

Parsing

Part I

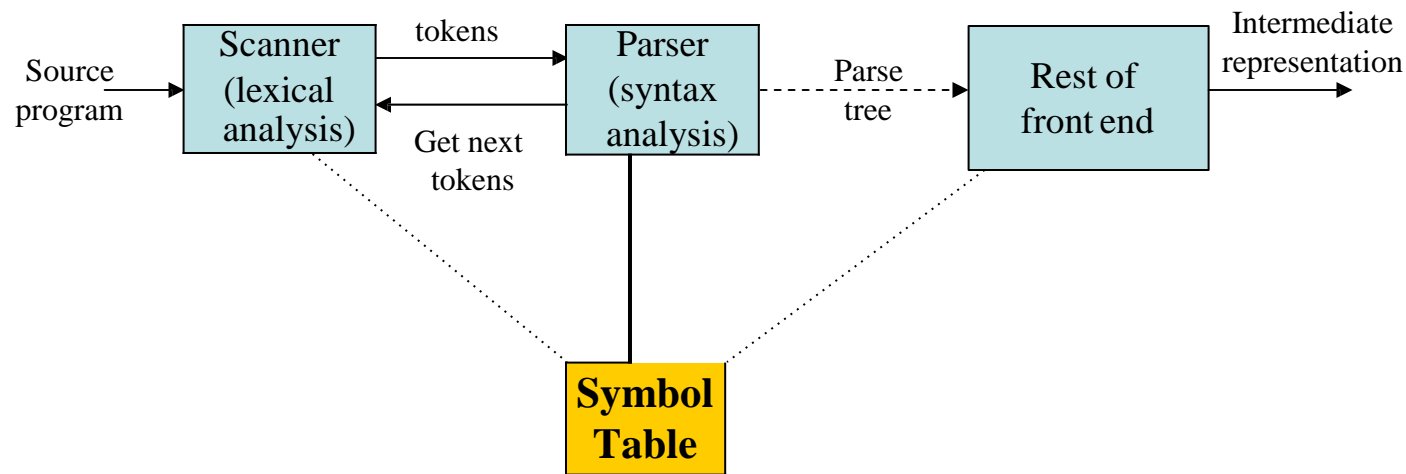
Language and Grammars

- Every (programming) language has precise rules
 - In English:
 - Subject Verb Object
 - In C
 - programs are made of functions
 - » Functions are made of statements etc.

Parsing

- A.K.A. Syntax Analysis
 - Recognize sentences in a language.
 - Discover the structure of a document/program.
 - Construct (implicitly or explicitly) a tree (called as a parse tree) to represent the structure.
 - The above tree is used later to guide translation.

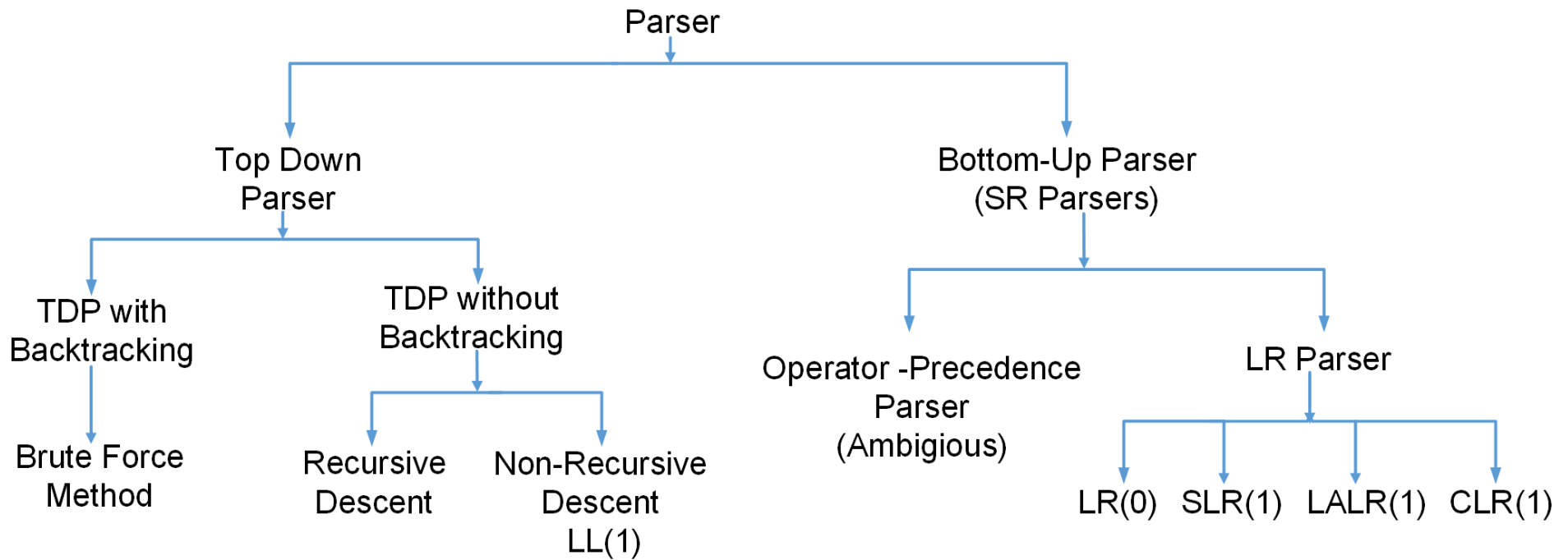
Role of the parser



- Verifies if the string of token can be generated from the grammar
- Error?
 - Report with a good descriptive, helpful message
 - Recover and continue parsing!
- Build a parse tree !!

