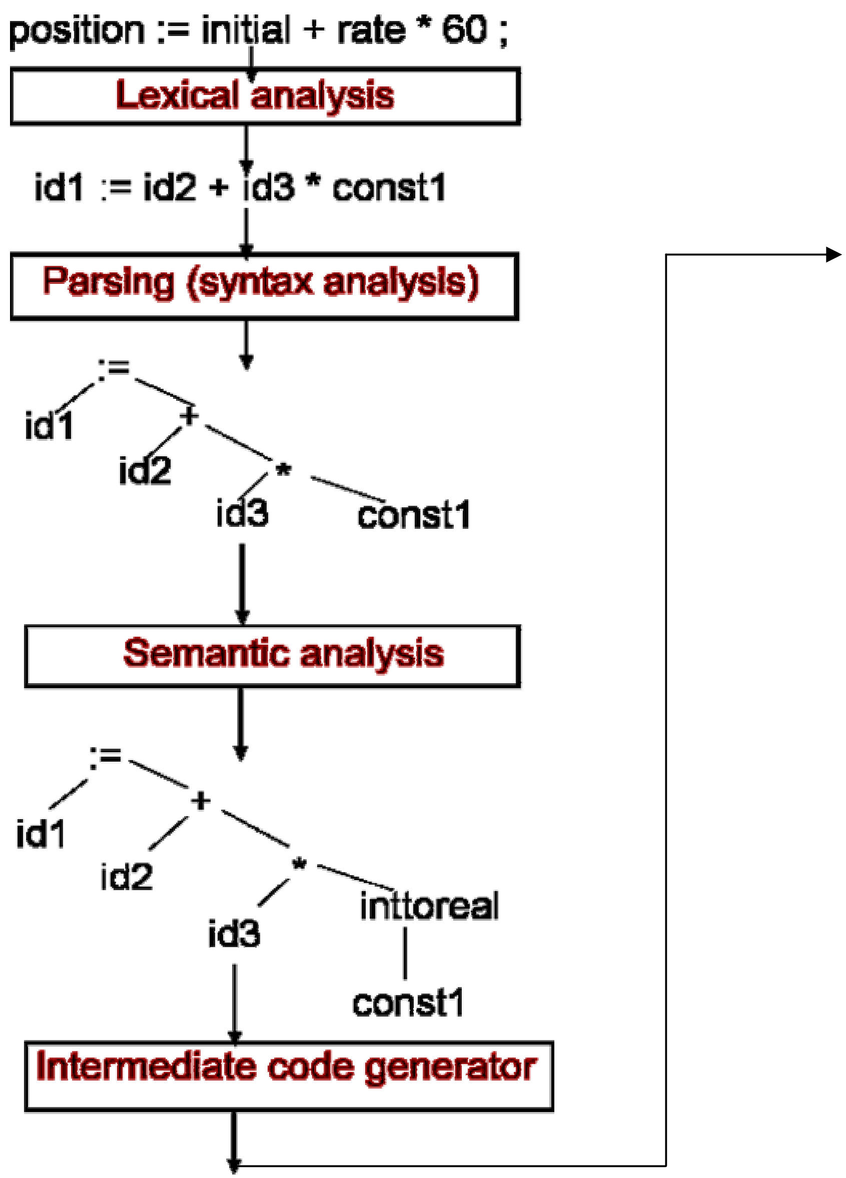
Syntax Analyzer (Parser)

Input: list of tokens produced by scanner
Output: tree which shows structure of program

Recap: Overview



temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3

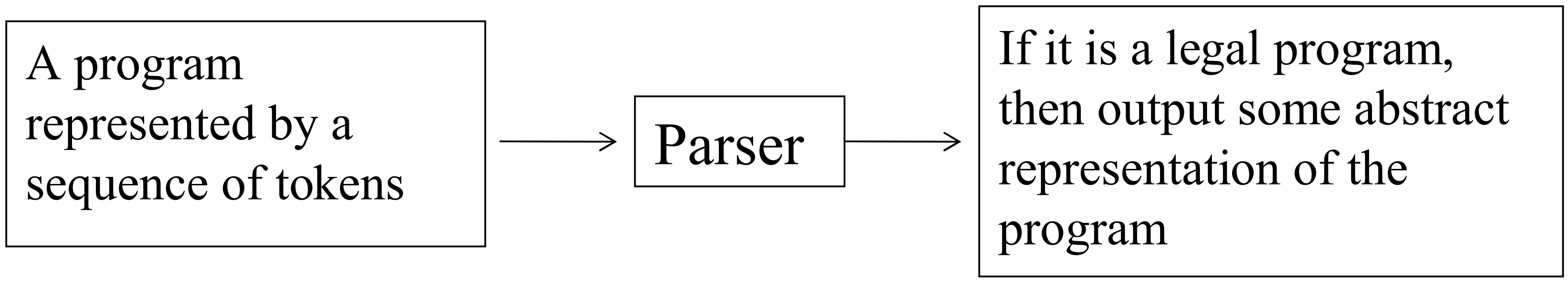
Code optimization

temp1 := id3 * 60.0
id1 := id2 + temp1

Code generator

MOVF ID3, R2
MULF #60.0, R2
MOVF ID3, R1
ADDF R2, R1
MOVF R1, ID1

Introduction



- Abstract representations of the input program:

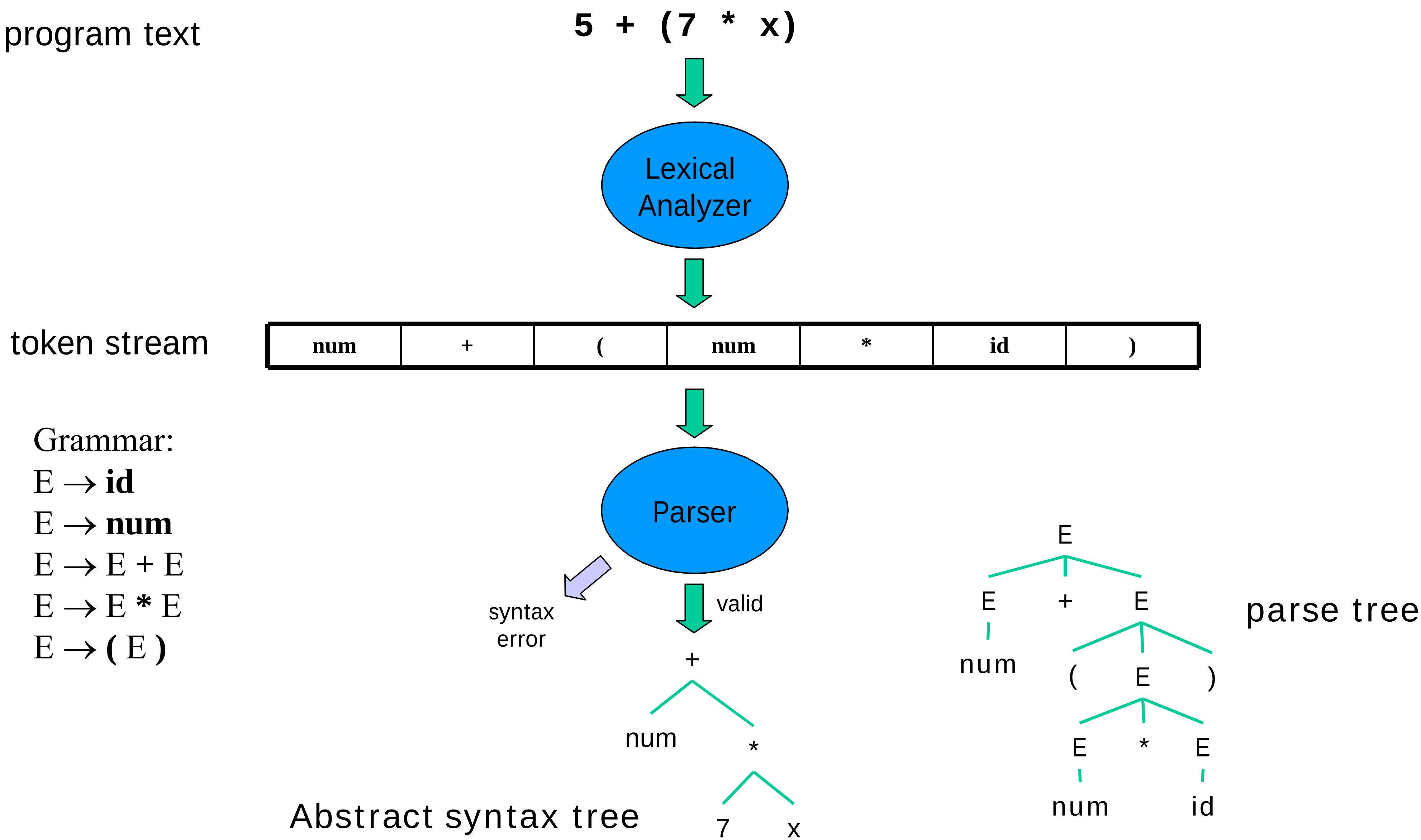
- abstract-syntax tree + symbol table

- intermediate code

- object code

- **Context free grammar (CFG)** is used to specify the structure of legal programs

