Aalto University

Introduction to Parsing

Lecture 3

Announcement

- WA1
  - Due today (10 June) at 11:59pm

The Functionality of the Parser

- **Input:** sequence of tokens from lexer
- **Output:** parse tree of the program
  (But some parsers never produce a parse tree . . .)

Example

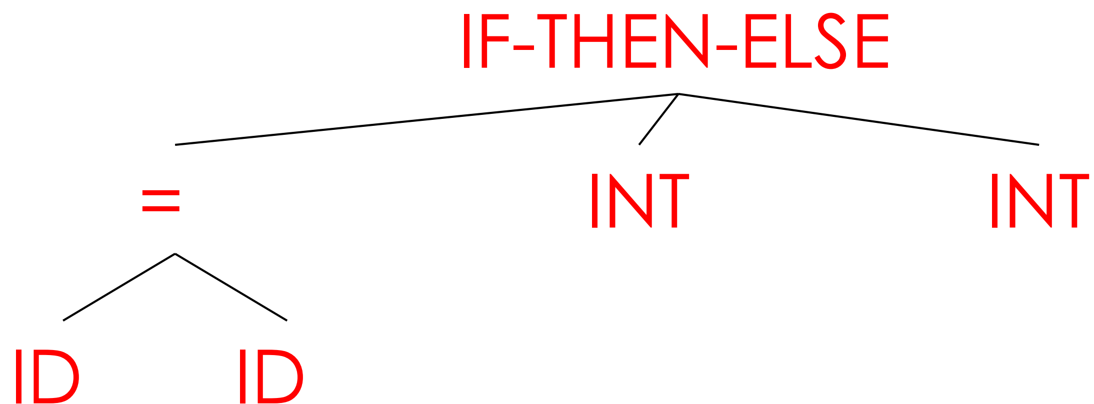
- Cool

if x = y then 1 else 2 fi

- Parser input

IF ID = ID THEN INT ELSE INT FI

- Parser output



Comparison with Lexical Analysis

Phase	*Input*	*Output*
Lexer	String of characters	String of tokens
Parser	String of tokens	Parse tree (may be implicit)

The Role of the Parser

- Not all strings of tokens are programs . . .
- . . . parser must distinguish between valid and invalid strings of tokens

- We need
  - A language for describing valid strings of tokens
  - A method for distinguishing valid from invalid strings of tokens

Context-Free Grammars

- Programming language constructs have recursive structure

- An STMT is
  if EXPR then STMT else STMT
  while EXPR do STMT end
  …
- Context-free grammars are a natural notation for this recursive structure

CFGs (Cont.)

- A CFG consists of
  - A set of *terminals* T
  - A set of *non-terminals* N
  - A *start symbol* S (a non-terminal)
  - A set of *productions*

$$X \rightarrow Y_1 Y_2 \ldots Y_n$$

$$\text{where } X \in N \text{ and } Y_i \in T \cup N \cup \{\varepsilon\}$$

Notational Conventions

- In these lecture notes
  - Non-terminals are written upper-case
  - Terminals are written lower-case
  - The start symbol is the left-hand side of the first production

Example of CFGs

Simple arithmetic expressions:

$$
\begin{aligned}
E \;\rightarrow\;& E * E \\
\mid\;& E + E \\
\mid\;& (E) \\
\mid\;& \mathrm{id}
\end{aligned}
$$

The Language of a CFG

Read productions as replacement rules:

$$X \rightarrow Y_1 \ldots Y_n$$

Means X can be replaced by $Y_1 \ldots Y_n$

Key Idea

1. Begin with a string consisting of the start symbol "S"
2. Replace any non-terminal X in the string by a the right-hand side of some production

$$X \to Y_1 \ldots Y_n$$

3. Repeat (2) until there are no non-terminals in the string

The Language of a CFG (Cont.)

More formally, write

$$X_1 \ldots X_{i-1} X_i X_{i+1} \ldots X_n \rightarrow X_1 \ldots X_{i-1} Y_1 \ldots Y_m X_{i+1} \ldots X_n$$

if there is a production

$$X_i \rightarrow Y_1 \ldots Y_m$$

The Language of a CFG (Cont.)

Write

$$X_1 \ldots X_n \rightarrow^* y_1 \ldots y_m$$

if

$$X_1 \ldots X_n \rightarrow \ldots \rightarrow \ldots \rightarrow y_1 \ldots y_m$$

in 0 or more steps

The Language of a CFG

Let G be a context-free grammar with start symbol S. Then the language $L(G)$ of G is:

$$\{a_1 \ldots a_n \mid S \rightarrow^* a_1 \ldots a_n \text{ and every } a_i \text{ is a terminal}\}$$

The sentential forms $SF(G)$ of G is:

$$\{X_1 \ldots X_n \mid S \rightarrow^* X_1 \ldots X_n \text{ and every } X_i \text{ is a terminal or non-terminal}\}$$

Therefore: $L(G) \subset SF(G)$

Terminals

- Terminals are so-called because there are no rules for replacing them

- Once generated, terminals are permanent

- Terminals ought to be tokens of the language

Examples

$L(G)$ is the language of CFG G

Strings of balanced parentheses $\{ (^i)^i \mid i \geq 0 \}$

Two grammars:

$$S \rightarrow (S)$$
$$S \rightarrow \varepsilon$$

OR

$$S \rightarrow (S)$$
$$\quad \mid \quad \varepsilon$$

Arithmetic Example

Simple arithmetic expressions:

$$E \rightarrow E+E \mid E * E \mid (E) \mid id$$

Some elements of the language:

id	id + id
(id)	id * id
(id) * id	id * (id)

