Next....

- Implementation of parsers
- Two approaches
  - Top-down
  - Bottom-up
- First: Top-Down
  - Easier to understand and program manually
- Then: Bottom-Up
  - More powerful and used by most parser generators

Top down Parsing

- Construction of parse tree for the input, starting from root and creating the nodes of the parse tree in *preorder.*

- *Equivalently,* top-down parsing can be viewed as finding a leftmost derivation for an input string.

Example: Top down Parsing

- Following grammar generates types of Pascals

```
type → simple
       | ↑ id
       | array [ simple] of type

simple → integer
         | char
         | num dotdot num
```

Examples generated by this grammar (valid Pascal types):

- **integer**
- **char**
- **1..10**
- **array[1..10] of integer**
- **array[char] of integer**
- **array[1..5] of array[1..10] of char**

Example … Parse Tree Construction Steps

1. Start with the root node

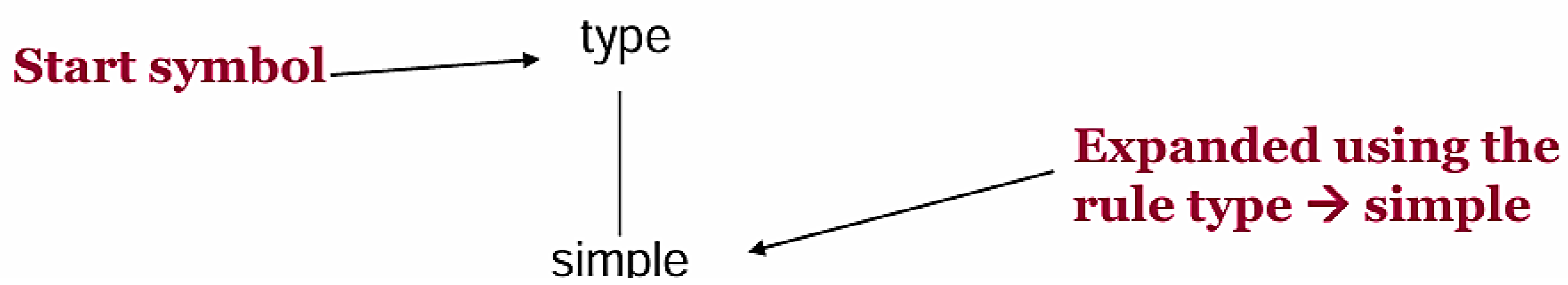
- The root is labeled with the **start symbol** of the grammar (something like <S> or <program>).

2. Repeat two steps until the tree is complete

- **Step 1: Expansion (Which production?)** Decides **how to expand** a non-terminal.
  - If the current node is a **non-terminal** (*like <expr> or <stmt>),* we must choose one of its grammar rules (productions).
  - Example: If <expr> → <expr> + <term> | <term>, we must decide **which production rule** applies.
  - After choosing, expand it by creating **child nodes** for each symbol in the production.
- **Step 2: Next node (Which node?)** **Decides where to expand next in the tree.**
  - After expanding, we look for the **next non-terminal** node in the tree that still needs to be expanded.
  - *This continues until all non-terminals are replaced with **terminal symbols*** (tokens from the input program).

