CSE-361:Compiler Design

Parsing : Part-I

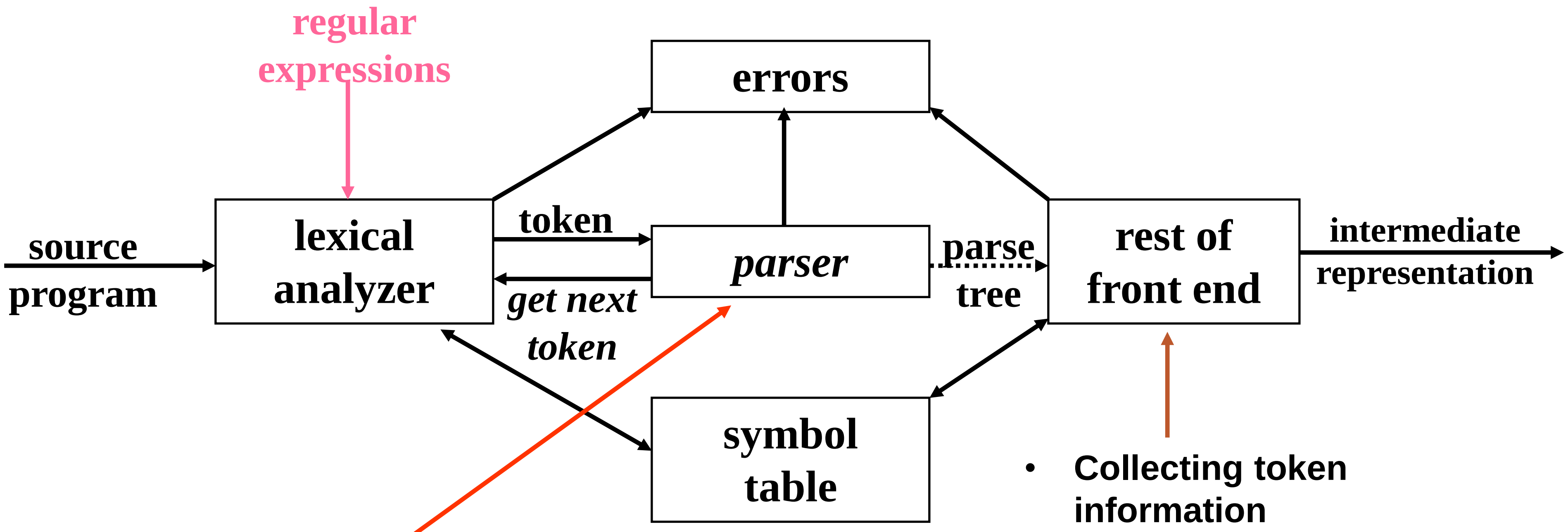
Language and Grammars

- 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

- 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.

Parsing During Compilation



- uses a grammar to check structure of tokens
- produces a parse tree
- syntactic errors and recovery
- recognize correct syntax
- report errors

- **Collecting token information**
- **Perform type checking**
- **Intermediate code generation**

Parsers

We categorize the parsers into two groups:

