Computation of FIRST & FOLLOW for a grammar G

$G = \langle \{E, \underline{E}, F, \underline{F}, T, \underline{T}\},$
$\{+, *, IDE, NUM\},$
E, Π (see fig.) $\rangle$

$E ::= F\ \underline{E}$
$\underline{E} ::= +\ F\ \underline{E}$
$\underline{E} ::= \varepsilon$
$T ::= Num$
$\underline{T} ::= Num$

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

Predictive Top-Down:

Recursive Descent using First and Follow - 1

Using functions First and Follow, we can define possibly linear, recursive descent parser.

Step 1: For each non-terminal A, Let $\{\alpha_i \mid 1 \le i \le n, A ::= \alpha_i \in P\}$ be the set of the production right sides of the syntactic category A. Then:

```
procedure $P_A$();
        begin
        case lookahead of
                **caseof($\alpha_1$)**  :  **codeof($\alpha_1$)**;
                **caseof($\alpha_2$)**  :  **codeof($\alpha_2$)**;
                ........
                **caseof($\alpha_n$)**  :  **codeof($\alpha_n$)**;
                end
        end;
```

Recursive Descent using First and Follow - 2
caseof and codeof

Step 2: For each production right side α_i, the applicability set of the production is *either* **first(α_i)** *or* **first(α_i)\{ε}+follow(A)**. Then:

$$\textbf{caseof}(\alpha_i) = \begin{cases} \textbf{first}(\alpha_i) & \textit{if } \varepsilon \notin \textit{first}(\alpha_i) \\ (\textbf{first}(\alpha_i)-\{\varepsilon\})\cup\textbf{follow}(A) & \textit{otherwise} \end{cases}$$

codeof(α_i) = *the same of slide1*

As an example: The complete definition of $P_{\underline{E}}$() is:

```
procedure $P_{\underline{E}}$();
begin
  case lookahead of
            + : begin match(+); $P_F$; $P_{\underline{E}}$ end
            $ : nop
  end
end;
```

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

Recursive Descent using First and Follow

Conclusive Remarks

Complexity: * linear **O(n) for n-size phrases**
* No backtrack: A Failure means “out of the language”

One-Pass: * Parser is **moving left-to-right, 1 input symbol a time**.
* Once parsed, the phrase is released in the output

Applicability: * **LL(k)** grammars (k=1 if used first as above)
* many Programming L. syntaxes are not LL(K) for any k.

Adaptability-Modifiability: * not at all
* changes in the syntax result in a **deep re-arrangement**, up to a **complete re-definition**

Applicability i.e. $LL(K)_{/k=1}$

Property 1: $\forall$ A::= α | β [equally {A::= α , A::= β}]
first(α)$\cap$**first**(β) = {}

Property 2: $\forall$ A::= α | β […]
if α =>*λ **then first**(β)$\cap$**follow**(A) = {}

Theorem. Let G be a context free grammar:

- **G$\in$LL(1) if and only if both Properties, 1 and 2, hold**
- **G admits predictive, linear, 1-Lookahead Symbol Parser if and only if both Properties, 1 and 2, hold**

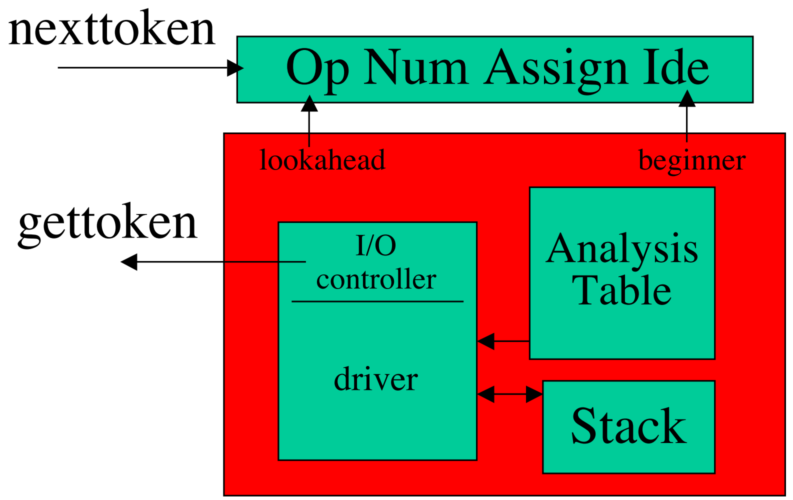
Adaptability of Recursive Descent vs. Adaptive Parser