Notes

The idea of a CFG is a big step. But:

- Membership in a language is "yes" or "no"; also need parse tree of the input
- Must handle errors gracefully
- Need an implementation of CFG's (e.g., bison)

Derivations and Parse Trees

A *derivation* is a sequence of productions

$$S \rightarrow \ldots \rightarrow \ldots \rightarrow \ldots \rightarrow \ldots \rightarrow \ldots$$

A derivation can be drawn as a tree

- Start symbol is the tree's root
- For a production $X \rightarrow Y_1 \ldots Y_n$ add children $Y_1 \ldots Y_n$ to node X

Derivation Example

- Grammar

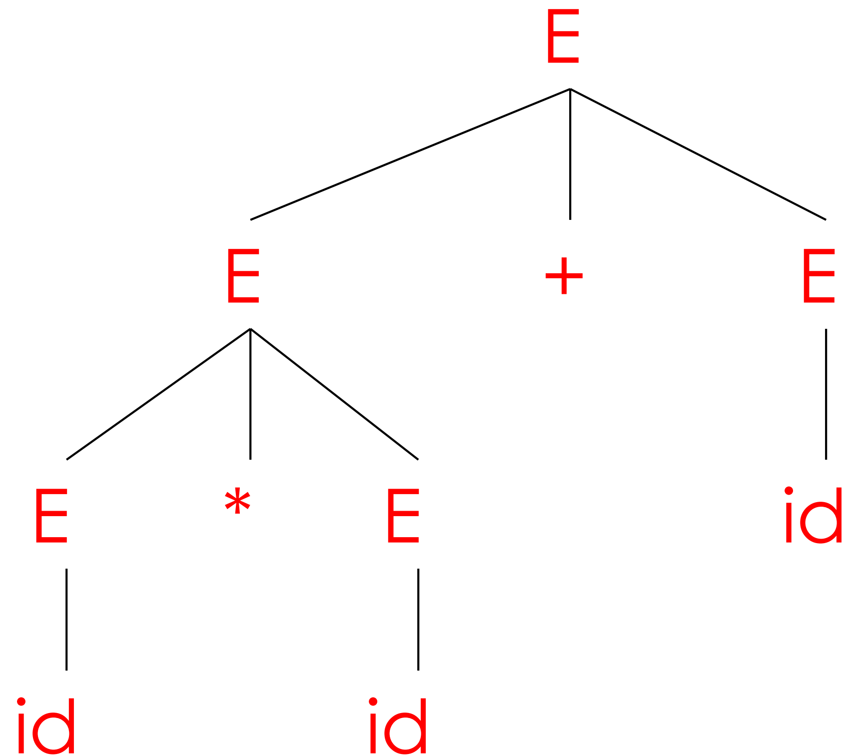
$$E \rightarrow E+E \mid E * E \mid (E) \mid id$$

- String

$$id * id + id$$

Derivation Example (Cont.)

$$
\begin{aligned}
& E \\
\to\ & E+E \\
\to\ & E*E+E \\
\to\ & id*E+E \\
\to\ & id*id+E \\
\to\ & id*id+id
\end{aligned}
$$

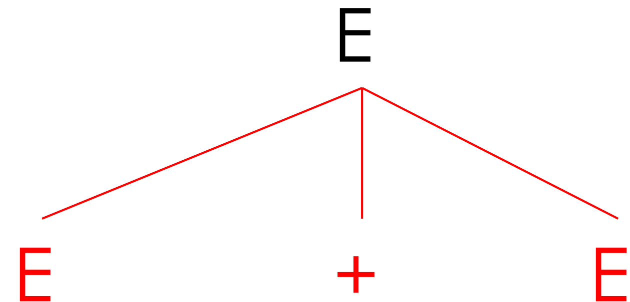


Derivation in Detail (1)

E

E

Derivation in Detail (2)

