Syntactic Analysis

Top-down Parsing

Parsing Techniques

Top-down parsers *(LL(1), recursive descent)*

- Start at the root of the parse tree and grow toward leaves
- Pick a production & try to match the input
- Bad "pick" ⇒ may need to backtrack
- Some grammars are backtrack-free *(predictive parsing)*

Bottom-up parsers *(LR(1), operator precedence)*

- Start at the leaves and grow toward root
- As input is consumed, encode possibilities in an internal state
- Start in a state valid for legal first tokens
- Bottom-up parsers handle a large class of grammars

Top-down Parsing

A top-down parser starts with the root of the parse tree
The root node is labeled with the goal symbol of the grammar

Top-down parsing algorithm:

Construct the root node of the parse tree
Repeat until the fringe of the parse tree matches the input string

1. *At a node labeled A, select a production with A on its lhs and, for each symbol on its rhs, construct the appropriate child*
2. *When a terminal symbol is added to the fringe and it doesn't match the fringe, backtrack*
3. *Find the next node to be expanded* *(label ∈ NT)*

- The key is picking the right production in step 1
  - *That choice should be guided by the input string*

Remember the Expression Grammar?

Example CFG:

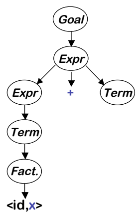
1	*Goal*	→	*Expr*
2	*Expr*	→	*Expr + Term*
3		|	*Expr - Term*
4		|	*Term*
5	*Term*	→	*Term * Factor*
6		|	*Term / Factor*
7		|	*Factor*
8	*Factor*	→	number
9		|	id

And the input x - 2 * y

Example

Let's try x – 2 * y :

Rule	Sentential Form	Input
—	Goal	↑x - 2 * y
1	Expr	↑x - 2 * y
2	Expr + Term	↑x - 2 * y
4	Term + Term	↑x - 2 * y
7	Factor + Term	↑x - 2 * y
9	<id,x> + Term	↑x - 2 * y
9	<id,x> + Term	x ↑- 2 * y

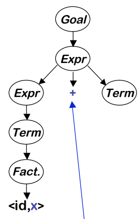


Leftmost derivation, choose productions in an order that exposes problems

Example

Let's try x – 2 * y :

Rule	Sentential Form	Input
—	Goal	↑x - 2 * y
1	Expr	↑x - 2 * y
2	Expr + Term	↑x - 2 * y
4	Term + Term	↑x - 2 * y
7	Factor + Term	↑x - 2 * y
9	<id,x> + Term	↑x - 2 * y
9	<id,x> + Term	x ↑- 2 * y

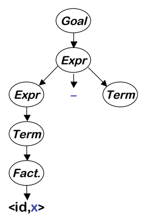


This worked well, except that "–" doesn't match "+"
The parser must backtrack to here

Example

Continuing with x – 2 * y :

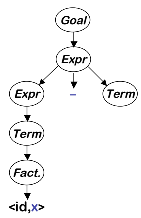
Rule	Sentential Form	Input
—	Goal	↑x – 2 * y
1	Expr	↑x – 2 * y
3	Expr – Term	↑x – 2 * y
4	Term – Term	↑x – 2 * y
7	Factor – Term	↑x – 2 * y
9	<id,x> – Term	↑x – 2 * y
9	<id,x> – Term	x ↑– 2 * y
—	<id,x> – Term	x – ↑2 * y



Example

Continuing with x – 2 * y :

Rule	Sentential Form	Input
—	Goal	↑x – 2 * y
1	Expr	↑x – 2 * y
3	Expr – Term	↑x – 2 * y
4	Term – Term	↑x – 2 * y
7	Factor – Term	↑x – 2 * y
9	<id,x> – Term	↑x – 2 * y
9	<id,x> – Term	x ↑– 2 * y
—	<id,x> – Term	x – ↑2 * y



This time, "–" and "–" matched

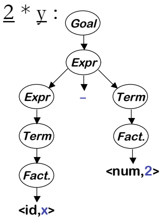
We can advance past "–" to look at "2"

⇒ Now, we need to expand *Term* - the last *NT* on the fringe

Example

Trying to match the “2” in x – 2 * y :

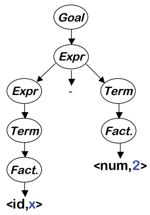
Rule	Sentential Form	Input
—	<id,x> - *Term*	x - ↑2 * y
7	<id,x> - *Factor*	x - ↑2 * y
9	<id,x> - <num,2>	x - ↑2 * y
—	<id,x> - <num,2>	x - 2 ↑* y



Example

Trying to match the “2” in x – 2 * y :

Rule	Sentential Form	Input
—	<id,x> - *Term*	x - ↑2 * y
7	<id,x> - *Factor*	x - ↑2 * y
9	<id,x> - <num,2>	x - ↑2 * y
—	<id,x> - <num,2>	x - 2 ↑* y



Where are we?

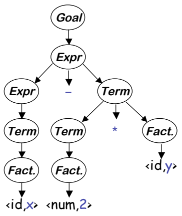
- “2” matches “2”
- We have more input, but no *NTs* left to expand
- The expansion terminated too soon

⇒ Need to backtrack

Example

Trying again with “2” in x – 2 * y :

Rule	Sentential Form	Input
—	<id,x> - *Term*	x - ↑2 * y
5	<id,x> - *Term* * *Factor*	x - ↑2 * y
7	<id,x> - *Factor* * *Factor*	x - ↑2 * y
8	<id,x> - <num,2> * *Factor*	x - ↑2 * y
—	<id,x> - <num,2> * *Factor*	x - 2 ↑* y
—	<id,x> - <num,2> * *Factor*	x - 2 * ↑y
9	<id,x> - <num,2> * <id,y>	x - 2 * ↑y
—	<id,x> - <num,2> * <id,y>	x - 2 * y↑



This time, we matched & consumed all the input

⇒ Success!

Another Possible Parse

Other choices for expansion are possible

consuming no input !