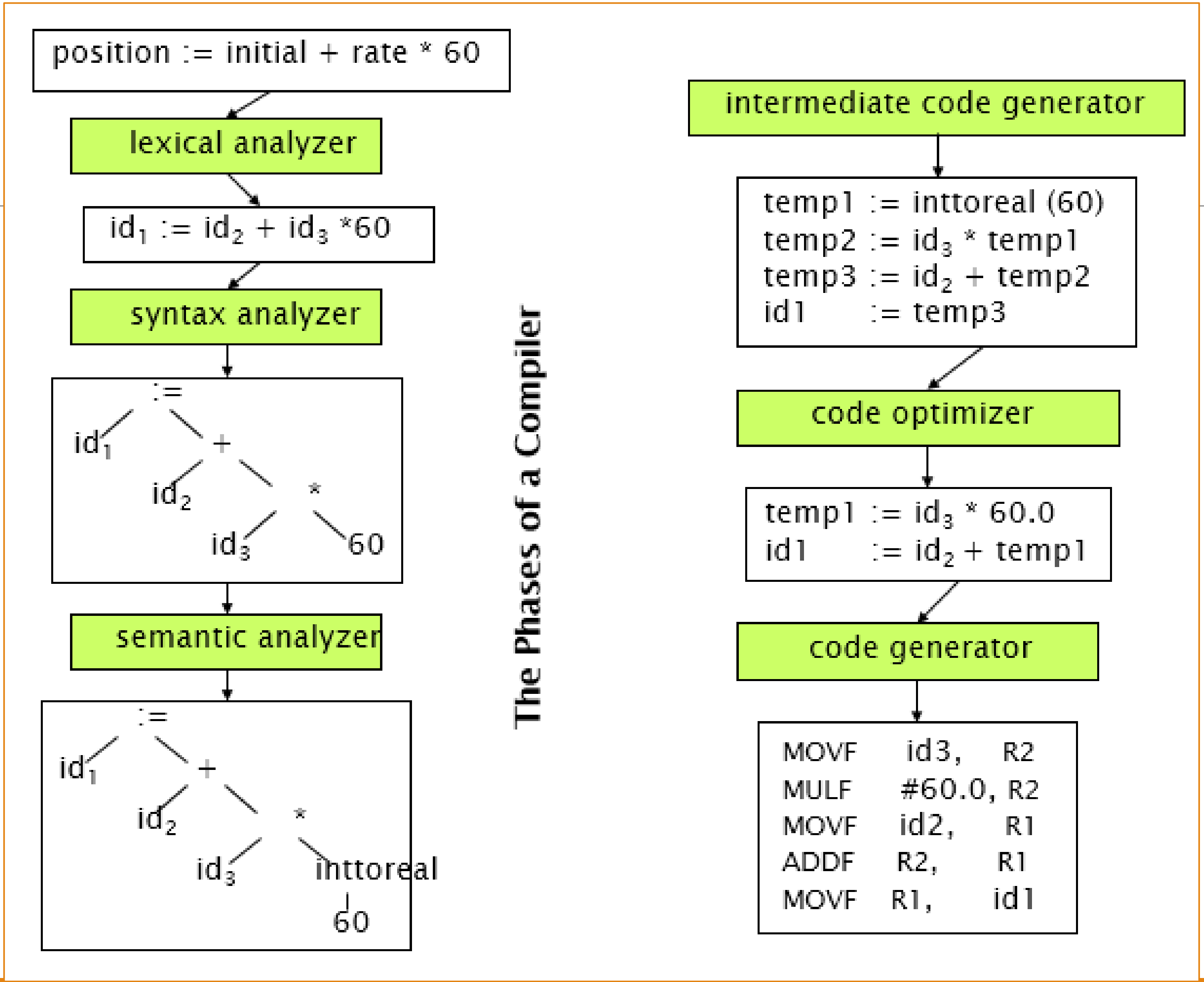
1. **Top- Down Parser**: The parse tree is created top to bottom, starting from the root

2. **Bottom-Up Parser**: The parse tree is created bottom to top, starting from the leaves

Both Top-Down and Bottom-Up parsers **scan the input from left to right (One symbol at a time)**

Efficient Top-Down and Bottom-Up parsers can be implement only for the sub-classes of context free grammars

- LL for top-down parsing
- LR for bottom-up parsing



The Phases of a Compiler
position := initial + rate * 60
lexical analyzer
$id_1 := id_2 + id_3 * 60$
syntax analyzer
:=
id_1
+
id_2
*
id_3
60
semantic analyzer
:=
id_1
+
id_2
*
id_3
inttoreal
60
intermediate code generator
temp1 := inttoreal (60)
temp2 := id_3 * temp1
temp3 := id_2 + temp2
id1 := temp3
code optimizer
temp1 := id_3 * 60.0
id1 := id_2 + temp1
code generator
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1

Errors in Programs

Syntax Error Identification / Handling

Recall typical error types:

1. **Lexical : Misspellings**

   if x<1 thenn y = 5;

2. **Syntactic : Omission, wrong order of tokens**

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

3. **Semantic : Incompatible types, undefined IDs**

   if (x+5) then

4. **Logical : Infinite loop / recursive call**

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

Majority of error processing occurs during syntax analysis

NOTE: Not all errors are identifiable !!

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 tokens until we see a “synchronizing” token.

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

- 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

Error Recovery Approaches: Phrase-Level Recovery

- Compiler corrects the program

  by deleting or inserting tokens

  ...so it can proceed to parse from where it was.

Example

while (x==4) y:= a + b

Insert do to fix the statement

- 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

Error Recovery Approaches: Error Productions

- Augment the CFG with “Error Productions”
- Now the CFG accepts anything!
- If “error productions” are used...

  Their actions:

  { print (“Error...”) }

- 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
  - 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 → y using minimum number changes (insertion, deletion etc.)
- Impractical algorithms - too time consuming

Syntactical Analysis

Each language definition has rules that describe the syntax of well formed programs.

- Format of the rules: **context-free grammars**
- Why not regular expressions/NFA's/DFA's?

