Syntactic Analysis (Parsing)

José Miguel Rivero

rivero@cs.upc.edu

Barcelona School of Informatics (FIB)

Universitat Politècnica de Catalunya BarcelonaTech (UPC)

FIB

Summary

- Linear Parsing Algorithms. (Counter)example
- Notations: BNF and Extended BNF
- Applying a Rule $A \rightarrow \alpha_i$
- Nullable, First and Follow
- Methods of Linear Parsing
  - Top-down Parsers
    - Grammar Restrictions in Top-down Parsing
      - Elimination of Left Recursion
      - Left Factoring
    - Types of Top-down Parsers
      - Table-driven Top-down Parser
      - Predictive Recursive Top-down Parser
  - Bottom-up Parsers

FIB

Linear Parsing Algorithms

- The list of tokens w will be visited only once, usually in a *left-to-right* traversal.
  The current token receives the name of *lookahead*
- Compute the derivation $S \Rightarrow^* w$ without *backtracking*.
  Sources of indeterminism at this point:

$$S \Rightarrow^* w_0 A_1 w_1 \ldots A_n w_n \Rightarrow$$

  - Which non-terminal A_i choose to expand?
  - If we have a rule of the form $A \to \gamma_1 | \ldots | \gamma_k$, which γ_j —if any— will be used?
    $\Rightarrow$ bear in mind the *lookahead* token.

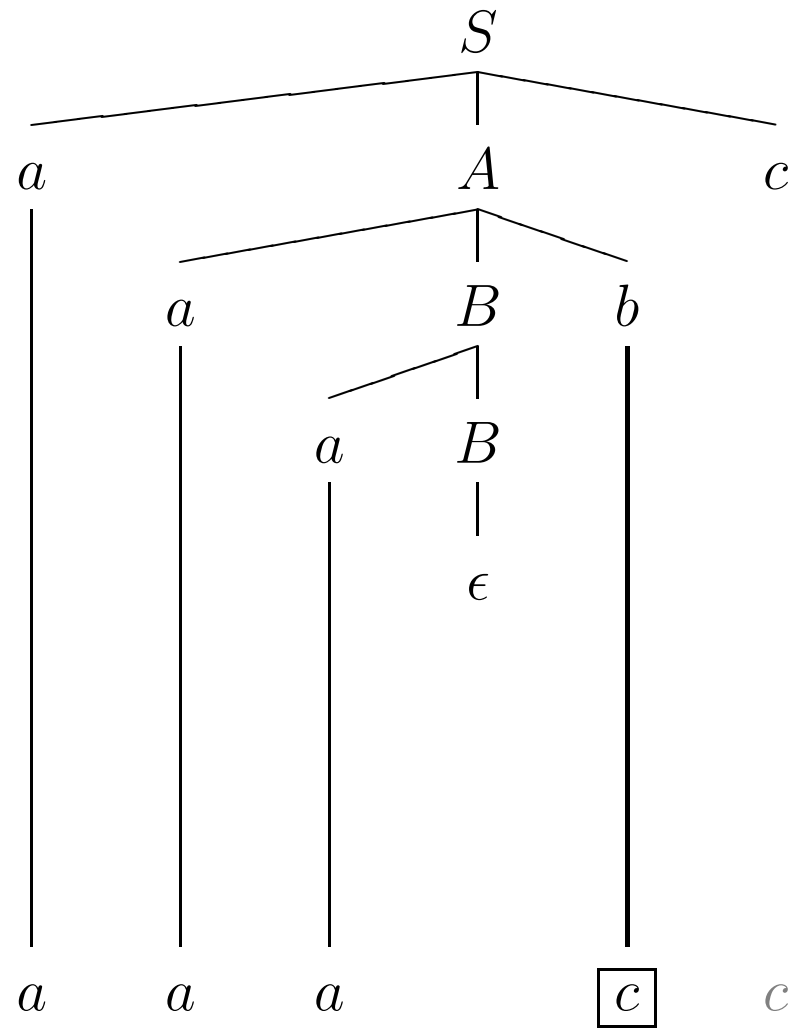
at most one γ_j may make sense

FIB

(Counter)example: backtracking

$$
\begin{aligned}
S &\rightarrow \quad aAc \\
&\quad\ \mid \quad c \\
A &\rightarrow \quad aBb \\
&\quad\ \mid \quad Bc \\
B &\rightarrow \quad aB \\
&\quad\ \mid \quad \epsilon
\end{aligned}
$$