Rule	Sentential Form	Input
—	*Goal*	↑x - 2 * y
1	*Expr*	↑x - 2 * y
2	*Expr* + *Term*	↑x - 2 * y
2	*Expr* + *Term* + *Term*	↑x - 2 * y
2	*Expr* + *Term* + *Term* + *Term*	↑x - 2 * y
2	*Expr* + *Term* + *Term* + …+ *Term*	↑x - 2 * y

This doesn’t terminate *(obviously)*

- Wrong choice of expansion leads to non-termination
- Non-termination is a bad property for a parser to have
- Parser must make the right choice

Left Recursion

Top-down parsers cannot handle left-recursive grammars

Formally,

A grammar is *left recursive* if $\exists\ A \in NT$ such that
$\exists$ a derivation $A \Rightarrow^+ A\alpha$, for some string $\alpha \in (NT \cup T)^+$

Our expression grammar is left recursive

- This can lead to non-termination in a top-down parser
- For a top-down parser, any recursion must be right recursion
- We would like to convert the left recursion to right recursion

Non-termination is a bad property in any part of a compiler

Eliminating Left Recursion

To remove left recursion, we can transform the grammar

Consider a grammar fragment of the form

$Fee \rightarrow Fee\ \alpha$
$\quad\ |\ \beta$

where neither α nor β start with *Fee*

We can rewrite this as

$Fee \rightarrow \beta\ Fie$
$Fie \rightarrow \alpha\ Fie$
$\quad\ |\ \varepsilon$

where *Fie* is a new non-terminal

This accepts the same language, but uses only right recursion

Eliminating Left Recursion

The expression grammar contains two cases of left recursion

Expr	→	*Expr* + *Term*	*Term*	→	*Term* * *Factor*
	|	*Expr* - *Term*		|	*Term* / *Factor*
	|	*Term*		|	*Factor*

Applying the transformation yields

Expr	→	*Term Expr′*	*Term*	→	*Factor Term′*
Expr′	|	+ *Term Expr′*	*Term′*	|	* *Factor Term′*
	|	- *Term Expr′*		|	/ *Factor Term′*
	|	ε		|	ε

These fragments use only right recursion

They retain the original left associativity

Eliminating Left Recursion

Substituting them back into the grammar yields

1	*Goal*	→	*Expr*
2	*Expr*	→	*Term Expr′*
3	*Expr′*	→	+ *Term Expr′*
4		|	- *Term Expr′*
5		|	ε
6	*Term*	→	*Factor Term′*
7	*Term′*	→	* *Factor Term′*
8		|	/ *Factor Term′*
9		|	ε
10	*Factor*	→	number
11		|	id
12		|	(*Expr*)

- This grammar is correct, if somewhat non-intuitive.
- It is left associative, as was the original
- A top-down parser will terminate using it.
- A top-down parser may need to backtrack with it.

Eliminating Left Recursion

The transformation eliminates immediate left recursion

What about more general, indirect left recursion ?

The general algorithm:

arrange the NTs into some order $A_1, A_2, \ldots, A_n$
for $i \leftarrow 1$ *to* n
  for $s \leftarrow 1$ *to* $i - 1$
    replace each production $A_i \rightarrow A_s\gamma$ *with* $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$,
      where $A_s \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ *are all the current productions for* A_s
    eliminate any immediate left recursion on A_i
      using the direct transformation

Must start with 1 to ensure that $A_1 \rightarrow A_1\,\beta$ is transformed

This assumes that the initial grammar has no cycles ($A_i \Rightarrow^+ A_i$), and no epsilon productions

And back

Eliminating Left Recursion

How does this algorithm work?

1. Impose arbitrary order on the non-terminals
2. Outer loop cycles through NT in order
3. Inner loop ensures that a production expanding A_i has no non-terminal A_s in its *rhs*, for $s < i$
4. Last step in outer loop converts any direct recursion on A_i to right recursion using the transformation showed earlier
5. New non-terminals are added at the end of the order & have no left recursion

At the start of the i^{th} outer loop iteration

For all $k < i$, *no production that expands* A_k *contains a non-terminal* A_s *in its rhs, for* $s < k$

Example

- Order of symbols: G, E, T

$G \rightarrow E$
$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow E \sim T$
$T \rightarrow$ id

Example

- Order of symbols: G, E, T

1. $A_i = G$

$G \rightarrow E$
$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow E \sim T$
$T \rightarrow$ id

Example

- Order of symbols: *G, E, T*

1. $A_i = G$	2. $A_i = E$
$G \to E$	$G \to E$
$E \to E + T$	$E \to T E'$
$E \to T$	$E' \to + T E'$
$T \to E \sim T$	$E' \to \varepsilon$
$T \to$ <u>id</u>	$T \to E \sim T$
	$T \to$ <u>id</u>

Example

- Order of symbols: *G, E, T*

1. $A_i = G$	2. $A_i = E$	3. $A_i = T, A_s = E$
$G \to E$	$G \to E$	$G \to E$
$E \to E + T$	$E \to T E'$	$E \to T E'$
$E \to T$	$E' \to + T E'$	$E' \to + T E'$
$T \to E \sim T$	$E' \to \varepsilon$	$E' \to \varepsilon$
$T \to$ <u>id</u>	$T \to E \sim T$	$T \to T E' \sim T$
	$T \to$ <u>id</u>	$T \to$ <u>id</u>

Go to Algorithm

Example

- Order of symbols: *G, E, T*

1. $A_i = G$	2. $A_i = E$	3. $A_i = T, A_s = E$	4. $A_i = T$
$G \to E$	$G \to E$	$G \to E$	$G \to E$
$E \to E + T$	$E \to T E'$	$E \to T E'$	$E \to T E'$
$E \to T$	$E' \to + T E'$	$E' \to + T E'$	$E' \to + T E'$
$T \to E \sim T$	$E' \to \varepsilon$	$E' \to \varepsilon$	$E' \to \varepsilon$
$T \to$ <u>id</u>	$T \to E \sim T$	$T \to T E' \sim T$	$T \to$ <u>id</u> T'
	$T \to$ <u>id</u>	$T \to$ <u>id</u>	$T' \to E' \sim T T'$
			$T' \to \varepsilon$

Roadmap (Where are we?)

We set out to study parsing

- Specifying syntax
  - Context-free grammars ✓
  - Ambiguity ✓
- Top-down parsers
  - Algorithm & its problem with left recursion ✓
  - Left-recursion removal ✓
- Predictive top-down parsing
  - The LL(1) condition
  - Simple recursive descent parsers
  - Table-driven LL(1) parsers

Picking the "Right" Production

If it picks the wrong production, a top-down parser may backtrack

Alternative is to look ahead in input & use context to pick correctly

How much lookahead is needed?

- In general, an arbitrarily large amount
- Use the Cocke-Younger, Kasami algorithm or Earley's algorithm

Fortunately,

- Large subclasses of CFGs can be parsed with limited lookahead
- Most programming language constructs fall in those subclasses

Among the interesting subclasses are *LL(1)* and *LR(1)* grammars

Predictive Parsing

Basic idea

Given $A \rightarrow \alpha \mid \beta$, *the parser should be able to choose between* α *and* β

FIRST Sets

For some *rhs* $\alpha \in G$, define FIRST(α) as the set of tokens that appear as the first symbol in some string that derives from α

That is, $\underline{x} \in$ FIRST(α) *iff* $\alpha \Rightarrow^* \underline{x}\,\gamma$, for some γ

We will defer the problem of how to compute FIRST sets until we look at the *LL(1)* table construction algorithm

Predictive Parsing

Basic idea

Given $A \rightarrow \alpha \mid \beta$, *the parser should be able to choose between* α *and* β

FIRST Sets

For some *rhs* $\alpha \in G$, define FIRST(α) as the set of tokens that appear as the first symbol in some string that derives from α

That is, $\underline{x} \in$ FIRST(α) *iff* $\alpha \Rightarrow^* \underline{x}\,\gamma$, for some γ

The LL(1) Property

If $A \rightarrow \alpha$ and $A \rightarrow \beta$ both appear in the grammar, we would like

$$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \varnothing$$

This would allow the parser to make a correct choice with a lookahead of exactly one symbol !