From text to abstract syntax



Grammar:

E → **id**

E → **num**

E → E **+** E

E → E ***** E

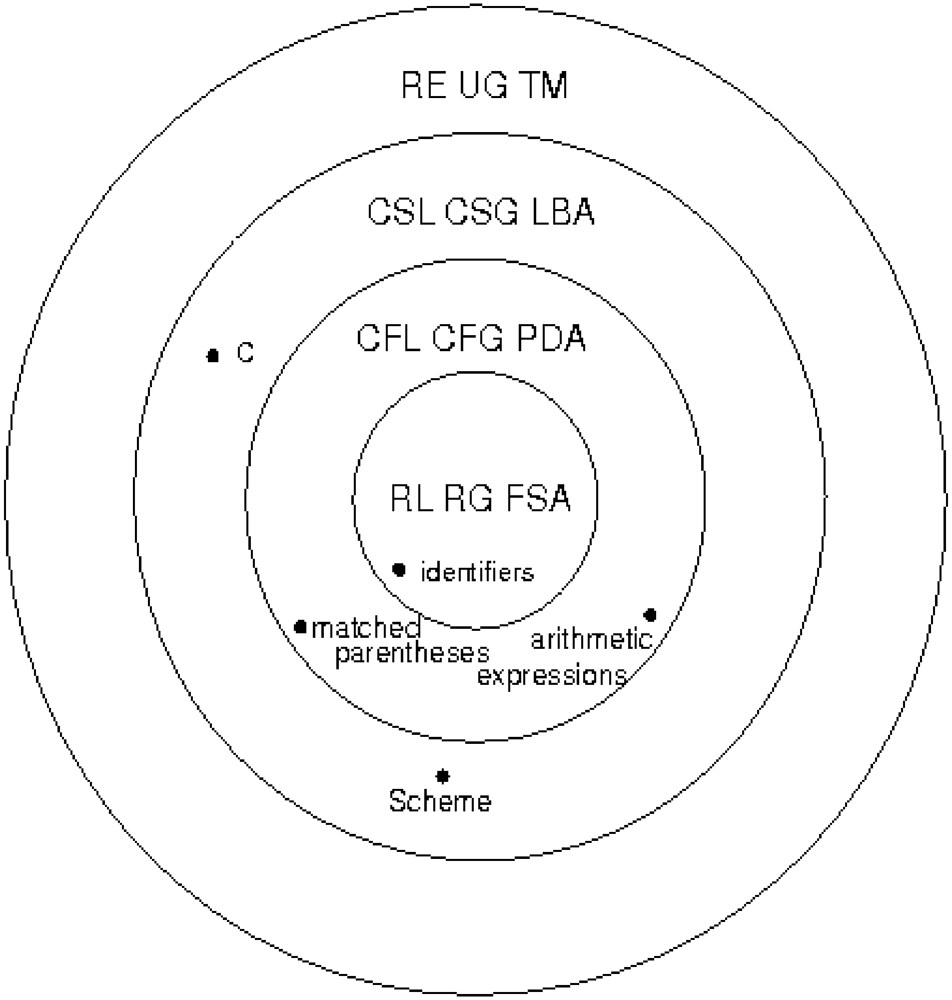
E → **(** E **)**

Goals of parsing

- Programming language has syntactic rules
  - Context-Free Grammars
- Decide whether program satisfies syntactic structure
  - Error detection
  - Error recovery
  - Simplification: rules on tokens
- Build Abstract Syntax Tree

Classes of Grammars
(The Chomsky Hierarchy)

- Type-0: Phrase structured (unrestricted) grammars
  – generate recursively enumerable (unrestricted) languages
  – include all formal grammars
  – implemented with Turing machines
- Type-1 : Context-sensitive grammars
  – generate context-sensitive languages
  – implemented with linear-bounded automata
- Type-2 : Context-free grammars
  – generate context-free languages
  – single non-terminal on left
  – non-terminals & terminals on right
  – implemented with pushdown automata
- Type-3 : Regular grammars
  – generate regular languages
  – no terminals or non-terminals here
  – implemented with finite state automata



Classes of Grammars
(The Chomsky Hierarchy)

Type 0, Phrase Structure (same as basic grammar definition)

Type 1, Context Sensitive

(1) $\alpha \to \beta$ where α is in $(N \cup \Sigma)^* N (N \cup \Sigma)^*$,

β is in $(N \cup \Sigma)+$, and length(α) $\leq$ length(β)

(2) $\gamma A \delta \to \gamma \beta \delta$ where A is in N, β is in $(N \cup \Sigma)^+$, and

γ and δ are in $(N \cup \Sigma)^*$

Type 2, Context Free

$A \to \beta$ where A is in N, β is in $(N \cup \Sigma)^*$

Linear

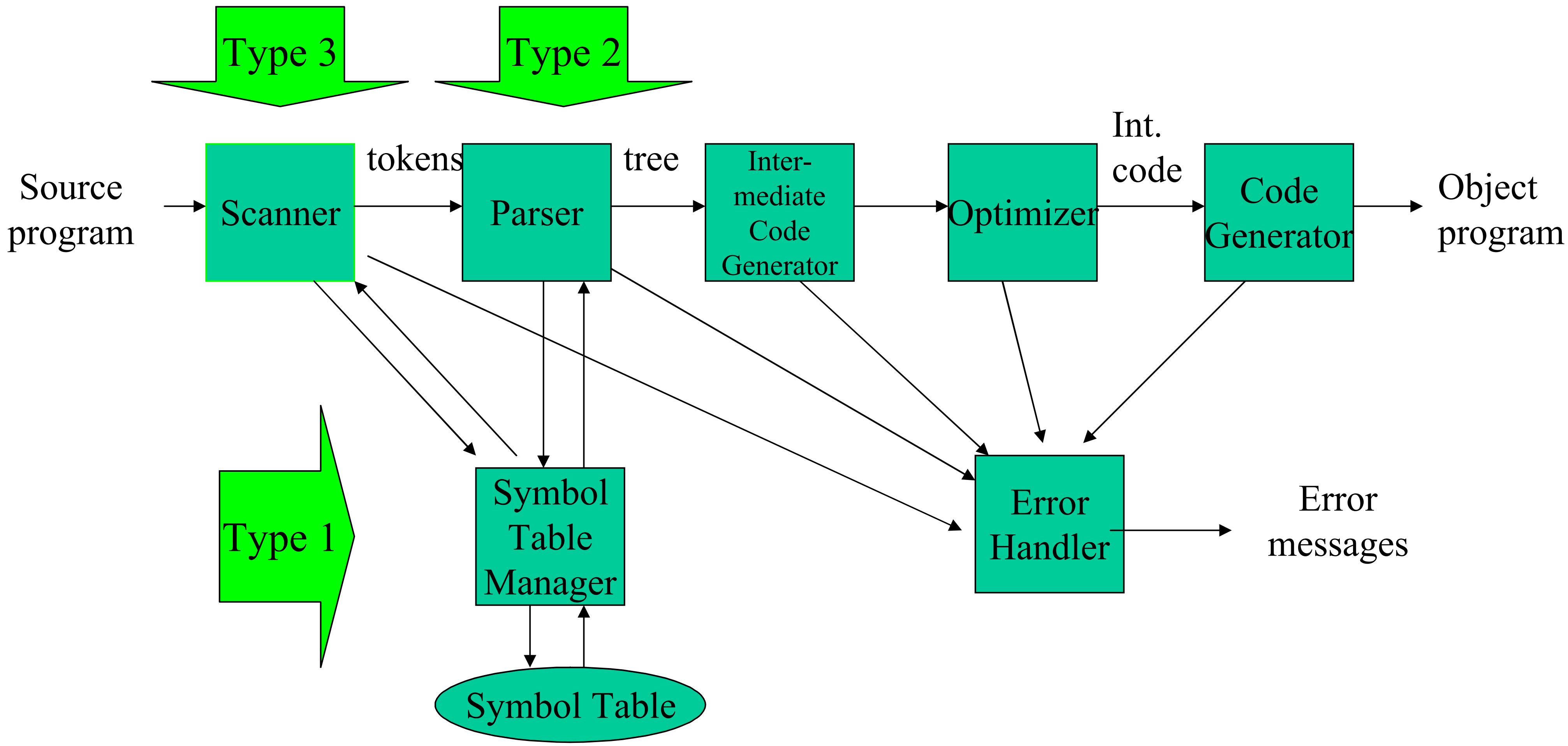
$A \to x$ or $A \to x B y$, where A and B are in N and x and y are in Σ^*

