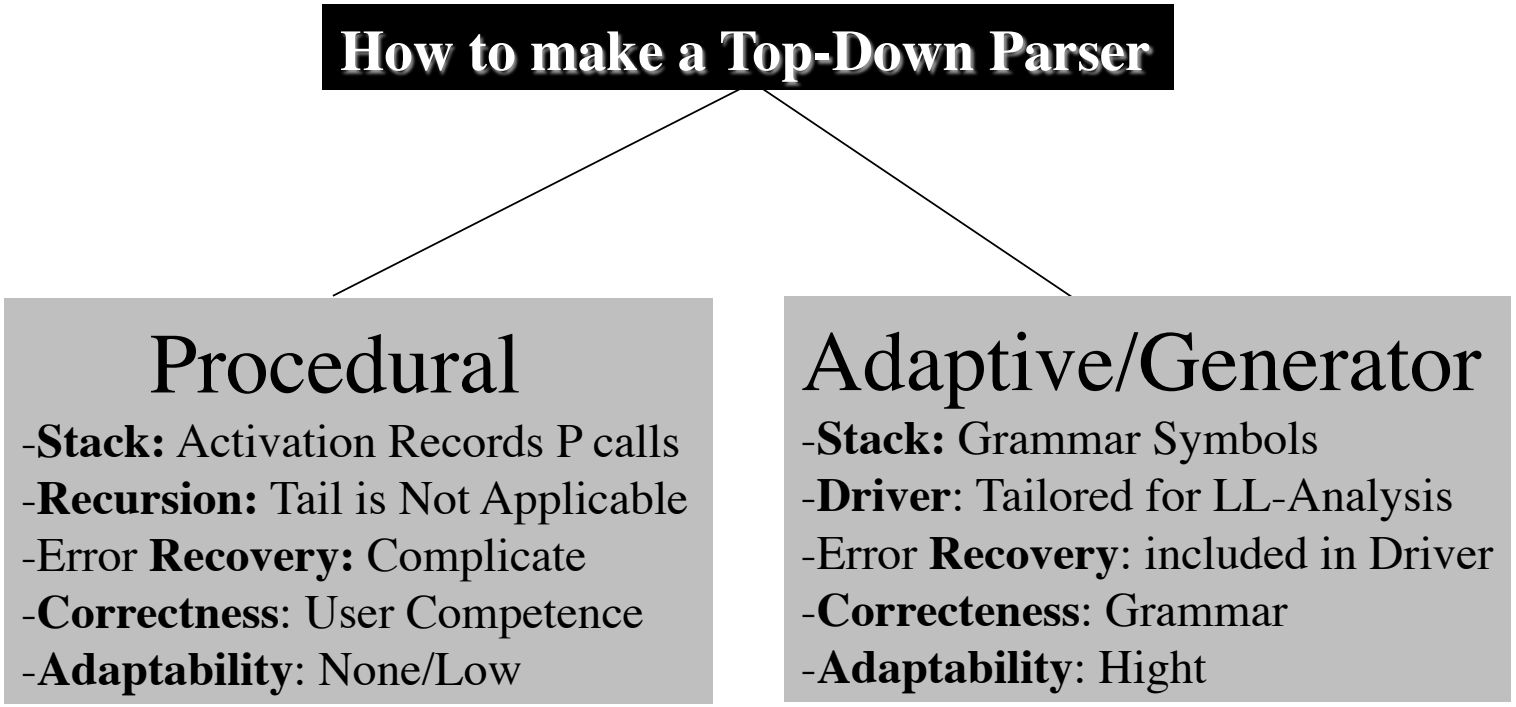
TOP-DOWN

How to make Top-Down Parsers



Construction of a Procedural, Recursive Descent Parser

A procedure P_A to each non-terminal A:

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

```
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 $\gamma \in L(A)=\{\gamma\ |...\}$

Construction of a Procedural, Recursive Descent Parser
Code of the X_i

code of the X_i

P_{xi}() *if* Xi is non-terminal

match(Xi) *if* Xi 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  P_E();,,,;
 Procedure  P_T();…;
 begin
   P_E();
   match(eof/$);
   accept()
 end;
```

```
Procedure  P_E();
 begin
   P_T();
   match(+);
   P_T()
 end;
```

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

Control state during the analysis

Invocations:	**Input:**
Parser()	id+id$
P_E	
P_T	
match(id)	
nexttoken	+id$
match(+)	
nexttoken	id$
P_T	
match(id)	
nexttoken	$
match($)	
accept()	