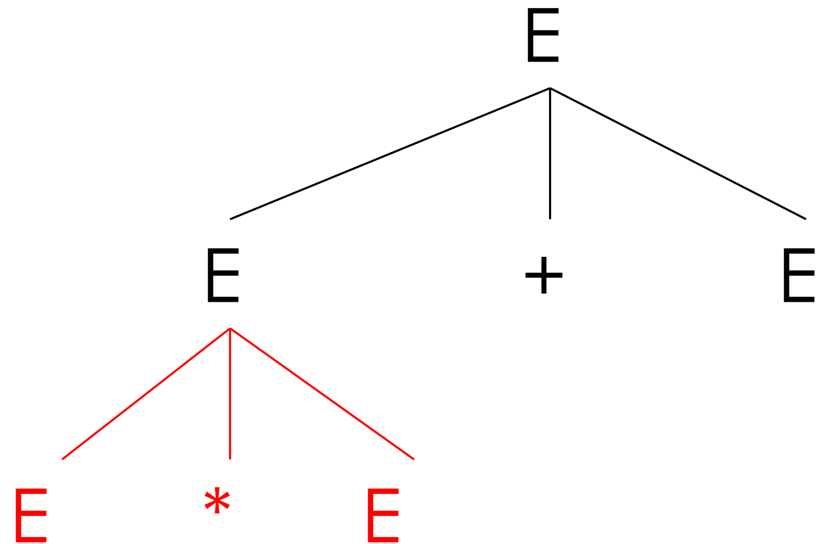


E

$\rightarrow$ E+E

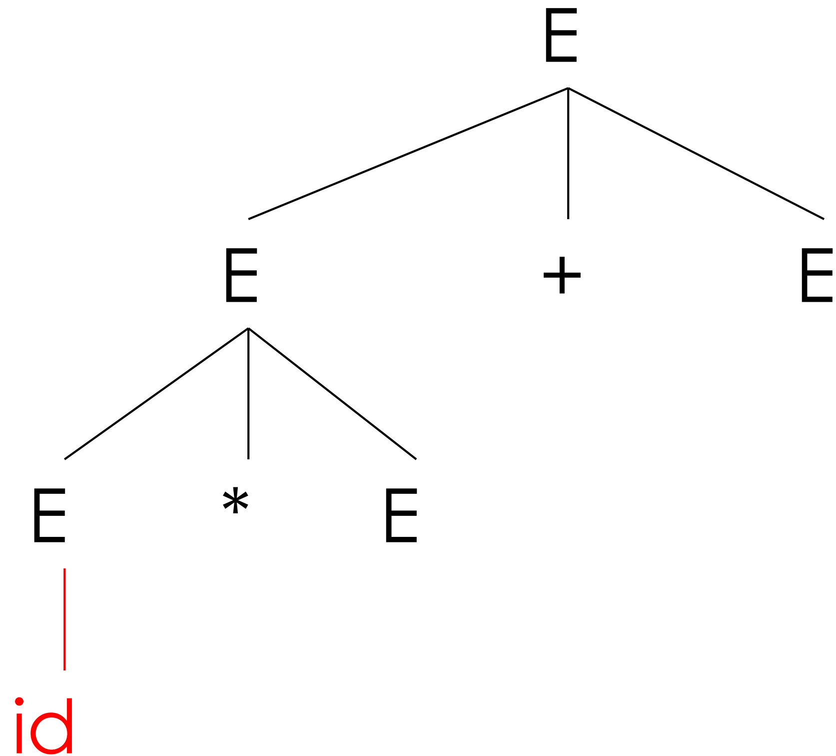
Derivation in Detail (3)

$$
\begin{aligned}
& E \\
\rightarrow\quad & E+E \\
\rightarrow\quad & E * E+E
\end{aligned}
$$



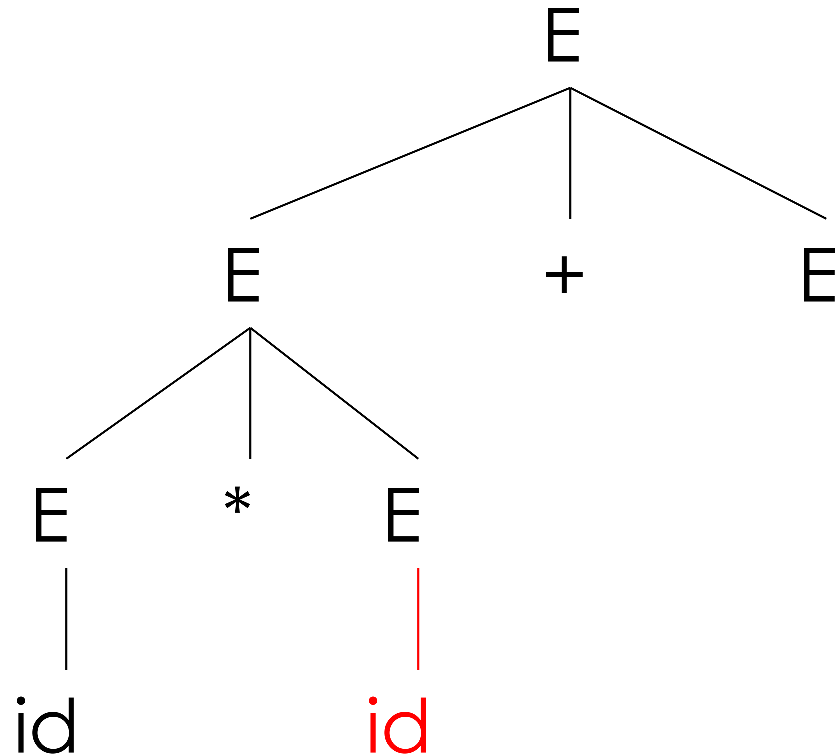
Derivation in Detail (4)

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E * E+E \\
\rightarrow\ & id * E + E
\end{aligned}
$$



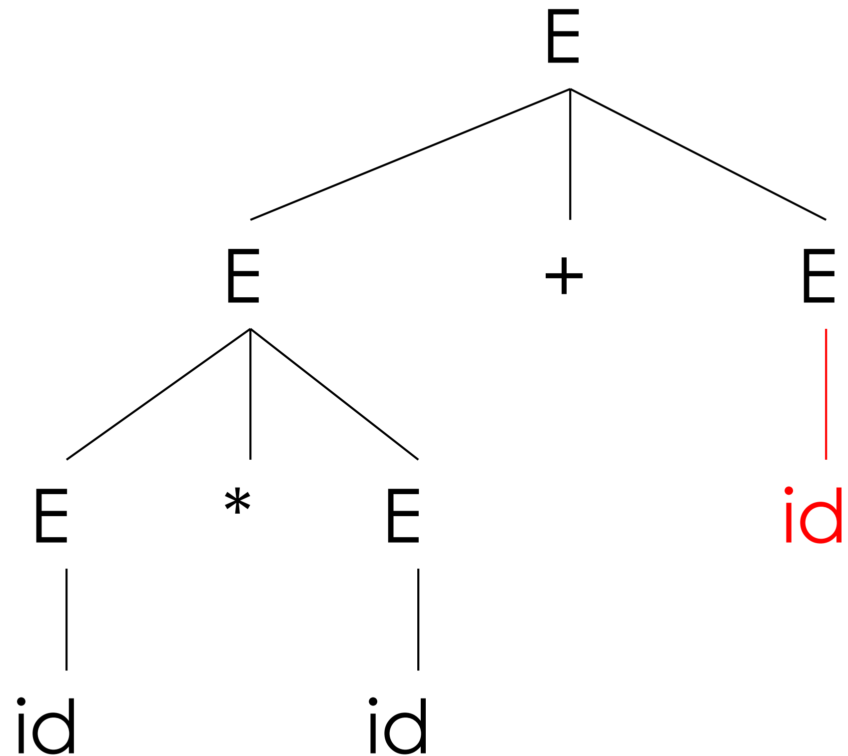
Derivation in Detail (5)

$$E$$
$$\rightarrow E+E$$
$$\rightarrow E * E+E$$
$$\rightarrow id * E + E$$
$$\rightarrow id * id + E$$