Rest of Front End

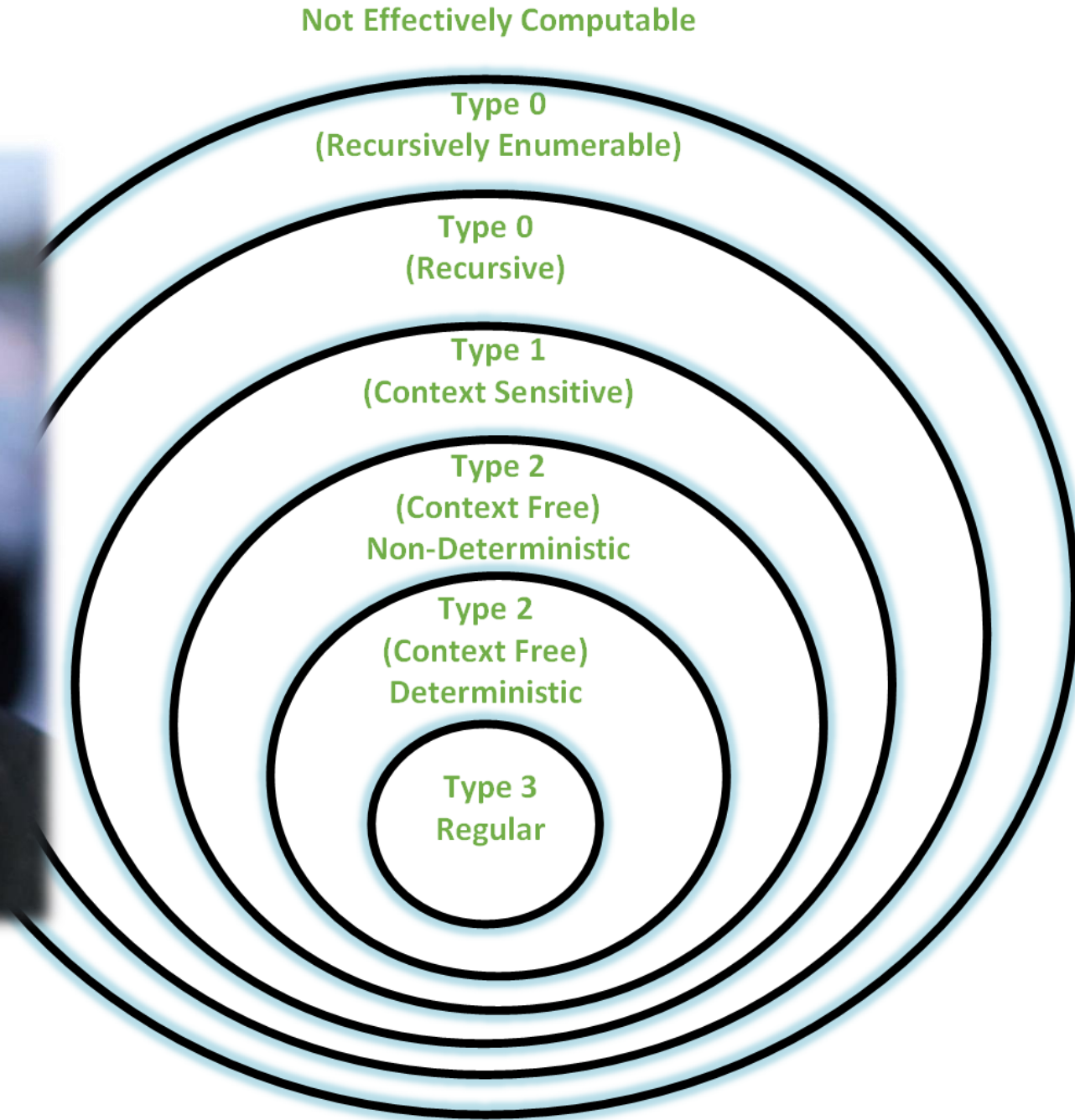
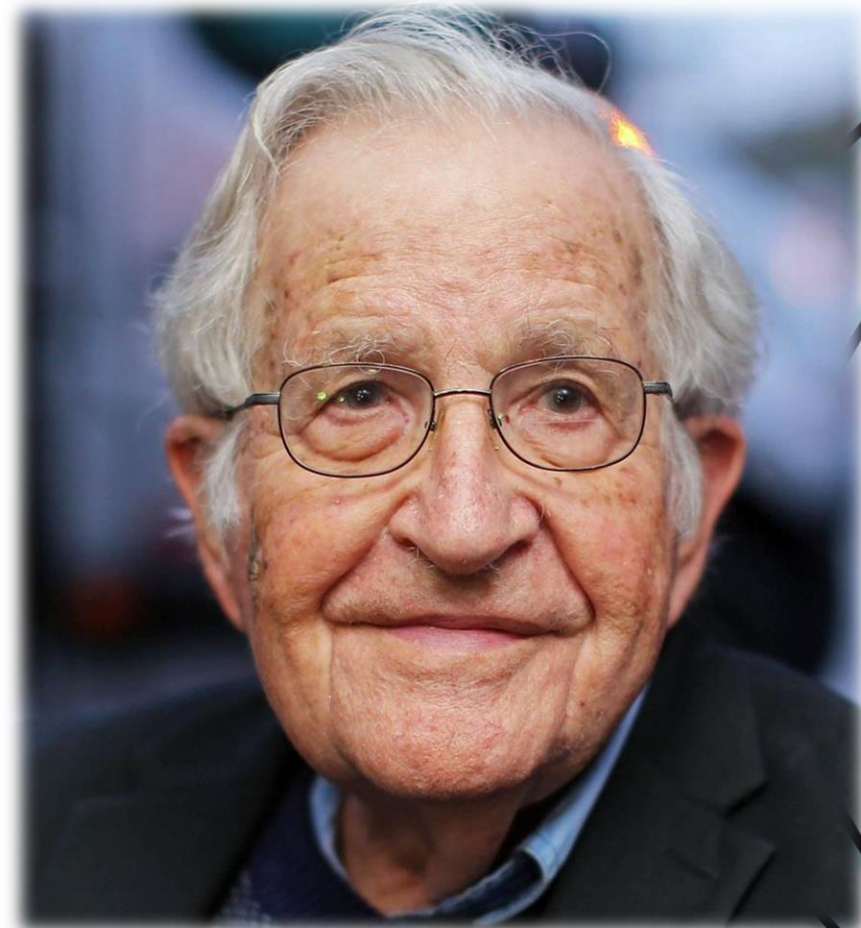
- Collecting token information
- Type checking
- Intermediate code generation



