

Top-down Parsing

CMPT 379: Compilers

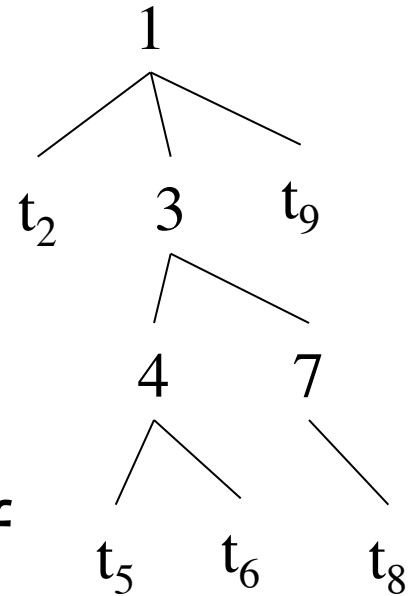
Instructor: Anoop Sarkar

anoopsarkar.github.io/compiler-class

Intro to Top-Down Parsing

- The parse tree is constructed
 - From the top
 - From the left to right
- Terminals are seen in the order of appearance in the token stream

$t_2 t_5 t_6 t_8 t_9$



Recursive Descent Parsing

- 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 from top-level non-terminal E
 - Try the rules for E in order

Recursive Descent Parsing

$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow \text{int}$

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

$T \rightarrow (E)$

Input:

$\text{int}_5 * \text{int}_2$

Try $E_0 \rightarrow T_1 + E_2$

Try $T_1 \rightarrow \text{int}$

Token int matches!

but $+$ does not match to input

Failure

Try $T_1 \rightarrow \text{int} * T_2$

Tokens int and $*$ match

Try $T_3 \rightarrow \text{int}$

Token int matches

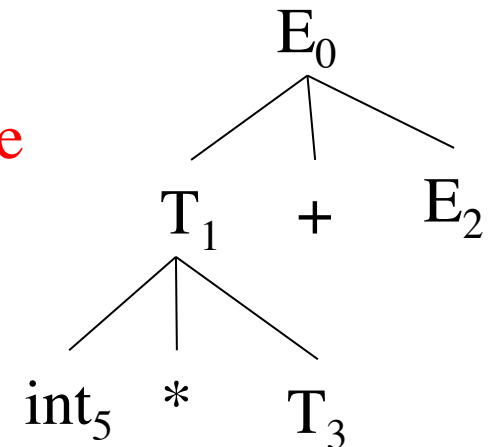
input is matched but tree should match $+ E_2$ **Failure**

Try $T_1 \rightarrow (E_3)$

Token $($ does not match **Failure**

has exhausted the choices for T_1

backtrack to choices for E_0



Recursive Descent Parsing

Try: $E_0 \rightarrow T_1$

Try $T_1 \rightarrow \text{int}$

Token **int** matches!

but no non-terminals left and

the input is not matched completely

Failure

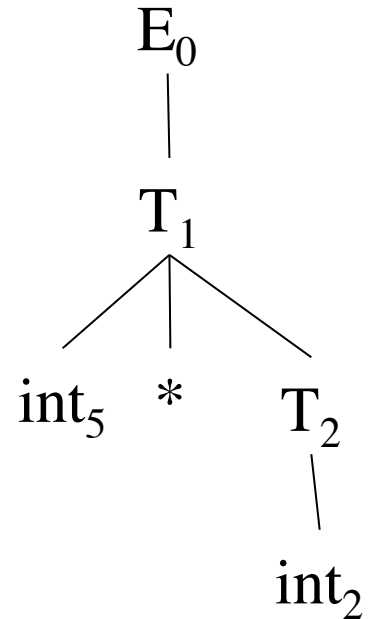
Try $T_1 \rightarrow \text{int} * T_2$

Tokens **int** , ***** match

Try $T_2 \rightarrow \text{int}$

Token **int** matches!