This is almost correct
See the next slide

Predictive Parsing

What about ε-productions?

$\Rightarrow$ They complicate the definition of LL(1)

If $A \rightarrow \alpha$ and $A \rightarrow \beta$ and $\varepsilon \in$ FIRST(α), then we need to ensure that FIRST(β) is disjoint from FOLLOW(α), too

Define FIRST$^+$(α) as

- FIRST(α) $\cup$ FOLLOW(α), if $\varepsilon \in$ FIRST(α)
- FIRST(α), otherwise

Then, a grammar is *LL(1)* iff $A \rightarrow \alpha$ and $A \rightarrow \beta$ implies

$$\text{FIRST}^+(\alpha) \cap \text{FIRST}^+(\beta) = \varnothing$$

FOLLOW(α) is the set of all words in the grammar that can legally appear immediately after an α

Predictive Parsing

Given a grammar that has the *LL(1)* property

- Can write a simple routine to recognize each *lhs*
- Code is both simple & fast

Consider $A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$, with

$\text{FIRST}^+(\beta_1) \cap \text{FIRST}^+(\beta_2) \cap \text{FIRST}^+(\beta_3) = \varnothing$

```
/* find an A */
if (current_word ∈ FIRST(β1))
    find a β1 and return true
else if (current_word ∈ FIRST(β2))
    find a β2 and return true
else if (current_word ∈ FIRST(β3))
    find a β3 and return true
else
    report an error and return false
```

Grammars with the *LL(1)* property are called *predictive grammars* because the parser can "predict" the correct expansion at each point in the parse.

Parsers that capitalize on the *LL(1)* property are called *predictive parsers*.

One kind of predictive parser is the *recursive descent* parser.

Recursive Descent Parsing

Recall the expression grammar, after transformation

1	*Goal*	→	*Expr*
2	*Expr*	→	*Term Expr'*
3	*Expr'*	→	+ *Term Expr'*
4		|	- *Term Expr'*
5		|	ε
6	*Term*	→	*Factor Term'*
7	*Term'*	→	* *Factor Term'*
8		|	/ *Factor Term'*
9		|	ε
10	*Factor*	→	<u>number</u>
11		|	<u>id</u>

This produces a parser with six *mutually recursive* routines:

- *Goal*
- *Expr*
- *EPrime*
- *Term*
- *TPrime*
- *Factor*

Each recognizes one *NT* or *T*

The term *descent* refers to the direction in which the parse tree is built.

Recursive Descent Parsing (Procedural)

A couple of routines from the expression parser

```
Goal( )
    token ← next_token( );
    if (Expr( ) = true & token = EOF)
        then next compilation step;
        else
            report syntax error;
            return false;

Expr( )
    if (Term( ) = false)
        then return false;
        else return Eprime( );
```

looking for EOF, found token

```
Factor( )
    if (token = Number) then
        token ← next_token( );
        return true;
    else if (token = Identifier) then
        token ← next_token( );
        return true;
    else
        report syntax error;
        return false;
```

EPrime, Term, & TPrime follow the same basic lines

looking for Number or Identifier, found token instead

Recursive Descent Parsing

To build a parse tree:

- Augment parsing routines to build nodes
- Pass nodes between routines using a stack
- Node for each symbol on *rhs*
- Action is to pop *rhs* nodes, make them children of *lhs* node, and push this subtree

To build an abstract syntax tree

- Build fewer nodes
- Put them together in a different order

```
Expr( )
    result ← true;
    if (Term( ) = false)
        then return false;
        else if (EPrime( ) = false)
            then result ← false;
            else
                build an Expr node
                pop EPrime node
                pop Term node
                make EPrime & Term
                    children of Expr
                push Expr node
    return result;
```

Success ⇒ build a piece of the parse tree

Left Factoring

What if my grammar does not have the LL(1) property?

⇒Sometimes, we can transform the grammar

The Algorithm

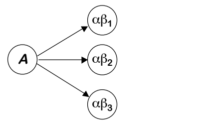
$\forall A \in NT$,
find the longest prefix α that occurs in two or more right-hand sides of A
if $\alpha \neq \varepsilon$ then replace all of the A productions,
$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$,
with
$A \rightarrow \alpha\ Z \mid \gamma$
$Z \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$
where Z is a new element of NT

Repeat until no common prefixes remain

Left Factoring

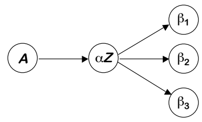
A graphical explanation for the same idea

$A \rightarrow \alpha\beta_1$
$\mid \alpha\beta_2$
$\mid \alpha\beta3$



becomes …

$A \rightarrow \alpha\ Z$
$Z \rightarrow \beta_1$
$\mid \beta_2$
$\mid \beta_n$



Left Factoring (An example)

Consider the following fragment of the expression grammar

Factor → **Identifier**
| **Identifier [*ExprList*]**
| **Identifier (*ExprList*)**

FIRST(rhs_1) = { Identifier }
FIRST(rhs_2) = { Identifier }
FIRST(rhs_3) = { Identifier }

After left factoring, it becomes

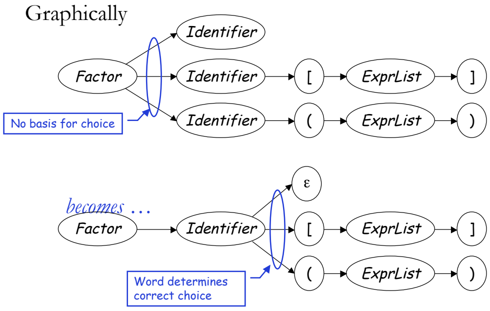
Factor → **Identifier *Arguments***
Arguments → **[*ExprList*]**
| **(*ExprList*)**
| ε

FIRST(rhs_1) = { Identifier }
FIRST(rhs_2) = { [}
FIRST(rhs_3) = { (}
FIRST(rhs_4) = FOLLOW(*Factor*)
⇒ It has the *LL(1)* property

This form has the same syntax, with the *LL(1)* property

Left Factoring

Graphically



Left Factoring (Generality)

Question

By *eliminating left recursion* and *left factoring*, can we transform an arbitrary CFG to a form where it meets the *LL(1)* condition? (and can be parsed predictively with a single token lookahead?)

Answer

Given a CFG that doesn't meet the *LL(1)* condition, it is undecidable whether or not an equivalent *LL(1)* grammar exists.