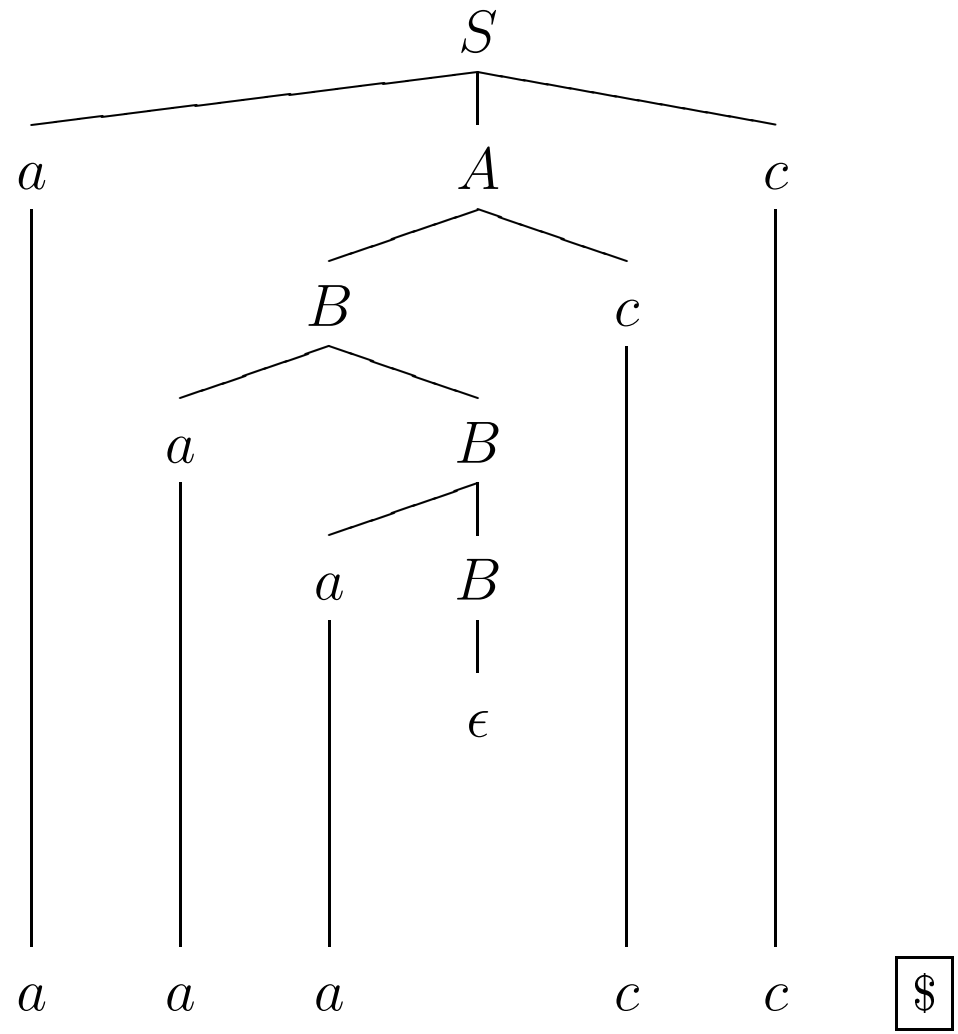
$w = a\,a\,a\,c\,c$



(Counter)example: backtracking

$$
\begin{aligned}
S &\rightarrow aAc \\
&\mid c \\
A &\rightarrow aBb \\
&\mid Bc \\
B &\rightarrow aB \\
&\mid \epsilon
\end{aligned}
$$

$w = a\,a\,a\,c\,c$



Exercises

- Find another simpler grammar that cannot be parsed with a linear algorithm.
  Give an input that demonstrate it.
- Characterize some conflicting grammars.

FIB

Notation. Backus-Naur Form (BNF)

- S, S' are the initial symbol of the grammar
- A, B, C, ..., are non-terminal symbols
- a, b, c, ..., and $\$$ are terminal symbols (tokens)
- X, Y, Z are terminal or non-terminal symbols
- u, v, w are words formed by terminal symbols, possibly empty (ϵ)
- α, β, γ are words formed by terminal and non-terminal symbols, possibly empty (ϵ)
- $A \to \alpha$ is a production rule.
- $A \to \alpha_1 \mid \ldots \mid \alpha_n$ is the group of rules for the non-terminal symbol A

FIB

Notation. Backus-Naur Form (BNF)

- $\gamma_1 A \gamma_2 \Rightarrow \gamma_1 \alpha \gamma_2$ is a derivation step with the rule $A \to \alpha$
- $\Rightarrow^*$ is the reflexive transitive closure of $\Rightarrow$
- Leftmost-derivation ($\Rightarrow^*_{ld}$) if every step:

  $$w A \beta \Rightarrow_{A \to \alpha} w \alpha \beta$$

- Rightmost-derivation ($\Rightarrow^*_{rd}$) if every step:

  $$\beta A w \Rightarrow_{A \to \alpha} \beta \alpha w$$

FIB

Extended Backus-Naur Form (EBNF)

Rules $A \to \alpha$ where α can take the form of a regular expression:

- $\alpha_1 \cdots \alpha_n$: concatenation, or ϵ
- $\alpha_1 | \ldots | \alpha_n$: alternatives
- α_1^* : 0 or more times α_1
- α_1^+ : 1 or more times α_1 ($\alpha_1 \alpha_1^*$)
- $\alpha_1?$: 0 or 1 times α_1 ($\alpha_1 | \epsilon$)
- (α_1) : parenthesis for breaking the standard precedence between operators

  $$\{\,|\,\} \prec_p \{\,\cdot\,\} \prec_p \{\,^*,\ ^+,\ ?\,\}$$
- a non-terminal symbol A, or a terminal symbol a

FIB

Extended Backus-Naur Form (EBNF)

In *ANTLR*, identifiers in capital letters denote terminals symbols, and identifiers beginning in lower case denote non-terminals.

Rules take the form id `:` reg_exp `;`

Example:

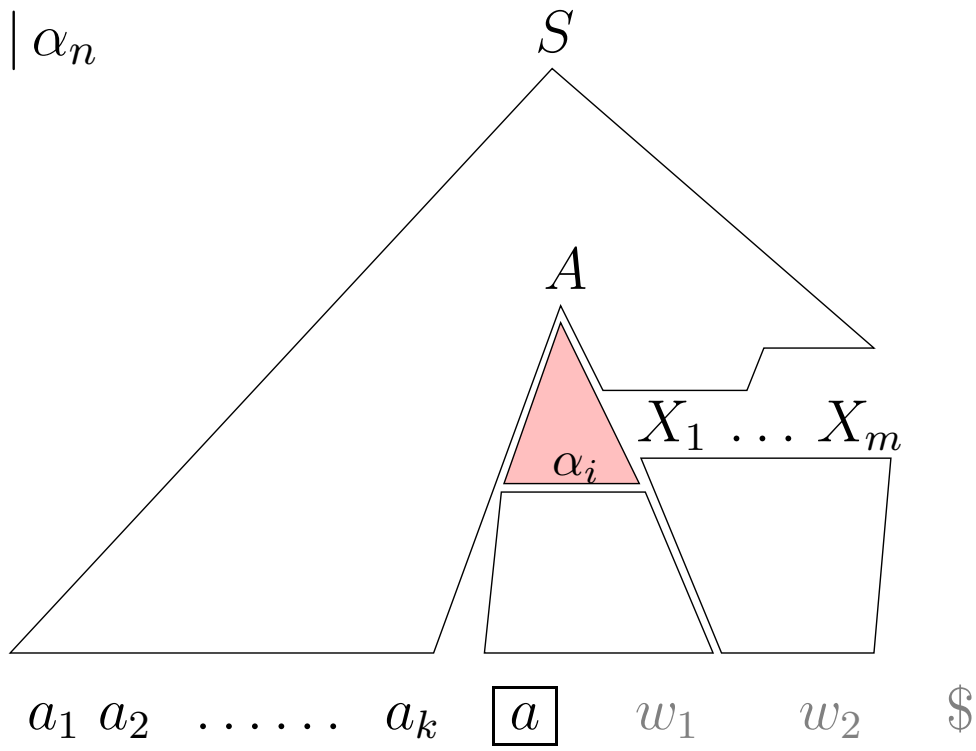
```
expr : term ( ( PLUS | MINUS ) term )* ;

term : factor ( ( TIMES | QUOTIENT ) factor )* ;

factor : IDENT ( LEFT_BRK expr RIGHT_BRK )?
       | NUM
       | LEFT_PAR expr RIGHT_PAR
       ;
```