- The need of **a deep revision of the R.D. parser code,** when (concrete) **syntax** has to be changed, is not an inspiring idea;
- The fact that it may happen also when, the **changes are in the grammar**, more than in the syntax, makes its use **even worse** (since grammar changes are in common use, in order to find a good grammar for the compiler needs: Trees, error detection and recovery,…
- Last but not least, the **writing from scratch**, of all the code for services that do not depend from the specific grammar, is a considerable time waste and source of code errors

Adaptive Parser is the solution for enhancing adaptability and also for costructing Parser Generators

Adaptive Parser - Parser Generator

A view of the Structure



- **Changes** in either the syntax or the grammar **result in changes in the Analysis Table**.

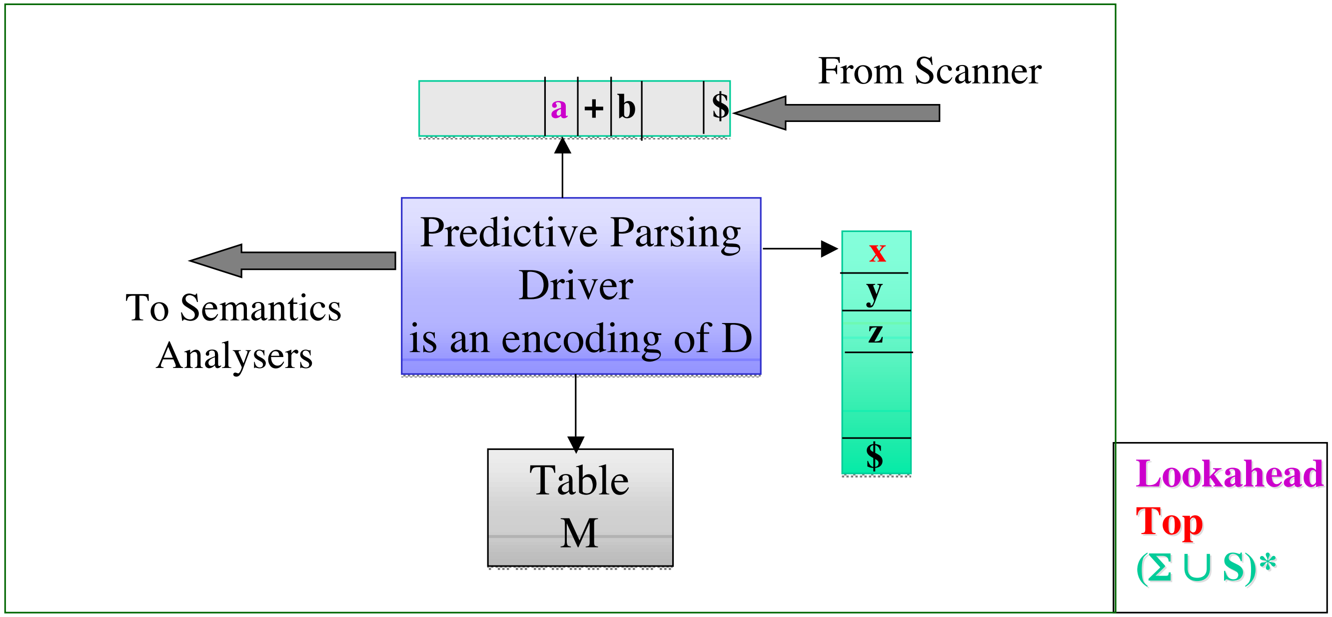
Push-Down Automata

are the perfect supports for predictive parsers

A **Pushdown Automaton** extends FSA and can be defined by the 6-tuple below:

$$\langle S,\ \Sigma,\ M: S \times \Sigma \dashrightarrow S,\ D: M \times (\Sigma \cup S)^* \dashrightarrow (\Sigma \cup S)^*,\ s_0 \in S,\ F \subseteq S \rangle$$

where **S, Σ, M,** s_0**, F** are the same of FSA, while **(Σ ∪ S)*** is a stack.