Example

$\{a^n\, 0\, b^n \mid n \geq 1\} \cup \{a^n\, 1\, b^{2n} \mid n \geq 1\}$ has no *LL(1)* grammar

Recursive Descent (Summary)

1. Build FIRST (and FOLLOW) sets
2. Massage grammar to have *LL(1)* condition
   a. Remove left recursion
   b. Left factor it
3. Define a procedure for each non-terminal
   a. Implement a case for each right-hand side
   b. Call procedures as needed for non-terminals
4. Add extra code, as needed
   a. Perform context-sensitive checking
   b. Build an IR to record the code

Can we automate this process?

FIRST and FOLLOW Sets

FIRST(α)

For some $\alpha \in T \cup NT$, define FIRST(α) as the set of tokens that appear as the first symbol in some string that derives from α

That is, $\underline{x} \in$ FIRST(α) *iff* $\alpha \Rightarrow^* \underline{x}\,\gamma$, for some γ

FOLLOW(α)

For some $\alpha \in NT$, define FOLLOW(α) as the set of symbols that can occur immediately after α in a valid sentence.

FOLLOW(S) = {EOF}, where S is the start symbol

To build FIRST sets, we need FOLLOW sets …

Computing FIRST Sets

Define FIRST as

- If $\alpha \Rightarrow^* \underline{a}\beta$, $\underline{a} \in T$, $\beta \in (T \cup NT)^*$, then $\underline{a} \in$ FIRST(α)
- If $\alpha \Rightarrow^* \varepsilon$, then $\varepsilon \in$ FIRST(α)
- If $\alpha \Rightarrow \beta_1\beta_2 \ldots \beta_k$ then $\underline{a} \in$ FIRST(α) if form some i a $\in$ FIRST(β_i) and $\varepsilon \in$ FIRST(β_1), …, FIRST(β_{i-1})

Note: if $\alpha = X\beta$, FIRST(α) = FIRST(X)

To compute FIRST

- Use a fixed-point method
- FIRST(A) $\in 2^{(T \cup \varepsilon)}$
- Loop is monotonic

$\Rightarrow$ Algorithm halts

Computing FIRST Sets

```
for each x ∈ T, FIRST(x) ← { x }
for each A ∈ NT, FIRST(A) ← Ø

while (FIRST sets are still changing)
    for each p ∈ P, of the form A→β,
        if β  is ε then
            FIRST(A) ← FIRST(A) ∪ { ε }
        else if β  is B_1B_2…B_k then begin
            FIRST(A) ← FIRST(A) ∪ ( FIRST(B_1) – { ε } )
            for i ← 1 to k–1 by 1 while ε ∈ FIRST(B_i )
                FIRST(A) ← FIRST(A) ∪ ( FIRST(B_{i +1}) – { ε } )
            if  i = k–1 and ε ∈ FIRST(B_k )
                then FIRST(A) ← FIRST(A) ∪ { ε }
            end
for each A ∈ NT
    if ε ∈ FIRST(A) then
        FIRST(A) ← FIRST(A) ∪ FOLLOW(A)
```

Computing FOLLOW Sets

Define FOLLOW as

- Place \$ in FOLLOW(*S*) where *S* is the start symbol
- If $A \rightarrow \alpha B \beta$ then any $(a/\varepsilon) \in$ FIRST(β) is in FOLLOW(A)
- If $A \rightarrow \alpha B$ or $A \rightarrow \alpha B \beta$ where $\varepsilon \in$ FIRST(β), then everything in FOLLOW(A) is in FOLLOW(B).

To compute FOLLOW

- Use a fixed-point method
- FOLLOW(*A*) $\in 2^{(T \cup \varepsilon)}$
- Loop is monotonic

⇒ Algorithm halts

Computing FOLLOW Sets

```
FOLLOW(S) ← {$ }
for each A ∈ NT, FOLLOW(A) ← Ø

while (FOLLOW sets are still changing)
    for each p ∈ P, of the form A→β_1β_2 … β_k
        FOLLOW(β_k) ← FOLLOW(β_k) ∪ FOLLOW(A)
        TRAILER ← FOLLOW(A)
        for i ← k down to 2
            if ε ∈ FIRST(β_i ) then
                FOLLOW(β_{i-1} ) ← FOLLOW(β_{i-1}) ∪ { FIRST(β_i ) – { ε }}
                                       ∪ TRAILER
            else
                FOLLOW(β_{i-1} ) ← FOLLOW(β_{i-1}) ∪ FIRST(β_i )
                TRAILER ← Ø
```

Building Top-down Parsers

Given an *LL(1)* grammar, and its FIRST & FOLLOW sets …

- Emit a routine for each non-terminal
  - Nest of if-then-else statements to check alternate rhs's
  - Each returns true on success and throws an error on false
  - Simple, working (, *perhaps ugly*,) code
- This automatically constructs a recursive-descent parser

I don't know of a system that does this …

Improving matters

- Nest of if-then-else statements may be slow
  - Good case statement implementation would be better
- What about a table to encode the options?
  - Interpret the table with a skeleton, as we did in scanning

Example: First and Follow Sets

$E \rightarrow T E'$
$E' \rightarrow +T E' \mid \varepsilon$
$T \rightarrow F T'$
$T' \rightarrow *F T' \mid \varepsilon$
$F \rightarrow (E) \mid \textbf{id}$

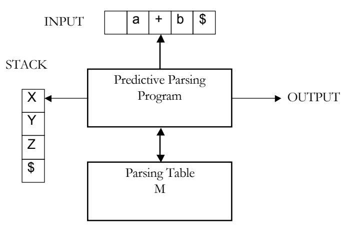
First(F) = { (, **id** } $\Rightarrow$ First(T) = First(E) = { (, **id** }
First(E') = { + , ε }
First(T') = { * , ε }

Follow(E) = { \$ } but since $F \rightarrow (E)$ then Follow(E) = {) , \$ }
Follow(E') = {) , \$ }
Follow(T) = Follow(T') = { + ,), \$ } because $E' \Rightarrow \varepsilon$
Follow(F) = { *, +,), \$ } because $T' \Rightarrow \varepsilon$

Table-driven Top-down Parsers

Non-Terminal	Input Symbol					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow$ id			$F \rightarrow (E)$		

Table-driven Top-down Parsers



Strategy

- Encode knowledge in a table
- Use standard "skeleton" parser to interpret the table

Building Top-down Parsers

Building the complete table

- Need a row for every *NT* & a column for every *T*
- Need a table-driven interpreter for the table
- Algorithm:
  - consider X the symbol on top of the symbol stack (TOS) and the current input symbol a
  - This tuple (X,a) determines the action as follows:
    - If X = a = \$ the parser halts and announces success
    - If X = a ≠ \$ the parser pops X off the stack and advances the input
    - If X is non-terminal, consults entry M[X,a] of parsing table M. If not an error entry, and is a production i.e., M[X,a] = { X → UVW } then replace X with WVU (reverse production RHS). If error invoke error recovery routine.

