

TOP-DOWN

Construction of a Procedural, Recursive Descent Parser

A procedure P_A to each non-terminal A :

Let $A ::= X_1 X_2 \dots X_n$

```
procedure  $P_A()$ ;  
begin  
  ** code for  $X_1$  **  
  ** code for  $X_2$  **  
  .....
```

```
  ** code for  $X_n$  **
```

```
end;
```

Construction INVARIANT

$P_A()$, applied to a Parser state having input γ , terminates with *success* iff $\gamma \in L(A) = \{\gamma \mid \dots\}$

Construction of a Procedural, Recursive Descent Parser

Code of the X_i

code of the X_i

$P_{x_i}()$ if X_i is non-terminal

match(X_i) if X_i is terminal

Definition of procedure match

```
procedure match(X:token);  
begin  
  if lookahead = X  
  then lookahead:= nextToken();  
  else fail()  
end;
```

The definitions of the auxiliary Procedures (see nextToken, fail) and of state variables:

- must be provided
- depend from the use context

Recursive Descent Parser: The Complete Structure

```
Procedure Parser();  
  **dichiarazioni procedure:  
    Ps, PA, PX1, ... **  
begin  
  Ps();  
  match(eof/$);  
  accept()  
end;
```

Example

```
E ::= T + T  
T ::= id
```

```
Procedure PE();  
begin  
  PT();  
  match(+);  
  PT()  
end;
```

```
Procedure PT();  
begin  
  match(id)  
end;
```

Recursive Descent Parser: The Running

```
Procedure Parser();  
Procedure PE();...;  
Procedure PT();...;  
begin  
  PE();  
  match(eof/$);  
  accept()  
end;
```

```
Procedure PE();  
begin  
  PT();  
  match(+);  
  PT()  
end;
```

```
Procedure PT();  
begin  
  match(id)  
end;
```

Control state during the analysis

```
Invocations:  
Parser()  
  PE  
  PT  
    match(id)  
      nexttoken  
    match(+)  
      nexttoken  
  PT  
    match(id)  
      nexttoken  
  match($)  
  accept()
```

```
Input:  
id+id$  
  
+id$  
  
id$  
  
$
```

- Simple to do but at a **poor efficiency**
- **What to do** when more than one production has the same left non-terminal?

Top-Down

What to do when more than...

$E ::= E + T$
 $E ::= T$
 $T ::= id$
 $T ::= num$

**All Choices
(direct)**

Backtracking

- Procedures ask for backtracking on choices
- Stack: Control States are stacked
- Case-statement similar to coding used in scanners defined by diagrams

Two Solutions

Predictive

LL Grammars (possibly via transformations)

- 1) *Lookahead* to remove nondeterminism on choice
- 2) *No (Removal of) Left Recursion*
- 3) *No (Removal of) Kleene's * operator*

Backtrack

Why Can it Work?

Let $G = \langle V, \Sigma, s \in V, P \rangle$. Then $\forall \alpha, \beta \in SF$

Monotony:

$\alpha \Rightarrow^* \beta$ only if $|\alpha|_{\Sigma} \leq |\beta|_{\Sigma}$

Persistency:

$\alpha \Rightarrow^* \beta$ only if $(\forall i, k_i, \alpha_i, \alpha_{i+k_i} \in \Sigma$ only if $\alpha_i \equiv \beta_j, \alpha_{i+k_i} \equiv \beta_{j+k_j}$ for some $j, k_j)$

where:

- $|\alpha|_{\Sigma}$ is the number of Σ symbols that are in α
- α_i is the i -th symbol, from left, in α

Predictive Top-Down

Remove nondeterminism on choice

- * By **looking ahead** an **initial trait** of the string, that has to be derived from a non-terminal, **at most one** production has to be applicable
- * *Lookahead* contains such an initial trait: k symbols
- * In the example below, procedure P_T behaves like a predictive 1-symbol of lookahead

```
T ::= id  
T ::= num
```

```
procedure  $P_T$ ();  
begin  
  case lookahead of  
    num: match(num);  
    id: match(id)  
  end  
end;
```

Predictive Top-Down:

1) Left Factoring

- * Lookahead of size $K > 1$ are hard to handle and expensive
- * $K = 1$ is not enough, below, when $|\alpha| > 0$

$$\begin{array}{l} E ::= \alpha \beta_1 \\ E ::= \alpha \beta_2 \end{array}$$

Left Factoring

consists in the replacement, below:

$$\begin{array}{l} E ::= \alpha B \\ B ::= \beta_1 \\ B ::= \beta_2 \end{array}$$

Predictive Top-Down:

2) Removal of Left Recursion

```
E ::= E + id  
E ::= id
```

```
procedure PE();  
begin  
  case lookahead of  
  id:  
  end  
end;
```

```
PE();
```

```
match(id);
```

Left Recursion Removal

```
A ::= A α  
A ::= β
```

```
A ::= β A  
A ::= α A  
A ::= ε
```

Lifting of Left Recursion

```
A ::= B α | γ  
B ::= A β | δ
```

```
A ::= B α | γ  
B ::= (B α | γ) β | δ
```

Predictive Top-Down:

3) Removal of Kleene's Star

Star Removal

$A ::= \alpha^* \beta$



$A ::= \underline{A} \beta$

$\underline{A} ::= \alpha \underline{A}$

$\underline{A} ::= \epsilon$

Predictive Top-Down

Doing it: Recursive Descent

Read lookahead and Select the right code of the procedure to be applied

What about the complexity?? (for string of n symbols and k productions per reduced symbol)

it is $O(n*k)$, provided that the grammar allows prediction...