SR = Shift Reduce
Shift means pushing
Reduce means popping
So it uses a stack

LR = **L**eft-to-right, **R**ightmost derivation in reverse
In TDP without backtracking, LR and Non-Determinism is not accepted
Only Operator Precedence Parser can pass Ambiguous Grammars

Chomsky Hierarchy



Naom Chomsky introduced a mathematical model of grammar which is effective for writing computer languages

Type of Grammar	Grammar Accepted	Language Accepted	Automaton
Type-0	Unrestricted (free) Grammar	Recursively Enumerable Language	Turing Machine (Most flexible and most powerful)
Type-1	Context Sensitive Grammar	Context Sensitive Language	Linear Bounded Automaton
Type-2	Context Free Grammar	Context Free Language	Pushdown Automata
Type-3	Regular Grammar	Regular Language	FSA (Least flexible and least powerful)



Power in the sense, type of languages for which it can construct a grammar
Most of the programming language will be under CGFs

Grammar

- A 4-tuple $G = \langle V_N, V_T, P, S \rangle$ of a language $L(G)$

Where, -- V_N is set of nonterminal symbols used to write the grammar

-- V_T is the set of terminals (set of words in the language $L(G)$)

-- P is a set of production rules such as $a \rightarrow b$ where a and b are strings on $V_N \cup V_T$ and atleast one of a belongs to V_N

-- S is a special symbol V_N called the start symbol of the grammar

- Strings in language $L(G)$ are those derived from S by applying the production rules from P
- Examples:

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid ID$$

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid \varepsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \varepsilon$$