Applying a Rule $A \rightarrow \alpha_i$

$A \rightarrow \alpha_1 \mid \ldots \mid \alpha_n$



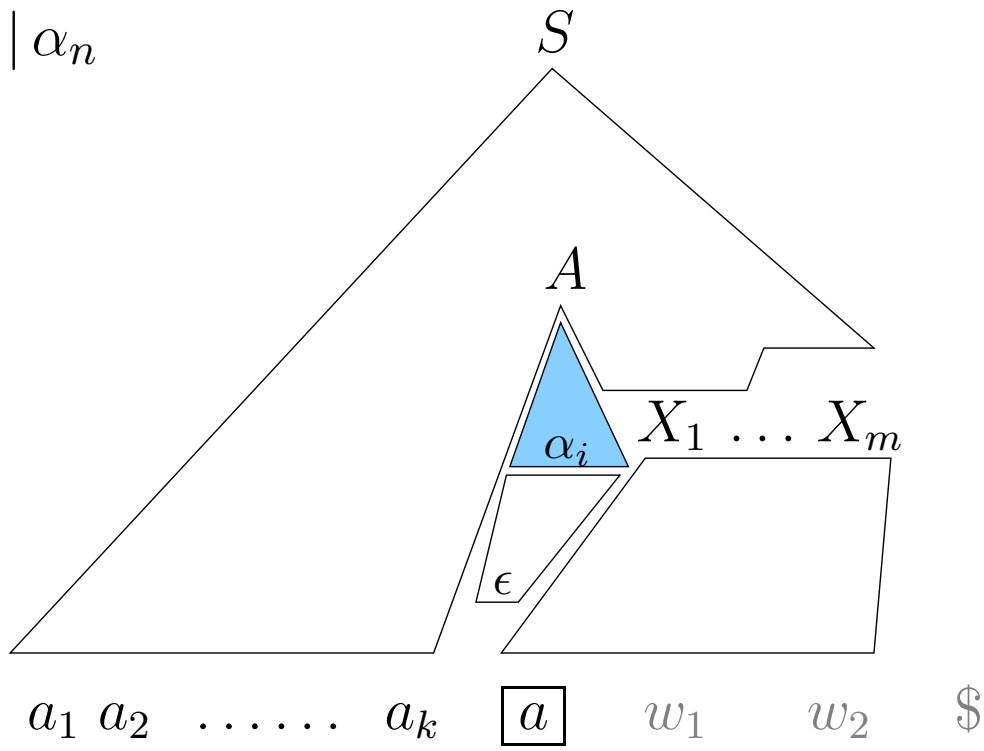
$A \Rightarrow \alpha_i \Rightarrow^* a\, w_1$

$a \in first(\alpha_i)$

$X_1 \ldots X_m \Rightarrow^* w_2$

Applying a Rule $A \rightarrow \alpha_i$

$A \rightarrow \alpha_1 \,|\, \ldots \,|\, \alpha_n$



$A \Rightarrow \alpha_i \Rightarrow^* \epsilon$

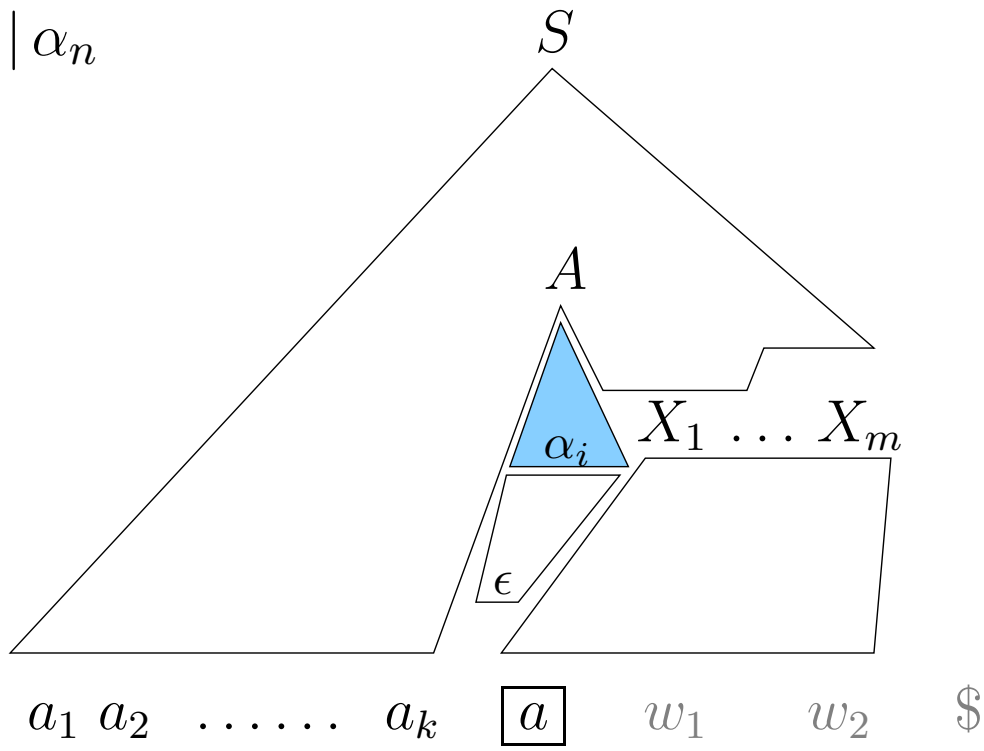
$nullable?(\alpha_i)$

$X_1 \ldots X_m \Rightarrow^* a\, w_1\, w_2$

$S\,\$ \Rightarrow^* a_1 \ldots a_k\, A\, a\, w_1\, w_2\, \$$

Applying a Rule $A \rightarrow \alpha_i$

$A \rightarrow \alpha_1 \mid \ldots \mid \alpha_n$



$A \Rightarrow \alpha_i \Rightarrow^* \epsilon$ $\quad$ $nullable?(\alpha_i)$

$S\,\$ \Rightarrow^* \gamma_1\, A\, a\, \gamma_2$ $\quad$ $a \in follow(A)$

Nullable, First, Follow

- Can α derive ϵ ?

$$nullable?(\alpha) \text{ iff } \alpha \Rightarrow^* \epsilon$$

- Set of terminals a that can be the first symbol of words derived from α

$$first(\alpha) = \{ a \mid \alpha \Rightarrow^* a\, w \}$$

- Set of terminals a that can follow A in a derivation from $S\,\$$

$$follow(A) = \{ a \mid S\,\$ \Rightarrow^* \gamma_1 A\, a\, \gamma_2 \}$$

FIB

To Sum Up ...