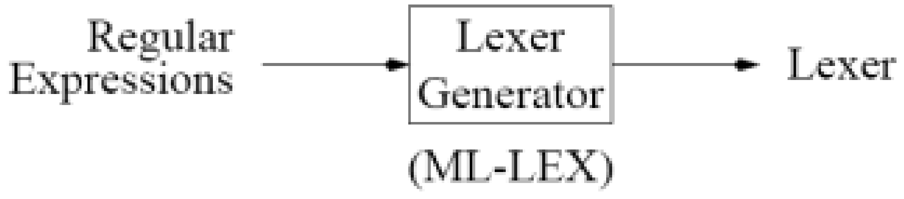
- Source program constructs have recursive structure:

digits = [0-9]+;
expr = digits | "(" expr "+" expr ")"

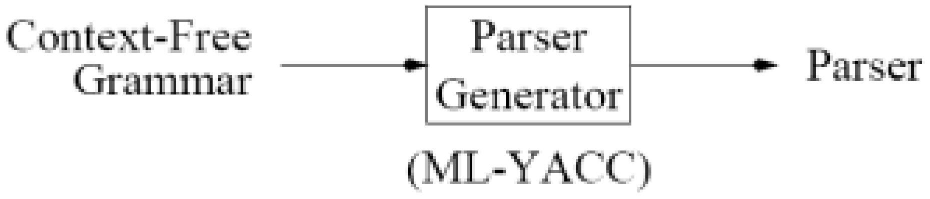
  - **Finite automata can't recognize recursive constructs**, **so cannot ensure expressions are well-bracketed:** a machine with *N* states cannot remember parenthesis—nesting depth greater than N
  - CFG's are more powerful, but also more costly to implement

CFG versus Regular Expression

Regular Expressions - describe lexical structure of tokens.



Context-Free Grammars - describe syntactic nature of programs.



CFG versus Regular Expression

Language: set of **strings**
String: finite sequence of **symbols** taken from finite **alphabet**

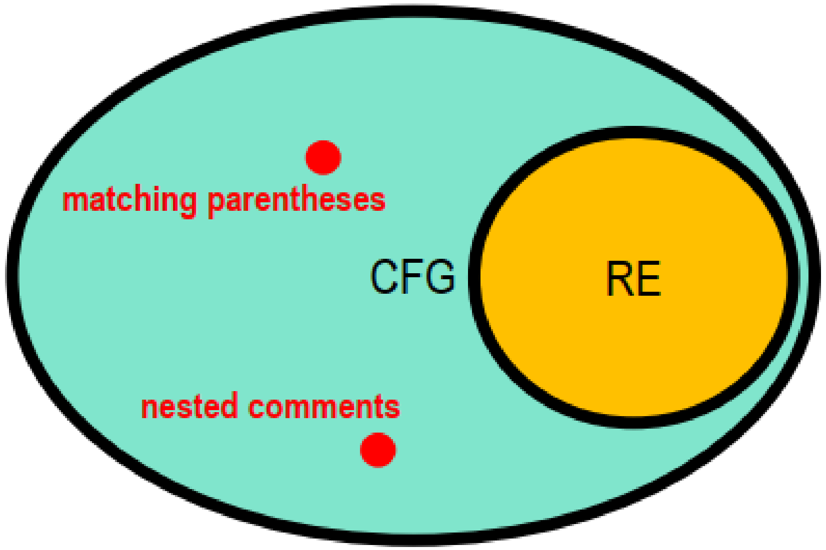
Regular expressions and CFG’s both describe languages, but over different alphabets

	Lexical analysis	Syntax analysis
symbols/alphabet	ASCII	token
strings	lists of tokens	lists of phrases
language	set of (legal) token sequences	set of (legal) phrase sequences (“programs”)

CFG versus Regular Expression

CFG's strictly more expressive than RE's:

Any language recognizable/generated by a RE can also be recognized/generated by a CFG, **but not vice versa**.



Also known as Backus-Naur Form (BNF, Algol 60)

Context Free Grammars (CFG)

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

  Keywords

  Token Classes

  Punctuation
- **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

Rule Alternative Notations

E → E + E
E → (E)
E → - E
E → ID

E → E + E
→ (E)
→ - E
→ ID

E → E + E
| (E)
| - E
| ID

E → E + E | (E) | - E | ID

All Notations are Equivalent

Notational Conventions

<u>Terminals</u>

- Lower-case letters early in the alphabet: *a, b, c*
- Operator symbols: +, -
- Punctuations symbols: parentheses, comma
- Boldface strings: **id** or **if**

<u>Nonterminals</u>:

- Upper-case letters early in the alphabet: *A, B, C*
- The letter *S* (start symbol)
- Lower-case italic names: *expr* or *stmt*

- Upper-case letters late in the alphabet, such as *X, Y, Z,* represent either nonterminals or terminals.
- Lower-case letters late in the alphabet, such as *u, v, …, z,* represent strings of terminals.