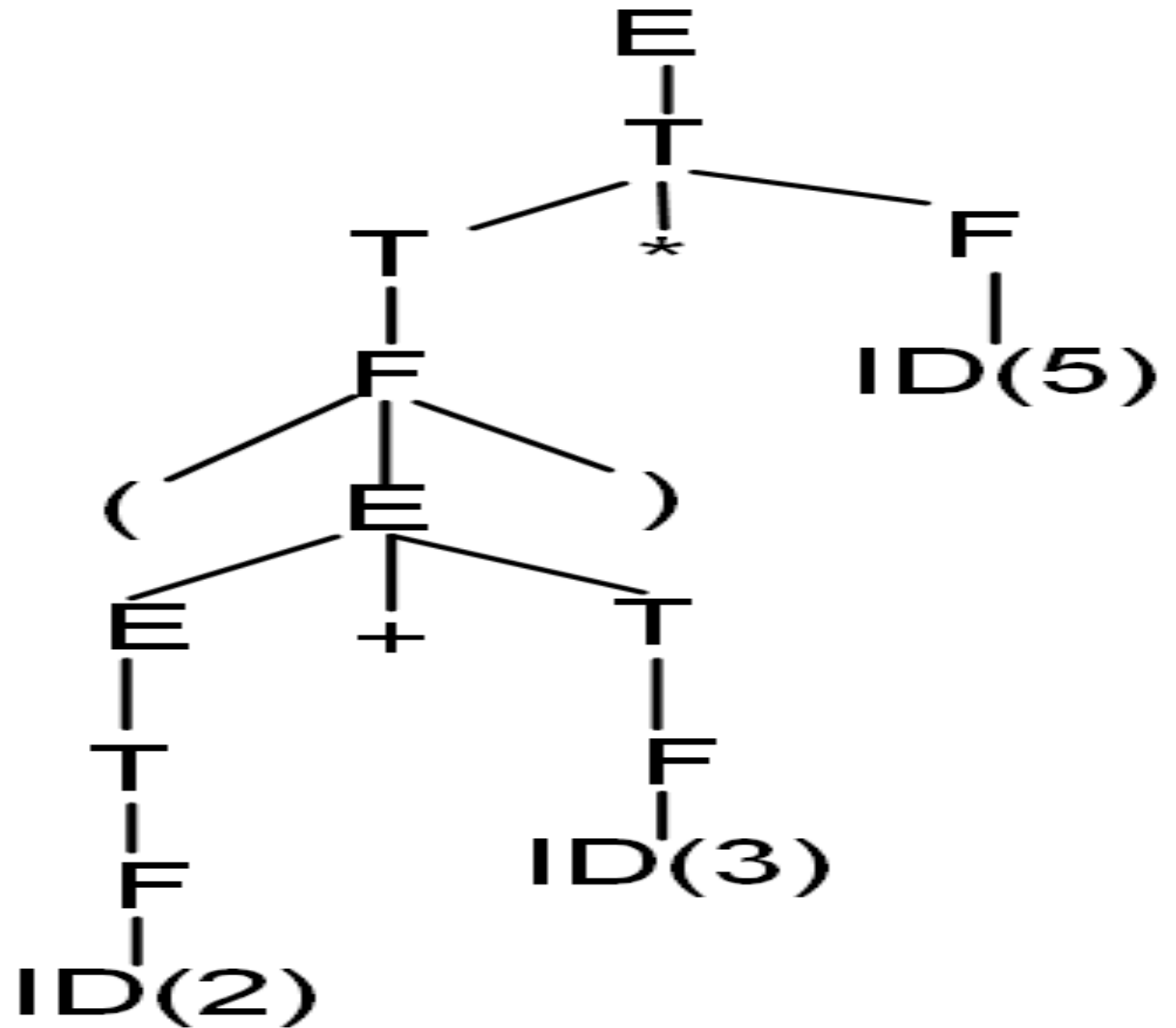
$$F \rightarrow (E) \mid ID$$

$(2+3)*5$

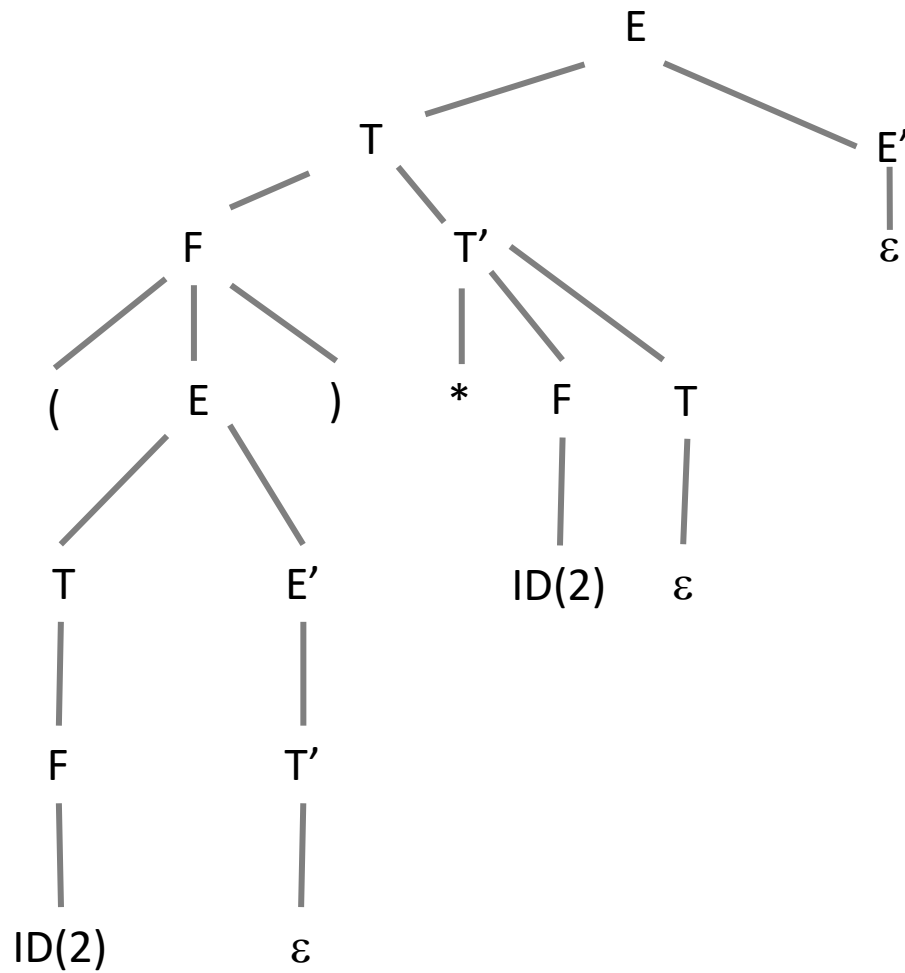
$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid ID$



$(2+3)*5$



$E \rightarrow TE'$
 $E' \rightarrow +TE' \mid \epsilon$
 $T \rightarrow FT'$
 $T' \rightarrow *FT' \mid \epsilon$
 $F \rightarrow (E) \mid ID$

$2+3*(4*5)$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid ID$

$E \rightarrow E + T$

$\rightarrow E + T * F$ [$T \rightarrow T * F$]

$\rightarrow E + T * (E)$ [$F \rightarrow (E)$]

$\rightarrow E + T * (T)$ [$E \rightarrow T$]

$\rightarrow E + T * (T * F)$ [$T \rightarrow T * F$]

$\rightarrow E + T * (T * id)$ [$F \rightarrow id$]

$\rightarrow E + T * (F * id)$ [$T \rightarrow F$]

$\rightarrow E + T * (id * id)$ [$F \rightarrow id$]

$\rightarrow E + F * (id * id)$ [$T \rightarrow F$]

$\rightarrow E + id * (id * id)$ [$F \rightarrow id$]

$\rightarrow T + id * (id * id)$ [$T \rightarrow F$]

$\rightarrow F + id * (id * id)$ [$F \rightarrow id$]

$\rightarrow id + id * (id * id)$

$\rightarrow 2 + 3 * (4 * 5)$

Errors in Programs

- **Lexical**

if x<1 then n y = 5:

“Typos”

- **Syntactic**

if ((x<1) & (y>5)) l ...