Type 3, Regular Expressions

(1) left linear $A \to B a$ or $A \to a$, where A and B are in N and a is in Σ

(2) right linear $A \to a B$ or $A \to a$, where A and B are in N and a is in Σ

The Chomsky Hierarchy and the Block Diagram of a Compiler



CFG vs. Regular Expressions

- CFG is more expressive than RE
  - Every language that can be described by regular expressions can also be described by a CFG
- Example : languages that are CFG but not RE
  - if-then-else statement, $\{a^n b^n \mid n>=1\}$
- Non-CFG
  - $L1=\{wcw \mid w \text{ is in } (a|b)^*\}$
  - $L2=\{a^n b^m c^n d^m \mid n>=1 \text{ and } m>=1\}$

Context Free Grammars

- CFGs
  - Add recursion to regular expressions
    - Nested constructions
  - Notation

expression → *identifier* | *number* | **`-`** *expression*
| **`(`** *expression* **`)`**
| *expression operator expression*

operator → **`+`** | **`-`** | **`*`** | **`/`**

- **`Terminal symbols`**
- *Non-terminal symbols*
- Production rule (i.e. substitution rule)

  terminal symbol → terminal and non-terminal symbols

Backus-Naur Form

- Backus-Naur Form (BNF)
  - Equivalent to CFGs in power
  - CFG

$$\begin{aligned} \textit{expression} \rightarrow\ & \textit{identifier} \mid \textit{number} \mid \texttt{-}\ \textit{expression} \\ & \mid \texttt{(}\ \textit{expression}\ \texttt{)} \\ & \mid \textit{expression operator expression} \\ \textit{operator} \rightarrow\ & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \end{aligned}$$

  - BNF

$$\begin{aligned} \langle\text{expression}\rangle \rightarrow\ & \langle\text{identifier}\rangle \mid \langle\text{number}\rangle \mid \text{-}\ \langle\text{expression}\rangle \\ & \mid (\ \langle\text{expression}\rangle\ ) \\ & \mid \langle\text{expression}\rangle\ \langle\text{operator}\rangle\ \langle\text{expression}\rangle \\ \langle\text{operator}\rangle \rightarrow\ & + \mid - \mid * \mid / \end{aligned}$$

Extended Backus-Naur Form

- Extended Backus-Naur Form (EBNF)
  - Adds some convenient symbols
    - Union |
    - Kleene star *
    - Meta-level parentheses ( )
  - It has the same expressive power

Extended Backus-Naur Form

- Extended Backus-Naur Form (EBNF)
  - It has the same expressive power

BNF

$\langle \text{digit} \rangle \rightarrow 0$
$\langle \text{digit} \rangle \rightarrow 1$
…
$\langle \text{digit} \rangle \rightarrow 9$
$\langle \text{unsigned\_integer} \rangle \rightarrow \langle \text{digit} \rangle$
$\langle \text{unsigned\_integer} \rangle \rightarrow \langle \text{digit} \rangle \langle \text{unsigned\_integer} \rangle$

EBNF

$\langle \text{digit} \rangle \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
$\langle \text{unsigned\_integer} \rangle \rightarrow \langle \text{digit} \rangle \langle \text{digit} \rangle^*$

Derivations

- A derivation shows how to generate a syntactically valid string
  - Given a CFG
  - Example:
    - CFG

$$
\begin{aligned}
\textit{expression} \rightarrow\ & \textit{identifier} \\
& \mid \textit{number} \\
& \mid \texttt{-}\ \textit{expression} \\
& \mid \texttt{(}\ \textit{expression}\ \texttt{)} \\
& \mid \textit{expression operator expression} \\
\textit{operator} \rightarrow\ & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}
\end{aligned}
$$

    - Derivation of

```
slope * x + intercept
```

Derivation Example

- Derivation of `slope * x + intercept`

expression ⇒ *expression operator* <u>*expression*</u>
⇒ *expression* <u>*operator*</u> `intercept`
⇒ <u>*expression*</u> + `intercept`
⇒ *expression operator* <u>*expression*</u> + `intercept`
⇒ *expression* <u>*operator*</u> `x` + `intercept`
⇒ <u>*expression*</u> * `x` + `intercept`
⇒ `slope` * `x` + `intercept`

expression ⇒* `slope` * `x` + `intercept`

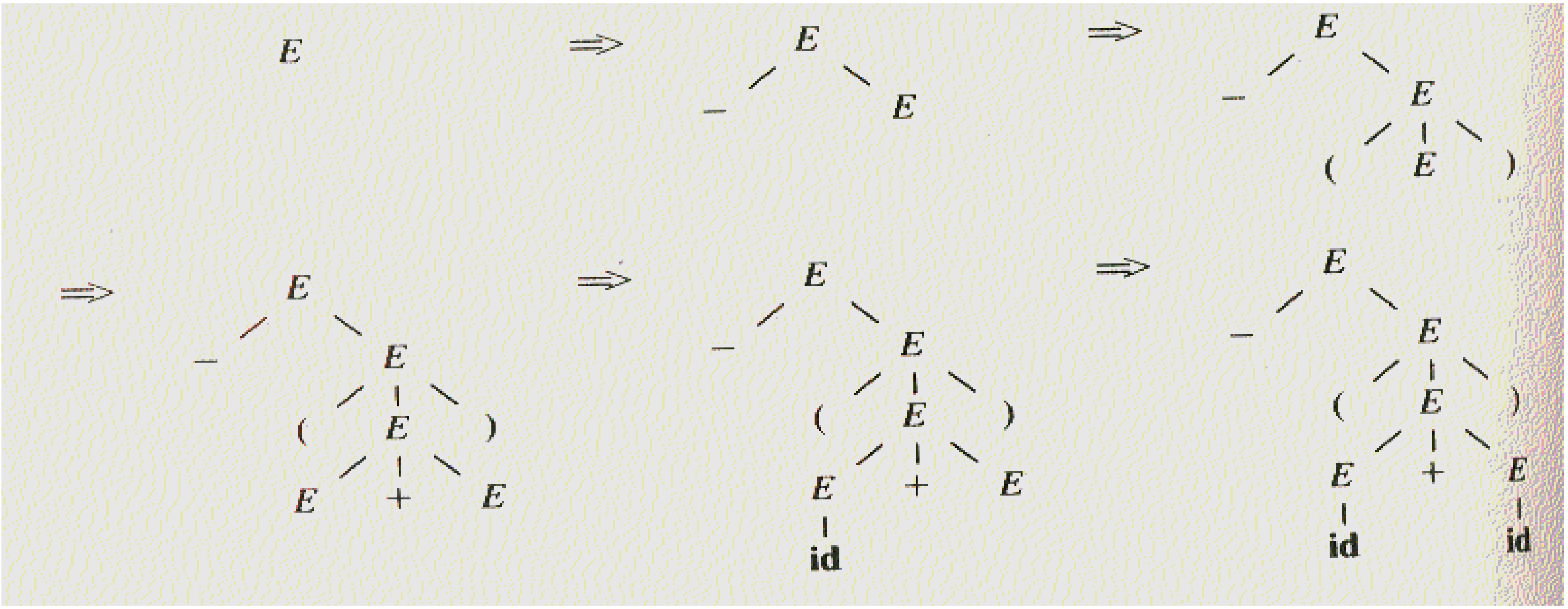
- Identifiers were not derived for simplicity

Parse Trees

- A parse tree is any tree in which
  - The root is labeled with S
  - Each leaf is labeled with a token a or ε
  - Each interior node is labeled by a nonterminal
  - If an interior node is labeled A and has children labeled X1,.. Xn, then A ::= X1...Xn is a production.