Notational Conventions

- Lower-case Greek letters, such as α, β, γ, represent strings of grammar symbols.
- Thus $A \rightarrow \alpha$ indicates that there is a single nonterminal A on the left side of the production and a string of grammar symbols α to the right of the arrow.
- If $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$, …., $A \rightarrow \alpha_k$ are all productions with A on the left, we may write:
- $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots. \mid \alpha_k$
- Unless otherwise started, the left side of the first production is the start symbol.

$$\mathbf{E \rightarrow E\ A\ E \mid (\ E\ ) \mid -E \mid id}$$

$$\mathbf{A \rightarrow + \mid - \mid * \mid / \mid \uparrow}$$

Summary of 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

A sequence of zero
Or more terminals
And nonterminals

α β γ …

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")

Context Free Grammars : A First Look

Production rules:

1. *assign_stmt* → ***id :=*** *expr ;*
2. *expr* → *expr operator term*
3. *expr* → *term*
4. *term* → ***id***
5. *term* → ***real***
6. *term* → ***integer***
7. *operator* → **+**
8. *operator* → **-**

Derivation: A sequence of grammar rule applications and substitutions that transform a starting non-term into a sequence of terminals / tokens.

Terminals: id real integer + - := ;
Nonterminals: *assign_stmt, expr, operator, term*
Start symbol: *assign_stmt*

Example Grammar: Simple Arithmetic Expressions

1. *expr* $\rightarrow$ *expr op expr*
2. *expr* $\rightarrow$ *(expr)*
3. *expr* $\rightarrow$ *- expr*
4. *expr* $\rightarrow$ *id*
5. *op* $\rightarrow$ +
6. *op* $\rightarrow$ -
7. *op* $\rightarrow$ *
8. *op* $\rightarrow$ /
9. *op* $\rightarrow$ $\uparrow$

9 Production rules

Terminals: id + - * / ↑ ( )
Nonterminals: *expr, op*
Start symbol: *expr*

Derivations

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

A "Derivation" of "(id*id)"

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

"Sentential Forms"

A sequence of terminals and nonterminals in a derivation

(id*E)

DERIVATIONS

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

$$\alpha A \gamma \Rightarrow \alpha\beta\gamma$$

Any sentential form containing a nonterminal (call it A)

... such that A matches the nonterminal in some rule.

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

$E \Rightarrow^*$ `(id*id)`

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

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

CFG Terminology

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."

$G_1 \approx G_2$ whenever $L(G_1) = L(G_2)$

In making a derivation...

Choose which nonterminal to expand

Choose which rule to apply

Derivation

Let's derive: *id = id + real – integer ;*

assign_stmt

→ *id = expr ;*

→ *id = expr operator term;*

→ *id = expr operator term operator term;*

→ *id = term operator term operator term;*

→ *id = id operator term operator term;*

→ *id = id + term operator term;*

→ *id = id + real operator term;*

→ *id = id + real - term;*

→ *id = id + real - integer;*

production rules:

assign_stmt → *id = expr ;*

expr → *expr operator term*

expr → *expr operator term*

expr → *term*

term → *id*

operator → +

term → *real*

operator → -

term → *integer*

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 ($\Rightarrow_{LM}^{*}$ means zero-or-more steps)

At each step in a leftmost derivation, we have

$w A \gamma \Rightarrow_{LM} w \beta \gamma$ where $A \rightarrow \beta$ 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 $S \Rightarrow_{LM}^{*} \alpha$ then we say α is a "left-sentential form."

Rightmost Derivation

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

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

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 ($\Rightarrow_{RM}{}^{*}$ means zero-or-more steps)

At each step in a rightmost derivation, we have

$\alpha A w \Rightarrow_{RM} \alpha\beta w$ where $A \rightarrow \beta$ 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 $S \Rightarrow_{RM}{}^{*} \alpha$ then we say α is a "right-sentential form."

Parse Tree

- A parse tree is a graphical representation of a derivation sequence of a sentential form.
- Tree nodes represent symbols of the grammar (nonterminals or terminals) and tree edges represent derivation steps.
- Inner nodes of a parse tree are non-terminal symbols.
- The leaves of a parse tree are terminal symbols.

Parse Tree

Two choices at each step in a derivation...

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

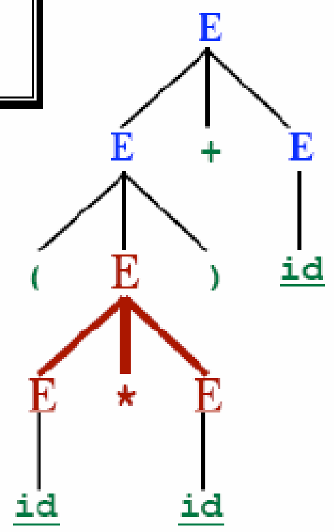
The parse tree remembers only this

Leftmost Derivation:

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

Input: *(id * id) + id*

1. E → E + E
2. → E * E
3. → (E)
4. → - E
5. → ID



PARSE TREE

Two choices at each step in a derivation...