{ ... { }

- **Semantic**

if (x+5) then ...

Type Errors Undefined IDs, etc.

- **Logical Errors**

if (i<9) then ... Should be <= not <

Bugs

Compiler cannot detect Logical Errors

Goals for Handling Errors

- Compiler works in a fashion in which it produces all the error messages together. But doing this is very difficult as the parser is working based on automata where it can be difficult to come to a clean state from another state.
- Reporting the error precisely
- Recovering from each error quickly enough to detect the subsequent errors
- The compiler should add minimum overhead to the processing of correct programs

Error Detection

- Much responsibility on Parser
 - Many errors are syntactic in nature
 - Precision/ efficiency of modern parsing method
 - Detect the error as soon as possible
- Challenges for error handler in Parser
 - Report error clearly and accurately
 - Recover from error and continue..
 - Should be efficient in processing
- Good news is
 - Simple mechanism can catch most common errors
- Errors don't occur that frequently!!
 - 60% programs are syntactically and semantically correct
 - 80% erroneous statements have only 1 error, 13% have 2
 - Most error are trivial : 90% single token error
 - 60% punctuation, 20% operator, 15% keyword, 5% other error

Adequate Error Reporting is Not a Trivial Task

- Difficult to generate clear and accurate error messages.

Example

```
function foo () {  
    ...  
    if (...) {  
        ...  
    } else {  
        ...  
        ...  
    }  
    ...  
}  
<eof>
```

Missing } here

Not detected until here

Example

```
int myVarr;  
...  
x = myVar;  
...
```

Misspelled ID here

Not detected until here

Error Recovery

- After first error recovered
 - Compiler must go on!
 - Restore to some state and process the rest of the input
- **Error-Correcting Compilers**
 - Issue an error message
 - Fix the problem
 - Produce an executable

Example

```
Error on line 23: "myVarr" undefined.  
"myVar" was used.
```

May not be a good Idea!!

- Guessing the programmers intention is not easy!

Error Recovery May Trigger More Errors!

- Inadequate recovery may introduce more errors
 - Those were not programmers errors

- **Example:**

```
int myVar flag ;
```

```
...
```

```
x := flag;
```

```
...
```

```
...
```

```
while (flag==0)
```

```
...
```

Declaration of flag is discarded

Variable flag is undefined

Variable falg is undefined

Too many Error message may be obscuring

- May bury the real message
- Remedy:
 - allow 1 message per token or per statement
 - Quit after a maximum (e.g. 100) number of errors

Error Recovery Approaches: Panic Mode

- Discard input symbol one at a time until one of designated set of synchronization token is found.
- Example of synchronization tokens are: `;`, `}`. It depends on the programming language
- If discarding a semicolon is not sufficient, it discards the subsequent statements till it come to a closing brace.
- So, an entire block can be skipped if necessary
- The key...
 - Good set of synchronizing tokens
 - Knowing what to do then
- Advantage
 - Simple to implement
 - Does not go into infinite loop
 - Commonly used
- Disadvantage
 - May skip over large sections of source with some errors


Example

Skip to next occurrence of
`} end ;`
Resume by parsing
the next statement

Error Recovery Approaches: Phrase-Level Recovery

- Compiler corrects the program
- Replace a prefix of remaining input by some string that allows the parser to continue
 - by deleting or inserting tokens
 - ...so it can proceed to parse from where it was.
- The key...
 - Don't get into an infinite loop
 - ...constantly inserting tokens and never scanning the actual source
- Generally used for **error-repairing** compilers
 - Difficulty: Point of error detection might be much later the point of error occurrence

Example:
while (x==4) y:= a + b
Insert **do** to fix the statement



Error Recovery Approaches: Error Productions

- There are set of production rules which tells how a valid statement is generated
- Compiler designer can think the possible errors an user can make and add some extra production rules called “Error Productions”
- Augment the CFG with “Error Productions”
- Now the CFG accepts anything!
- Suppose, a production rule defines:
 $s \rightarrow \text{if Condition then Statement else Statement}$
- While designing a compiler, a common error the user can do is to forgetting the **then**. So, the compiler designer can add a error production:
 $s \rightarrow \text{if Condition Statement else Statement}$
- Used with...
 - LR (Bottom-up) parsing
 - Parser Generators

Error Recovery Approaches: Global Correction

- Theoretical Approach
- Find the minimum change to the source to yield a valid program with least cost correction
 - Insert tokens, delete tokens, swap adjacent tokens



- Global Correction Algorithm
 - Input:** grammatically incorrect input string x ; grammar G
 - Output:** grammatically correct string y
 - Algorithm:** converts $x \rightarrow y$ using minimum number changes (insertion, deletion etc.)
 - Another example, undeclared variable x
- Impractical algorithms - too time consuming

Context Free Grammars (CFG)

- A **context free grammar** is a formal model that consists of:

- **Terminals**

Keywords

Token classes

Punctuations

- **Non-terminals**

Any symbol appearing on the lefthand side of any rule

- **Start Symbol**

Usually the non-terminal on the lefthand side of the first rule

- **Rules (or “Productions”)**

BNF: Backus-Naur Form /

Backus Normal Form

Stmt ::= if Expr **then** Stmt **else** Stmt

Expression \rightarrow Expression + Term

Expression \rightarrow Expression – Term

Expression \rightarrow Term

Term \rightarrow Term * Factor

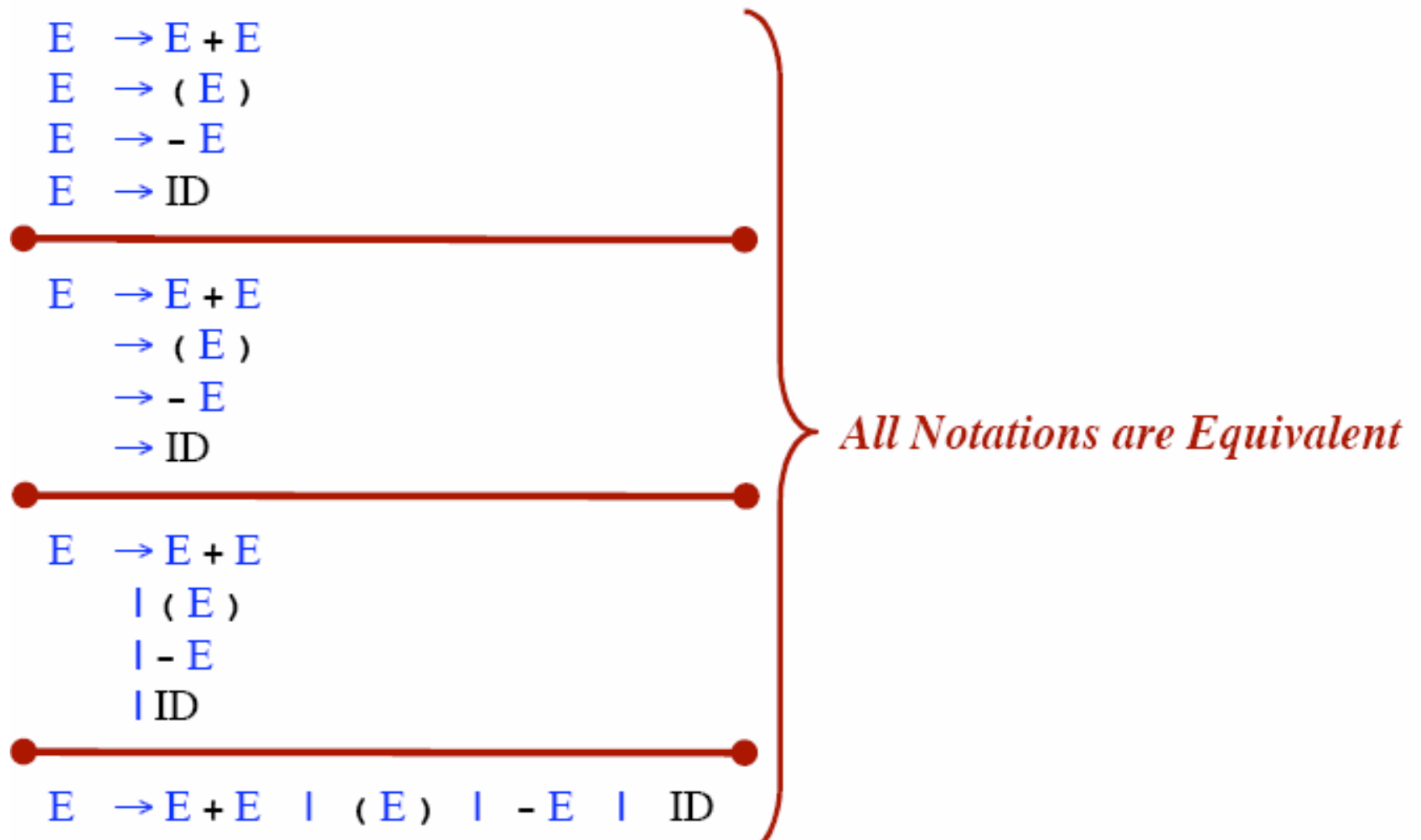
Term \rightarrow Term / Factor

Term \rightarrow Factor

Factor \rightarrow (Expression)

Factor \rightarrow **id**

Rule Alternative Notations



Notational Conventions

Terminals

a b c ...

Nonterminals

A B C ...

S

Expr

Grammar Symbols (Terminals or Nonterminals)

X Y Z U V W ...

Strings of Symbols

α β γ ...

A sequence of zero
Or more terminals
And nonterminals

Strings of Terminals

x y z u v w ...

Including ϵ

Examples

$A \rightarrow \alpha B$

A rule whose righthand side ends with a nonterminal

$A \rightarrow x \alpha$

A rule whose righthand side begins with a string of terminals (call it “x”)

Derivations

- Productions are treated as rewriting rules to generating a string

1. $E \rightarrow E + E$
2. $\rightarrow E * E$
3. $\rightarrow (E)$
4. $\rightarrow - E$
5. $\rightarrow ID$

A “Derivation” of “(id*id)”

$$E \Rightarrow (E) \Rightarrow (E * E) \Rightarrow (\underline{id} * E) \Rightarrow (\underline{id} * \underline{id})$$

“Sentential Forms”

A sequence of terminals and nonterminals in a derivation

(id*E)

Derivation

If $A \rightarrow \beta$ is a rule, then we can write

$$\underbrace{\alpha A \gamma}_{\uparrow} \Rightarrow \alpha \beta \gamma$$

*Any sentential form containing a nonterminal (call it A)
... such that A matches the nonterminal in some rule.*

Derives in zero-or-more steps \Rightarrow^*

$$E \Rightarrow^* (\underline{id} * \underline{id})$$

If $\alpha \Rightarrow^* \beta$ and $\beta \Rightarrow \gamma$, then $\alpha \Rightarrow^* \gamma$

Derives in one-or-more steps \Rightarrow^+

Given

G A grammar
S The Start Symbol

Define

$L(G)$ The language generated
 $L(G) = \{ w \mid S \Rightarrow^+ w \}$

“Equivalence” of CFG’s

If two CFG’s generate the same language, we say they are “**equivalent**.”

$$G_1 \approx G_2 \text{ whenever } L(G_1) = L(G_2)$$

In making a derivation...