LL(1) Skeleton Parser

```
token ← next_token()
push EOF onto Stack
push the start symbol, S, onto Stack
TOS ← top of Stack
loop forever
  if TOS = EOF and token = EOF then
     break & report success
  else if TOS is a terminal then
     if TOS matches token then
        pop Stack                      // recognized TOS
        token ← next_token()
     else report error looking for TOS
  else                                 // TOS is a non-terminal
     if TABLE[TOS,token] is A→ B1B2…Bk then
        pop Stack                      // get rid of A
        push Bk, Bk-1, …, B1           // in that order
     else report error expanding TOS
  TOS ← top of Stack
```

exit on success

Building Top-down Parsers

Building the complete table

- Need a row for every *NT* & a column for every *T*
- Need an algorithm to build the table

Filling in M[X,y], X ∈ *NT*, y ∈ *T*

1. Entry is the rule $X \rightarrow \beta$, if y ∈ FIRST(β)
2. Entry is the rule $X \rightarrow \varepsilon$ if y ∈ FOLLOW(X) and $X \rightarrow \varepsilon \in G$
3. Entry is **error** if neither 1 nor 2 define it

If any entry is defined multiple times, G is not *LL(1)*

This is the *LL(1)* table construction algorithm

Table-driven Top-down Parsers

Non-Terminal	Input Symbol					
	id	**+**	*****	**(**	**)**	**$**
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow$ **id**			$F \rightarrow (E)$		

Table-driven Top-down Parsers

STACK	INPUT	OUTPUT
$E	id + id * id$	

```
token ← nextToken()
push EOF onto Stack
push the start symbol, S, onto Stack
TOS ← top of Stack
loop forever
  if TOS = EOF and token = EOF then
     break & report success
  else if TOS is a terminal then
     if TOS matches token then
        pop Stack                      // recognized TOS
        token ← nextToken()
     else report error looking for TOS
  else                                 // TOS is a non-terminal
     if M[TOS,token] is A→ B1B2…Bk then
        pop Stack                      // get rid of A
        push Bk, Bk-1, …, B1           // in that order
     else report error expanding TOS
  TOS ← top of Stack
```

Non-Terminal	Input Symbol					
	id	**+**	*****	**(**	**)**	**$**
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow$ id			$F \rightarrow (E)$		

M[X,a]

Table-driven Top-down Parsers

STACK	INPUT	OUTPUT
$\$E$	id + id * id$	
$\$E'T$	id + id * id$	$E \to TE'$

```
token ← nextToken()
push EOF onto Stack
push the start symbol, S, onto Stack
TOS ← top of Stack
loop forever
  if TOS = EOF and token = EOF then
    break & report success
  else if TOS is a terminal then
    if TOS matches token then
      pop Stack                // recognized TOS
      token ← nextToken()
    else report error looking for TOS
  else                         // TOS is a non-terminal
    if M[TOS,token] is A→ B1B2…Bk then
      pop Stack                // get rid of A
      push Bk, Bk-1, …, B1     // in that order
    else report error expanding TOS
  TOS ← top of Stack
```

Non-Terminal	Input Symbol: id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to \text{id}$			$F \to (E)$		

M[X,a]

Table-driven Top-down Parsers

STACK	INPUT	OUTPUT
$\$E$	id + id * id$	
$\$E'T$	id + id * id$	$E \to TE'$
$\$E'T'F$	id + id * id$	$T \to FT'$

```
token ← nextToken()
push EOF onto Stack
push the start symbol, S, onto Stack
TOS ← top of Stack
loop forever
  if TOS = EOF and token = EOF then
    break & report success
  else if TOS is a terminal then
    if TOS matches token then
      pop Stack                // recognized TOS
      token ← nextToken()
    else report error looking for TOS
  else                         // TOS is a non-terminal
    if M[TOS,token] is A→ B1B2…Bk then
      pop Stack                // get rid of A
      push Bk, Bk-1, …, B1     // in that order
    else report error expanding TOS
  TOS ← top of Stack
```

Non-Terminal	Input Symbol: id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to \text{id}$			$F \to (E)$		

M[X,a]

Table-driven Top-down Parsers

STACK	INPUT	OUTPUT
$\$E$	id + id * id$	
$\$E'T$	id + id * id$	$E \to TE'$
$\$E'T'F$	id + id * id$	$T \to FT'$
$\$E'T'\text{id}$	id + id * id$	$F \to \text{id}$

```
token ← nextToken()
push EOF onto Stack
push the start symbol, S, onto Stack
TOS ← top of Stack
loop forever
  if TOS = EOF and token = EOF then
    break & report success
  else if TOS is a terminal then
    if TOS matches token then
      pop Stack                // recognized TOS
      token ← nextToken()
    else report error looking for TOS
  else                         // TOS is a non-terminal
    if M[TOS,token] is A→ B1B2…Bk then
      pop Stack                // get rid of A
      push Bk, Bk-1, …, B1     // in that order
    else report error expanding TOS
  TOS ← top of Stack
```

Non-Terminal	Input Symbol: id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to \text{id}$			$F \to (E)$		

M[X,a]

Table-driven Top-down Parsers

STACK	INPUT	OUTPUT
$\$E$	id + id * id$	
$\$E'T$	id + id * id$	$E \to TE'$
$\$E'T'F$	id + id * id$	$T \to FT'$
$\$E'T'\text{id}$	id + id * id$	$F \to \text{id}$
$\$E'T'$	+ id * id$	

```
token ← nextToken()
push EOF onto Stack
push the start symbol, S, onto Stack
TOS ← top of Stack
loop forever
  if TOS = EOF and token = EOF then
    break & report success
  else if TOS is a terminal then
    if TOS matches token then
      pop Stack                // recognized TOS
      token ← nextToken()
    else report error looking for TOS
  else                         // TOS is a non-terminal
    if M[TOS,token] is A→ B1B2…Bk then
      pop Stack                // get rid of A
      push Bk, Bk-1, …, B1     // in that order
    else report error expanding TOS
  TOS ← top of Stack
```

Non-Terminal	Input Symbol: id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to \text{id}$			$F \to (E)$		

M[X,a]

Table-driven Top-down Parsers

STACK	INPUT	OUTPUT
$E	id + id * id$	
$E'T	id + id * id$	E → TE'
$E'T'F	id + id * id$	T → FT'
$E'T'id	id + id * id$	F → id
$E'T'	+ id * id$	
$E'	+ id * id$	T' → ε

```
token ← nextToken()
push EOF onto Stack
push the start symbol, S, onto Stack
TOS ← top of Stack
loop forever
  if TOS = EOF and token = EOF then
     break & report success
   else if TOS is a terminal then
     if TOS matches token then
        pop Stack                  // recognized TOS
        token ← nextToken()
     else report error looking for TOS
   else                            // TOS is a non-terminal
     if M[TOS,token] is A→ B1B2…Bk then
        pop Stack                  // get rid of A
        push Bk, Bk-1, …, B1       // in that order
     else report error expanding TOS
  TOS ← top of Stack
```