$$
\begin{array}{rcl}
S \rightarrow & aAc & \\
\quad | & c & \\
A \rightarrow & aBb & \\
\quad | & Bc & \\
B \rightarrow & aB & \\
\quad | & \epsilon &
\end{array}
\qquad
\begin{array}{rcl}
first(aB) & = & \{\, a \,\} \\
first(\epsilon) & = & \emptyset \\
first(B) & = & first(aB) \cup first(\epsilon) \\
 & = & \{\, a \,\} \\
first(aBb) & = & \{\, a \,\} \\
first(Bc) & = & \{\, a, c \,\} \\
first(A) & = & first(aBb) \cup first(Bc) \\
 & = & \{\, a, c \,\} \\
first(aAc) & = & \{\, a \,\} \\
first(c) & = & \{\, c \,\} \\
first(S) & = & first(aAc) \cup first(c) \\
 & = & \{\, a, c \,\}
\end{array}
$$

FIB

To Sum Up ...

$$
\begin{array}{rcl}
S \rightarrow & aAc & \\
 & | \; c & \\
A \rightarrow & aBb & \\
 & | \; Bc & \\
B \rightarrow & aB & \\
 & | \; \epsilon &
\end{array}
\qquad
\begin{array}{lcl}
first(aB) & = & \{ a \} \\
first(\epsilon) & = & \emptyset \\
nullable?(\epsilon) & = & true \\
follow(B) & = & \{ b, c \}
\end{array}
$$

FIB

$nullable?(\alpha)$ **(BNF)**

$$nullable?(\alpha) \text{ iff } \alpha \Rightarrow^* \epsilon$$

- $nullable?(a) = false$
- $nullable?(A) = true$ if $\exists\, A \to \alpha \in G \ \wedge\ nullable(\alpha)$
  $= false$ otherwise
- $nullable?(X_1 \ldots X_n) = true$ if $\forall i : 1 \leq i \leq n : nullable(X_i)$
  $= false$ otherwise

FIB

$nullable?(\alpha)$ **(Extended BNF)**

$$nullable?(\alpha) \text{ iff } \alpha \Rightarrow^* \epsilon$$

- $nullable?(\alpha_1\,\alpha_2) = true$ if $nullable?(\alpha_1) \wedge nullable?(\alpha_2)$
  $= false$ otherwise
- $nullable?(\alpha_1 \,|\, \alpha_2) = true$ if $nullable?(\alpha_1) \vee nullable?(\alpha_2)$
  $= false$ otherwise
- $nullable?(\alpha_1^*) = true$
- $nullable?(\alpha_1^+) = true$ if $nullable?(\alpha_1)$
  $= false$ otherwise
- $nullable?(\alpha_1?) = true$

FIB

$first(\alpha)$ (BNF)

$$first(\alpha) = \{\, a \mid \alpha \Rightarrow^* a\, w \,\}$$

- $first(a) = \{a\}$
- $first(A) \supseteq first(\alpha_j)$ if $\exists\, A \to \alpha_j \in G$
- $first(X_1 \ldots X_n) \supseteq first(X_i)$ if $nullable?(X_1 \ldots X_{i-1})$

FIB

$first(\alpha)$ (Extended BNF)

$$first(\alpha) = \{\, a \mid \alpha \Rightarrow^* a\, w \,\}$$

- $first(\alpha_1\, \alpha_2) = first(\alpha_1)$ if $\neg\, nullable?(\alpha_1)$
  $first(\alpha_1\, \alpha_2) = first(\alpha_1) \cup first(\alpha_2)$ if $nullable?(\alpha_1)$
- $first(\alpha_1 \mid \alpha_2) = first(\alpha_1) \cup first(\alpha_2)$
- $first(\alpha_1^*) = first(\alpha_1)$
- $first(\alpha_1^+) = first(\alpha_1)$
- $first(\alpha_1?) = first(\alpha_1)$

$follow(A)$ **(BNF)**

$$follow(A) = \{\, a \mid S\,\$ \Rightarrow^* \gamma_1 A\, a\, \gamma_2 \,\}$$

- $follow(S) \supseteq \{\,\$\,\}$
- $follow(B) \supseteq first(\beta)$ if $\exists\, A \to \alpha B \beta \in G$
- $follow(B) \supseteq follow(A)$ if $\exists\, A \to \alpha B \in G$

  $follow(B) \supseteq follow(A)$ if $\exists\, A \to \alpha B \beta \in G \;\wedge$ $nullable?(\beta)$

$follow(\alpha)$ (Extended BNF)

If α occurs in G: $follow(\alpha) = \{\, a \mid S\,\$ \Rightarrow^* \gamma_1 \alpha\, a\, \gamma_2 \,\}$

- $follow(\alpha) \supseteq follow(A)$ if $\exists\, A \to \alpha \in G$
- $follow(\alpha_1) \supseteq first(\alpha_2)$ if $\alpha_1\,\alpha_2$ occurs in G
- $follow(\alpha_2) \supseteq follow(\alpha_1\,\alpha_2)$ if $\alpha_1\,\alpha_2$ occurs in G
- $follow(\alpha_1) \supseteq follow(\alpha_1\,\alpha_2)$ if $\alpha_1\,\alpha_2$ occurs in G $\wedge\, nullable?(\alpha_2)$
- $follow(\alpha_1) \supseteq follow(\alpha_1 \,|\, \alpha_2)$ if $\alpha_1 \,|\, \alpha_2$ occurs in G
- $follow(\alpha_2) \supseteq follow(\alpha_1 \,|\, \alpha_2)$ if $\alpha_1 \,|\, \alpha_2$ occurs in G
- $follow(\alpha_1) \supseteq first(\alpha_1) \cup follow(\alpha_1^*)$ if α_1^* occurs in G
- $follow(\alpha_1) \supseteq first(\alpha_1) \cup follow(\alpha_1^+)$ if α_1^+ occurs in G
- $follow(\alpha_1) \supseteq follow(\alpha_1?)$ if $\alpha_1?$ occurs in G

FIB

Methods of Linear Parsing

The list of tokens will be traversed *left-to-right*. Decisions to proceed take into account one token of lookahead.