Derivation in Detail (6)

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E * E+E \\
\rightarrow\ & id * E + E \\
\rightarrow\ & id * id + E \\
\rightarrow\ & id * id + id
\end{aligned}
$$



Notes on Derivations

- A parse tree has
  - Terminals at the leaves
  - Non-terminals at the interior nodes

- An in-order traversal of the leaves is the original input

- The parse tree shows the association of operations, the input string does not

Left-most and Right-most Derivations

- The example was a *left-most* derivation
  - At each step, replaced the left-most non-terminal
- There is an equivalent notion of a *right-most* derivation

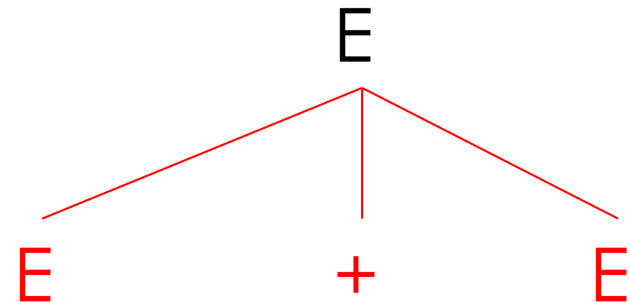
$$
\begin{array}{ll}
 & E \\
\rightarrow & E+E \\
\rightarrow & E+id \\
\rightarrow & E*E+id \\
\rightarrow & E*id+id \\
\rightarrow & id*id+id
\end{array}
$$

Right-most Derivation in Detail (1)

E

E

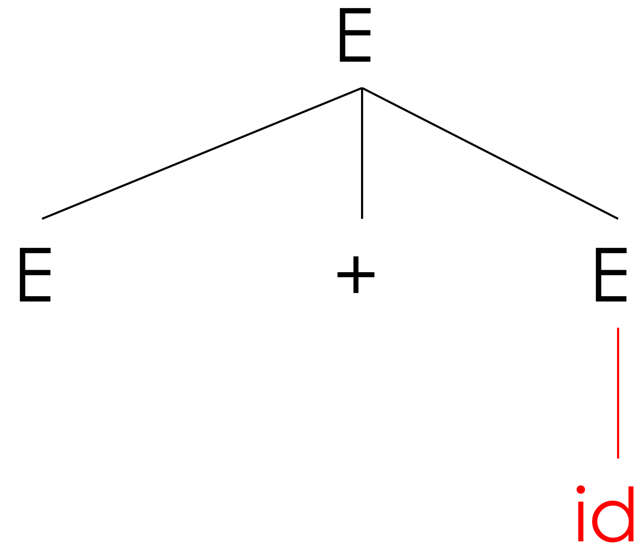
Right-most Derivation in Detail (2)



E

$\rightarrow$ E+E

Right-most Derivation in Detail (3)



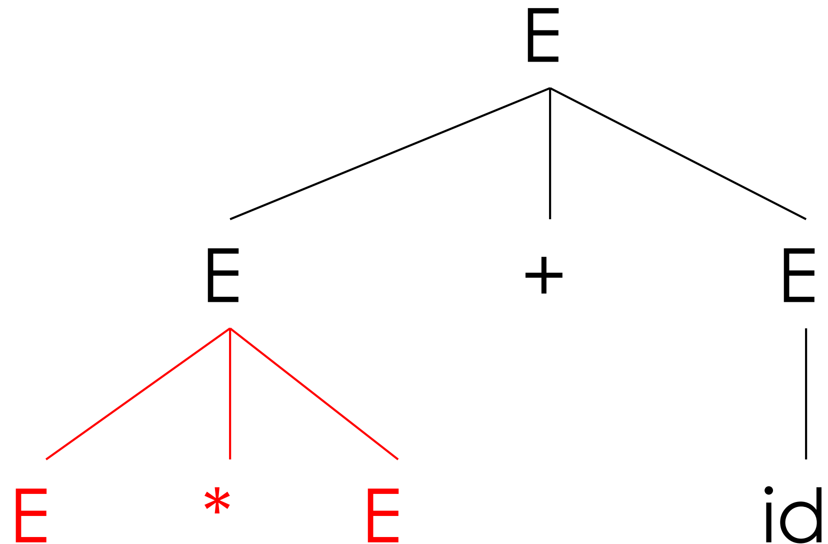
E

$\rightarrow E+E$

$\rightarrow E+id$

Right-most Derivation in Detail (4)

$$E$$
$$\rightarrow E+E$$
$$\rightarrow E+id$$
$$\rightarrow E * E + id$$



Right-most Derivation in Detail (5)