Non-Terminal	Input Symbol: id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

M[X,a]

Table-driven Top-down Parsers

STACK	INPUT	OUTPUT
$E	id + id * id$	
$E'T	id + id * id$	E → TE'
$E'T'F	id + id * id$	T → FT'
$E'T'id	id + id * id$	F → id
$E'T'	+ id * id$	
$E'	+ id * id$	T' → ε
$E'T+	+ id * id$	E' → +TE'

```
token ← nextToken()
push EOF onto Stack
push the start symbol, S, onto Stack
TOS ← top of Stack
loop forever
  if TOS = EOF and token = EOF then
     break & report success
   else if TOS is a terminal then
     if TOS matches token then
        pop Stack                  // recognized TOS
        token ← nextToken()
     else report error looking for TOS
   else                            // TOS is a non-terminal
     if M[TOS,token] is A→ B1B2…Bk then
        pop Stack                  // get rid of A
        push Bk, Bk-1, …, B1       // in that order
     else report error expanding TOS
  TOS ← top of Stack
```

Non-Terminal	Input Symbol: id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

M[X,a]

Table-driven Top-down Parsers

STACK	INPUT	OUTPUT
$E	id + id * id$	
$E'T	id + id * id$	E → TE'
$E'T'F	id + id * id$	T → FT'
$E'T'id	id + id * id$	F → id
$E'T'	+ id * id$	
$E'	+ id * id$	T' → ε
$E'T+	+ id * id$	E' → +TE'
$E'T	id * id$	

```
token ← nextToken()
push EOF onto Stack
push the start symbol, S, onto Stack
TOS ← top of Stack
loop forever
  if TOS = EOF and token = EOF then
     break & report success
   else if TOS is a terminal then
     if TOS matches token then
        pop Stack                  // recognized TOS
        token ← nextToken()
     else report error looking for TOS
   else                            // TOS is a non-terminal
     if M[TOS,token] is A→ B1B2…Bk then
        pop Stack                  // get rid of A
        push Bk, Bk-1, …, B1       // in that order
     else report error expanding TOS
  TOS ← top of Stack
```

Non-Terminal	Input Symbol: id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

M[X,a]

Table-driven Top-down Parsers

STACK	INPUT	OUTPUT
$E	id + id * id$	
$E'T	id + id * id$	E → TE'
$E'T'F	id + id * id$	T → FT'
$E'T'id	id + id * id$	F → id
$E'T'	+ id * id$	
$E'	+ id * id$	T' → ε
$E'T+	+ id * id$	E' → +TE'
$E'T	id * id$	
$E'T'F	id * id$	T → FT'

```
token ← nextToken()
push EOF onto Stack
push the start symbol, S, onto Stack
TOS ← top of Stack
loop forever
  if TOS = EOF and token = EOF then
     break & report success
   else if TOS is a terminal then
     if TOS matches token then
        pop Stack                  // recognized TOS
        token ← nextToken()
     else report error looking for TOS
   else                            // TOS is a non-terminal
     if M[TOS,token] is A→ B1B2…Bk then
        pop Stack                  // get rid of A
        push Bk, Bk-1, …, B1       // in that order
     else report error expanding TOS
  TOS ← top of Stack
```

Non-Terminal	Input Symbol: id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

M[X,a]

Table-driven Top-down Parsers

STACK	INPUT	OUTPUT
$\$E$	$\text{id} + \text{id} * \text{id}\$$	
$\$E'T$	$\text{id} + \text{id} * \text{id}\$$	$E \rightarrow TE'$
$\$E'T'F$	$\text{id} + \text{id} * \text{id}\$$	$T \rightarrow FT'$
$\$E'T'\text{id}$	$\text{id} + \text{id} * \text{id}\$$	$F \rightarrow \text{id}$
$\$E'T'$	$+ \text{id} * \text{id}\$$	
$\$E'$	$+ \text{id} * \text{id}\$$	$T' \rightarrow \varepsilon$
$\$E'T+$	$+ \text{id} * \text{id}\$$	$E' \rightarrow +TE'$
$\$E'T$	$\text{id} * \text{id}\$$	
$\$E'T'F$	$\text{id} * \text{id}\$$	$T \rightarrow FT'$
$\$E'T'\text{id}$	$\text{id} * \text{id}\$$	$F \rightarrow \text{id}$

```
token ← nextToken()
push EOF onto Stack
push the start symbol, S, onto Stack
TOS ← top of Stack
loop forever
  if TOS = EOF and token = EOF then
    break & report success
  else if TOS is a terminal then
    if TOS matches token then
      pop Stack                  // recognized TOS
      token ← nextToken()
    else report error looking for TOS
  else                           // TOS is a non-terminal
    if M[TOS,token] is A→ B1B2…Bk then
      pop Stack                  // get rid of A
      push Bk, Bk-1, …, B1       // in that order
    else report error expanding TOS
  TOS ← top of Stack
```

Non-Terminal	Input Symbol: id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

M[X,a]

Table-driven Top-down Parsers

STACK	INPUT	OUTPUT
$\$E$	$\text{id} + \text{id} * \text{id}\$$	
$\$E'T$	$\text{id} + \text{id} * \text{id}\$$	$E \rightarrow TE'$
$\$E'T'F$	$\text{id} + \text{id} * \text{id}\$$	$T \rightarrow FT'$
$\$E'T'\text{id}$	$\text{id} + \text{id} * \text{id}\$$	$F \rightarrow \text{id}$
$\$E'T'$	$+ \text{id} * \text{id}\$$	
$\$E'$	$+ \text{id} * \text{id}\$$	$T' \rightarrow \varepsilon$
$\$E'T+$	$+ \text{id} * \text{id}\$$	$E' \rightarrow +TE'$
$\$E'T$	$\text{id} * \text{id}\$$	
$\$E'T'F$	$\text{id} * \text{id}\$$	$T \rightarrow FT'$
$\$E'T'\text{id}$	$\text{id} * \text{id}\$$	$F \rightarrow \text{id}$
$\$E'T'$	$* \text{id}\$$	

```
token ← nextToken()
push EOF onto Stack
push the start symbol, S, onto Stack
TOS ← top of Stack
loop forever
  if TOS = EOF and token = EOF then
    break & report success
  else if TOS is a terminal then
    if TOS matches token then
      pop Stack                  // recognized TOS
      token ← nextToken()
    else report error looking for TOS
  else                           // TOS is a non-terminal
    if M[TOS,token] is A→ B1B2…Bk then
      pop Stack                  // get rid of A
      push Bk, Bk-1, …, B1       // in that order
    else report error expanding TOS
  TOS ← top of Stack
```