- Top-down parsers (LL(1))
  - Build the AST from the root to the leaves (top-down)
  - Follow a left-most derivation in forward direction
  - More intuitive: can be *manually* written
  - **Grammars may need preprocessing**
- Bottom-up parsers (LR(1))
  - Build the AST from the leaves to the root (bottom-up)
  - Follow a right-most derivation in *backward* direction
  - Less intuitive than top-down parsers
  - Slightly more powerful

FIB

Elimination of Left Recursion

Grammar G:

$$E \rightarrow E + \texttt{id} \mid \texttt{id}$$

$$first(E + \texttt{id}) = \{\texttt{id}\}$$

$$first(\texttt{id}) = \{\texttt{id}\}$$

$$first(E + \texttt{id}) \cap first(\texttt{id}) \neq \emptyset$$

$$w = \texttt{id} + \texttt{id} + \texttt{id}$$

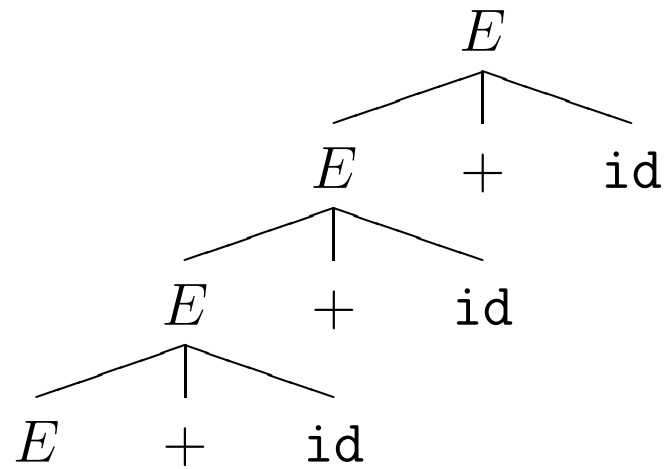
FIB

Elimination of Left Recursion

Grammar G:

$$E \rightarrow E + \texttt{id} \mid \texttt{id}$$

$$w = \texttt{id} + \texttt{id} + \texttt{id}$$



$$\boxed{\texttt{id}} \quad + \quad \texttt{id} \quad + \quad \texttt{id}$$

Elimination of Left Recursion (BNF)

$$
\begin{array}{rcl}
A & \rightarrow & A\,\alpha_1 \\
& | & \ldots \\
& | & A\,\alpha_n \\
& | & \beta_1 \\
& | & \ldots \\
& | & \beta_m
\end{array}
$$

Transform into right recursion:

$$
\begin{array}{rcl}
A & \rightarrow & \beta_1\,A' \\
& | & \ldots \\
& | & \beta_m\,A'
\end{array}
$$

$$
\begin{array}{rcl}
A' & \rightarrow & \alpha_1\,A' \\
& | & \ldots \\
& | & \alpha_n\,A' \\
& | & \epsilon
\end{array}
$$

FIB

Elimination of Left Recursion (EBNF)

$$
\begin{array}{rcl}
A & \rightarrow & A\,\alpha_1 \\
& | & \ldots \\
& | & A\,\alpha_n \\
& | & \beta_1 \\
& | & \ldots \\
& | & \beta_m
\end{array}
$$

Extended BNF: use regular expressions

$$A \rightarrow (\,\beta_1 \,|\, \ldots \,|\, \beta_m\,)\,(\,\alpha_1 \,|\, \ldots \,|\, \alpha_n\,)^*$$

$$
\begin{array}{rcl}
A & \rightarrow & B\,(A')^* \\
B & \rightarrow & \beta_1 \,|\, \ldots \,|\, \beta_m \\
A' & \rightarrow & \alpha_1 \,|\, \ldots \,|\, \alpha_n
\end{array}
$$

FIB

Exercises

$$
\begin{array}{rcl}
LI & \rightarrow & LI\ I \\
 & | & I
\end{array}
$$

$$
\begin{array}{rcl}
E & \rightarrow & E + T \\
 & | & T \\
T & \rightarrow & T * F \\
 & | & F \\
F & \rightarrow & (E) \\
 & | & \texttt{id}
\end{array}
$$

Indirect left recursion:

$$
\begin{array}{rcl}
A & \rightarrow & B\,d \\
B & \rightarrow & C\,e \\
C & \rightarrow & A\,f \\
 & | & g
\end{array}
$$

Left Factoring

$$
\begin{array}{rcl}
E & \rightarrow & T + E \\
 & | & T \\
 & & \\
T & \rightarrow & \texttt{id} \\
 & | & (E)
\end{array}
$$

$$first(T + E) = first(T) = \{\texttt{id}, (\}$$

$$first(T) = \{\texttt{id}, (\}$$

$$first(T + E) \cap first(T) \neq \emptyset$$

$$first(\texttt{id}) = \{\texttt{id}\}$$

$$first((E)) = \{(\}$$

$$first(\texttt{id}) \cap first((E)) = \emptyset$$

FIB

Left Factoring (BNF)

$$
\begin{array}{rcl}
A & \rightarrow & \beta\ \alpha_1 \\
& | & \ldots \\
& | & \beta\ \alpha_n \\
& | & \gamma_1 \\
& | & \ldots \\
& | & \gamma_m
\end{array}
$$

$$
\begin{array}{rcl}
A & \rightarrow & \beta\ A' \\
& | & \gamma_1 \\
& | & \ldots \\
& | & \gamma_m \\
 & & \\
A' & \rightarrow & \alpha_1 \\
& | & \ldots \\
& | & \alpha_n
\end{array}
$$

FIB

Left Factoring (EBNF)

$$
\begin{array}{rcl}
A & \rightarrow & \beta\ \alpha_1 \\
& | & \ldots \\
& | & \beta\ \alpha_n \\
& | & \gamma_1 \\
& | & \ldots \\
& | & \gamma_m
\end{array}
$$

Extended BNF:

$$A \rightarrow \beta\ (\ \alpha_1 \mid \ldots \mid \alpha_n\ ) \mid \gamma_1 \mid \ldots \mid \gamma_m$$

$$
\begin{array}{rcl}
A & \rightarrow & \beta\ A' \mid \gamma_1 \mid \ldots \mid \gamma_m \\
A' & \rightarrow & \alpha_1 \mid \ldots \mid \alpha_n
\end{array}
$$

Exercises

$$
\begin{array}{rcl}
P & \rightarrow & \texttt{if } C \texttt{ then } P \texttt{ endif} \\
 & | & \texttt{if } C \texttt{ then } P \texttt{ else } P \texttt{ endif} \\
 & | & \texttt{p} \\
C & \rightarrow & \texttt{c}
\end{array}
$$

$$
\begin{array}{rcl}
I & \rightarrow & LE \ \texttt{':='} \ E \\
 & | & \texttt{write '('} \ E \ \texttt{')'} \\
 & | & \texttt{id '('} \ E \ \texttt{')'} \\
E & \rightarrow & \texttt{id} \\
 & | & \texttt{num} \\
LE & \rightarrow & \texttt{id}
\end{array}
$$

