Compiler Design

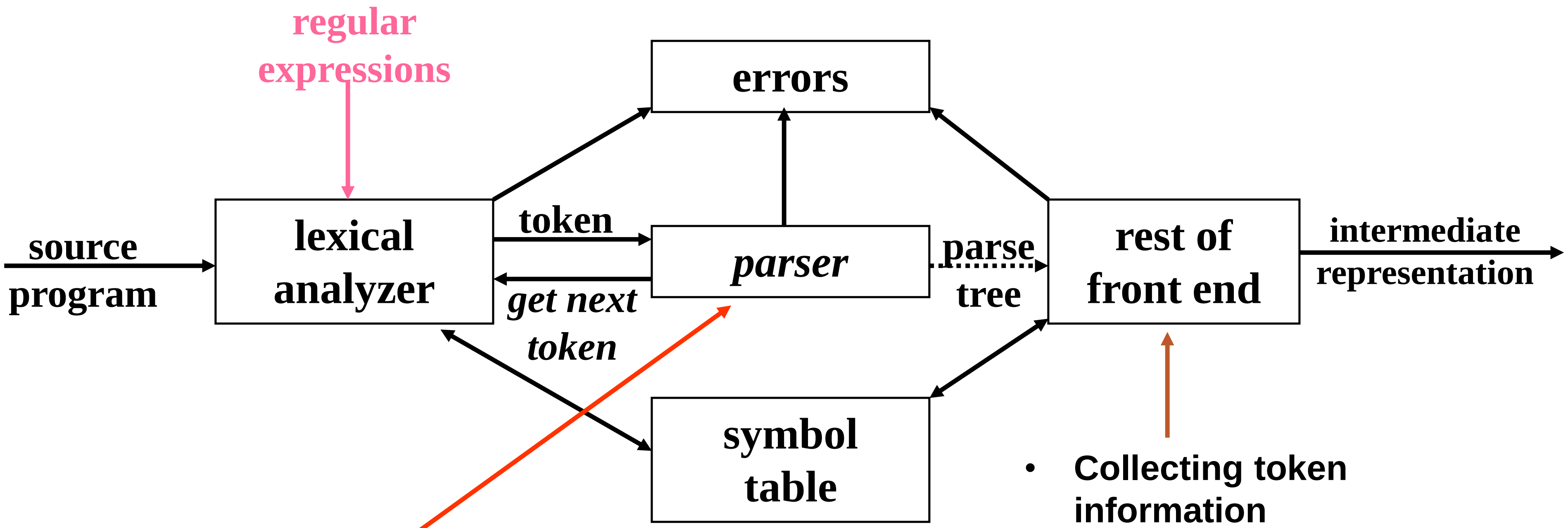
Language and Grammars

- 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.

Parsing During Compilation



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

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

Errors in Programs

- **Lexical**
  - if x<1 thenn y = 5;
  - "Typos"
- **Syntactic**
  - if ((x<1) & (y>5))) ...
  - { ... { ... _ ... }
- **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

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 {
   ...
                    Missing } here

...
}
<eof>                   Not detected until here
```

Example

```
int myVarr;
...                     Misspelled ID here
x = myVar;
...                     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

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 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 implemented only for sub-classes of context-free grammars.
  - LL for top-down parsing
  - LR for bottom-up parsing

Context Free Grammars (CFG)

A context-free grammar has four components: G = (V, Σ, P, S)

- ✓A set of **non-terminals** (V). Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.
- ✓A set of tokens, known as **terminal symbols** (Σ). Terminals are the basic symbols from which strings are formed.
- ✓A set of **productions** (P). The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of a **non-terminal** called the left side of the production, an arrow, and a sequence of tokens and/or **on- terminals**, called the right side of the production.
- ✓One of the non-terminals is designated as the **start symbol** (S); from where the production begins.

Example of CFG:

G = (V, Σ, P, S)Where:

V = { Q, Z, N }

Σ = { 0, 1 }

P = { Q → Z | Q → N | Q → Ɛ | Z → 0Q0 |

N → 1Q1 }

S = { Q }

- This grammar describes palindrome language, such as: 1001, 11100111, 00100, 1010101, 11111, etc.