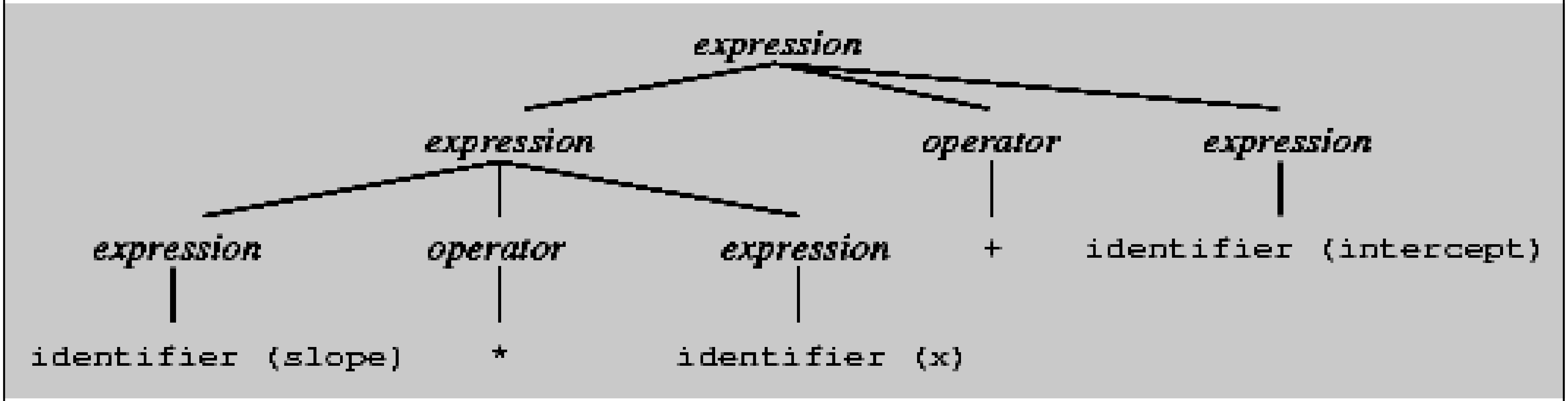
Parse Trees and Derivations

E ::= E + E | E * E | E - E | - E | (E) | id



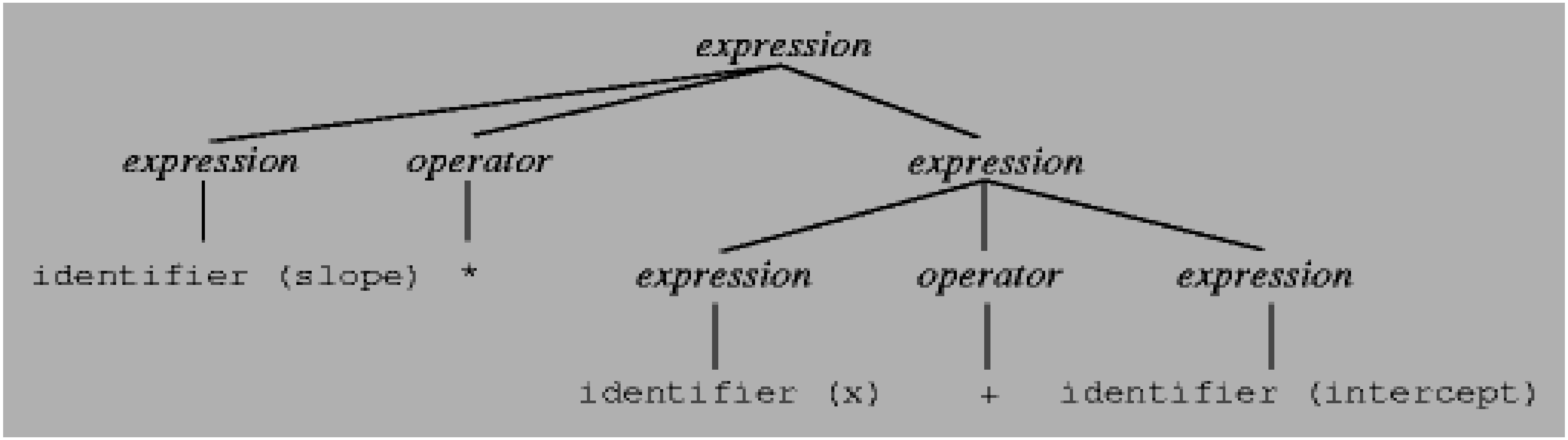
Parse Trees

- A parse is graphical representation of a derivation
- Example



Ambiguous Grammars

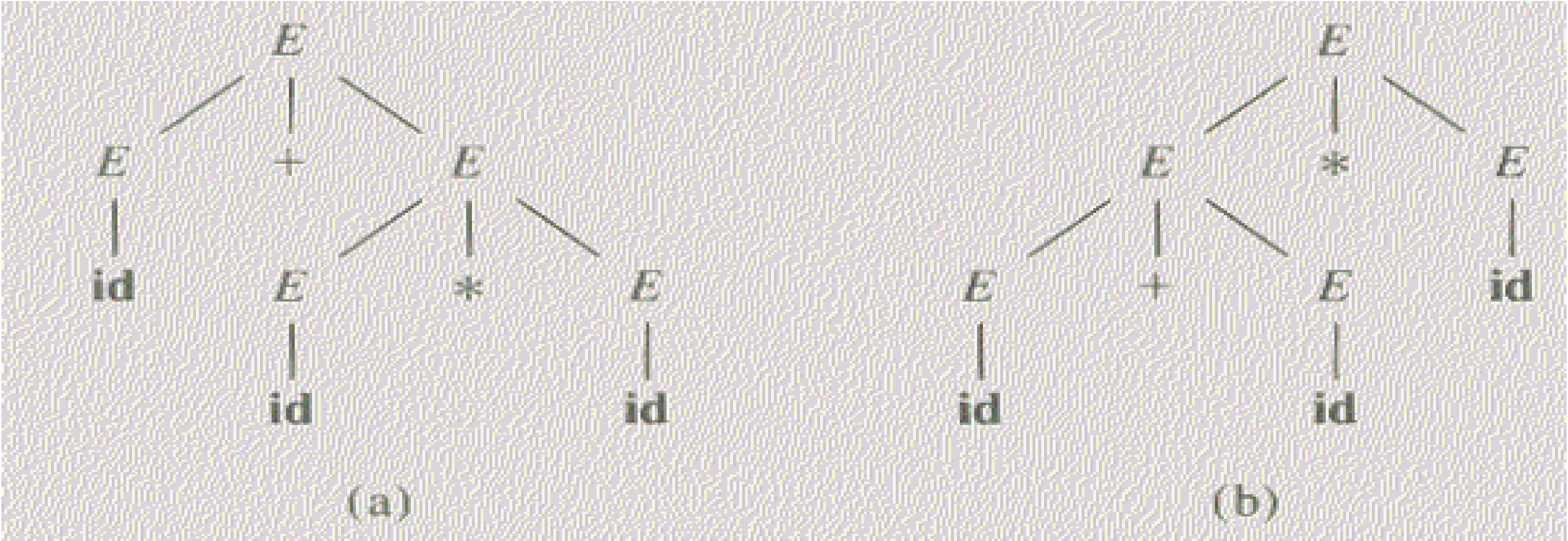
- Alternative parse tree
  - same expression
  - same grammar



  - This grammar is ambiguous

Ambiguity

- A grammar that produces more than one parse tree for some sentence is said to be *ambiguous*.