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

TOP-DOWN

Construction of a Procedural, Recursive Descent Parser

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

Two Solutions

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

Predictive

LL Grammars (possibly via transformations)

1) *Lookahead* to remove nondeterminism on choise
2) *No Left Recursion* (Removal of)
3) *No Kleene * operator* (Removal of)

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** $(\forall i,ki\geq 0,\ \alpha_i,\alpha_{i+ki} \in \Sigma$ **only if** $\alpha_i \equiv \beta_j,\ \alpha_{i+ki} \equiv \beta_{j+kj}$ **fo** $i,kj\geq 0)$

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

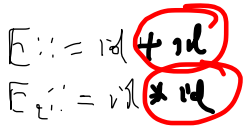
```
T::= id
T::= num
```

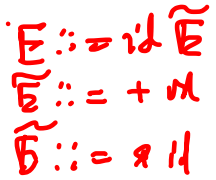
```
procedure P_T();
begin
  case lookahead of
            num: match(num);
            id: match(id)
  end
end;
```

Predictive Top-Down:
1) Left Factoring

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

$E ::= \alpha\ \beta_1$
$E ::= \alpha\ \beta_2$





Left Factoring
consists in the replacement, below:

$E ::= \alpha\ B$
$B ::= \beta_1$
$B ::= \beta_2$

Predictive Top-Down:
2) Removal of Left Recursion

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

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

→ $P_E()$;
→ match(id);

Left Recursion Removal

A::= A α
A::= β

→

A::= β A
A::= α A
A ::= ε

Lifting of Left Recursion

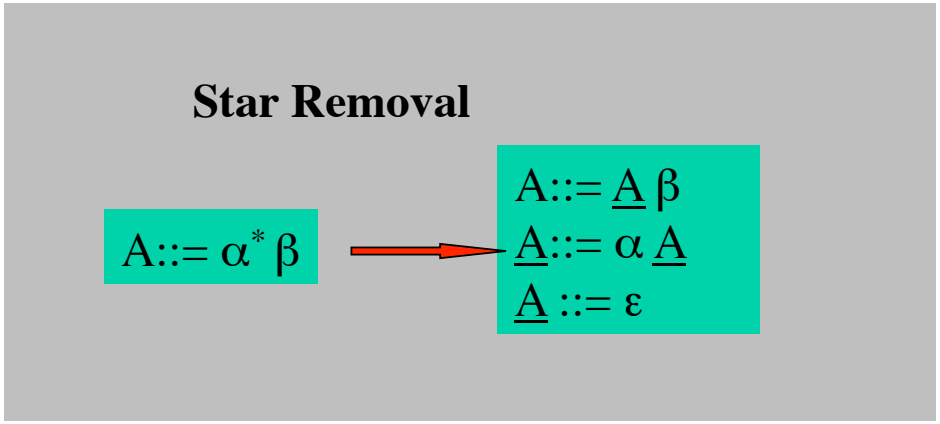
A::= B α | γ
B::= A β | δ

→

A::= B α | γ
B::= (B α | γ) β | δ
B::= BαB | γβ | δ

Predictive Top-Down:

3) Removal of Kleene's Star



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 strings of n symbols and k productions per reduced symbol)

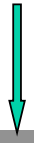
It is O(n*k), provided that the grammar allows prediction….

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

It is perfect (but expensive in time/space, since procedures and procedure calls)