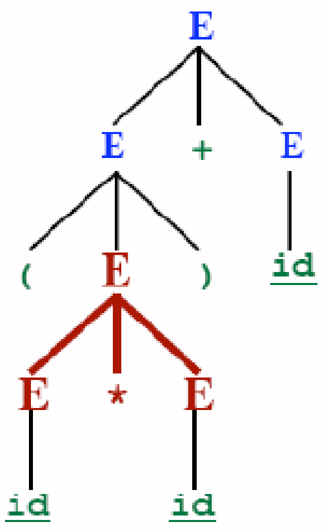
- Which non-terminal to expand
- Which rule to use in replacing it

The parse tree remembers only this

Input: *(id * id) + id*

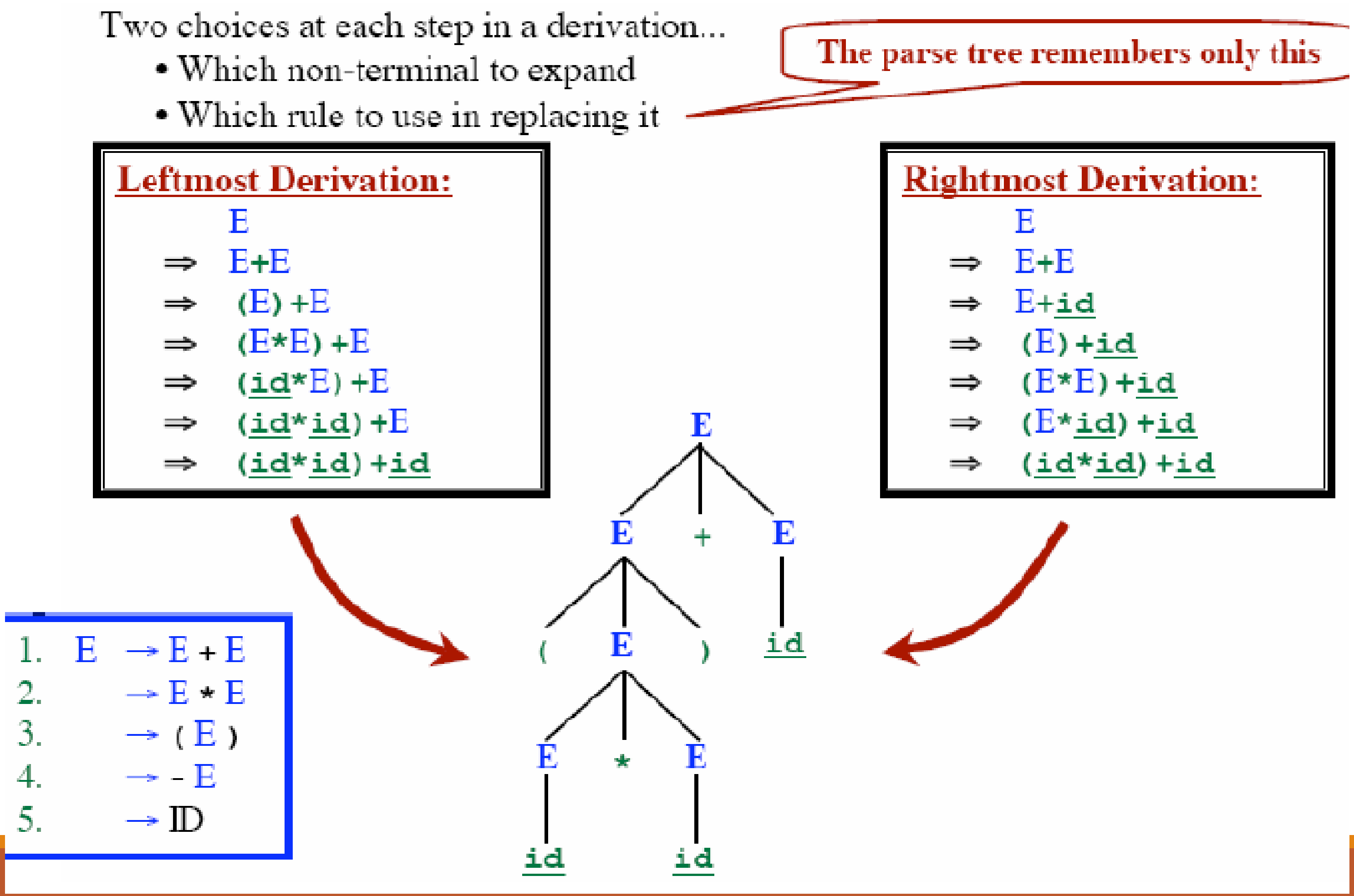
Rightmost Derivation:

E
⇒ E+E
⇒ E+id
⇒ (E)+id
⇒ (E*E)+id
⇒ (E*id)+id
⇒ (id*id)+id



1. E → E + E
2. → E * E
3. → (E)
4. → - E
5. → 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
⇒ E+E
⇒ (E)+E
⇒ (E*E)+E
⇒ (id*E)+E
⇒ (id*id)+E
⇒ (id*id)+id
Rightmost Derivation:
E
⇒ E+E
⇒ E+id
⇒ (E)+id
⇒ (E*E)+id
⇒ (E*id)+id
⇒ (id*id)+id
1. E → E + E
2. → E * E
3. → (E)
4. → - E
5. → ID

Parse Tree

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

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

Lefttmost 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

Ambiguous Grammar

Ambiguity:

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

Therefore:

- Several leftmost derivations
- Several rightmost derivations

A grammar that produces more than one parse tree for any input sentence is said to be an ambiguous grammar.

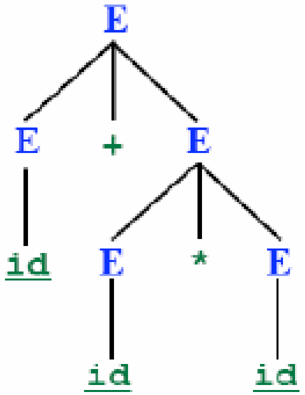
Ambiguous Grammar

1. E → E + E
2. → E * E
3. → (E)
4. → - E
5. → ID

Input: id+id*id

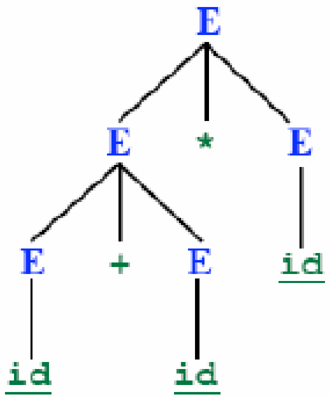
Leftmost Derivation #1

E
⇒ E+E
⇒ id+E
⇒ id+E*E
⇒ id+id*E
⇒ id+id*id



Leftmost Derivation #2

E
⇒ E*E
⇒ E+E*E
⇒ id+E*E
⇒ id+id*E
⇒ id+id*id

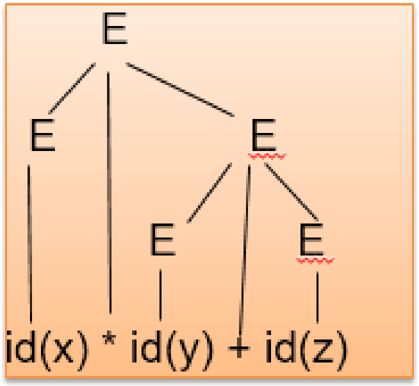


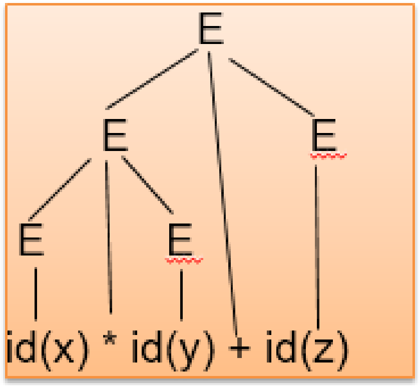
Two different parse trees!!
Which derivation of the parse tree is correct??

Ambiguous Grammar

- What about this grammar?

$$
\begin{aligned}
E ::= {} & E + E \\
| \; & E - E \\
| \; & E * E \\
| \; & E \,/\, E \\
| \; & \text{num} \\
| \; & \text{id}
\end{aligned}
$$





- Operators **+ - * /** have the same precedence!
- It is *ambiguous*: has more than one parse tree for the same input sequence (depending which derivations are applied each time)

Ambiguous Grammar

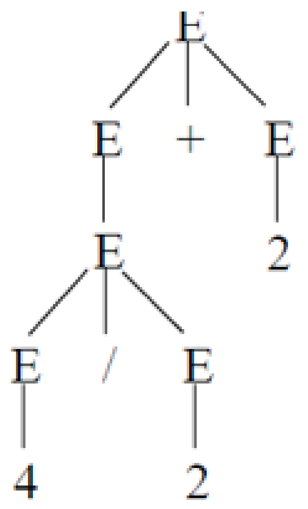
E::= E + E
E::= E - E
E::= E * E
E::= E / E

E::= (E)
E::= num
E::= id

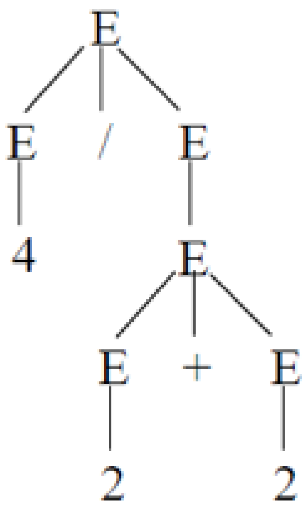
- ***Is this an ambiguous grammar?*** → **YES**
- **Example:**
  - Find a derivation for the expression: **4 / 2 + 2**