Types of Top-down Parsers

- Table Driven parsers (iterative)
  - Parsing algorithm is fixed, driven by a decision table
  - Table M is built from the grammar G.
    Empty boxes correspond to syntax errors

M	a_1	$\ldots$	a	$\ldots$	a_n	\$
A_1						
$\vdots$						
A			$A \rightarrow \alpha_k$			
$\vdots$						
A_m						

Types of Top-down Parsers

- Table Driven parsers (iterative)
  - Parsing algorithm is fixed, driven by a decision table
  - Table M is built from the grammar G.
    Empty boxes correspond to syntax errors
- Recursive predictive parsers
  - Parsing algorithm is formed by a set of mutually recursive functions
  - Each rule $A \to \alpha$ generates the code of its function

```
void A(void) {
    // Code generated from α
}
```

  - `Gencode` describes how to translate a rule to the associated function

Table-driven Top-down Parser

M	a_1	$\ldots$	a	$\ldots$	a_n	\$
A_1						
$\vdots$						
A			$A \rightarrow \alpha_k$			
$\vdots$						
A_m						

$$A \rightarrow \alpha_1 \mid \ldots \mid \alpha_k \mid \ldots \mid \alpha_o$$

$A \rightarrow \alpha_k \in M[A, a]$ if :

- $a \in first(\alpha_k)$, or
- $nullable?(\alpha_k)$ and $a \in follow(A)$

Table-driven Top-down Parser

Algorithm to build the parser table $M[A, a]$

for all rule $A \to \alpha \in G$ do

add $A \to \alpha$ to $M[A, a]$ if:

- $a \in first(\alpha)$ or,
- $nullable?(\alpha)$ and $a \in follow(A)$

FIB

Exercises

Complete the table M for the following grammars, indicating the possible problems.

Grammar G_1 :

$$
\begin{array}{rcl}
E & \rightarrow & E + T \\
 & | & T \\
T & \rightarrow & T * F \\
 & | & F \\
F & \rightarrow & \texttt{id} \\
 & | & (E)
\end{array}
$$

Grammar G_2 :

$$
\begin{array}{rcl}
E & \rightarrow & T\, E' \\
E' & \rightarrow & + T\, E' \\
 & | & \epsilon \\
T & \rightarrow & F\, T' \\
T' & \rightarrow & * F\, T' \\
 & | & \epsilon \\
F & \rightarrow & \texttt{id} \\
 & | & (E)
\end{array}
$$

FIB

Exercises

Complete the table M for the following grammars, indicating the possible problems.

Grammar G_3 :

$$
\begin{array}{rcl}
P & \rightarrow & \texttt{if } C \texttt{ then } P \\
 & | & \texttt{if } C \texttt{ then } P \texttt{ else } P \\
 & | & \texttt{p} \\
C & \rightarrow & \texttt{c}
\end{array}
$$

Grammar G_4 :

$$
\begin{array}{rcl}
P & \rightarrow & \texttt{if } C \texttt{ then } P \ P' \\
 & | & \texttt{p} \\
P' & \rightarrow & \epsilon \\
 & | & \texttt{else } P \\
C & \rightarrow & \texttt{c}
\end{array}
$$

Exercises

Complete the table M for the following grammars, indicating the possible problems.

Grammar G_5 :

$$
\begin{array}{rcl}
P & \rightarrow & \texttt{if } C \texttt{ then } P \; P' \\
 & | & \texttt{p} \\
P' & \rightarrow & \texttt{endif} \\
 & | & \texttt{else } P \texttt{ endif} \\
C & \rightarrow & \texttt{c}
\end{array}
$$

