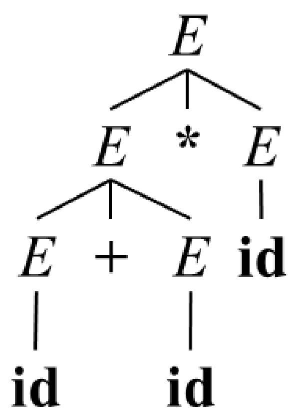
Syntax Analysis



Limitations of Regular Languages

- Weakest formal languages widely used

- Lots of applications
  - search/replace
  - lexical analysis

- Main limitation
  - regular languages cannot count past $|S|$

Limitations of Regular Languages

- Example: balanced parentheses

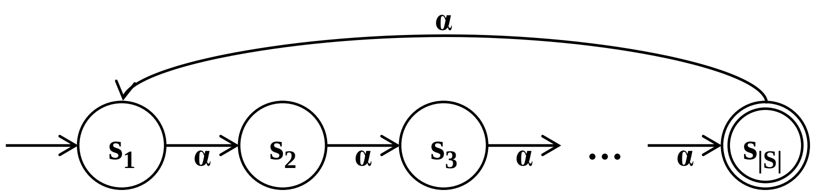
$$\{ (^i\,)^i \mid i \geq 0 \}$$

()
(())
((((()))))
…

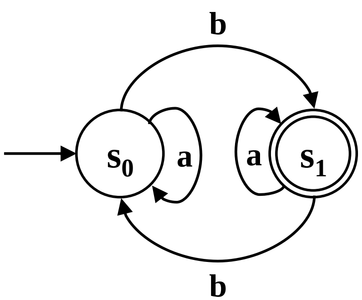
- Any balanced structure
  - nested expressions
  - if-else
  - statement blocks

Limitations of Regular Languages

- Regular languages can count up to $|S|$ and modulo $|S|$
  - $\{\ \alpha^{i|S|} \mid i \geq 0\ \}$