**Succeed! No non-terminal left in the tree,
input is totally matched**



$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow \text{int}$

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

$T \rightarrow (E)$

Input:

`int5 * int2`

Preliminaries

- **TOKEN**: the type of all tokens
 - Special tokens INT, OPEN, CLOSE, PLUS, TIMES
- The global **next** points to the next token in the input

Implementing Productions

- Define boolean functions that check the token string for match of

- A given token terminal

```
bool term(TOKEN tok) { return *next++ == tok; }
```

- A given production of S (the n-th)

```
bool Sn() {...}
```

- Any production of S

```
bool S() {...}
```

- These functions advance **next**

Implementing Productions

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

- For production $E \rightarrow T$
`bool E1() { return T(); }`
- For production $E \rightarrow T + E$
`bool E2() { return T() && term(PLUS) && E(); }`
- For all productions of E (with backtracking)
`bool E() {
 TOKEN *save = next;
 return (next= save, E1()) || (next= save,
 E2()); }`

Implementing Productions

- For non-terminal T

```
bool T1() { return terms(OPEN) && E() && term(CLOSE); }
```

```
bool T2() { return terms(INT) && term(TIMES) && T(); }
```

```
bool T3() { return terms(INT); }
```

```
bool T() {  
    TOKEN *save = next;  
    return (next= save, T1())  
        || (next= save, T2())  
        || (next= save, T3()); }
```

$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow (E)$

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

$T \rightarrow \text{int}$

Recursive Descent Parsing

- To start the parser
 - Initialize next to point to the first token
 - Invoke $E()$
- Note how this simulates our previous example
- Easy to implement
- But not always working!

Left-Recursion in Recursive Descent Parsing

- Consider a production $S \rightarrow S a$
 - `bool S1() { return S() && term(a); }`
 - `bool S() { return S1(); }`
- $S()$ will get into an infinite loop
- Left-recursive grammar has a nonterminal S
 - $S \rightarrow \uparrow + \dots S \dots$
- Recursive descent parsing does not work for left-recursive grammars

Elimination of Left Recursion

- Consider the left recursive grammar
 - $S \rightarrow S a \mid b$
- S generates all strings starting with 'b' and followed by a number of 'a'
- Can rewrite using right-recursion
 - $S \rightarrow b S'$
 - $S' \rightarrow a S' \mid \epsilon$

No Immediate Left Recursion

- In general for immediate left recursion
 - $S \rightarrow S \alpha_1 \mid \dots \mid S \alpha_n \mid \beta_1 \mid \dots \mid \beta_m$
- 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
 - $S \rightarrow \beta_1 S' \mid \dots \mid \beta_m S'$
 - $S' \rightarrow \alpha_1 S' \mid \dots \mid \alpha_n S' \mid \varepsilon$

No Immediate Left Recursion

$T \Rightarrow T * F \Rightarrow T * F * F \Rightarrow F * F * F$

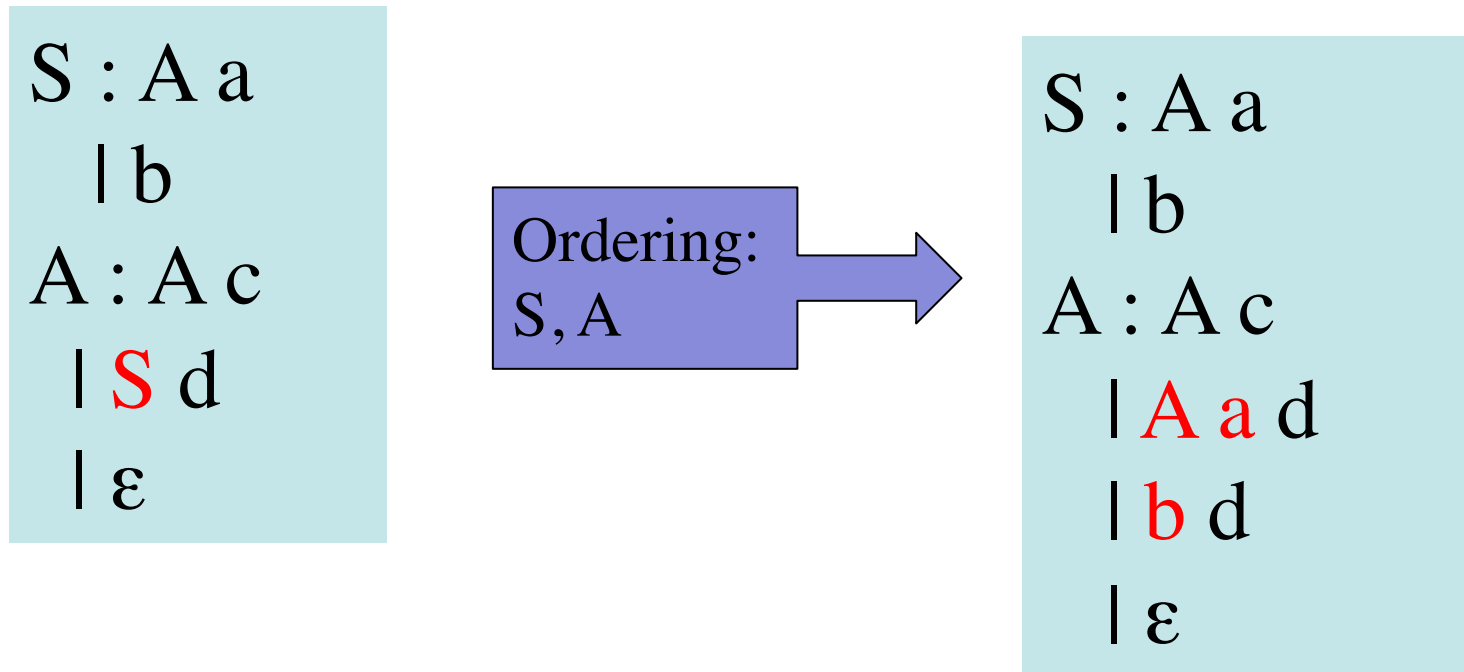
$T : T * F$
 $\quad | F$
 $F : a$
 $\quad | b$
 $\quad | c$