From Scanner

a + b $

Predictive Parsing Driver
is an encoding of D

To Semantics
Analysers

Table
M

x
y
z
$

Lookahead
Top
(Σ ∪ S)*

Push-Down Automata

The **definition of D** for LL(1) grammars

The function D for LL(1)

```
- top = lookahead = $: stop (with success)
- top = lookahead ≠ $: pop;
                       lookahead:= nexttoken
- top ∈S: pop;
          push(α)
          where  M(top,lookahead) =  top::=α
```

Push-Down Automata

The definition of Table M for LL(1) grammars

For each grammar production $A::=\alpha_i$

+ $\forall a \in (first(\alpha_i)-\varepsilon)$,
    $M(A,a) := A ::= \alpha_i$

+ if $\varepsilon \in first(\alpha_i)$ then:
    $\forall\ b \in follow(A)$,
        $M(A,b) := A ::= \alpha_i$

+ All the remaining table entries are marked “failure”

Predictive Paser: Adaptive/Generator

To Do: In Summary

Grammar Transformation:

- **Left Factoring**
- **Left Recursion Removal**
- **Kleene's Star Removal**

Construction of table M

computation of FIRST e FOLLOW

Example

Table Construction

Apply the construction of the adaptive/generator to a grammar (already transformed)

0.E::= F $\underline{E}$
1.$\underline{E}$::= + F $\underline{E}$
2.$\underline{E}$::= ε
3.T::= Num
4.$\underline{T}$::= Num
5.F::= T $\underline{F}$
6.$\underline{F}$ ::= ∗ T $\underline{F}$
7.$\underline{F}$ ::= ε
8.T::= Ide $\underline{T}$
9.$\underline{T}$::= ε

First(F $\underline{E}$) = {Ide, Num}
First(+ F $\underline{E}$) = {+}
First(T $\underline{F}$) = {Ide, Num}
First(∗ T $\underline{F}$) = {∗}

Fw($\underline{E}$) = Fw(E) = {\$}
Fw($\underline{F}$) = Fw(F) = First($\underline{E}$\$) = {+,\$}
Fw($\underline{T}$) = Fw(T) = First($\underline{F}$)∪Fw(F)
= {∗}∪{+,\$}

	'+	'*	Ide	Num	$
E			0	0	
$\underline{E}$	1				2
F			5	5	
$\underline{F}$	7	6			7
T			8	3	
$\underline{T}$	9	9		4	9

Example

Use

Apply the parser in the analysis of the string x+3*5: Show all the states (input/ stack) of the parser.

0.E::= F $\underline{E}$
1.$\underline{E}$::= + F $\underline{E}$
2.$\underline{E}$::= ε
3.T::= Num
4.$\underline{T}$::= Num
5.F::= T $\underline{F}$
6.$\underline{F}$ ::= * T $\underline{F}$
7.$\underline{F}$ ::= ε
8.T::= Ide $\underline{T}$
9.$\underline{T}$::= ε

	'+	'*	Ide	Num	$
E			0	0	
$\underline{E}$	1				2
F			5	5	
$\underline{F}$	7	6			7
T			8	3	
$\underline{T}$	9	9		4	9

x+3*5$	x+3*5$	x+3*5$	x+3*5$	+3*5$	+3*5$	+3*5$	+3*5$
E $	F $\underline{E}$ $	T $\underline{F}$ $\underline{E}$ $	I $\underline{T}$ $\underline{F}$ $\underline{E}$ $	$\underline{T}$ $\underline{F}$ $\underline{E}$ $	$\underline{F}$ $\underline{E}$ $	$\underline{E}$ $	+ F $\underline{E}$ $