Table-driven Top-down Parser Algorithm

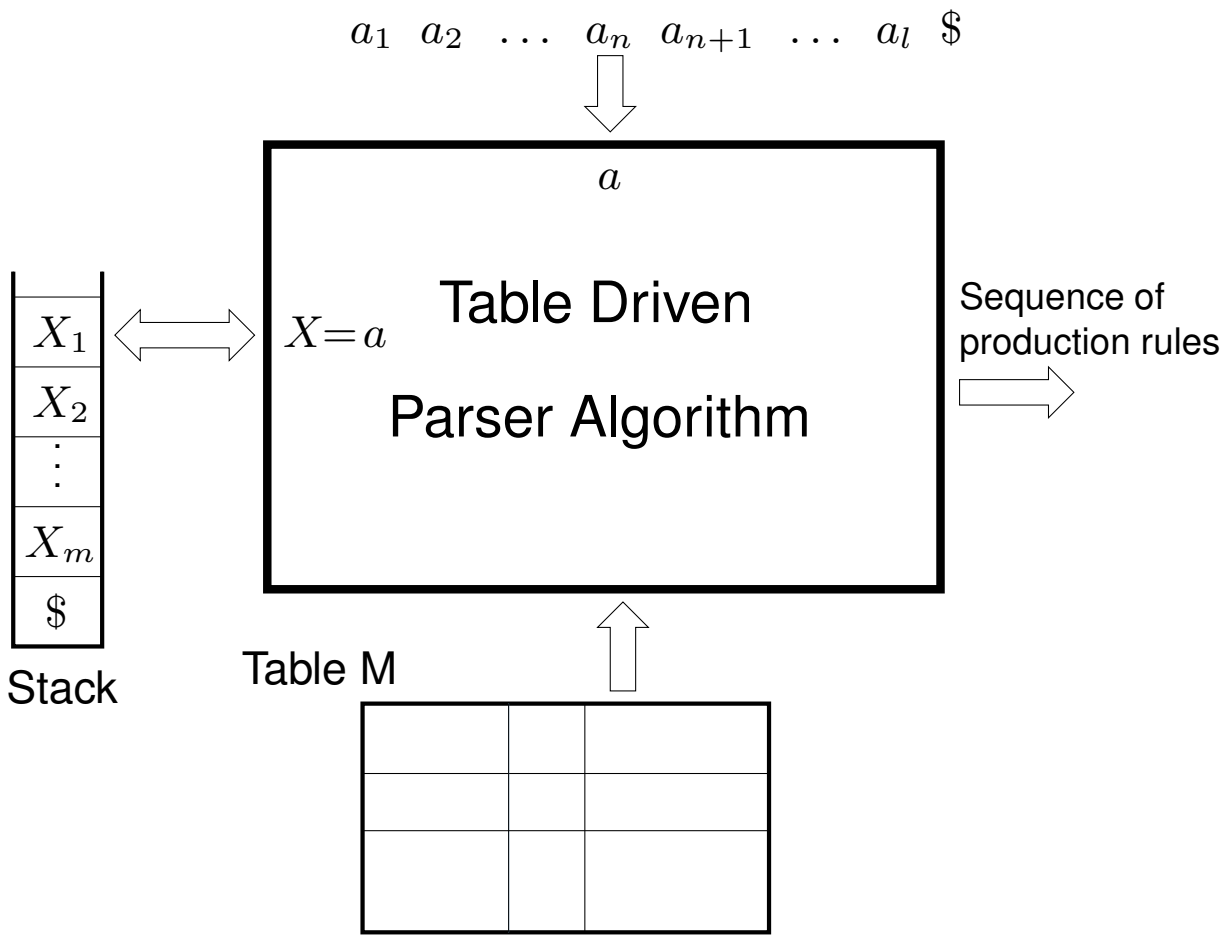


Table-driven Top-down Parser Algorithm

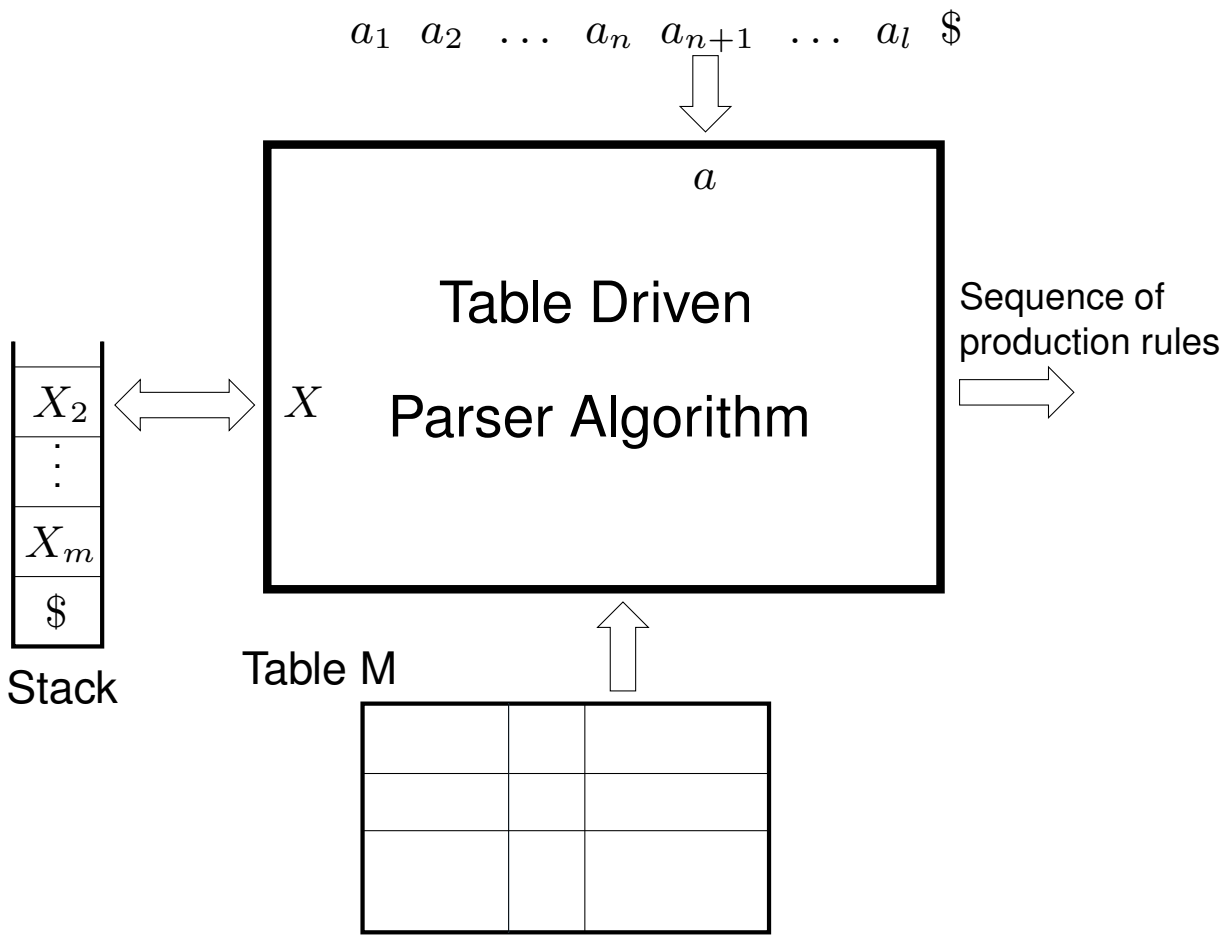


Table-driven Top-down Parser Algorithm

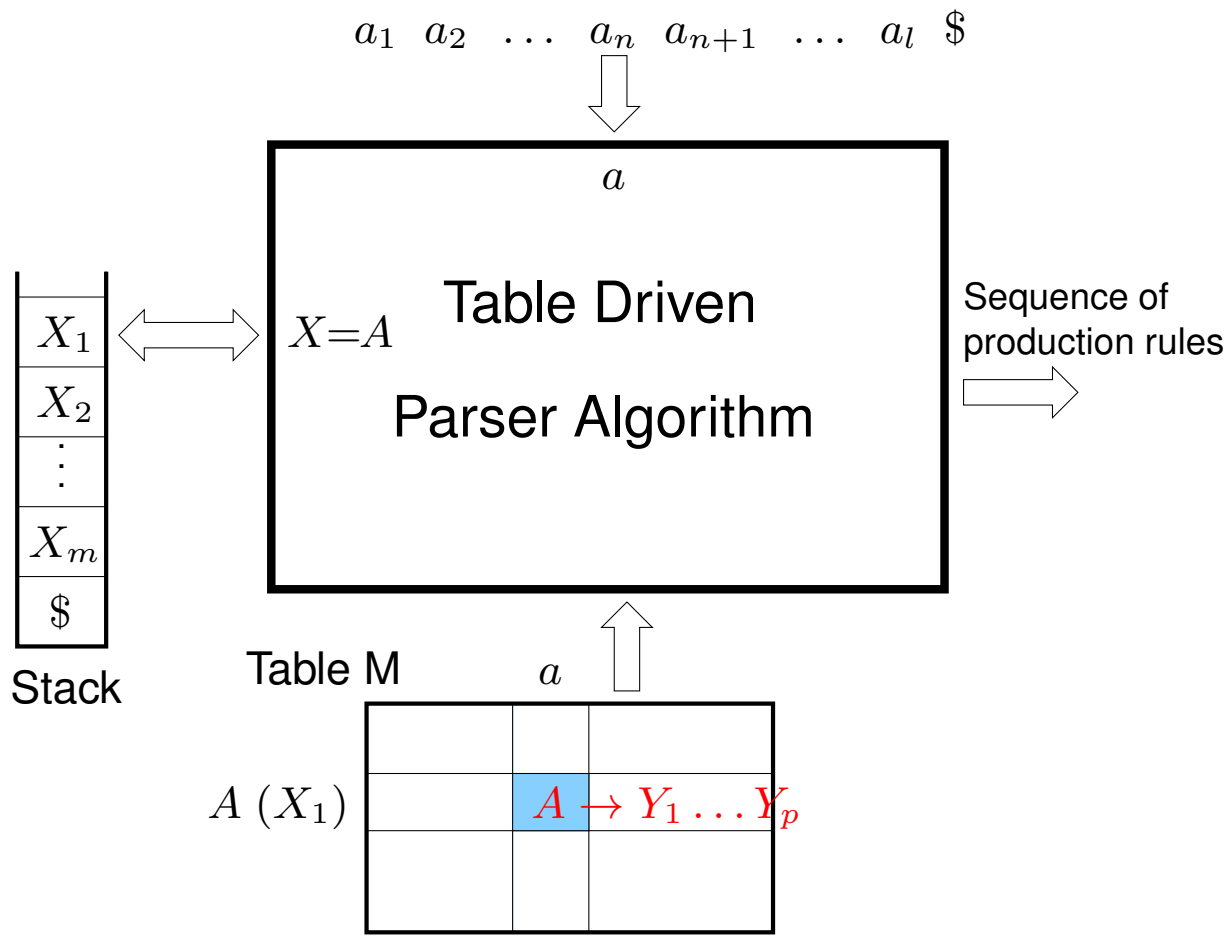


Table-driven Top-down Parser Algorithm

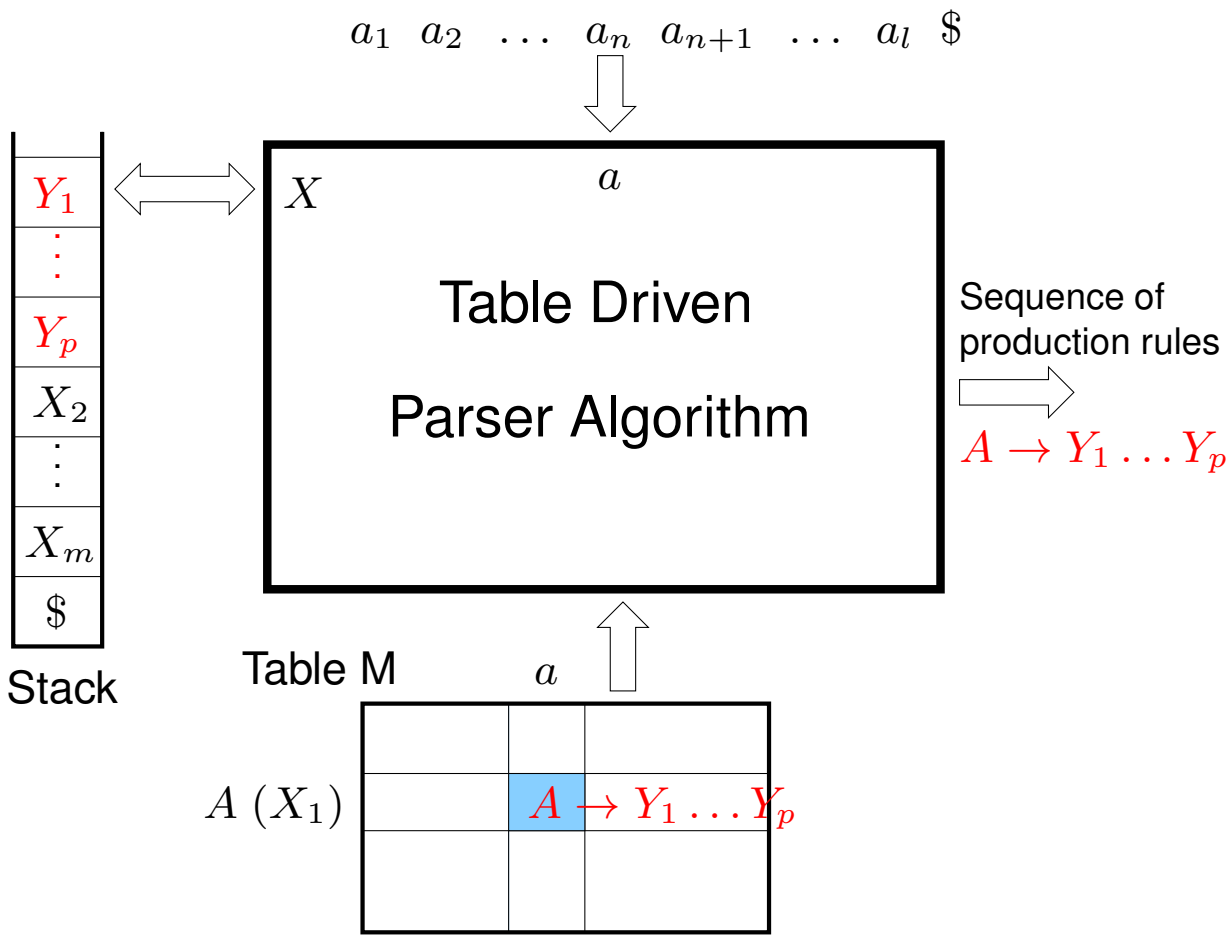


Table-driven Top-down Parser Algorithm

$Stk := EmptyStack(\,);\ PushStack(Stk,\ \$);\ PushStack(Stk,\ S);$

$X := TopStack(Stk);\ \ a := FirstToken(\,);$

while $X \neq \$$ **do**

    if X *is terminal* **then**

        if $X = a$ **then**

            $PopStack(Stk);\ \ a := NextToken(\,);$

        else

            throw syntax error

    else // X *is non−terminal*

        if $M[X, a]$ *is empty* (*is error*) **then**

            throw syntax error

        else // $M[X, a] = X \rightarrow Y_1 \ldots Y_p$

            emit production rule $X \rightarrow Y_1 \ldots Y_p$

            $PopStack(Stk);$ **for** $i := p$ **downto** 1 **do** $PushStack(Stk,\ Y_i);$

    $X := TopStack(Stk);$

endwhile