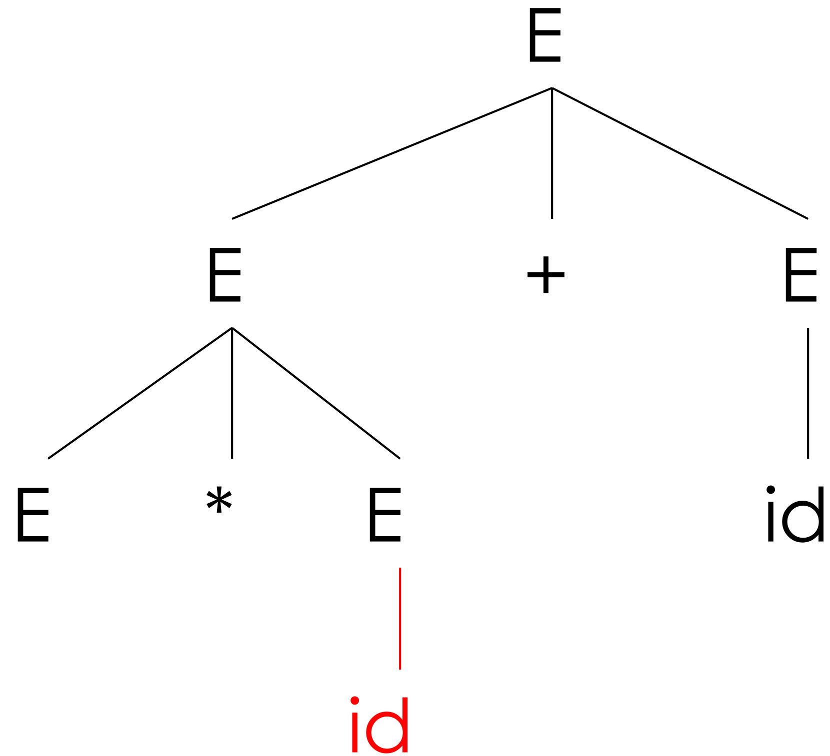
$$E$$

$$\rightarrow E+E$$

$$\rightarrow E+id$$

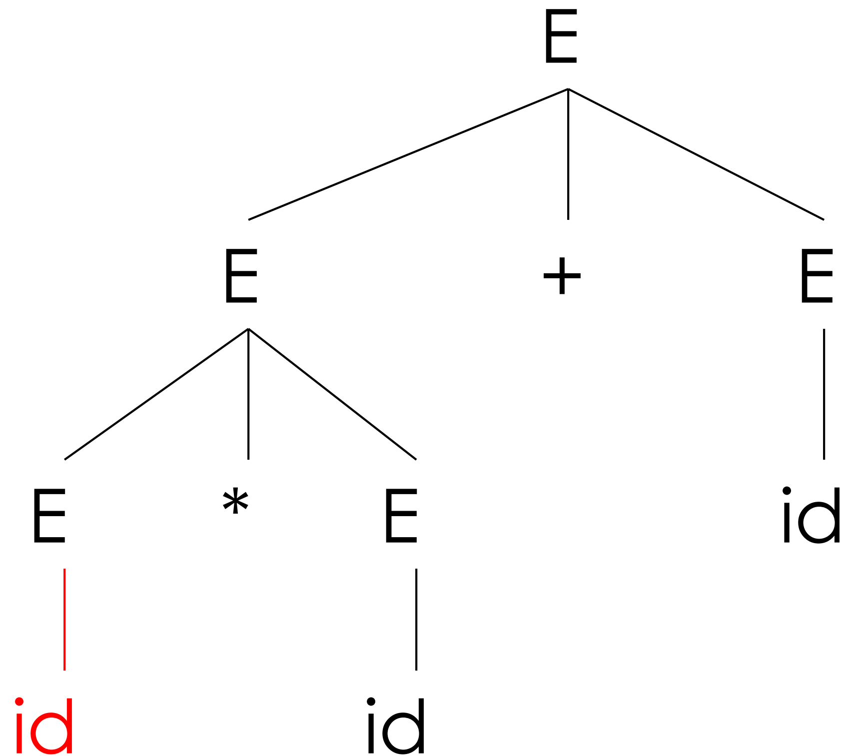
$$\rightarrow E * E + id$$

$$\rightarrow E * id + id$$



Right-most Derivation in Detail (6)

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E+id \\
\rightarrow\ & E * E + id \\
\rightarrow\ & E * id + id \\
\rightarrow\ & id * id + id
\end{aligned}
$$



Derivations and Parse Trees

- Note that right-most and left-most derivations have the same parse tree

- The difference is the order in which branches are added

Summary of Derivations

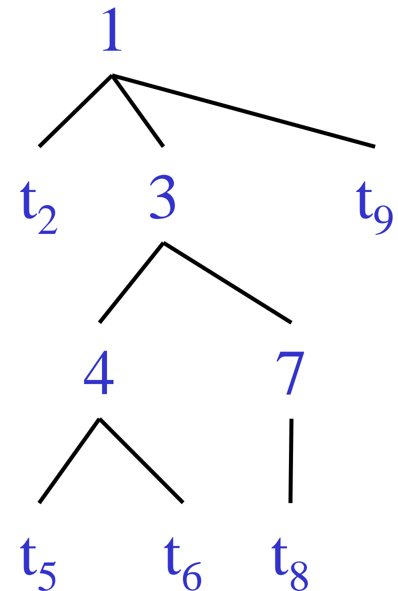
- We are not just interested in whether $s \in L(G)$
  - We need a parse tree for s
- A derivation defines a parse tree
  - But one parse tree may have many derivations
- Left-most and right-most derivations are important in parser implementation