  - $\{\ (a^*b^*)^* \mid |b| \bmod 2 = 1\}$



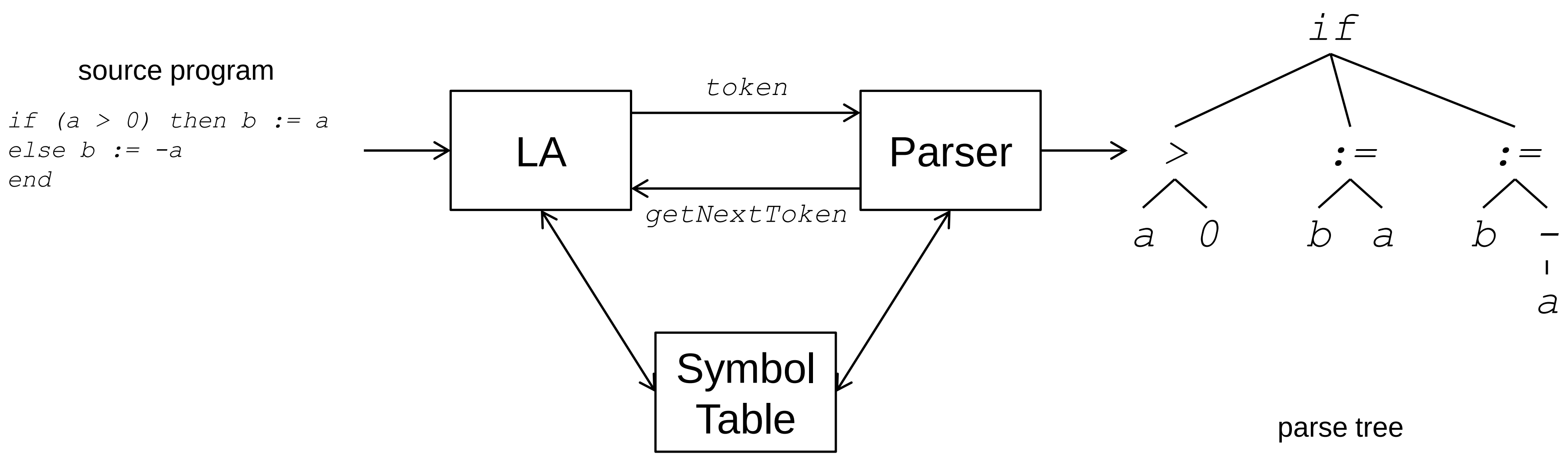
$a^*ba^*(ba^*ba^*)^*$

  - $\{\ (^i\ )^i \mid i \geq 0\ \}$

regular languages can count one thing, but not two

Role of Syntax Analysis

- Input: tokenized input stream from the lexer
- Output: parse tree of program



Context-Free Grammars

Context-Free Grammars

- Not all (valid) token strings are also valid programs

```
if (if while end begin) := (a + (* 3( -) 2);
```

- Formalism to describe valid strings of tokens
  - parsing: an algorithm to distinguish valid from invalid strings of tokens
  - context-free grammars are a natural notation for describing recursive structures

```
begin
   begin
      begin end
   end
end
```

Context-Free Grammars

- **Formal Definition**
  A context-free grammar (CFG) $\boldsymbol{G} = (T, N, P, S)$ consists of
  - **terminals** $\boldsymbol{T}$
    basic symbols from which the strings belonging to the grammar are formed. Aka *tokens*.
  - **nonterminals** $\boldsymbol{N}$
    syntactic variables that denote sets of strings and impose a hierarchical structure on the language.
  - **productions** $\boldsymbol{P}$
    specifications on how terminals and nonterminals can be combined to form strings. Productions have the form

    nonterminal → { terminal | nonterminal | ε }

    head *body*
  - a **start symbol** $\boldsymbol{S}$
    one of the nonterminals. The set of strings that can be generated starting with the start symbol denotes the language generated by the grammar.

Context-Free Grammars

- Example: balanced parentheses

$$S \rightarrow \mathbf{(}\ S\ \mathbf{)}$$
$$S \rightarrow \varepsilon$$

- nonterminals: $N = \{ S \}$
- terminals: $T = \{ \mathbf{(}, \mathbf{)} \}$
- start symbol: S
  (unless explicitly stated: nonterminal on LHS of first production)
- productions: $\{ S \rightarrow \mathbf{(}S\mathbf{)}, S \rightarrow \varepsilon \}$

Context-Free Grammars

- Productions are **replacement rules**
  - start with the start symbol S
  - replace nonterminal X by the RHS of a production of the form

    $$X \rightarrow Y_1 Y_2 \ldots Y_n$$
  - repeat until no nonterminals are left.

$$S \rightarrow (\,S\,)$$
$$S \rightarrow \varepsilon$$

$$S \underset{S \rightarrow (S)}{\rightarrow} (S) \underset{S \rightarrow (S)}{\rightarrow} ((S)) \underset{S \rightarrow (S)}{\rightarrow} (((S))) \underset{S \rightarrow \varepsilon}{\rightarrow} ((())).$$

Context-Free Grammars

- Another Example

expression	→	*expression* + *term*
expression	→	*expression* – *term*
expression	→	*term*
term	→	*term* ***** *factor*
term	→	*term* / *factor*
term	→	*factor*
factor	→	**(** *expression* **)**
factor	→	**id**

Context-Free Grammars

- **Notational Conventions**
  - terminals: a, b, c, +, -, ., (,), 0, 1, 2, .., 9, **if**, **else**
    - lowercase letters, operator and punctuation symbols, digits, bold strings
  - nonterminals: A, B, C, S, *expression*, *term*
    - uppercase letters, lowercase *italic* names
  - grammar symbols: X, Y, Z
    - either non-terminals or terminals
  - strings of terminals: u, v, z
  - strings of grammar symbols: α, β, γ
    - $A \rightarrow \alpha$
  - empty string: ε
  - end-of-input marker: $
  - alternatives: merge bodies of productions with common head