Choose which nonterminal to expand

Choose which rule to apply

Leftmost Derivation

In a derivation... always expand the leftmost nonterminal.

$$\begin{aligned} & E \\ \Rightarrow & E + E \\ \Rightarrow & (E) + E \\ \Rightarrow & (E * E) + E \\ \Rightarrow & (\underline{id} * E) + E \\ \Rightarrow & (\underline{id} * \underline{id}) + E \\ \Rightarrow & (\underline{id} * \underline{id}) + \underline{id} \end{aligned}$$

- | | |
|----|-----------------------|
| 1. | $E \rightarrow E + E$ |
| 2. | $\rightarrow E * E$ |
| 3. | $\rightarrow (E)$ |
| 4. | $\rightarrow - E$ |
| 5. | $\rightarrow ID$ |

Let \Rightarrow_{LM} denote a step in a leftmost derivation ($\Rightarrow_{\text{LM}}^*$ means zero-or-more steps)

At each step in a leftmost derivation, we have

$$wA\gamma \Rightarrow_{\text{LM}} w\beta\gamma \quad \text{where } A \rightarrow \beta \text{ is a rule}$$

(Recall that w is a string of terminals.)

Each sentential form in a leftmost derivation is called a “**left-sentential form.**”

If $S \Rightarrow_{\text{LM}}^* \alpha$ then we say α is a “**left-sentential form.**”

Rightmost Derivation

In a derivation... always expand the rightmost nonterminal.

E
 $\Rightarrow E + E$
 $\Rightarrow E + \underline{id}$
 $\Rightarrow (E) + \underline{id}$
 $\Rightarrow (E * E) + \underline{id}$
 $\Rightarrow (E * \underline{id}) + \underline{id}$
 $\Rightarrow (\underline{id} * \underline{id}) + \underline{id}$

- | | |
|----|-----------------------|
| 1. | $E \rightarrow E + E$ |
| 2. | $\rightarrow E * E$ |
| 3. | $\rightarrow (E)$ |
| 4. | $\rightarrow - E$ |
| 5. | $\rightarrow ID$ |

Let \Rightarrow_{RM} denote a step in a rightmost derivation ($\Rightarrow_{\text{RM}}^*$ means zero-or-more steps)

At each step in a rightmost derivation, we have

$$\alpha A w \Rightarrow_{\text{RM}} \alpha \beta w \quad \text{where } A \rightarrow \beta \text{ is a rule}$$

(Recall that w is a string of terminals.)

Each sentential form in a rightmost derivation is called a “**right-sentential form.**”

If $S \Rightarrow_{\text{RM}}^* \alpha$ then we say α is a “**right-sentential form.**”

Parse Tree

Two choices at each step in a derivation...

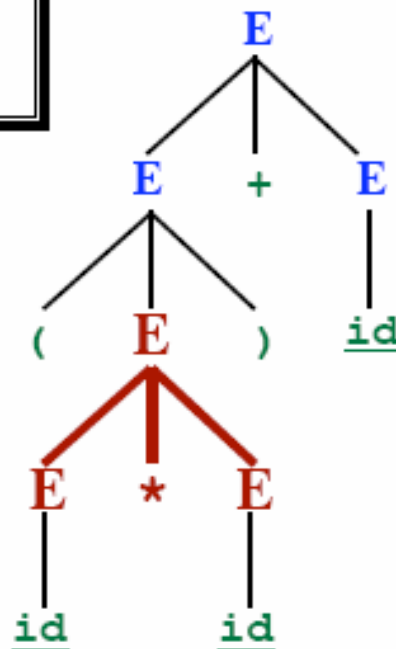
- Which non-terminal to expand
- Which rule to use in replacing it

The parse tree remembers only this

Leftmost Derivation:

E
 $\Rightarrow E + E$
 $\Rightarrow (E) + E$
 $\Rightarrow (E * E) + E$
 $\Rightarrow (id * E) + E$
 $\Rightarrow (id * id) + E$
 $\Rightarrow (id * id) + id$

1. $E \rightarrow E + E$
2. $\rightarrow E * E$
3. $\rightarrow (E)$
4. $\rightarrow - E$
5. $\rightarrow ID$



Parse Tree

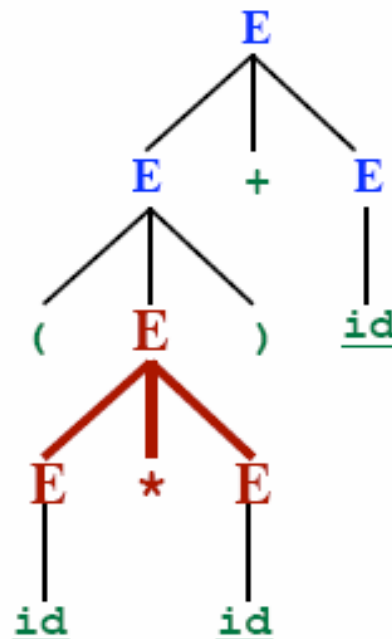
Two choices at each step in a derivation...