Non-Terminal	Input Symbol: id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

M[X,a]

Table-driven Top-down Parsers

STACK	INPUT	OUTPUT
$\$E$	$\text{id} + \text{id} * \text{id}\$$	
$\$E'T$	$\text{id} + \text{id} * \text{id}\$$	$E \rightarrow TE'$
$\$E'T'F$	$\text{id} + \text{id} * \text{id}\$$	$T \rightarrow FT'$
$\$E'T'\text{id}$	$\text{id} + \text{id} * \text{id}\$$	$F \rightarrow \text{id}$
$\$E'T'$	$+ \text{id} * \text{id}\$$	
$\$E'$	$+ \text{id} * \text{id}\$$	$T' \rightarrow \varepsilon$
$\$E'T+$	$+ \text{id} * \text{id}\$$	$E' \rightarrow +TE'$
$\$E'T$	$\text{id} * \text{id}\$$	
$\$E'T'F$	$\text{id} * \text{id}\$$	$T \rightarrow FT'$
$\$E'T'\text{id}$	$\text{id} * \text{id}\$$	$F \rightarrow \text{id}$
$\$E'T'$	$* \text{id}\$$	
$\$E'T'F*$	$* \text{id}\$$	$T' \rightarrow *FT'$

```
token ← nextToken()
push EOF onto Stack
push the start symbol, S, onto Stack
TOS ← top of Stack
loop forever
  if TOS = EOF and token = EOF then
    break & report success
  else if TOS is a terminal then
    if TOS matches token then
      pop Stack                  // recognized TOS
      token ← nextToken()
    else report error looking for TOS
  else                           // TOS is a non-terminal
    if M[TOS,token] is A→ B1B2…Bk then
      pop Stack                  // get rid of A
      push Bk, Bk-1, …, B1       // in that order
    else report error expanding TOS
  TOS ← top of Stack
```

Non-Terminal	Input Symbol: id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

M[X,a]

Table-driven Top-down Parsers

STACK	INPUT	OUTPUT
$\$E$	$\text{id} + \text{id} * \text{id}\$$	
$\$E'T$	$\text{id} + \text{id} * \text{id}\$$	$E \rightarrow TE'$
$\$E'T'F$	$\text{id} + \text{id} * \text{id}\$$	$T \rightarrow FT'$
$\$E'T'\text{id}$	$\text{id} + \text{id} * \text{id}\$$	$F \rightarrow \text{id}$
$\$E'T'$	$+ \text{id} * \text{id}\$$	
$\$E'$	$+ \text{id} * \text{id}\$$	$T' \rightarrow \varepsilon$
$\$E'T+$	$+ \text{id} * \text{id}\$$	$E' \rightarrow +TE'$
$\$E'T$	$\text{id} * \text{id}\$$	
$\$E'T'F$	$\text{id} * \text{id}\$$	$T \rightarrow FT'$
$\$E'T'\text{id}$	$\text{id} * \text{id}\$$	$F \rightarrow \text{id}$
$\$E'T'$	$* \text{id}\$$	
$\$E'T'F*$	$* \text{id}\$$	$T' \rightarrow *FT'$
$\$E'T'F$	$\text{id}\$$	

```
token ← nextToken()
push EOF onto Stack
push the start symbol, S, onto Stack
TOS ← top of Stack
loop forever
  if TOS = EOF and token = EOF then
    break & report success
  else if TOS is a terminal then
    if TOS matches token then
      pop Stack                  // recognized TOS
      token ← nextToken()
    else report error looking for TOS
  else                           // TOS is a non-terminal
    if M[TOS,token] is A→ B1B2…Bk then
      pop Stack                  // get rid of A
      push Bk, Bk-1, …, B1       // in that order
    else report error expanding TOS
  TOS ← top of Stack
```

Non-Terminal	Input Symbol: id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

M[X,a]

Table-driven Top-down Parsers

STACK	INPUT	OUTPUT
$E	id + id * id$	
$E'T	id + id * id$	E → TE'
$E'T'F	id + id * id$	T → FT'
$E'T'id	id + id * id$	F → id
$E'T'	+ id * id$	
$E'	+ id * id$	T' → ε
$E'T+	+ id * id$	E' → +TE'
$E'T	id * id$	
$E'T'F	id * id$	T → FT'
$E'T'id	id * id$	F → id
$E'T'	* id$	
$E'T'F*	* id$	T' → *FT'
$E'T'F	id$	
$E'T'id	id$	F → id

```
token ← nextToken()
push EOF onto Stack
push the start symbol, S, onto Stack
TOS ← top of Stack
loop forever
  if TOS = EOF and token = EOF then
     break & report success
   else if TOS is a terminal then
     if TOS matches token then
        pop Stack                 // recognized TOS
        token ← nextToken()
     else report error looking for TOS
   else                           // TOS is a non-terminal
     if M[TOS,token] is A→ B1B2…Bk then
        pop Stack                 // get rid of A
        push Bk, Bk-1, …, B1      // in that order
     else report error expanding TOS
  TOS ← top of Stack
```

Non-Terminal	Input Symbol: id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

M[X,a]

Table-driven Top-down Parsers

STACK	INPUT	OUTPUT
$E	id + id * id$	
$E'T	id + id * id$	E → TE'
$E'T'F	id + id * id$	T → FT'
$E'T'id	id + id * id$	F → id
$E'T'	+ id * id$	
$E'	+ id * id$	T' → ε
$E'T+	+ id * id$	E' → +TE'
$E'T	id * id$	
$E'T'F	id * id$	T → FT'
$E'T'id	id * id$	F → id
$E'T'	* id$	
$E'T'F*	* id$	T' → *FT'
$E'T'F	id$	
$E'T'id	id$	F → id
$E'T'	$	

```
token ← nextToken()
push EOF onto Stack
push the start symbol, S, onto Stack
TOS ← top of Stack
loop forever
  if TOS = EOF and token = EOF then
     break & report success
   else if TOS is a terminal then
     if TOS matches token then
        pop Stack                 // recognized TOS
        token ← nextToken()
     else report error looking for TOS
   else                           // TOS is a non-terminal
     if M[TOS,token] is A→ B1B2…Bk then
        pop Stack                 // get rid of A
        push Bk, Bk-1, …, B1      // in that order
     else report error expanding TOS
  TOS ← top of Stack
```

Non-Terminal	Input Symbol: id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

M[X,a]