Is it suitable for one-pass parsing ????

it is perfect (but expensive in time/space)

Predictive Top-Down To Do: In Summary

Grammar 3-step Transformation:
Kleene's Star Removal
Left Factorization
Left Recursion Removal



Lookahead:
Computation of: First and Follow



Definition of *Parser()* and its auxiliaries

Example: Part1

The grammar

$E ::= F (+ F)^*$

$F ::= F * T$

$F ::= T$

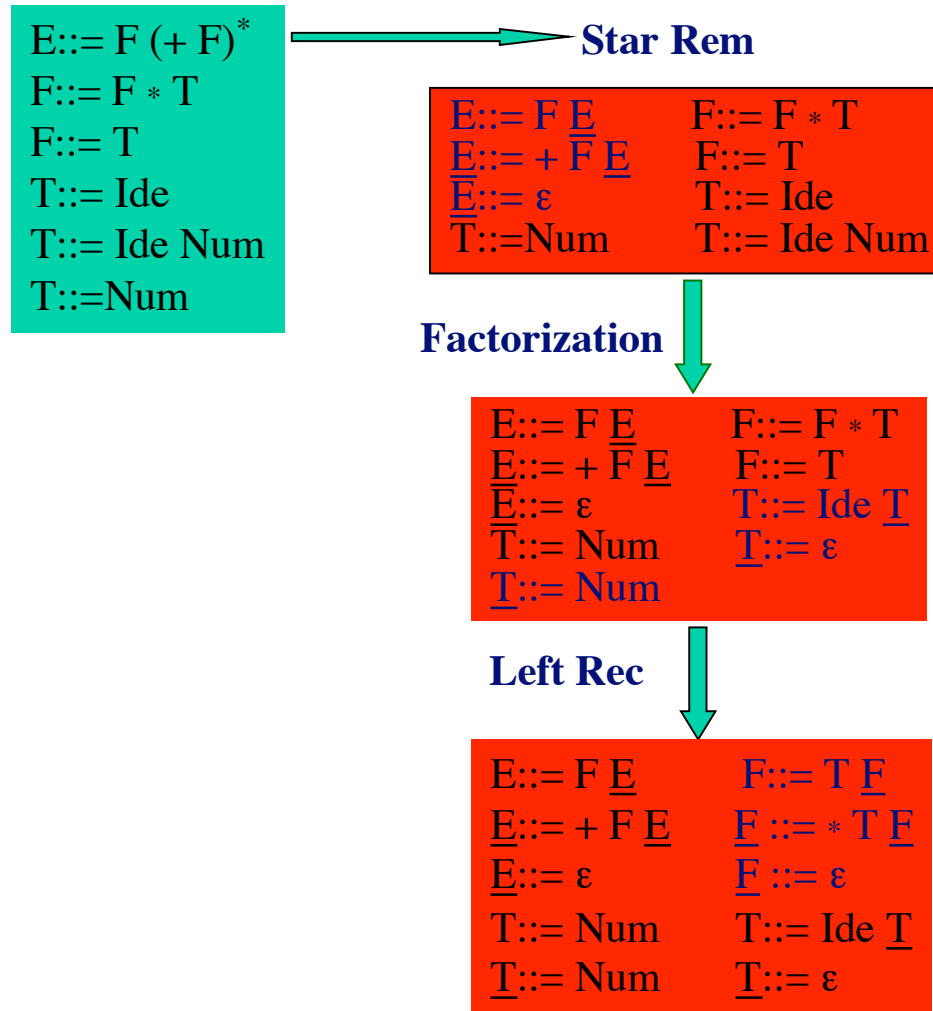
$T ::= \text{Ide}$

$T ::= \text{Ide Num}$

$T ::= \text{Num}$

Apply the 3-Step Transformation

Example: Part2



Example: Part3

```
E ::= F E      F ::= T F
E ::= + F E    F ::= * T F
E ::= ε        F ::= ε
T ::= Num      T ::= Ide T
T ::= Num      T ::= ε
```

The procedure body selects the code to be run on the basis of the value of lookahead

```
procedure P_T();
begin
  case lookahead of
    ??? : match(Num);
    ??? : begin match(Ide); P_T() end
  end
end;
```

```
procedure P_E();
begin
  case lookahead of
    ??? : begin P_F(); P_E() end
    ??? : fail();
  end
end;
```

- How to replace **???**, is simple for P_T .
- look at the first symbol that can be derived from LSF that is right side of the production to be apply
- What about $P_E P_F P_E P_F P_T$?