Introduction to Top-Down Parsing

Recursive Descent

Intro to Top-Down Parsing: The Idea

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

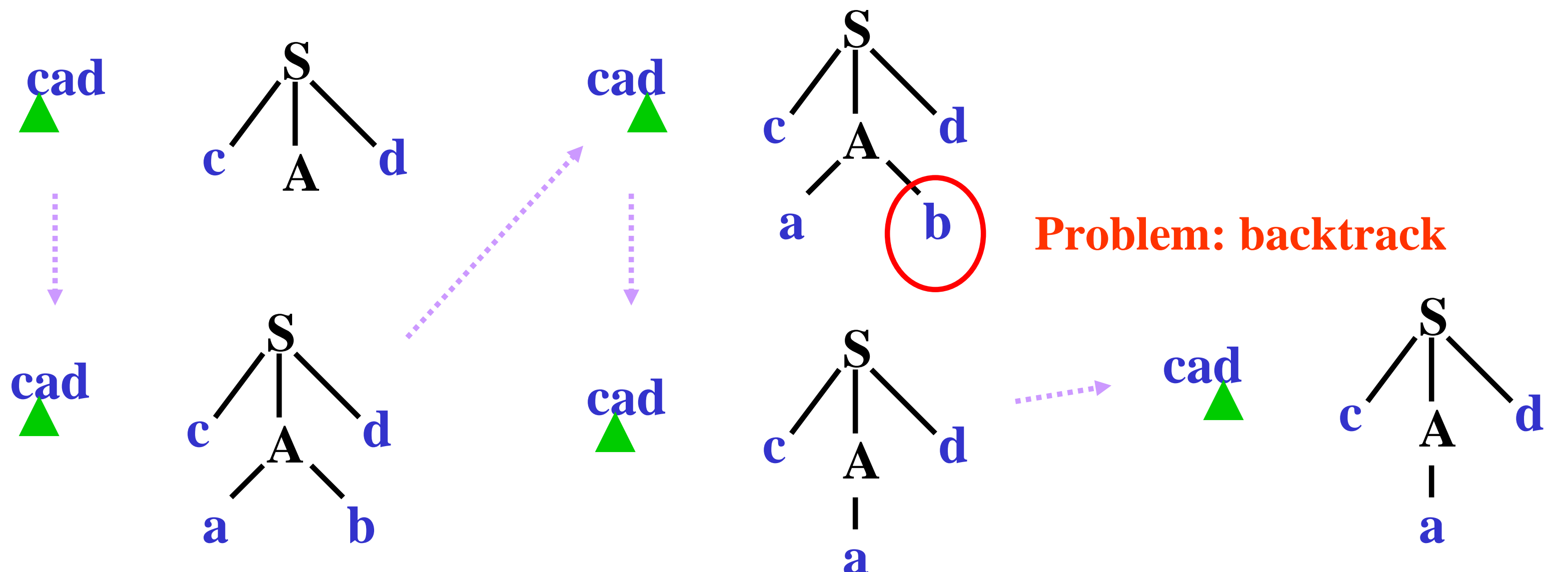


- Terminals are seen in order of appearance in the token stream:

t_2 t_5 t_6 t_8 t_9

Non-Predictive Top-Down Parsing

- Choose production rule based on input symbol
- May require backtracking to correct a wrong choice.
- Example: $S \to c\ A\ d$, $A \to ab \mid a$ — input: **cad**



Predictive Top-Down Parsing

Parser Never Backtracks

For Example, Consider:

Type → Simple   **Start symbol**

| ↑ **id**

| **array** [Simple] **of** Type

Simple → **integer**

| **char**

| **num dotdot num**

Suppose input is :

array [**num dotdot num**] **of integer**

Parsing would begin with

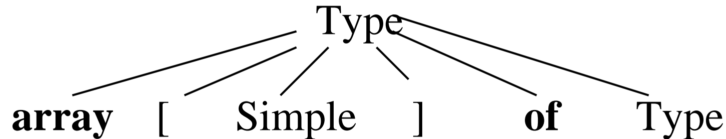
Type → **???**

Predictive Parsing Example

Lookahead symbol

Input : **array** [**num dotdot num**] **of integer**

Type
?

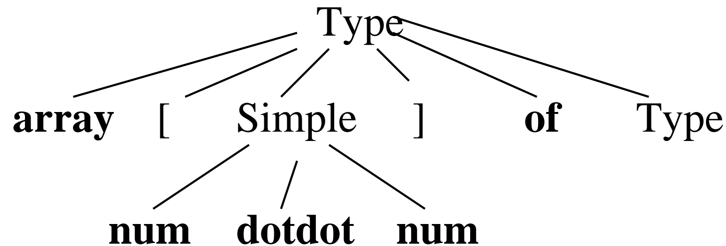


Start symbol

Type → simple
| ↑ **id**
| **array** [Simple] **of** Type
Simple → **integer**
| **char**
| **num dotdot num**

Lookahead symbol

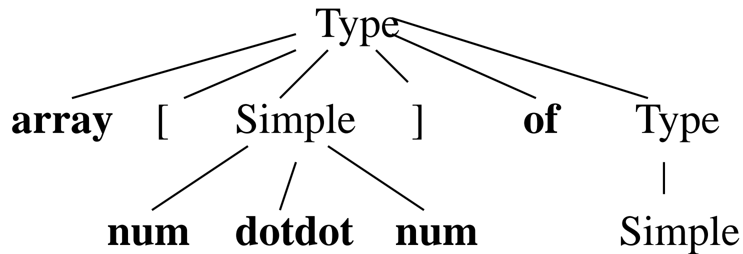
Input : **array** [**num dotdot num**] **of integer**