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*

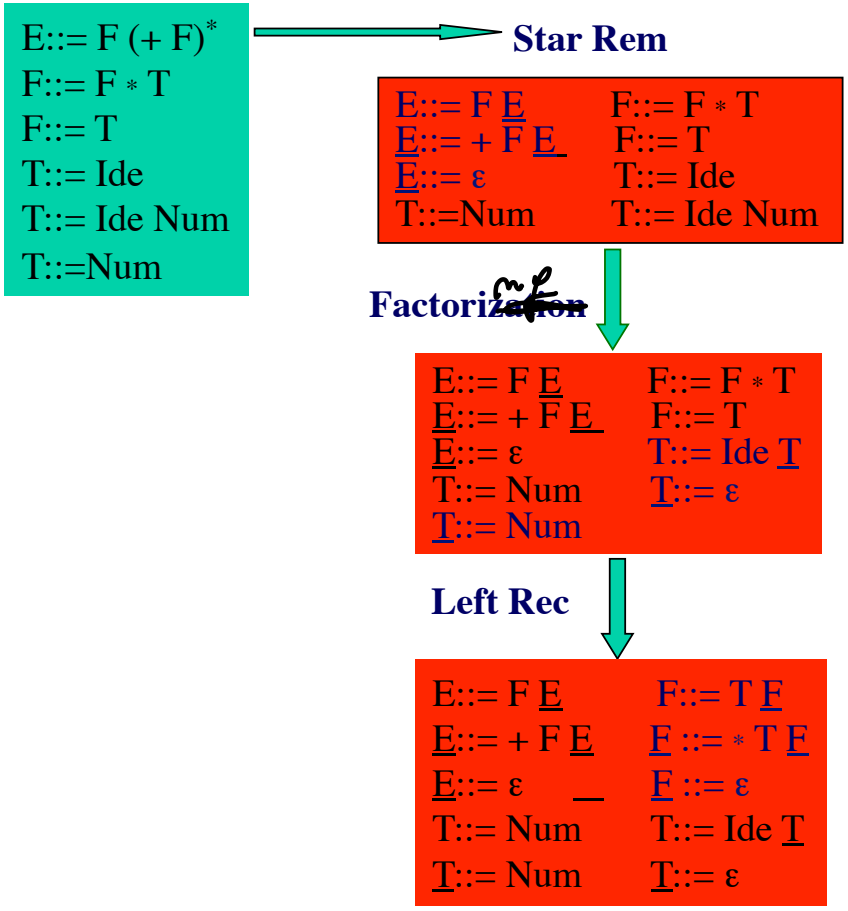
Example: Part1

The grammar

E::= F (+ F)*
F::= F * T
F::= T
T::= Ide
T::= Ide Num
T::= Num

Apply the 3-Step Trasformation

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;
```

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

Property. For predictive 1-lookahead symbols grammar **G=<V,Σ,s∈V,Π>**, the following derivations always apply:

- **if** $\alpha A \beta \Rightarrow_l^* \alpha a \gamma \delta$ **and** $A ::= a\gamma' \in \Pi$ **then**
  $\alpha A \beta \Rightarrow_l \alpha a \gamma' \beta \Rightarrow_l^* \alpha a \gamma \delta$
- **if** $\alpha A \beta \Rightarrow_l^* \alpha a \gamma \delta$ **and** $A ::= \varepsilon \in \Pi$ **and** $\beta \Rightarrow_l^* a\gamma'$ **then**
  $\alpha A \beta \Rightarrow_l \alpha \beta \Rightarrow_l^* \alpha a \gamma \delta$

Predictive Top-Down:

Definition of the function First

first:$(\Sigma \cup V)^* \rightarrow 2^{\Sigma \cup \{\varepsilon\}}$
first$(\gamma)=\{a \in \Sigma \mid \gamma \Rightarrow^* a\gamma'\} \cup \{\varepsilon \mid \gamma \Rightarrow^* \lambda\}$

first(c)={c} $\quad \forall c \in \Sigma$
first(s)=$\cup_{s::=\alpha \in P}$ first(α)
first(ε)={ε}
first($x_1 \ldots x_n$)=[$\cup_{i<k}$ (first(x_i)-{ε})]$\cup$first(x_k)
***for* maximum $1 \le k < n$, $x_1 \ldots x_{k-1} \Rightarrow^* \lambda$**

Apply to:

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

Predictive Top-Down:

Definition of the function Follows

follow: V->$2^{\Sigma \cup \{\$\}}$

follow(A)$\subseteq \{a \in \Sigma \cup \{\$\} \mid \alpha A \gamma \in LSF \ \& \ \gamma\$ \Rightarrow^*_1 a\gamma'\}$

follow(A)= $\bigcup_{\{B::=\alpha A\beta\}}$ **(first(β)-{ε})**

$\cup$

$\bigcup_{\{B::=\alpha A\beta \mid \beta \Rightarrow^* \lambda\}}$ **follow(B)**

$\alpha A \gamma \$ \Rightarrow^*_L \alpha A a \gamma'$

{eof/$}∈**follow**($S_0$) for distinct (start) symbol S_0

Apply to:

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(E) = FIRST(F E) = {num, ide}

FIRST(F) = FIRST(T F) = {num, ide}

FIRST(E)

FIRST(T) = {num, ide}