Rule Alternative Notations

$E \rightarrow E + E$
$E \rightarrow (E)$
$E \rightarrow - E$
$E \rightarrow ID$

$E \rightarrow E + E$
$\quad \rightarrow (E)$
$\quad \rightarrow - E$
$\quad \rightarrow ID$

$E \rightarrow E + E$
$\quad \mid (E)$
$\quad \mid - E$
$\quad \mid ID$

$E \rightarrow E + E \mid (E) \mid - E \mid ID$

All Notations are Equivalent

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

- A derivation is basically a sequence of production rules, in order to get the input string. During parsing, we take two decisions for some sentential form of input:
- Deciding the non-terminal which is to be replaced.
- Deciding the production rule, by which, the non-terminal will be replaced.

To decide which non-terminal to be replaced with production rule, we can have two options.

Derivations

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

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

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

"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

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

Any sentential form containing a nonterminal (call it A*)*

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

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

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

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."

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

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 ($\Rightarrow_{LM}{}^*$ 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	→ E + E
2.		→ E * E
3.		→ (E)
4.		→ - E
5.		→ 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...

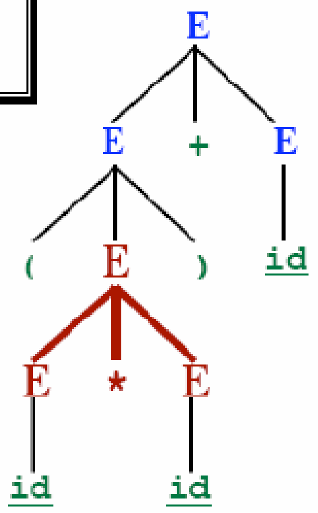
- Which non-terminal to expand
- Which rule to use in replacing it

The parse tree remembers only this

Leftmost Derivation:

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

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



Parse Tree

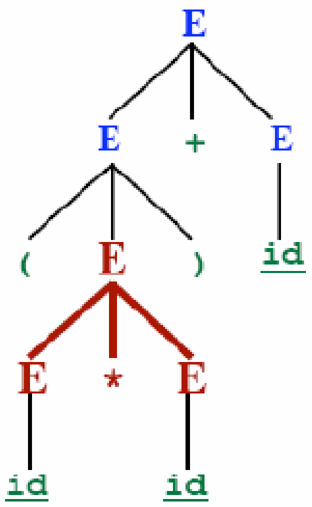
Two choices at each step in a derivation...