Contd.,

- Parse: array [num dotdot num] of integer



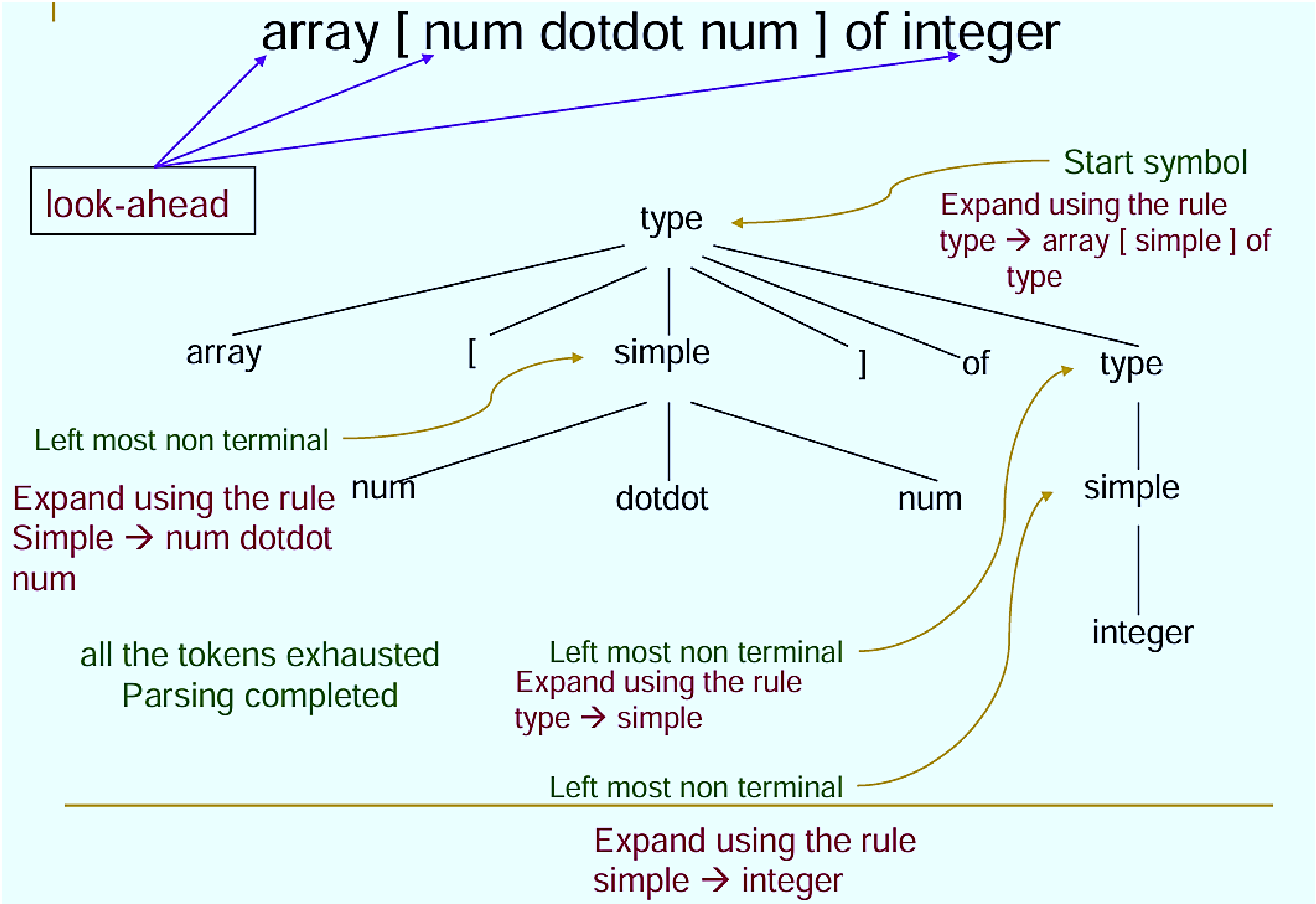
Problem with Expansion

- The non-terminal **simple** can **never generate a string starting with "array"**.
- But our input starts with the token array.
- This means parsing fails at this stage, and the parser would need **backtracking** to try other rules.

Back-tracking is not desirable therefore, take help of a “look ahead” token here (array).

The current token is treated as look-ahead token.