Eliminating Ambiguity

- There is no deterministic way of finding out whether a grammar is ambiguous and how to fix it. In order to remove ambiguity, we follow some heuristics.
- There are three parts to this:

1. Add a non-terminal for each precedence level
2. Isolate the corresponding part of the grammar
3. Force the parser to recognize the high-precedence sub expressions first

```
E -> E + E | E − E
       | E * E | E / E
       | (E) | var
```

```
E -> E + T | E − T | T
T -> T * F | T  / F  | F
F -> (E) | var
```

Eliminating Left-Recursion

- Direct left-recursion

$$A ::= A\alpha \mid \beta$$

$$\Downarrow$$

$$A ::= \beta A'$$
$$A' ::= \alpha A' \mid \varepsilon$$

$$A ::= A\alpha 1 \mid \ldots \mid A\alpha m \mid \beta 1 \mid \ldots \mid \beta n$$

$$\Downarrow$$

$$A ::= \beta 1 A' \mid \ldots \mid \beta n A'$$
$$A' ::= \alpha 1 A' \mid \ldots \mid \alpha n A' \mid \varepsilon$$

Eliminating Indirect Left-Recursion

- Indirect left-recursion

- Algorithm

$$S ::= Aa \mid b$$
$$A ::= Ac \mid Sd \mid \varepsilon$$

Arrange the nonterminals in some order $A_1,\ldots,A_n$.
for (i in 1..n) {
    for (j in 1..i-1) {
        replace each production of the form $A_i ::= A_j\gamma$ by the
        productions $A_i ::= \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$ where
            $A_j ::= \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$
    }
    eliminate the immediate left recursion among A_i productions
}

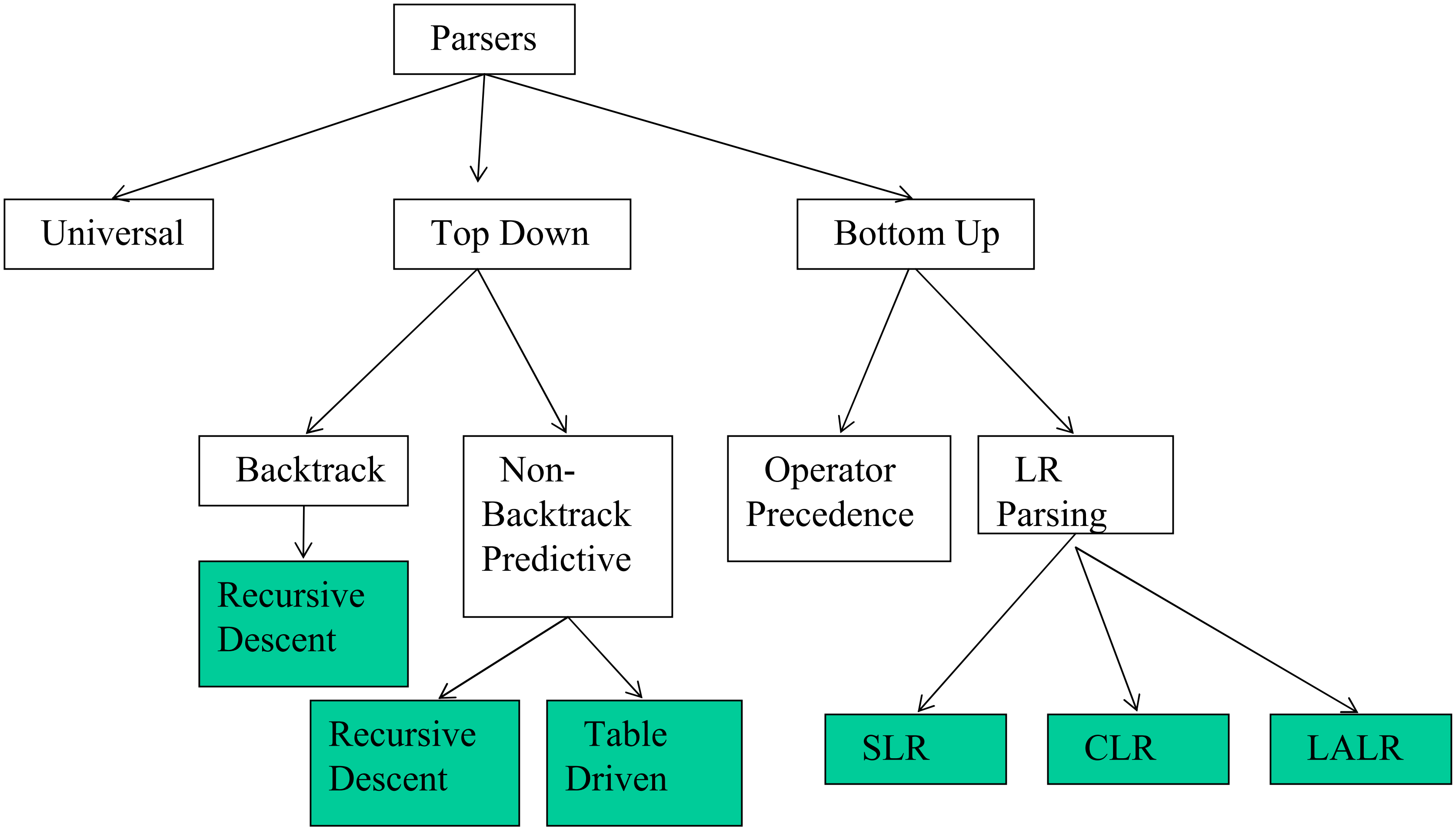
Left Factoring

$$A ::= \alpha\beta 1 \mid \ldots \mid \alpha\beta n \mid \gamma$$

$$\downarrow$$

$$A ::= \alpha A' \mid \gamma$$
$$A' ::= \beta 1 \mid \ldots \mid \beta n$$

Types of Parsers



Top-Down Parsing

- Start from the start symbol and build the parse tree top-down
- Apply a production to a nonterminal. The right-hand of the production will be the children of the nonterminal
- Match terminal symbols with the input
- May require backtracking
- Some grammars are backtrack-free (predictive)

TDP

- The parse tree is created top to bottom.
- Top-down parser
  - Recursive-Descent Parsing
    - Backtracking is needed (If a choice of a production rule does not work, we backtrack to try other alternatives.)
    - It is a general parsing technique, but not widely used.
    - Not efficient
  - Predictive Parsing
    - no backtracking
    - efficient
    - needs a special form of grammars (LL(1) grammars).
    - **Recursive Predictive Parsing is a special form of Recursive Descent parsing without backtracking.**
    - **Non-Recursive (Table Driven) Predictive Parser is also known as LL(1) parser.**

Construct Parse Trees Top-Down

– Start with the tree of one node labeled with the start symbol and repeat the following steps 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 is added to the fringe that doesn't match the input string, backtrack
3. Find the next node to be expanded

– Minimize the number of backtracks

Example

Left-recursive

```
E ::=   T
        | E + T
        | E - T
T ::=   F
        | T * F
        | T / F
F ::=   id
        | number
        | (E)
```

Right-recursive

```
E ::=   T E'
E'::=   + T E'
        | - T E'
        | e
T::=    F T'
T' ::=  * F T'
        | / F T'
        | e
F ::=   id
        | number
        | (E)
```

x - 2 * y

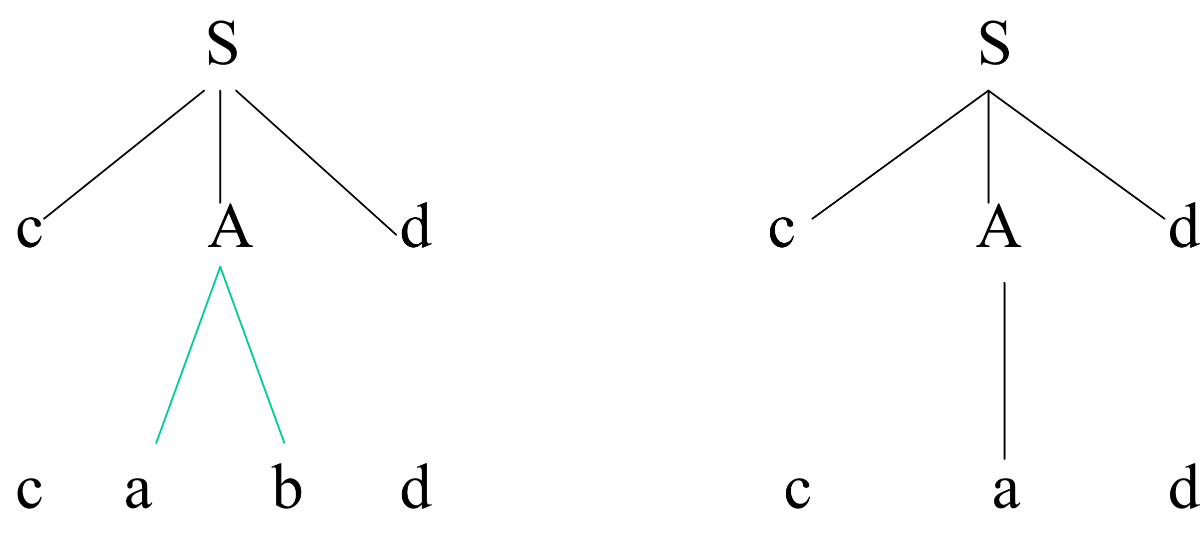
Recursive-Descent Parsing (uses Backtracking)

- Backtracking is needed.
- It tries to find the left-most derivation.
- Grammar rule of a non-terminal "A" is viewed as a definition of a procedure that will recognize "A".