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

The parse tree remembers only this

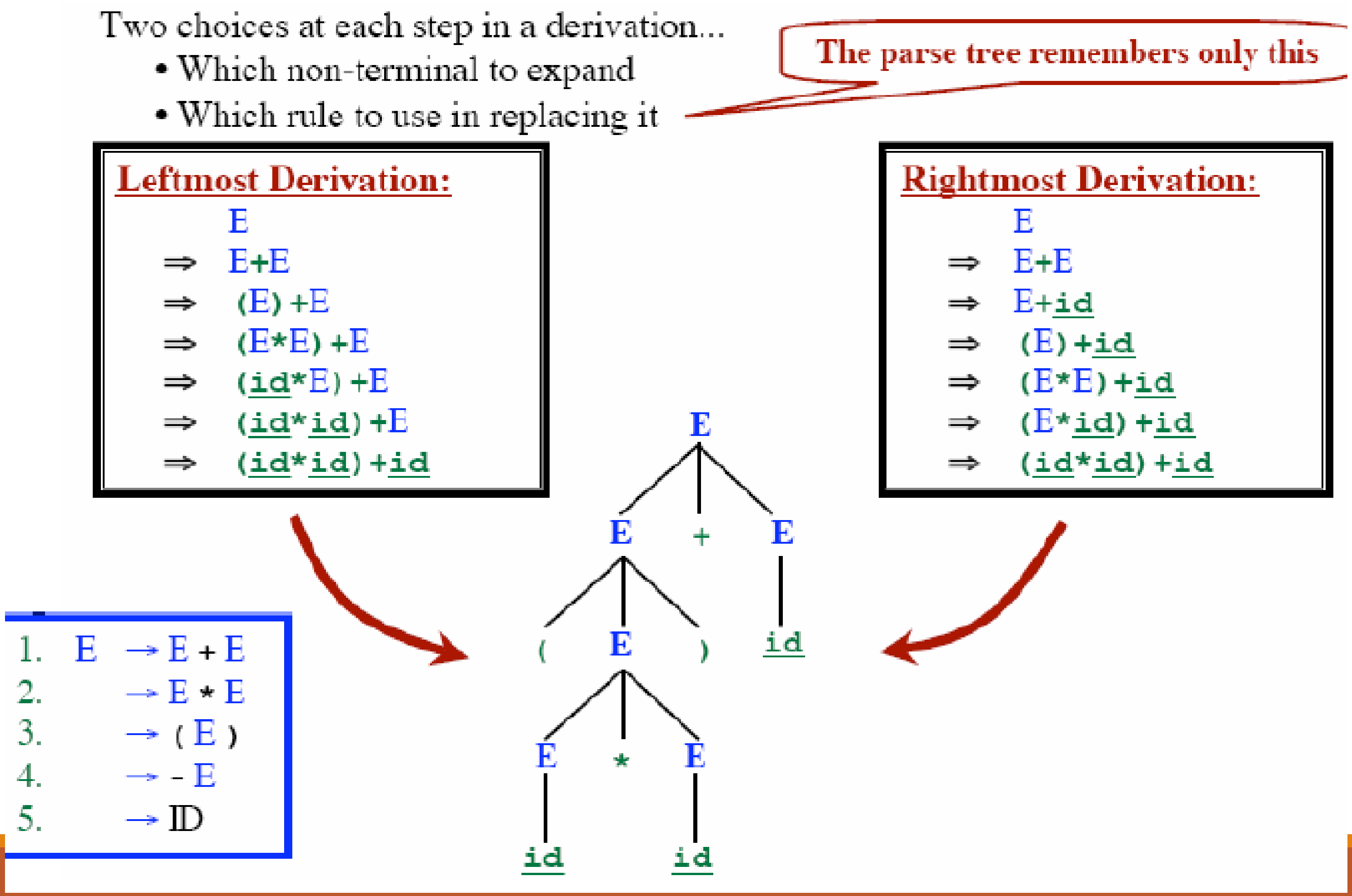
Rightmost Derivation:

E
⇒ 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
E
E + E
(E) id
E * E
id 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.

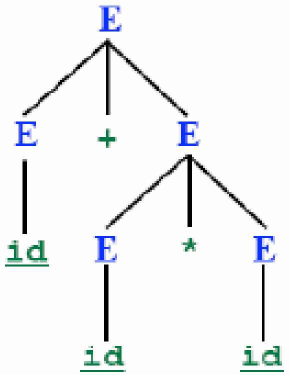
Ambiguous Grammar

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

Input: id+id*id

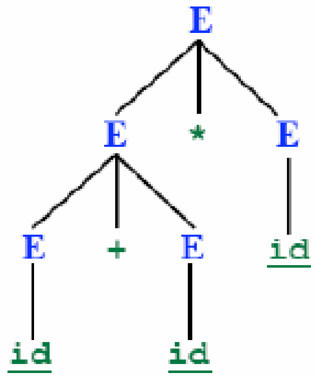
Leftmost Derivation #1

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



Ambiguous Grammar

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

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

Ambiguous Grammar

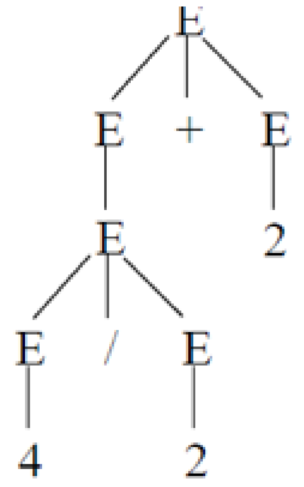
- Why are ambiguous grammars problematic?