Ambiguous Grammar

E::= E + E
E::= E - E
E::= E * E
E::= E / E
E::= (E)
E::= num
E::= id

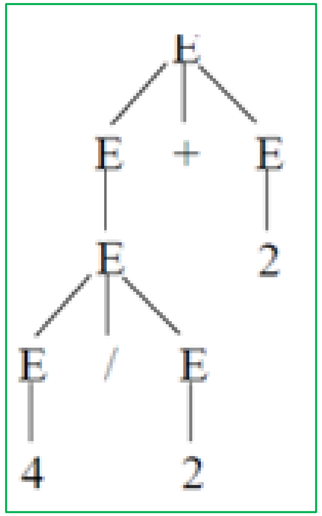


4 / 2 + 2



Problems of Ambiguous Grammar

- *Problem*: compilers use parse trees to interpret meaning of parsed expressions.
  - Different parse trees may have different meanings, resulting in different interpreted results.
  - For example, does 4 / 2 + 2 equal 4 or 1 ?



This parse tree gives the Right Answer.

4 / 2 + 2 = 4

Problems of Ambiguous Grammar

- It is often possible to transform an ambiguous grammar into an equivalent unambiguous grammar.
- In our grammar,
  - * has higher precedence than +
  - each operator associates to the left

The Ambiguous Grammar does not consider the Precedence and Associativity

SOLUTION: REMOVING AMBIGUITY

- For the most parsers, the grammar must be unambiguous.
- unambiguous grammar

  → unique selection of the parse tree for a sentence
- *We should eliminate the ambiguity in the grammar during the design phase of the compiler.*
- An **unambiguous** grammar should be written to eliminate the ambiguity.
- We have to prefer one of the parse trees of a sentence (generated by an ambiguous grammar) to disambiguate that grammar to restrict to this choice.

Ambiguous Grammar: How to Solve Associativity Problem?

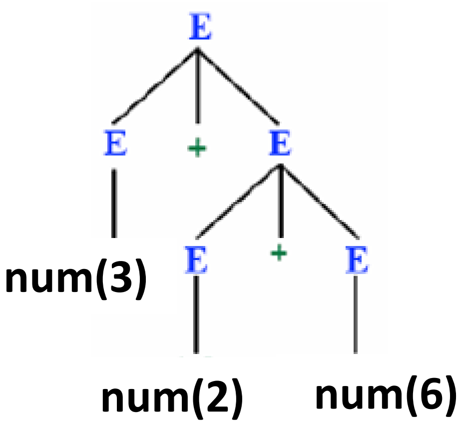
Operators with

Input: 3 + 2 + 6

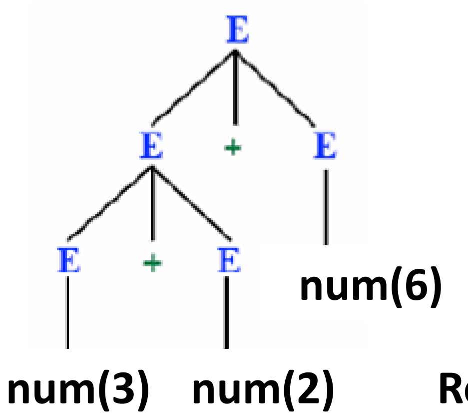
Ambiguous Grammar

1. E → E + E
2. E → E * E
3. E → num

Parse Tree -1



Parse Tree -2



- **Two different parse trees!!**
- **According to the Grammar both are correct.**
- **But Parse Tree-1 gives WRONG answer and Parse Tree-2 gives RIGHT answer.**

Removing the associativity problem from the Grammar:

1. E → E + num
2. E → E * num
3. E→ num

Still it is an Ambiguous Grammar!!

- ✓ **Operators with same precedence must be resolved by Associativity**
- ✓ **Some operators have left associativity (+, -, *, /) and some operators have right associativity (^)**

Ambiguous Grammar:
How to Solve Associativity Problem?(2)