No left
recursion

$T : F T'$
 $T' : * F T'$
 $\quad | \epsilon$
 $F : a$
 $\quad | b$
 $\quad | c$

$T \Rightarrow F T' \Rightarrow F * F T' \Rightarrow F * F * F T' \Rightarrow F * F * F$

Remove General Left Recursion



Immediate Left Recursion

$S : A a$
| b
 $A : A c$
| $A a d$
| $b d$
| ϵ

Remove
Left
Recursion



$S : A a$
| b
 $A : b d A'$
| A'
 $A' : c A'$
| $a d A'$
| ϵ

General Left Recursion

Input: grammar G with no cycles $A \rightarrow A$ or empty rules $A \rightarrow \varepsilon$

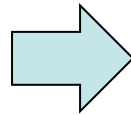
Output: grammar with no left recursion

Arrange nonterminals in order $A_1, A_2, A_3, \dots, A_n$

```
for i = 1 to n {  
  for j = 1 to i-1 {  
    replace each rule  $A_i \rightarrow A_j \alpha$  where  $A_j \rightarrow \beta_1 \mid \dots \mid \beta_m$  with  
    the rules  $A_i \rightarrow \beta_1 \alpha \mid \dots \mid \beta_m \alpha$   
  }  
  remove immediate left recursion among  $A_i$  rules  
}
```

Remove General Left Recursion

$S : A a$
 $| b$
 $A : A c$
 $| S d$
 $| B$
 $B : B e$
 $| A f$
 $| S g$
 $| h$



$S \rightarrow A a$
 $| b$
 $A \rightarrow b d A'$
 $| B A'$
 $A' \rightarrow a d A'$
 $| c A'$
 $| \epsilon$

$B \rightarrow b d A' a g B'$
 $| b d A' f B'$
 $| b g B'$
 $| h B'$
 $B' \rightarrow A' a g B'$
 $| A' f B'$
 $| e B'$
 $| \epsilon$

Summary of Recursive Descent

- Simple and general parsing strategy
 - Left-recursion must be eliminated first
 - Most of the time manually (but it can be done automatically)
 - Backtracking is inefficient
 - In practice, backtracking is eliminated by restricting the grammar
 - Used in production compilers (e.g. gcc front-end)

How to compute: Does $X \Rightarrow^* \epsilon$?

- The question 'Does $X \Rightarrow^* \epsilon$?' can be written as the predicate: nullable(X)

Nullable = {} (set containing nullable non-terminals)

Changed = True

While (changed):

 changed = False

 if X is not in Nullable:

 if

 1. $X \rightarrow \epsilon$ is in the grammar, or

 2. $X \rightarrow Y_1 \dots Y_n$ is in the grammar and Y_i is in Nullable for all i

 then

 add X to Nullable; changed = True