Predictive Parsing Example

Lookahead symbol

Input : **array** [**num dotdot num**] **of integer**

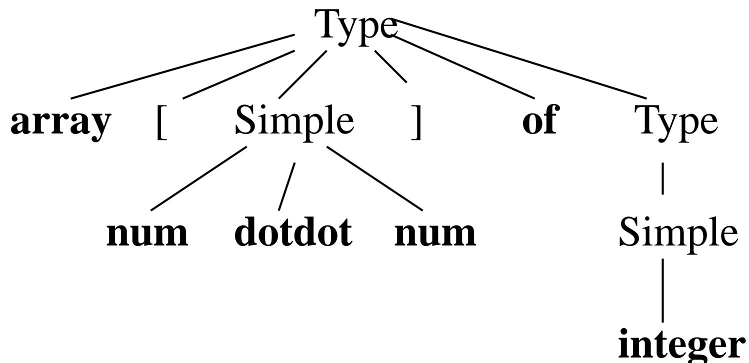


Start symbol

Type → Simple
| ↑ **id**
| **array** [Simple] **of** Type
Simple → **integer**
| **char**
| **num dotdot num**

Lookahead symbol

Input : **array** [**num dotdot num**] **of integer**



Predictive Recursive Descent

- Parser is implemented by N + 1 subroutines, where N is the number grammar non-terminals

- There is one subroutine for attempting to Match tokens in the input stream

- There is also one subroutine for each non-terminal with two tasks:
    1. Deciding on the next production to use
    2. Applying the selected production

Predictive Recursive Descent (Cont.)

Procedure "Match" checks if the token matches the expected input

```
procedure Match ( expected_token ) ;
  {
      if  lookahead = expected_token then
              lookahead  :=  get_next_token
      else error
   }
```

Predictive Recursive Descent (Cont.)

- The subroutine for each non-terminals has two tasks:
  1. Selecting the appropriate production
  2. Applying the chosen production

- Selection is done based on the result of a number of if-then-else statements
- Applying a production is implemented by calling the match procedure or other subroutines, based on the rhs of the production

Predictive Recursive Descent (Cont.)

Subroutine "Simple" for the given example:

Type → Simple
 | ↑ **id**
 | **array** [Simple] **of** Type
Simple → **integer**
 | **char**
 | **num dotdot num**

```
procedure Simple ;
   { if lookahead = integer then call Match ( integer );
        else if  lookahead = char then call Match ( char );
             else if lookahead = num
                   then { call Match ( num ); call Match ( dotdot );
                          call Match ( num ) }
                   else error
   }
```

Predictive Recursive Descent (Cont.)

Subroutine "Type" for the given example:

Type → Simple
  | ↑ **id**
  | **array** [Simple] **of** Type
Simple → **integer**
  | **char**
  | **num dotdot num**

```
procedure Type ;
   {if lookahead  is in { integer, char, num } then call Simple;
       else if lookahead  = '↑'  then  { call Match ( '↑' ) ;  call Match( id ) }
          else if lookahead  = array
                    then { call Match( array ); call Match( '[' ); call Simple;
                             call Match( ']' ); call Match( of ); call Type }
               else  error
   }
```

How to write tests for selecting the appropriate production rule?

Basic Tools:

First: Let α be a string of grammar symbols. First(α) is the set that includes every terminal that appears leftmost in α or in any string originating from α.
NOTE: If $\alpha \Rightarrow \in$, then $\in$ is First(α).

Follow: Let A be a non-terminal. Follow(A) is the set of terminals a that can appear directly to the right of A in some sentential form. ($S \Rightarrow \alpha A a \beta$, for some α and β).
NOTE: If $S \Rightarrow \alpha A$, then \$ is Follow(A).

Computing First Sets

Definition

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

Algorithm sketch:

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

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

First(( ) = { (}

First( )) = {) }

First(int) = { int }

First(+) = { + }

First(*) = { * }

First(T) = {int, (}

First(E) = {int, (}

First(X) = {+, ε }

First(Y) = {*, ε }

Computing Follow Sets

- Definition:

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

- 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 → T X   X → + E | ε

  T → (E) | int Y   Y → * T | ε

- Follow sets

  Follow(E) = {), $}

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

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

  Follow(X) = {), $}

General Form of R.D. Procedures

$A \rightarrow \alpha \mid \beta \mid \delta$

Case 1: when $\varepsilon \notin$ First (A),

procedure A;
   {if *lookahead* is in First (α) then use $A \rightarrow \alpha$
      else if *lookahead* is in First (β) then use $A \rightarrow \beta$
         else if *lookahead* is in First (δ) then $A \rightarrow \delta$
            else error /* if *lookahead* $\notin$ First (A) */
   }

General Form of R.D. Procedures (Cont.)

$A \rightarrow \alpha \mid \beta \mid \delta$

Case 2: when $\varepsilon \in$ First (A),

Let's assume $\varepsilon \in$ First (α),

($\alpha = \varepsilon$ or $\alpha = B_1B_2 \ldots B_n$, where B_i is a non-terminal and $\varepsilon \in$ First (B_i))

```
procedure A;
  {if lookahead is in First ( α ) then use A → α
   else if lookahead is in First ( β ) then use A → β
     else if lookahead is in First ( δ ) then use A → δ
       else if lookahead is in Follow ( A ) then exit
         else error /* lookahead ∉ First ( A ) ∪ Follow ( A ) */
  }
```

This line is needed only if $\alpha = B_1B_2 \ldots B_n$ (points to the line "if lookahead is in First (α) then use A → α")

If $\alpha = B_1B_2 \ldots B_n$ use $A \rightarrow \alpha$ (instead of exit) (points to the line "then exit")