- Which non-terminal to expand
- Which rule to use in replacing it

The parse tree remembers only this

Rightmost Derivation:

E
 $\Rightarrow E + E$
 $\Rightarrow E + \underline{id}$
 $\Rightarrow (E) + \underline{id}$
 $\Rightarrow (E * E) + \underline{id}$
 $\Rightarrow (E * \underline{id}) + \underline{id}$
 $\Rightarrow (\underline{id} * \underline{id}) + \underline{id}$



1. $E \rightarrow E + E$
2. $\rightarrow E * E$
3. $\rightarrow (E)$
4. $\rightarrow - E$
5. $\rightarrow ID$

Parse Tree

Two choices at each step in a derivation...

- Which non-terminal to expand
- Which rule to use in replacing it

The parse tree remembers only this

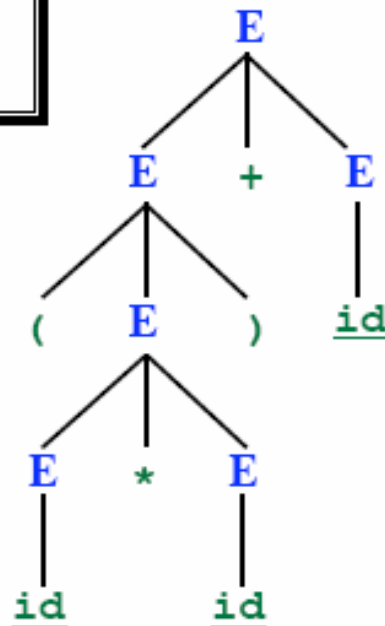
Leftmost Derivation:

E
 $\Rightarrow E + E$
 $\Rightarrow (E) + E$
 $\Rightarrow (E * E) + E$
 $\Rightarrow (\underline{id} * E) + E$
 $\Rightarrow (\underline{id} * \underline{id}) + E$
 $\Rightarrow (\underline{id} * \underline{id}) + \underline{id}$

Rightmost Derivation:

E
 $\Rightarrow E + E$
 $\Rightarrow E + \underline{id}$
 $\Rightarrow (E) + \underline{id}$
 $\Rightarrow (E * E) + \underline{id}$
 $\Rightarrow (E * \underline{id}) + \underline{id}$
 $\Rightarrow (\underline{id} * \underline{id}) + \underline{id}$

1. $E \rightarrow E + E$
2. $\rightarrow E * E$
3. $\rightarrow (E)$
4. $\rightarrow - E$
5. $\rightarrow ID$



Parse Tree

Given a leftmost derivation, we can build a parse tree.

Given a rightmost derivation, we can build a parse tree.

Leftmost Derivation of

(id*id)+id

Rightmost Derivation of

(id*id)+id

Same Parse Tree



Every parse tree corresponds to...

- A single, unique leftmost derivation
- A single, unique rightmost derivation

Ambiguity:

However, one input string may have several parse trees!!!

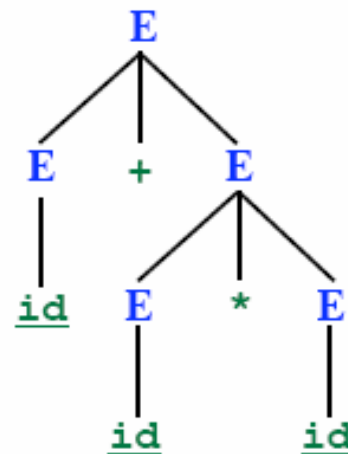
Therefore:

- Several leftmost derivations
- Several rightmost derivations

Ambiguous Grammar

Leftmost Derivation #1

E
 $\Rightarrow E + E$
 $\Rightarrow \underline{id} + E$
 $\Rightarrow \underline{id} + E * E$
 $\Rightarrow \underline{id} + \underline{id} * E$
 $\Rightarrow \underline{id} + \underline{id} * \underline{id}$

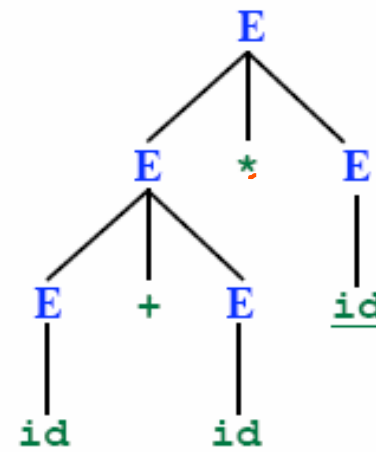


1. $E \rightarrow E + E$
2. $\rightarrow E * E$
3. $\rightarrow (E)$
4. $\rightarrow - E$
5. $\rightarrow ID$

Input: $id + id * id$

Leftmost Derivation #2

E
 $\Rightarrow E * E$
 $\Rightarrow E + E * E$
 $\Rightarrow \underline{id} + E * E$
 $\Rightarrow \underline{id} + \underline{id} * E$
 $\Rightarrow \underline{id} + \underline{id} * \underline{id}$



Ambiguous Grammar

- If we consider the precedence of this addition and multiplication,
 - Multiplication has higher precedence. So in the parse tree where addition is done at first will not be accepted
 - But if the precedence is not mentioned, both of them are correct

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid ID$

- This E-T-F grammar does not have any ambiguity
- Because, it says, for applying the rule first T has to be derived where T takes care of the multiplication function

Ambiguous Grammar

- More than one Parse Tree for some sentence.
 - The grammar for a programming language may be ambiguous
 - Need to modify it for parsing.
- Also: Grammar may be left recursive.
- Need to modify it for parsing.