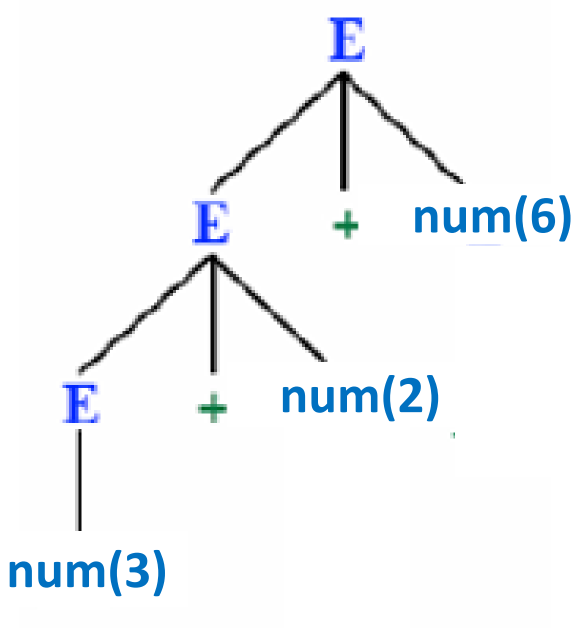
Input: 3 + 2 + 6

Removing the associativity problem from the Grammar:

1. E → E + num
2. E → E * num
3. E → num

Still it is an Ambiguous Grammar!!

Parse Tree



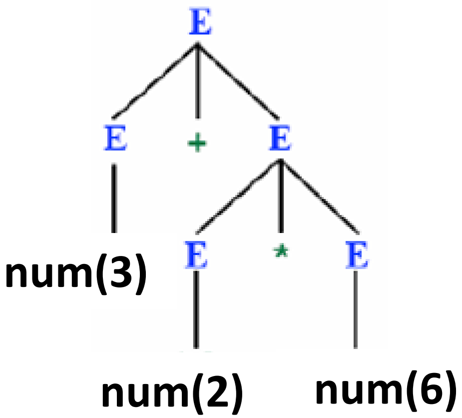
Ambiguous Grammar:
How to Solve Precedence Problem?

Input: 3 + 2 * 6

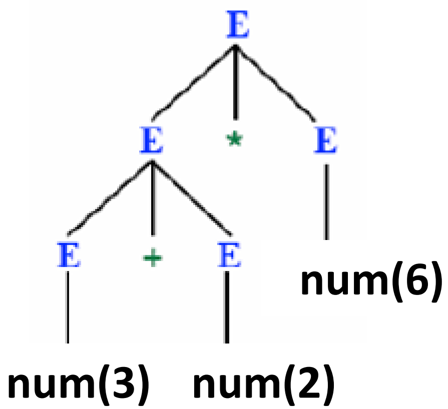
Ambiguous Grammar

1. E → E + E
2. E → E * E
3. E → num

Parse Tree -1



Parse Tree -2



- Two different parse trees!!
- According to the Grammar both are correct.
- But Parse Tree-1 gives RIGHT answer and Parse Tree-2 gives WRONG answer.

After Conversion to Unambiguous Grammar:

1. E → E + T | T
2. T → T * F | F
3. F → num

- ✓ Lower precedence operation rules should be declared in the upper level in the Grammar
- ✓ Higher precedence operation rules should be declared in the lower level in the Grammar

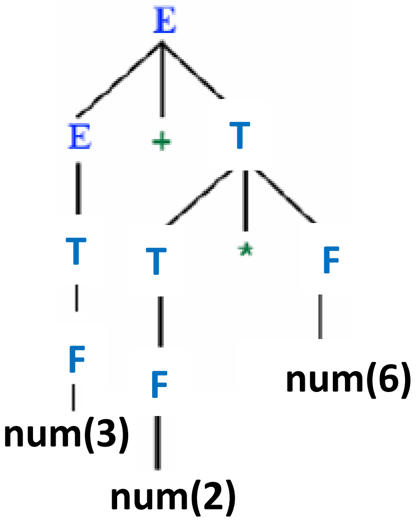
Ambiguous Grammar:
How to Solve Precedence Problem?(2)

Input: 3 + 2 * 6

After Conversion to Unambiguous Grammar:

1. E → E + T | T
2. T → T * F | F
3. F → num

Parse Tree

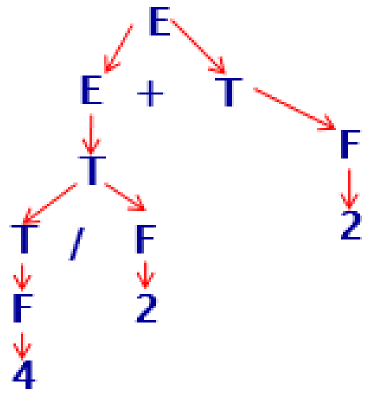


UNAMBIGUOUS GRAMMAR

E::= E + E
E::= E - E
E::= E * E
E::= E / E
E::= (E)
E::= num
E::= id

Ambiguous to unambiguous grammar

E ::= E + T
E ::= E - T
E ::= T
T::= T * F
T::= T / F
T::= F
F ::= id
F ::= num
F ::= (E)



Reading Materials

- Chapter -4 of your Text book:
  - Compilers: Principles, Techniques, and Tools

The End