  $(4 / 2) + 2 = 4$ or $4 / (2 + 2) = 1$

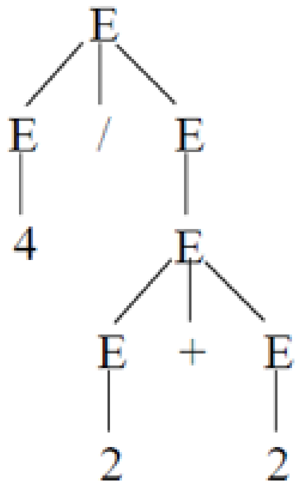
- It is often possible to transform an ambiguous grammar into an equivalent unambiguous grammar.
- In our grammar,
  - * has higher precedence than +
  - each operator associates to the left

Ambiguous Grammar

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



4 / 2 + 2



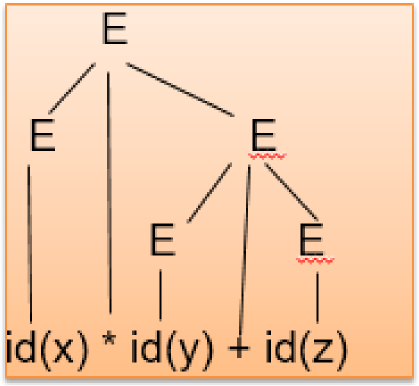
Ambiguous Grammar

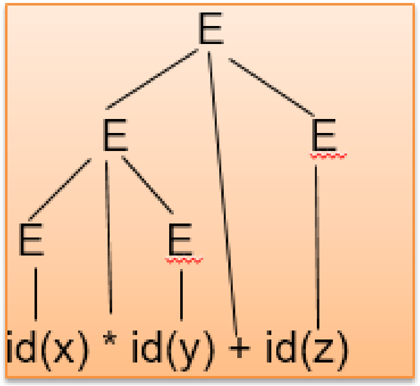
- For the most parsers, the grammar must be unambiguous.
- unambiguous grammar
  - → unique selection of the parse tree for a sentence
- *We should eliminate the ambiguity in the grammar during the design phase of the compiler.*
- An **unambiguous** grammar should be written to eliminate the ambiguity.
- 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

- What about this grammar?

```
E ::= E + E
    | E - E
    | E * E
    | E / E
    | num
    | id
```





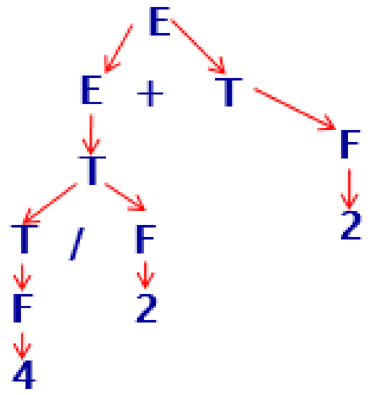
- Operators **+ - * /** have the same precedence!
- It is *ambiguous*: has more than one parse tree for the same input sequence (depending which derivations are applied each time)

Unambiguous Grammar

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

Ambiguous to unambiguous grammar

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



Predictive Parsing

- The goal is to construct a top-down parser that never backtracks
- Always leftmost derivations
- We must transform a grammar in two ways:
  - eliminate left recursion
  - perform left factoring
- These rules eliminate most common causes for backtracking although they do not guarantee a completely backtrack-free parsing

Left Recursion: Infinite Looping problem

A grammar is left-recursive if it has a non-terminal A, such that there is a derivation :

$$A \stackrel{+}{\Rightarrow} A\alpha, \quad \text{for some } \alpha.$$

Top-Down parsing can't reconcile this type of grammar, since it could consistently make choice which wouldn't allow termination.

$$A \Rightarrow A\alpha \Rightarrow A\alpha\alpha \Rightarrow A\alpha\alpha\alpha \ldots \text{etc.} \quad A \rightarrow A\alpha \mid \beta$$

So we have to convert our left-recursive grammar into an equivalent grammar which is not left-recursive.

The left-recursion may appear in a single step of the derivation (*immediate left-recursion*), or may appear in more than one step of the derivation.