$S \rightarrow cAd$

$A \rightarrow ab \mid a$

input: cad



fails, backtrack

Recursive Descent Parser- Example

- A separate recursive procedure is written for every non-terminals

```
Procedure S()
{
    if input = 'c'
    {
    Advance();     //procedure that is written to advance the input pointer to next position
    A();
    if input = 'd'
    {
    Advance();
    return true;
    }
    else return false;
    else return false;
}
```

Cont.

```
Procedure A()
{
isave=in-ptr;                  // i-save saves the input pointer position before each alternate to facilitate
     backtracking
If input ='a'
{    Advance();
     if input = 'b'
     {
               Advance();
               return true;
     }
}
In-ptr=isave
If input ='a'
{    Advance();
     return true;
}
return false;
return false;
}
```

Cont.

- Problems??
- Left recursion – ambiguity as how many times to call? Solution – eliminate it
- Backtracking – when more than one alternative in the rule. Solution – left factoring
- Very difficult to identify the position of the errors

- To eliminate left recursion:

Ex. $A \rightarrow A\alpha|\beta$

Soln: $A \rightarrow \beta A'$

$A' \rightarrow \alpha A'|\epsilon$

Predictive Parser

a grammar $\xrightarrow[\text{left recursion}]{\text{eliminate}}$ $\xrightarrow[\text{factor}]{\text{left}}$ a grammar suitable for predictive parsing (a LL(1) grammar)

- When re-writing a non-terminal in a derivation step, a predictive parser can uniquely choose a production rule by just looking the current symbol in the input string.

$A \to \alpha_1 \mid \ldots \mid \alpha_n$ input: ... a ....

current token

Predictive Parser (example)

stmt → `if` ...... |
`while` ...... |
`begin` ...... |
`for` .....

- When we are trying to write the non-terminal *stmt*, if the current token is `if` we have to choose first production rule.
- When we are trying to write the non-terminal *stmt*, we can uniquely choose the production rule by just looking the current token.
- We eliminate the left recursion in the grammar, and left factor it. But it may not be suitable for predictive parsing (not LL(1) grammar).

Recursive Predictive Parsing

- Each non-terminal corresponds to a procedure.

Ex: $A \rightarrow aBb$ (This is only the production rule for A)

proc A {

- match the current token with a, and move to the next token;
- call 'B';
- match the current token with b, and move to the next token;

}

Recursive Predictive Parsing (cont.)

$A \rightarrow aBb \mid bAB$

```
proc A {
    case of the current token {
        'a':  - match the current token with a, and move to the next token;
              - call 'B';
              - match the current token with b, and move to the next token;
        'b':  - match the current token with b, and move to the next token;
              - call 'A';
              - call 'B';
    }
}
```

Recursive Predictive Parsing (cont.)

- When to apply ε-productions.

  $A \rightarrow aA \mid bB \mid \varepsilon$

- If all other productions fail, we should apply an ε-production. For example, if the current token is not a or b, we may apply the ε-production.
- Most correct choice: We should apply an ε-production for a non-terminal A when the current token is in the follow set of A (which terminals can follow A in the sentential forms).

Recursive Predictive Parsing (Example)

$A \rightarrow aBe \mid cBd \mid C$

$B \rightarrow bB \mid \varepsilon$

$C \rightarrow f$

```
proc A {
    case of the current token {
        a:  - match the current token with a,
              and move to the next token;
            - call B;
            - match the current token with e,
              and move to the next token;
        c:  - match the current token with c,
              and move to the next token;
            - call B;
            - match the current token with d,
              and move to the next token;
        f:  - call C
    }
}
```

first set of C

```
proc C {  match the current token with f,
          and move to the next token; }
```

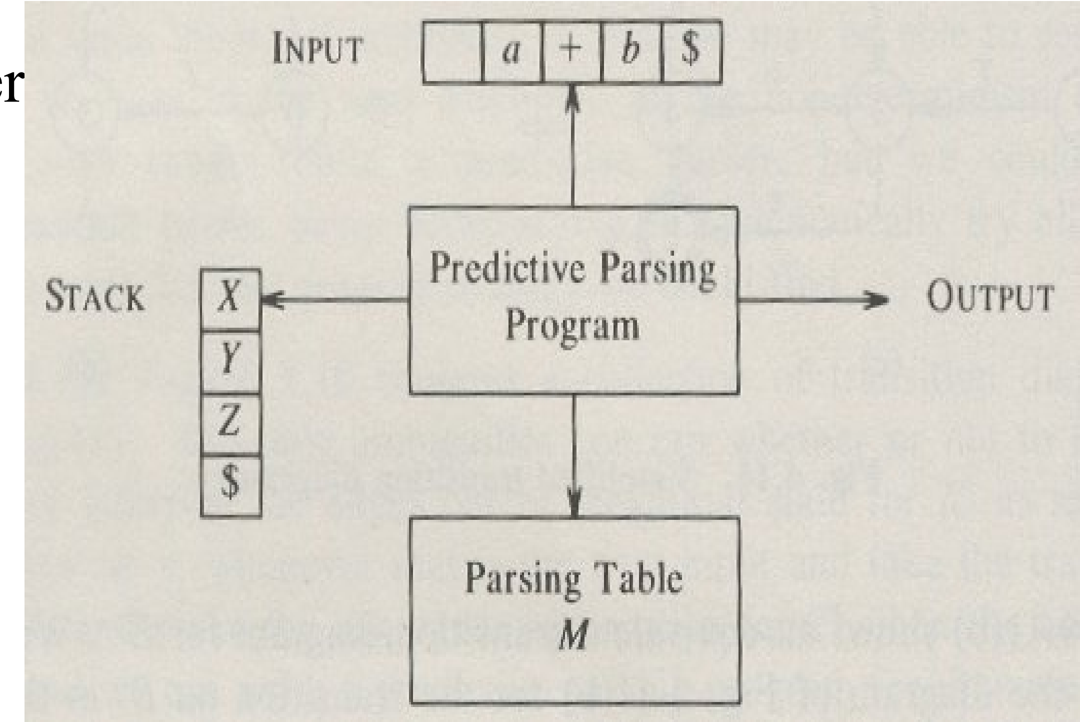
```
proc B {
    case of the current token {
        b:  - match the current token with b,
              and move to the next token;
            - call B
        e,d:  do nothing
    }
}
```

follow set of B

Non-Recursive Predictive Parsing - LL(1) Parser

- An **LL parser** is a top-down parser for a subset of the context-free grammars. It parses the input from **L**eft to right, and constructs a **L**eftmost derivation of the sentence
- Non-Recursive predictive parsing is a table-driven parser.
- It is a top-down parser.
- It is also known as LL(1) Parser

An LL parser is called an LL(*k*) parser if it uses *k* tokens of lookahead when parsing a sentence



LL(1) Parser

input buffer

- our string to be parsed. We will assume that its end is marked with a special symbol $.

output

- a production rule representing a step of the derivation sequence (left-most derivation) of the string in the input buffer.

stack

- contains the grammar symbols
- at the bottom of the stack, there is a special end marker symbol $.
- initially the stack contains only the symbol $ and the starting symbol S. $S ← initial stack
- when the stack is emptied (ie. only $ left in the stack), the parsing is completed.

parsing table

- a two-dimensional array M[A,a]
- each row is a non-terminal symbol
- each column is a terminal symbol or the special symbol $
- each entry holds a production rule.

LL(1) Parser – Parser Actions

parsing table

```
set ip to point to the first symbol of w$;
repeat
      let X be the top stack symbol and a the symbol pointed to by ip;
      if X is a terminal or $ then
            if X = a then
                  pop X from the stack and advance ip
            else error()
      else        /* X is a nonterminal */
            if M[X, a] = X → Y1Y2 · · · Yk then begin
                  pop X from the stack;
                  push Yk, Yk−1, . . . , Y1 onto the stack, with Y1 on top;
                  output the production X → Y1Y2 · · · Yk
            end
            else error()
until X = $      /* stack is empty */
```

LL(1) Parser – Parser Actions

- The symbol at the top of the stack (say X) and the current symbol in the input string (say a) determine the parser action.
- There are four possible parser actions.