Diagram represents **leftmost derivation and** parsing of an array type declaration

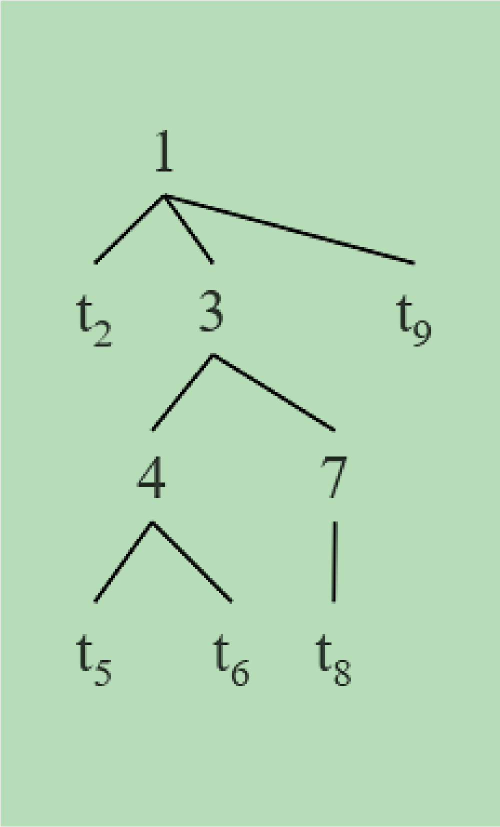


Introduction to Top-Down Parsing

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

  t2 t5 t6 t8 t9

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



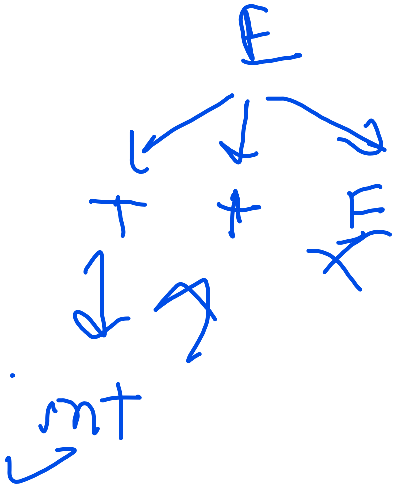
Recursive Descent Parsing

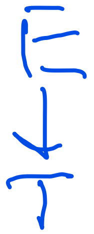
- Consider the grammar

  $E \rightarrow T + E \mid T$

  $T \rightarrow int \mid int * T \mid (E)$

The parser starts at the start symbol and calls functions that expand non-terminals to match input tokens.





- Token stream is: int5 * int2
- Start with top-level non-terminal E
- Try the rules for E in order

Recursive Descent Parsing. Example (Cont.)

Token stream: int5 * int2

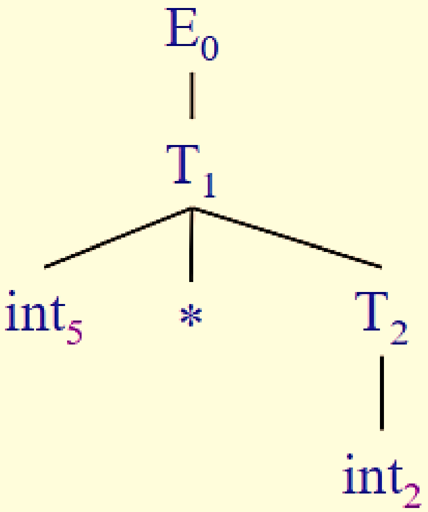
- Try $E_0 \rightarrow T_1 + E_2$
- Then try a rule for $T_1 \rightarrow (E_3)$
  - But (does not match input token int_5 ✗ mismatch.
- Try $T_1 \rightarrow int$. Token matches.
  - But + after T_1 does not match input token * ✗ mismatch.
- Try $T_1 \rightarrow int * T_2$
  - This will match but + after T_1 will be unmatched ✗ mismatch.
- Has exhausted the choices for T_1
  - Backtrack to choice for E_0

$$E \rightarrow T + E \mid T$$
$$T \rightarrow (E) \mid int \mid int * T$$

Recursive Descent Parsing. Example (Cont.)