Immediate Left Recursion

- $A \to A\, \alpha \mid \beta$ where β does not start with A

$\Downarrow$ eliminate immediate left recursion

$A \to \beta\, A'$ where A' is a new nonterminal

$A' \to \alpha\, A' \mid \varepsilon$ an equivalent grammar

More General (but still immediate):

$$A \to A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots \mid \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots$$

Transform into:

$$A \to \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \ldots$$
$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \ldots \mid \varepsilon$$

Immediate Left Recursion Elimination: example

Our Example :

$E \rightarrow E + T \mid T \quad\longrightarrow\quad \begin{cases} E \rightarrow TE' \\ E' \rightarrow + TE' \mid \in \end{cases}$

$T \rightarrow T * F \mid F \quad\longrightarrow\quad \begin{cases} T \rightarrow FT' \\ T' \rightarrow * FT' \mid \in \end{cases}$

$F \rightarrow (E) \mid id \quad\longrightarrow\quad F \rightarrow (E) \mid id$

Left Recursion in More Than One Step

- A grammar cannot be immediately left-recursive, but it still can be left-recursive.
- By just eliminating the immediate left-recursion, we may not get a grammar which is not left-recursive.

Example:

$S \rightarrow Af \mid b$

$A \rightarrow Ac \mid Sd \mid e$

Is A left recursive? Yes.

Is S left recursive? Yes, but not immediate left recursion. $S \Rightarrow Af \Rightarrow Sdf$

Left Recursion in More Than One Step: Elimination

Approach:

Look at the rules for S only (ignoring other rules)... No left recursion.

Look at the rules for A...

Do any of A's rules start with S? Yes.

$A \rightarrow Sd$

Get rid of the S. Substitute in the righthand sides of S.

$A \rightarrow Afd \mid bd$

The modified grammar:

$S \rightarrow Af \mid b$

$A \rightarrow Ac \mid Afd \mid bd \mid e$

Now eliminate immediate left recursion involving A.

$S \rightarrow Af \mid b$

$A \rightarrow bdA' \mid eA'$

$A' \rightarrow cA' \mid fdA' \mid \varepsilon$

Left Recursion in More Than One Step: Elimination

The Original Grammar:

$S \rightarrow Af \mid b$

$A \rightarrow Ac \mid Sd \mid Be$

$B \rightarrow Ag \mid Sh \mid k$

So Far:

$S \rightarrow Af \mid b$

$A \rightarrow bdA' \mid BeA'$

$A' \rightarrow cA' \mid fdA' \mid \varepsilon$

Left Recursion in More Than One Step: Elimination

The Original Grammar:

$S \rightarrow Af \mid b$
$A \rightarrow Ac \mid Sd \mid Be$
$B \rightarrow Ag \mid Sh \mid k$

So Far:

$S \rightarrow Af \mid b$
$A \rightarrow bdA' \mid BeA'$
$A' \rightarrow cA' \mid fdA' \mid \varepsilon$
$B \rightarrow Ag \mid Sh \mid k$

Look at the B rules next; Does any righthand side start with "S"?

Left Recursion in More Than One Step: Elimination

The Original Grammar:

$S \rightarrow Af \mid b$

$A \rightarrow Ac \mid Sd \mid Be$

$B \rightarrow Ag \mid Sh \mid k$

So Far:

$S \rightarrow Af \mid b$

$A \rightarrow bdA' \mid BeA'$

$A' \rightarrow cA' \mid fdA' \mid \varepsilon$

$B \rightarrow Ag \mid Afh \mid bh \mid k$

Substitute, using the rules for "S"

$Af\ldots \mid b\ldots$

Left Recursion in More Than One Step: Elimination

The Original Grammar:

S → Af | b
A → Ac | Sd | Be
B → Ag | Sh | k

So Far:

S → Af | b
A → bdA' | BeA'
A' → cA' | fdA' | ε
B → Ag | Afh | bh | k

Does any righthand side start with "A"?

Left Recursion in More Than One Step: Elimination

The Original Grammar:

S → Af | b
A → Ac | Sd | Be
B → Ag | Sh | k

So Far:

S → Af | b
A → bdA' | BeA'
A' → cA' | fdA' | ε
B → bdA'g | BeA'g | Afh | bh | k

Substitute, using the rules for "A"
bdA'... | BeA'...