1. If X and a are \$ ➔ parser halts (successful completion)
2. If X and a are the same terminal symbol (different from \$)

   ➔ parser pops X from the stack, and moves the next symbol in the input buffer.
3. If X is a non-terminal

   ➔ parser looks at the parsing table entry M[X,a]. If M[X,a] holds a production rule $X \rightarrow Y_1 Y_2 ... Y_k$, it pops X from the stack and pushes $Y_k, Y_{k-1}, ..., Y_1$ into the stack. The parser also outputs the production rule $X \rightarrow Y_1 Y_2 ... Y_k$ to represent a step of the derivation.
4. none of the above ➔ error
   - all empty entries in the parsing table are errors.
   - If X is a terminal symbol different from a, this is also an error case.

LL(1) Parser – Example1

$S \to aBa$

$B \to bB \mid \varepsilon$

	a	b	$
S	$S \to aBa$		
B	$B \to \varepsilon$	$B \to bB$	

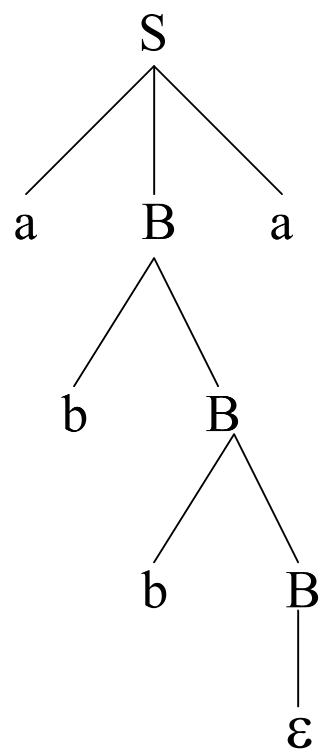
LL(1) Parsing Table

stack	input	output
$S	abba$	$S \to aBa$
$aBa	abba$	
$aB	bba$	$B \to bB$
$aBb	bba$	
$aB	ba$	$B \to bB$
$aBb	ba$	
$aB	a$	$B \to \varepsilon$
$a	a$	
$	$	accept, successful completion

LL(1) Parser – Example1 (cont.)

Outputs: $S \rightarrow aBa$ $B \rightarrow bB$ $B \rightarrow bB$ $B \rightarrow \varepsilon$

Derivation(left-most): $S \Rightarrow aBa \Rightarrow abBa \Rightarrow abbBa \Rightarrow abba$



parse tree

LL(1) Parser – Example2

$E \rightarrow TE'$
$E' \rightarrow +TE' \;|\; \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \;|\; \varepsilon$
$F \rightarrow (E) \;|\; id$

	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	$F \rightarrow id$			$F \rightarrow (E)$		

LL(1) Parser – Example2

stack	input	output
$E	id+id$	$E \rightarrow TE'$
$E'T	id+id$	$T \rightarrow FT'$
$E' T'F	id+id$	$F \rightarrow id$
$ E' T'id	id+id$	
$ E' T'	+id$	$T' \rightarrow \varepsilon$
$ E'	+id$	$E' \rightarrow +TE'$
$ E' T+	+id$	
$ E' T	id$	$T \rightarrow FT'$
$ E' T' F	id$	$F \rightarrow id$
$ E' T'id	id$	
$ E' T'	$	$T' \rightarrow \varepsilon$
$ E'	$	$E' \rightarrow \varepsilon$
$	$	accept

Constructing LL(1) Parsing Tables

1. Eliminate left recursion in grammar G
2. Perform left factoring on the grammar G
3. Find FIRST and FOLLOW on the symbols in grammar G
4. Construct the predictive parse table
5. Check if the given input string can be accepted by the parser

Compute FIRST

- If α is a terminal symbol ‘a’ then FIRST(α)={a}

For example, for gramamr rule A -> a, FIRST(a)={a}

- If α is a non-terminal symbol ‘X’ and X -> aα,

then FIRST(X)={a}

For example for grammar rule A->aBC, FIRST(A) = FIRST(aBC) = {a}

- If α is a non-terminal ‘X’ and X-> Ɛ, then FIRST(X)={Ɛ}

For example for grammer rule A->Ɛ, FIRST(A)={Ɛ}

- If X -> $Y_{1,}Y_{2,}..Y_n$ then add to FIRST($Y_{1,}Y_{2,}..Y_n$) all the non- Ɛ symbols of FIRST(Y_1). Also add the non- Ɛ symbols of FIRST(Y_2) if Ɛ is in FIRST(Y_1), the non- Ɛ symbols of FIRST(Y_3) if Ɛ is in both FIRST(Y_1) and in FIRST(Y_2), and so on. Finally add Ɛ to FIRST($Y_{1,}Y_{2,}..Y_n$) if, for all i, FIRST(Y_i) contains Ɛ.

For example for rules: X -> Yb and Y -> a | Ɛ

FIRST(X)=FIRST(Yb)=FIRST(Y)={a, b}

FIRST Example

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E) \mid id$

FIRST(F) = {(,id}
FIRST(T′) = {*, ε}
FIRST(T) = {(,id}
FIRST(E′) = {+, ε}
FIRST(E) = {(,id}

FIRST(TE′) = {(,id}
FIRST(+TE′) = {+}
FIRST(ε) = {ε}
FIRST(FT′) = {(,id}
FIRST(*FT′) = {*}
FIRST(ε) = {ε}
FIRST((E)) = {(}
FIRST(id) = {id}

Compute FOLLOW (for non-terminals)

FOLLOW of a non-terminal A is a set of terminals that follow or occur to the right of A

- If S is the start symbol ➔ $ is in FOLLOW(S)

- if $A \rightarrow \alpha B \beta$ is a production rule
  ➔ everything in FIRST(β) is FOLLOW(B) except ε

- If ($A \rightarrow \alpha B$ is a production rule) or
  ($A \rightarrow \alpha B \beta$ is a production rule and ε is in FIRST(β))
  ➔ everything in FOLLOW(A) is in FOLLOW(B).

We apply these rules until nothing more can be added to any follow set.

FOLLOW Example

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E) \mid id$

FOLLOW(E) = { \$,) }
FOLLOW(E') = { \$,) }
FOLLOW(T) = { +,), \$ }
FOLLOW(T') = { +,), \$ }
FOLLOW(F) = {+, *,), \$ }

Constructing LL(1) Parsing Table -- Algorithm

- for each production rule $A \rightarrow \alpha$ of a grammar G
  - for each terminal a in FIRST(α)

    ➔ add $A \rightarrow \alpha$ to M[A,a]
  - If ε in FIRST(α)

    ➔ for each terminal a in FOLLOW(A) add $A \rightarrow \alpha$ to M[A,a]
  - If ε in FIRST(α) and \$ in FOLLOW(A)

    ➔ add $A \rightarrow \alpha$ to M[A,\$]

- All other undefined entries of the parsing table are error entries.

Constructing LL(1) Parsing Table -- Example

$E \to TE'$	FIRST(TE')={(,id}	➔ $E \to TE'$ into M[E,(] and M[E,id]
$E' \to +TE'$	FIRST($+TE'$)={+}	➔ $E' \to +TE'$ into M[E',+]
$E' \to \varepsilon$	FIRST(ε)={ε} but since ε in FIRST(ε) and FOLLOW(E')={$,)}	➔ none ➔ $E' \to \varepsilon$ into M[E',$] and M[E',)]
$T \to FT'$	FIRST(FT')={(,id}	➔ $T \to FT'$ into M[T,(] and M[T,id]
$T' \to *FT'$	FIRST($*FT'$)={*}	➔ $T' \to *FT'$ into M[T',*]
$T' \to \varepsilon$	FIRST(ε)={ε} but since ε in FIRST(ε) and FOLLOW(T')={$,),+}	➔ none ➔ $T' \to \varepsilon$ into M[T',$], M[T',)] and M[T',+]
$F \to (E)$	FIRST((E))={(}	➔ $F \to (E)$ into M[F,(]
$F \to id$	FIRST(id)={id}	➔ $F \to id$ into M[F,id]

LL(1) Grammars

- A grammar whose parsing table has no multiply-defined entries is said to be LL(1) grammar.

one input symbol used as a look-head symbol to determine parser action

LL(1) left most derivation

input scanned from left to right

- ***The parsing table of a grammar may contain more than one production rule. In this case, we say that it is not a LL(1) grammar.***

A Grammar which is not LL(1)

$S \to i\,C\,t\,S\,E \quad | \quad a$
$E \to e\,S \quad | \quad \varepsilon$
$C \to b$

FOLLOW(S) = { \$,e }
FOLLOW(E) = { \$,e }
FOLLOW(C) = { t }

FIRST(iCtSE) = {i}
FIRST(a) = {a}
FIRST(eS) = {e}
FIRST(ε) = {ε}
FIRST(b) = {b}

	a	**b**	**e**	**i**	**t**	**$**
S	$S \to a$			$S \to iCtSE$		
E			$E \to e\,S$ $E \to \varepsilon$			$E \to \varepsilon$
C		$C \to b$				

two production rules for M[E,e]

Problem ➔ ambiguity

A Grammar which is not LL(1) (cont.)