Context-Free Grammars

- Example

$$\begin{aligned}
\textit{expression} &\rightarrow \textit{expression} + \textit{term} \\
\textit{expression} &\rightarrow \textit{expression} - \textit{term} \\
\textit{expression} &\rightarrow \textit{term} \\
\textit{term} &\rightarrow \textit{term} * \textit{factor} \\
\textit{term} &\rightarrow \textit{term} \,/\, \textit{factor} \\
\textit{term} &\rightarrow \textit{factor} \\
\textit{factor} &\rightarrow \textbf{(}\ \textit{expression}\ \textbf{)} \\
\textit{factor} &\rightarrow \textbf{id}
\end{aligned}$$

$$\Rightarrow$$

$$\begin{aligned}
E &\rightarrow E + T \mid E - T \mid T \\
T &\rightarrow T * F \mid T / F \mid F \\
F &\rightarrow \textbf{(}\ E\ \textbf{)} \mid \textbf{id}
\end{aligned}$$

Derivational View

- Every production is a replacement rule
- Starting with the start symbol, repeatedly apply a production until all nonterminals have been eliminated

- Notation: $A \Rightarrow$ applied production

- Example: derivation for –(id)

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

$$E \Rightarrow -E \Rightarrow \mathbf{-(}\, E \,\mathbf{)} \Rightarrow \mathbf{-(\, id\,)}$$

- $\alpha \Rightarrow^{*} \alpha$: derives in **zero** or more steps
- $\alpha \Rightarrow^{+} \alpha$: derives in **one** or more steps

Languages and Grammars

- A sentence of $G = (T, N, P, S)$, w, is a string of terminals for which a derivation

$$S \Rightarrow^* w$$

  exists

- The language of a grammar, $L(G)$, is the set of sentences generated by G.

$$\{ a_1 \ldots a_n \mid \forall i \; a_i \in T \wedge S \Rightarrow^* a_1 \ldots a_n \}$$

- Not surprisingly, a context-free language is generated by a context-free grammar.

- Two grammars G_1 and G_2 are said to be equivalent if $L(G_1) = L(G_2)$

Languages and Grammars

- Simplified extract from SnuPL/1

$$
\begin{array}{lll}
\textit{expression} & \rightarrow & \textit{expression} + \textit{expression} \\
& | & \textit{expression}\ \mathbf{*}\ \textit{expression} \\
& | & \mathbf{(}\ \textit{expression}\ \mathbf{)} \\
& | & \mathbf{id}
\end{array}
$$

- some valid strings

id

id + id

id + (id * id)

id + id + id + id + id

…

Languages and Grammars

- Simplified extract from SnuPL/1

statSequence	→	*statement* { **;** *statement* }
statement	→	**if (** *expression* **) then** *statSequence* [**else** *statSequence*] **end**
	|	**while (** *expression* **) do** *statSequence* **end**
	|	*ident* **(** [*expression* { **,** *expression* }] **)**
	|	**return** [*expression*]
	|	*ident* **:=** *expression*
expression	→	*ident* **+** *ident*
	|	*ident* ***** *ident*
	|	*ident*

Leftmost and Rightmost Derivations

- For each derivation
  - choose which nonterminal to replace
  - pick a production with that nonterminal as its head

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

$$\mathbf{-\ (id + id)}$$

$$E \Rightarrow -E \Rightarrow \mathbf{-(}\ E\ \mathbf{)} \Rightarrow \mathbf{-(}\ E + E\ \mathbf{)} \Rightarrow \mathbf{-(\ id} + E\ \mathbf{)} \Rightarrow \mathbf{-(\ id + id\ )}$$

$$E \Rightarrow -E \Rightarrow \mathbf{-(}\ E\ \mathbf{)} \Rightarrow \mathbf{-(}\ E + E\ \mathbf{)} \Rightarrow \mathbf{-(}\ E + \mathbf{id\ )} \Rightarrow \mathbf{-(\ id + id\ )}$$

