E \mid \epsilon \\ T \rightarrow (E) \mid \text{int } Y & Y \rightarrow * T \mid \epsilon \end{array}$$
- Where in the line of Y we put $Y \rightarrow * T$?
 - In the lines of $\text{First}(*T) = \{ * \}$
- Where in the line of Y we put $Y \rightarrow \epsilon$?
 - In the lines of $\text{Follow}(Y) = \{ \$, +,) \}$

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)
- There are tools that build LL(1) tables

Review

- For some grammars there is a simple parsing strategy
 - Predictive parsing (LL(1))
 - Once you build the LL(1) table, you can write the parser by hand
- Next: a more powerful parsing strategy for grammars that are not LL(1)