- What do we have to do if the resulting parsing table contains multiply defined entries?
  - If we didn't eliminate left recursion, eliminate the left recursion in the grammar.
  - If the grammar is not left factored, we have to left factor the grammar.
  - If its (new grammar's) parsing table still contains multiply defined entries, that grammar is ambiguous or it is inherently not a LL(1) grammar.
- A left recursive grammar cannot be a LL(1) grammar.
  - $A \rightarrow A\alpha \mid \beta$
    - ➔ any terminal that appears in FIRST(β) also appears FIRST($A\alpha$) because $A\alpha \Rightarrow \beta\alpha$.
    - ➔ If β is ε, any terminal that appears in FIRST(α) also appears in FIRST($A\alpha$) and FOLLOW(A).
- A grammar is not left factored, it cannot be a LL(1) grammar
  - $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$
    - ➔ any terminal that appears in FIRST($\alpha\beta_1$) also appears in FIRST($\alpha\beta_2$).
- An ambiguous grammar cannot be a LL(1) grammar.

Properties of LL(1) Grammars

- A grammar G is LL(1) if and only if the following conditions hold for two distinctive production rules $A \rightarrow \alpha$ and $A \rightarrow \beta$

1. Both α and β cannot derive strings starting with same terminals.
2. At most one of α and β can derive to ε.
3. If β can derive to ε, then α cannot derive to any string starting with a terminal in FOLLOW(A).

Example

- Construct predictive parse table for the following grammar. Also show parser actions for the input string - (a,a)

  S->a | ↑ | (T)

  T->T,S | S

- Eliminate left recursion
- No left factor
- First
- Follow
- Construct parsing table – check multiple entries
- Show Actions

Cont.

- Eliminate left recursion

S-> a | ↑ | (T)

T-> ST’

T’-> ,ST’ | Ɛ

- Its not needed to left factor
- FIRST

FIRST(S)={a, ↑, (}

FIRST(T)=FIRST(ST’)=FIRST(S)={a, ↑,(}

FIRST(T’)={, , Ɛ}

- FOLLOW

FOLLOW(S)={$,) and ,}

FOLLOW(T)={) }

FOLLOW(T’)={) }

- Is following grammar LL(1)? Also trace input string - ibtaea

S -> iCtSS` | a

S` -> eS | Ɛ

C ->b

Motivation Behind First & Follow

First: Is used to help find the appropriate production to follow given the top-of-the-stack non-terminal and the current input symbol.

Example: If $A \rightarrow \alpha$, and a is in First(α), then when a=input, replace A with α (in the stack).

(a is one of first symbols of α, so when A is on the stack and a is input, POP A and PUSH α.

Follow: Is used when First has a conflict, to resolve choices, or when First gives no suggestion. When $\alpha \rightarrow \epsilon$ or $\alpha \stackrel{*}{\Rightarrow} \epsilon$, then what follows A dictates the next choice to be made.

Example: If $A \rightarrow \alpha$, and b is in Follow(A), then when $\alpha \stackrel{*}{\Rightarrow} \epsilon$ and b is an input character, then we expand A with α, which will eventually expand to ϵ, of which b follows!

($\alpha \stackrel{*}{\Rightarrow} \epsilon$: i.e., First(α) contains ϵ.)

Error Recovery Techniques

- Panic-Mode Error Recovery
  - Skipping the input symbols until a synchronizing token is found.
- Phrase-Level Error Recovery
  - Each empty entry in the parsing table is filled with a pointer to a specific error routine to take care that error case.
- Error-Productions
  - If we have a good idea of the common errors that might be encountered, we can augment the grammar with productions that generate erroneous constructs.
  - When an error production is used by the parser, we can generate appropriate error diagnostics.
  - Since it is almost impossible to know all the errors that can be made by the programmers, this method is not practical.
- Global-Correction
  - Ideally, we we would like a compiler to make as few change as possible in processing incorrect inputs.
  - We have to globally analyze the input to find the error.
  - This is an expensive method, and it is not in practice.

Panic-Mode Recovery

Assume a non-terminal on the top of the stack.

1. Idea:

   skip symbols on the input until a token in a selected set of *synchronizing* tokens is found.

2. The choice for a synchronizing set is important.

   Some ideas:

   a. Define the synchronizing set of A to be FOLLOW(A). then skip input until a token in FOLLOW(A) appears and then pop A from the stack. Resume parsing...

   b. Add symbols of FIRST(A) into synchronizing set. In this case we skip input and once we find a token in FIRST(A) we resume parsing from A.

   c. Productions that lead to $\in$ if available might be used.

3. **If a terminal appears on top of the stack and does not match to the input => pop it and continue parsing (issuing an error message saying that the terminal was inserted).**

General Approach: Modify the empty cells of the Parsing Table.

1. if M[A,a] = {empty} and a belongs to Follow(A) then we set M[A,a] = "synch"

Error-recovery Strategy :

If A=top-of-the-stack and a=current-input,

1. If A is NT and M[A,a] = {empty} then skip a from the input.
2. If A is NT and M[A,a] = {synch} then pop A.
3. If A is a terminal and A!=a then pop token (essentially inserting it).

Revised Parsing Table / Example

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)	—	—

From Follow sets. Pop top of stack NT

“synch” action

Skip input symbol

Revised Parsing Table / Example

STACK	INPUT	Remark
$E	+ id * + id$	error, skip +
$E	id * + id$	
$E’T	id * + id$	
$E’T’F	id * + id$	
$E’T’id	id * + id$	
$E’T’	* + id$	
$E’T’F*	* + id$	
$E’T’F	+ id$	error, M[F,+] = synch
$E’T’	+ id$	F has been popped
$E’	+ id$	
$E’T+	+ id$	
$E’T	id$	
$E’T’F	id$	
$E’T’id	id$	
$E’T’	$	
$E’	$	
$	$	

Phrase-Level Error Recovery

- Each empty entry in the parsing table is filled with a pointer to a special error routine which will take care that error case.
- These error routines may:
  - change, insert, or delete input symbols.
  - issue appropriate error messages
  - pop items from the stack.
- We should be careful when we design these error routines, because we may put the parser into an infinite loop.

How to Implement TD Parser

- Stack – Easy to handle.
- Input Stream – Responsibility of lexical analyzer
- Key Issue – How is parsing table implemented ?

One approach: Assign unique IDS

Non-terminal	INPUT SYMBOL					
	id	+	*	(	)	$
E	E→TE'			E→TE'	synch	synch
E'		E'→+TE'			E'→∈	E'→∈
T	T→FT'	synch		T→FT'	synch	synch
T'		T'→∈	T'→*FT'		T'→∈	T'→∈
F	F→id	synch	synch	F→(E)	synch	synch

All rules have unique IDs

synch actions

Also for blanks which handle errors

Revised Parsing Table:

Non-terminal	INPUT SYMBOL					
	id	+	*	(	)	$
E	1	18	19	1	9	10
E'	20	2	21	22	3	3
T	4	11	23	4	12	13
T'	24	6	5	25	6	6
F	8	14	15	7	16	17

1 $E \rightarrow TE'$
2 $E' \rightarrow +TE'$
3 $E' \rightarrow \epsilon$
4 $T \rightarrow FT'$
5 $T' \rightarrow *FT'$
6 $T' \rightarrow \epsilon$
7 $F \rightarrow (E)$
8 $F \rightarrow id$

9 – 17 : Sync Actions

18 – 25 : Error Handlers

How is Parser Constructed ?

One large CASE statement:

```
state = M[ top(s), current_token ]
switch (state)
  {
   case 1:   proc_E_TE'( ) ;
             break ;
   …
   case 8:  proc_F_id( ) ;
            break ;
   case 9:   proc_sync_9( ) ;
             break ;
   …
   case 17:  proc_sync_17( ) ;
             break ;
   case 18:
   …
   case 25:
  }
```

Procs to handle errors

Combine → put in another switch

Some sync actions may be same

Some error handlers may be similar