Example: Part4

```
E ::= F E      F ::= T F
E ::= + F E   F ::= * T F
E ::= ε        F ::= ε
T ::= Num      T ::= Ide T
T ::= Num     T ::= ε
```

...first symbols derived from right side of the production...
For T: **Num =>* Num, Ide=>*Ide**. Then:

```
procedure P_T();
begin
  case lookahead of
    Num : match(Num);
    Ide : begin match(Ide); P_T end
  end
end;
```

...first symbols derived from right side of the production...

For E: **FE =>* TFE** . Then:

```
procedure P_E();
begin
  case lookahead of
    Num, Ide : begin P_F(); P_E() end
    otherwise : fail();
  end
end;
```

For E: **FE =>* +FE** , **ε =>* .** Then:

```
procedure P_E();
begin
  case lookahead of
    + : begin match("+"); P_F(); P_E() end
    ??? : return;
  end
end;
```


Predictive Top-Down: First and Follow Traits

The grammar on the right is a predictive k-lookahead [k=1] symbols, grammar. It copes perfectly with linear top-down parsers.

The main feature of these grammars is the property below.

Π : Produzioni

$E ::= F \underline{E}$

$\underline{E} ::= + F \underline{E}$

$\underline{E} ::= \epsilon$

$T ::= \text{Num}$

$\underline{T} ::= \text{Num}$

$F ::= T \underline{F}$

$\underline{F} ::= * T \underline{F}$

$\underline{F} ::= \epsilon$

$T ::= \text{Ide } \underline{T}$

$\underline{T} ::= \epsilon$

Property. For predictive 1-lookahead symbols grammar $G = \langle V, \Sigma, s \in V, \Pi \rangle$, the following derivations always apply:

- if $\alpha A \beta \xRightarrow{*} \alpha \gamma \delta$ and $A ::= a \gamma' \in \Pi$ then
 $\alpha A \beta \xRightarrow{*} \alpha a \gamma' \beta \xRightarrow{*} \alpha \gamma \delta$
- if $\alpha A \beta \xRightarrow{*} \alpha \gamma \delta$ and $A ::= \epsilon \in \Pi$ and $\beta \xRightarrow{*} a \gamma'$ then
 $\alpha A \beta \xRightarrow{*} \alpha \beta \xRightarrow{*} \alpha \gamma \delta$

Predictive Top-Down:

Definition of the function **First**

first: $(\Sigma \cup \{ \epsilon \})^* \rightarrow 2^{\Sigma \cup \{ \epsilon \}}$

first(γ) = $\{ a \in \Sigma \mid \gamma \Rightarrow^* a\gamma' \} \cup \{ \epsilon \mid \gamma \Rightarrow^* \lambda \}$

first(c) = $\{ c \} \quad \forall c \in \Sigma$

first(s) = $\bigcup_{s ::= \alpha \in P} \text{first}(\alpha)$

first(ϵ) = $\{ \epsilon \}$

first($x_1 \dots x_n$) = $[\bigcup_{i < k} (\text{first}(x_i) - \{ \epsilon \})] \cup \text{first}(x_k)$
for maximum $1 \leq k < n, x_1 \dots x_{k-1} \Rightarrow^* \lambda$

Apply to:

$\underline{E} ::= F \underline{E}$	$\underline{F} ::= T \underline{F}$
$\underline{E} ::= + F \underline{E}$	$\underline{F} ::= * T \underline{F}$
$\underline{E} ::= \epsilon$	$\underline{F} ::= \epsilon$
$\underline{T} ::= \text{Num}$	$\underline{T} ::= \text{Ide } \underline{T}$
$\underline{T} ::= \text{Num}$	$\underline{T} ::= \epsilon$

Predictive Top-Down:

Definition of the function Follows

follow: V → 2^{Σ ∪ { \$ }}

follow(A) ⊇ { a ∈ Σ ∪ { \$ } | αAγ ∈ LSF & γ\$ ₁ ⇒* aγ' }

follow(A) = ∪_{B ::= αAβ} (first(β) - {ε})

∪_{B ::= αAβ | β ⇒* λ} follow(B)

{eof/\$} ∈ follow(S₀) for distinct (start) symbol S₀

Apply to:

<u>E</u> ::= F <u>E</u>	<u>F</u> ::= T <u>F</u>
<u>E</u> ::= + F <u>E</u>	<u>F</u> ::= * T <u>F</u>
<u>E</u> ::= ε	<u>F</u> ::= ε
<u>T</u> ::= Num	<u>T</u> ::= Ide <u>T</u>
<u>T</u> ::= Num	<u>T</u> ::= ε