Left Recursion in More Than One Step: Elimination

The Original Grammar:

S → Af | b
A → Ac | Sd | Be
B → Ag | Sh | k

So Far:

S → Af | b
A → bdA' | BeA'
A' → cA' | fdA' | ε
B → bdA'g | BeA'g | bdA'fh | BeA'fh | bh | k

Substitute, using the rules for "A"
bdA'... | BeA'...

Left Recursion in More Than One Step: Elimination

The Original Grammar:

S → Af | b
A → Ac | Sd | Be
B → Ag | Sh | k

So Far:

S → Af | b
A → bdA' | BeA'
A' → cA' | fdA' | ε
B → bdA'g | BeA'g | bdA'fh | BeA'fh | bh | k

Finally, eliminate any immediate Left recursion involving "B"

Next Form

S → Af | b
A → bdA' | BeA'
A' → cA' | fdA' | ε
B → bdA'gB' | bdA'fhB' | bhB' | kB'
B' → eA'gB' | eA'fhB' | ε

Left Recursion in More Than One Step: Elimination

The Original Grammar:

S → Af | b

A → Ac | Sd | Be | C

B → Ag | Sh | k

C → BkmA | AS | j

If there is another nonterminal, then do it next.

So Far:

S → Af | b

A → bdA' | BeA' | CA'

A' → cA' | fdA' | ε

B → bdA'gB' | bdA'fhB' | bhB' | kB' | CA'gB' | CA'fhB'

B' → eA'gB' | eA'fhB' | ε

Algorithm for Eliminating Left Recursion

Assume the nonterminals are ordered $A_1, A_2, A_3, \ldots$

(In the example: S, A, B)

for **each** nonterminal A_i (for i = 1 to N) **do**

  for **each** nonterminal A_j (for j = 1 to i-1) **do**

    Let $A_j \to \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots \mid \beta_N$ be all the rules for A_j

    if there is a rule of the form

      $A_i \to A_j\alpha$

    then replace it by

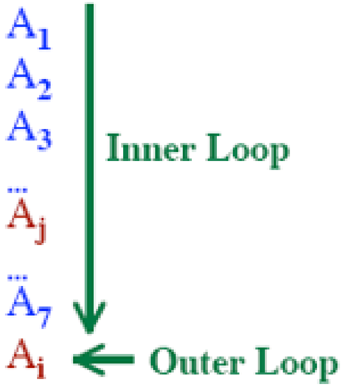
      $A_i \to \beta_1\alpha \mid \beta_2\alpha \mid \beta_3\alpha \mid \ldots \mid \beta_N\alpha$

    endIf

  endFor

  Eliminate immediate left recursion among the A_i rules

endFor



Left Factoring: Common Prefix Problem

Problem : Uncertain which of 2 rules to choose:

$$\begin{aligned} stmt &\rightarrow \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\ &\mid \textbf{if } expr \textbf{ then } stmt \end{aligned}$$

When do you know which one is valid ?

What's the general form of *stmt* ?

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

α : **if** *expr* **then** *stmt*

β_1: **else** *stmt* β_2 : $\in$

Transform to:

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 \mid \beta_2$

EXAMPLE:

$stmt \rightarrow \textbf{if } expr \textbf{ then } stmt\ rest$

$rest \rightarrow \textbf{else } stmt \mid \in$

Left Factoring : Example

$A \rightarrow abB \mid aB \mid cdg \mid cdeB \mid cdfB$

$\Downarrow$

$A \rightarrow aA' \mid cdg \mid cdeB \mid cdfB$

$A' \rightarrow bB \mid B$

$\Downarrow$

$A \rightarrow aA' \mid cdA''$

$A' \rightarrow bB \mid B$

$A'' \rightarrow g \mid eB \mid fB$

Left Factoring : Example

$A \rightarrow ad \mid a \mid ab \mid abc \mid b$

$$\Downarrow$$

$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid b \mid bc$

$$\Downarrow$$

$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid bA''$

$A'' \rightarrow \varepsilon \mid c$

The End