Table-driven Top-down Parsers

STACK	INPUT	OUTPUT
$E	id + id * id$	
$E'T	id + id * id$	E → TE'
$E'T'F	id + id * id$	T → FT'
$E'T'id	id + id * id$	F → id
$E'T'	+ id * id$	
$E'	+ id * id$	T' → ε
$E'T+	+ id * id$	E' → +TE'
$E'T	id * id$	
$E'T'F	id * id$	T → FT'
$E'T'id	id * id$	F → id
$E'T'	* id$	
$E'T'F*	* id$	T' → *FT'
$E'T'F	id$	
$E'T'id	id$	F → id
$E'T'	$	
$E'	$	T' → ε

```
token ← nextToken()
push EOF onto Stack
push the start symbol, S, onto Stack
TOS ← top of Stack
loop forever
  if TOS = EOF and token = EOF then
     break & report success
   else if TOS is a terminal then
     if TOS matches token then
        pop Stack                 // recognized TOS
        token ← nextToken()
     else report error looking for TOS
   else                           // TOS is a non-terminal
     if M[TOS,token] is A→ B1B2…Bk then
        pop Stack                 // get rid of A
        push Bk, Bk-1, …, B1      // in that order
     else report error expanding TOS
  TOS ← top of Stack
```

Non-Terminal	Input Symbol: id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

M[X,a]

Table-driven Top-down Parsers

STACK	INPUT	OUTPUT
$E	id + id * id$	
$E'T	id + id * id$	E → TE'
$E'T'F	id + id * id$	T → FT'
$E'T'id	id + id * id$	F → id
$E'T'	+ id * id$	
$E'	+ id * id$	T' → ε
$E'T+	+ id * id$	E' → +TE'
$E'T	id * id$	
$E'T'F	id * id$	T → FT'
$E'T'id	id * id$	F → id
$E'T'	* id$	
$E'T'F*	* id$	T' → *FT'
$E'T'F	id$	
$E'T'id	id$	F → id
$E'T'	$	
$E'	$	T' → ε
$	$	E' → ε

```
token ← nextToken()
push EOF onto Stack
push the start symbol, S, onto Stack
TOS ← top of Stack
loop forever
  if TOS = EOF and token = EOF then
     break & report success
   else if TOS is a terminal then
     if TOS matches token then
        pop Stack                 // recognized TOS
        token ← nextToken()
     else report error looking for TOS
   else                           // TOS is a non-terminal
     if M[TOS,token] is A→ B1B2…Bk then
        pop Stack                 // get rid of A
        push Bk, Bk-1, …, B1      // in that order
     else report error expanding TOS
  TOS ← top of Stack
```

Non-Terminal	Input Symbol: id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

M[X,a]

Table-driven Top-down Parsers

STACK	INPUT	OUTPUT
$E	id + id * id$	
$E'T	id + id * id$	E → TE'
$E'T'F	id + id * id$	T → FT'
$E'T'id	id + id * id$	F → id
$E'T'	+ id * id$	
$E'	+ id * id$	T' → ε
$E'T +	+ id * id$	E' → + TE'
$E'T	id * id$	
$E'T'F	id * id$	T → FT'
$E'T'id	id * id$	F → id
$E'T'	* id$	
$E'T'F *	* id$	T' → * FT'
$E'T'F	id$	
$E'T'id	id$	F → id
$E'T'	$	
$E'	$	T' → ε
$	$	E' → ε

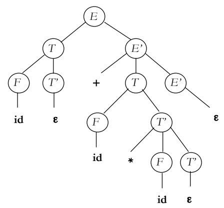
```
token ← nextToken()
push EOF onto Stack
push the start symbol, S, onto Stack
TOS ← top of Stack
loop forever
  if TOS = EOF and token = EOF then
     break & report success
   else if TOS is a terminal then
     if TOS matches token then
        pop Stack                    // recognized TOS
        token ← nextToken()
     else report error looking for TOS
   else                              // TOS is a non-terminal
     if M[TOS,token] is A→ B1B2…Bk then
        pop Stack                    // get rid of A
        push Bk, Bk-1, …, B1   // in that order
     else report error expanding TOS
  TOS ← top of Stack
```

Non-Terminal	Input Symbol					
	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

M[X,a]

Table-driven Top-down Parsers

STACK	INPUT	OUTPUT
$E	id + id * id$	
$E'T	id + id * id$	E → TE'
$E'T'F	id + id * id$	T → FT'
$E'T'id	id + id * id$	F → id
$E'T'	+ id * id$	
$E'	+ id * id$	T' → ε
$E'T +	+ id * id$	E' → + TE'
$E'T	id * id$	
$E'T'F	id * id$	T → FT'
$E'T'id	id * id$	F → id
$E'T'	* id$	
$E'T'F *	* id$	T' → * FT'
$E'T'F	id$	
$E'T'id	id$	F → id
$E'T'	$	
$E'	$	T' → ε
$	$	E' → ε



Error Recovery in Predictive Parsing

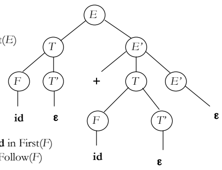
- What happens when M[X,a] is empty?
- Announce Error, Stop and Terminate ! ?
- Engage in Error Recovery mode:
  - Panic-mode:
    - skip symbols on the input until a token in a synchronizing (synch) set of tokens appears on the input;
    - complete entries to the table
  - Phrase-level mode:
    - invoke an external (possibly programmer-defined) procedure that manipulates the stack and the input;
    - less structure, more ad-hoc

Panic-Mode Error Recovery

- No universally accepted method
- Heuristics to fill in empty table entries include:
  - Place all symbols in Follow(A) a synch set of the non-terminal A; skip input tokens until on elements of synch is seen and then pop A
    - Pretends like we have seen A and successfully parsed it.
  - Use hierarchical relation between grammar symbols (*e.g.,* expr and stats). Use First(H) as synch of lower non-terminal symbols.
    - In effect skip or ignore lower constructs poping then off the stack
  - Add First(A) to synch set of A without poping. Skip input until they match
    - Try to move on to the beginning of the next occurrence of A
  - If A ⇒ ε, then try to use this production as default and proceed
  - If a terminal cannot be matched, pop it from the stack
    - In effect mimicing its insertion in the input stream

Panic-mode Error Recovery Example

STACK	INPUT	REMARK
$E	) id * + id$	error: skip) until **id** in First(*E*)
$E	id * + id$	
$E' T	id * + id$	
$E' T' F	id * + id$	
$E' T' id	id * + id$	
$E' T'	* + id$	
$E' T' F *	* + id$	
$E' T' F	+ id$	error: could skip + until **id** in First(*F*) or pop *F* because + is in Follow(*F*)
$E' T'	+ id$	
$E'	+ id$	
$E' T' +	+ id$	
$E' T	id$	
$E' T' F	id$	
$E' T' id	id$	
$E' T'	$	
$E'	$	
$	$	



Summary

- Top-Down Parsing
  - Predictive-Procedural Parsing
  - Eliminating Left-Recursion & Left Factoring
  - First and Follow Sets
  - Table-driven Parsing
- Error Recovery