Token stream: int5 * int2

- Try $E_0 \to T_1$
- Follow same steps as before for T_1
  - And succeed with $T_1 \to int_5 * T_2$ and $T_2 \to int_2$
  - With the following parse tree



$E \to T + E \mid T$
$T \to (E) \mid int \mid int * T$

Recursive Descent Parsing → Notes

- Easy to implement by hand
- Somewhat inefficient (due to backtracking)
- But does not always work …

This is **left-recursive** because the non-terminal **S** appears at the **leftmost position** on the right-hand side of its own production.

$$S \Rightarrow Sa \Rightarrow Saa \Rightarrow Saaa \Rightarrow \cdots$$

When Recursive Descent Does Not Work

- Consider a production $S \rightarrow S\ a$

```
bool S1() { return S() && term(a); }
bool 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 α
- Recursive descent does not work in such cases

Why Recursive Descent Fails

Predictive Parser

- Recursive descent works when the grammar is **predictive** (LL(1)):
  - No left recursion
  - No ambiguity
  - Parser can decide the production by looking at the next input token (1 lookahead)

Fixing the Grammar (Remove Left Recursion)

We transform the left-recursive rule:

$$S \rightarrow Sa \mid \varepsilon$$

Eliminate left recursion by introducing a new non-terminal:

$$S \rightarrow \varepsilon\, S'$$

$$S' \rightarrow a\, S' \mid \varepsilon$$

Now the grammar is **right-recursive**, suitable for recursive descent parsing.

Elimination of Left Recursion

$S \to S\alpha \mid \beta$

$S \to \beta S'$

$S' \to \alpha S' \mid \varepsilon$

- Consider the left-recursive grammar

  $S \to S\ \alpha \mid \beta$

- S generates all strings starting with a β and followed by any number of α's

- The grammar can be rewritten using right-recursion

  $S \to \beta\ S'$

  $S' \to \alpha\ S' \mid \varepsilon$

Contd.,

Ex1: $P \rightarrow P + Q \mid Q$

$A \rightarrow A \;\; \alpha \mid \beta$

$A \rightarrow \beta A'$
$A' \rightarrow \alpha A' \mid \varepsilon$

$P \rightarrow QP'$
$P' \rightarrow +QP' \mid \varepsilon$

Ex2: $S \rightarrow S0S1S \mid 01$

$A \rightarrow A \;\; \alpha \mid \beta$

$A \rightarrow \beta A'$
$A' \rightarrow \alpha A' \mid \varepsilon$

$S \rightarrow 01S'$
$S' \rightarrow 0S1SS' \mid \varepsilon$

Contd.,

Ex3: $A \rightarrow (B) \mid b$

$B \rightarrow B\,x\,A \mid A$

$A \rightarrow A \quad \alpha \mid \beta$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$

$A \rightarrow (B) \mid b$

$B \rightarrow AB'$

$B' \rightarrow xAB' \mid \varepsilon$

Example

- Consider grammar for arithmetic expressions

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

- $E \rightarrow E + T$ is **immediate left recursion.**
- $T \rightarrow T * F$ is also **immediate left recursion.**
- F is fine (no left recursion).
- If we try recursive descent directly, it'll **loop forever** on E or T.

Step-by-step Transformation

$A \rightarrow \beta A'$
$A' \rightarrow \alpha A' \mid \varepsilon$

For E: $E \rightarrow E + T \mid T$

Here, $\alpha = + T$, $\beta = T$

Rewrite:

$E \rightarrow T E'$

$E' \rightarrow + T E' \mid \varepsilon$

For T: $T \rightarrow T * F \mid F$

Here, $\alpha = * F$, $\beta = F$

Rewrite:

$T \rightarrow F T'$

$T' \rightarrow * F T' \mid \varepsilon$

$F \rightarrow (E) \mid id$

No left recursion → unchanged.

Contd.,

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

Given Grammer

$\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow$

$E \rightarrow T E'$
$E' \rightarrow + T E' \mid \epsilon$
$T \rightarrow F T'$
$T' \rightarrow * F T' \mid \epsilon$
$F \rightarrow (E) \mid id$

After removal of left recursion, the grammar becomes