3*5$	3*5$	3*5$	*5$	*5$	5$	5$	$	$	$
F $\underline{E}$ $	T $\underline{F}$ $\underline{E}$ $	N $\underline{F}$ $\underline{E}$ $	$\underline{F}$ $\underline{E}$ $	* T $\underline{F}$ $\underline{E}$ $	T $\underline{F}$ $\underline{E}$ $	N $\underline{F}$ $\underline{E}$ $	$\underline{F}$ $\underline{E}$ $	$\underline{E}$ $	$

Top Down:

Concluding Remarks -1

1. Consider the language **T** $=\{u^n v^k z^m \mid n,k,m > 0, \mathbf{n < m}\}$

   a) Give a grammar G such that L(G) = **T**

   b) Is G ∈LL(1) ?

   c) Have transformations of G, if any, predictive parsers ?

2. Consider the language **T** $=\{u^n v^k z^m \mid n,k,m > 0, \mathbf{n > m}\}$

   a) Give a grammar G such that L(G) = **T**

   b) Is G ∈LL(1) ?

   c) Have transformations of G, if any, predictive parsers ?

3. Is **LL(1)-inclusion**, decidable for Context Free Languages ?:
   (Equally, let **F** ≡ ∀**T**, ∃G: (L(G)=**T** and G ∈LL(1)).
   Is **F** a computable decision function ?

Top Down:

Concluding Remarks -2

4. Are LL(1)-Grammars strongly included in LL(K+1)-Grammars ?

5. Are LL(1)-Languages strongly included in LL(K+1)-Languages ?

6. What about conditions for LL(k)

let $G = \langle V, \Sigma, s, \prod \rangle$

$(\forall A ::= \beta_1 | \beta_2 \in \prod)$ and $((\forall \gamma): s \;{}_{l}{=}\!{>}^{*}\, \alpha A \gamma)$

$first_k(\beta_1\, \gamma) \cap first_k\, (\beta_2\, \gamma) = \{\}$

$(\forall A ::= \beta_1 | \beta_2 \in \prod)$

$first_k(\beta_1\, follow_k(A)) \cap first_k(\beta_2\, follow_k(A)) = \{\}$

Top Down:

Concluding Remarks -3

6. Definition of first_k e follow_k

$\forall G=<V, \Sigma, s, P>$,

- $\forall \gamma \in (\Sigma \cup V)^{*}$,

  $\text{first}_k(\gamma)=\{\alpha \mid \gamma \;{}_{l}\!\Rightarrow^{*} \alpha\gamma' \wedge (|\alpha|<k \supset |\gamma'|=0)\}$

  $\cup \{\varepsilon \mid \gamma \;{}_{l}\!\Rightarrow^{*} \lambda\}$

- $\forall A \in V$,

  $\text{follow}_k(A)=\{\alpha \mid \exists \delta A\gamma \in LSF_G, \alpha \in \text{first}_k(\gamma\$)$

Top Down: Implementations

Parser Predittivo

Recursive Descent

-**Stack:** Activation Records P calls
-**Recursion:** Tail is Not Applicable
-Error **Recovery:** Complicate
-**Correctness**: User Competence
-**Adaptability**: Low

Adaptive/Generator

-**Stack:** Grammar Symbols
-**Driver**: Tailored for LL-Analysis
-Error **Recovery**: included in Driver
-**Correcteness**: Grammar
-**Adaptability**: Hight

Adaptive is better because:

Tailored for LL (1) the code is written in a suitable language and possibly tested or verified, only once. (2) the code has been designed to interface in the most suitable and efficient way for the used platform.

Recovery (1) It requires the knowledge of specific techniques (2) The implementation may result hard to do when the recovery structures have to traverse the language control stasks (as in the recursive descent parsers).

Correctness: (1) Only limited to grammar correcteness; (2) Safe transformations, from the grammar to the analyser, are used

Adaptability: The grammar of a language (not the syntax) is changed during implementation. For example, Javac adopted a LALR grammar for Java, obtained after many changes that affected the Abstract Syntax Tree of programs.