Conditions for R.D. Parsing to be Predictive

R.D. parsing is Predictive when for all $A \to \alpha \mid \beta$

1. First(α) $\cap$ First(β) = $\emptyset$;
   besides, only one of α or β can derive ε

2. if α derives ε, then Follow(A) $\cap$ First(β) = $\emptyset$

Error Handling

- Purpose of the compiler is
  - To detect non-valid programs
  - To translate the valid ones
- Many kinds of possible errors (e.g. in C)

Error kind	Example	Detected by …
Lexical	… $ …	Lexer
Syntax	… x *% …	Parser
Semantic	… int x; y = x(3); …	Type checker
Correctness	your favorite program	Tester/User

Syntax Error Handling

- Error handler should
  - Report errors accurately and clearly
  - Recover from an error quickly
  - Not slow down compilation of valid code

- Good error handling is not easy to achieve

Approaches to Syntax Error Recovery

- From simple to complex
  - Panic mode
  - Error productions
  - Automatic local or global correction

Error Recovery: Panic Mode

- Simplest, most popular method
- When an error is detected:
  - Discard tokens until one with a clear role is found
  - Continue from there
- Such tokens are called <u>synchronizing</u> tokens
  - Typically the statement or expression terminators

Syntax Error Recovery: Error Productions

- Idea: specify in the grammar known common mistakes

- Essentially promotes common errors to alternative syntax

- Example:
  - Write 5 x instead of 5 * x
  - Add the production $E \rightarrow \ldots \mid E\ E$

- Disadvantage
  - Complicates the grammar

Error Recovery: Local and Global Correction

- Idea: find a correct "nearby" program
  - Try token insertions and deletions
  - Exhaustive search

- Disadvantages:
  - Hard to implement
  - Slows down parsing of correct programs
  - "Nearby" is not necessarily "the intended" program

Syntax Error Recovery: Past and Present

- Past
  - Slow recompilation cycle (even once a day)
  - Find as many errors in one cycle as possible

- Present
  - Quick recompilation cycle
  - Users tend to correct one error/cycle
  - Complex error recovery is less compelling
  - Panic-mode seems enough

Implementing Panic-Mode Recovery

- The choice for the synchronizing set is important for improving the performance of the panic mode method.
- We define First(A) $\cup$ Follow(A) as the synchronizing set of non-terminal A.

Implementing Panic-Mode Recovery (Cont.)

Suppose the parser is in subroutine *A*, the current token is a, and a syntax error is detected:

1. if a $\notin$ Follow (*A*),
   Report the error by 'illegal a found on line N',
   where N is the line number of token a,
   then get the next token from the scanner, and then call *A*.

2. if a $\in$ Follow (*A*),
   Report the error by: 'missing b on line N',
   where b is a simple token that can be derived from *A*,
   and N is the line number of token a,
   then resume parsing by exiting from *A*.

Implementing Panic-Mode Recovery (Cont.)

- Suppose the parser is in subroutine Match, the current token is a, and the expected token is b:
- Report the error by the message:
  - 'missing b on line N, where N is the line number of token a, and
  - then resume parsing by exiting from Match.

Example of PM Error Recovery

Follow(Type) = { $ }

```
procedure Type ;
  { if lookahead  is in { integer, char, num } then call Simple;
      else if lookahead  = '↑'  then  { call Match ( '↑' ) ; call Match( id ) }
         else if lookahead  = array then {
                  call Match( array ); call Match( '[' ); call Simple;
                  call Match( ']' ); call Match( of ); call Type }
              else  if lookahead is in { $ }
                       then print ( 'missing integer on line N' )
                       else {    print ( 'illegal lookahead on line N' );
                                 lookahed := get_next_token;
                                 call Type }
  }
```

Example of PM Error Recovery (Cont.)

Follow(Simple) = { $,] }

```
procedure Simple ;
   { if lookahead = integer then call Match ( integer );
       else if  lookahead = char then call Match ( char );
            else if lookahead = num
                  then {   call Match ( num  );
                           call Match ( dotdot  );
                           call Match ( num  )
                            }
                  else  if lookahead is in { $, ] }
                           then print ( 'missing integer on line N' )
                           else {    print ( 'illegal lookahead on line N' );
                                     lookahed := get_next_token;
                                     call Simple }
   }
```

Example of PM Error Recovery (Cont.)

```
procedure Match ( expected_token ) ;
  {
    if  lookahead = expected_token then
        lookahead  :=  get_next_token
      else print ( 'missing expected_token on line N' )
   }
```

Question?

- Consider the grammar

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

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

- Write Recursive Descent Procedures including panic mode error recovery for all non-terminals.
- Write a step-by-step parsing of input 'int * int'
- Draw the parse tree of the input

Question?

How many strings does the following grammar generate?

- 7
- 15
- 2
- 8
- 16
- 4

$$A \to B\ B$$
$$B \to C\ C$$
$$C \to 1 \mid 2$$

Question?

How many strings does the following grammar generate?

$$A \rightarrow B\ B$$
$$B \rightarrow C\ C$$
$$C \rightarrow 1 \mid 2 \mid \varepsilon$$

- 16
- 31
- 15
- 12
- 64
- 63
- 32
- 11

Question?

Which of the following is a valid derivation of the given grammar?

$S \rightarrow aXa$

$X \rightarrow \varepsilon \mid bY$

$Y \rightarrow \varepsilon \mid cXc \mid d$

- S
  aXa
  abYa
  acXca
  acca

- S
  aa

- S
  aXa
  abYa
  abcXca
  abcbYca
  abcbdca

- S
  aXa
  abYa
  abcXcda
  abccda

Question?

Which of the following is a valid parse tree for the given grammar?

$S \rightarrow aXa$

$X \rightarrow \varepsilon \mid bY$

$Y \rightarrow \varepsilon \mid cXc \mid d$