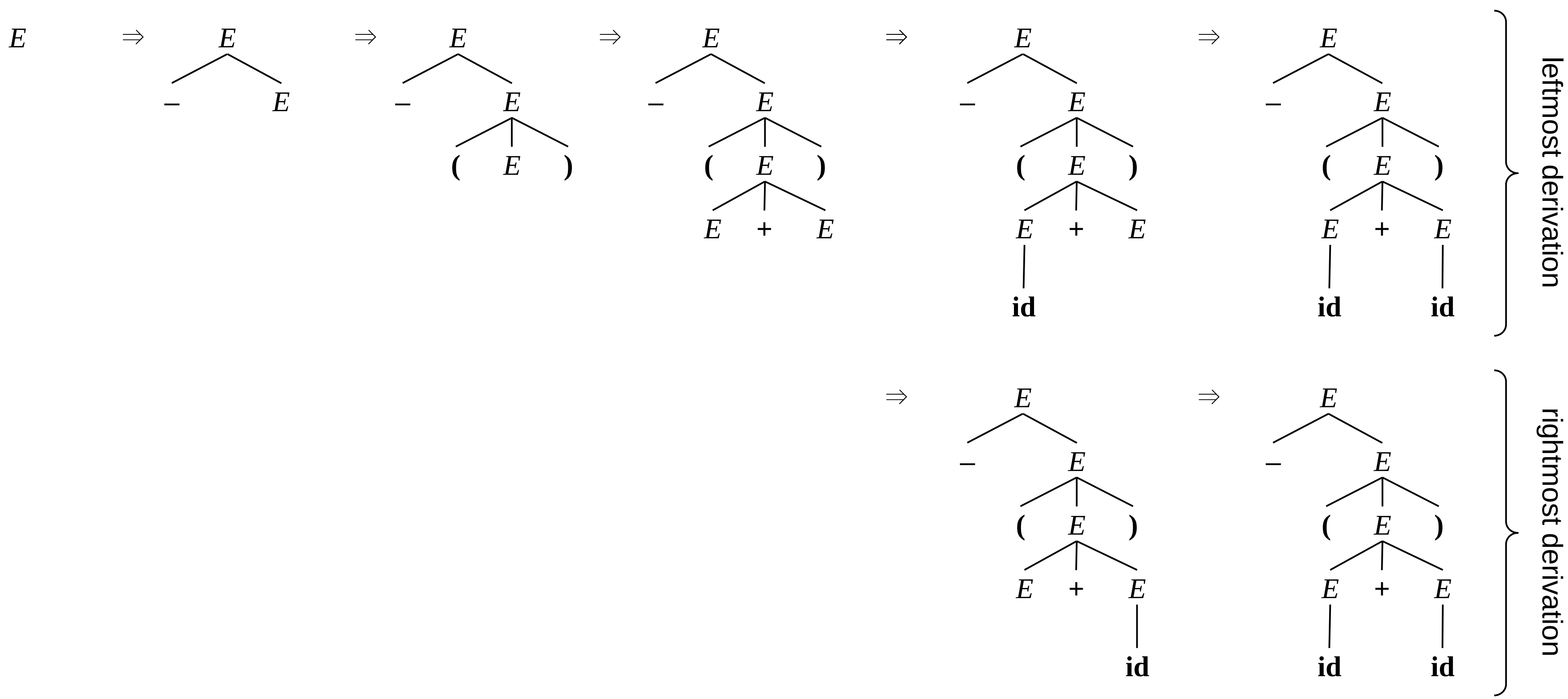
- leftmost derivation: always pick *leftmost* nonterminal
- rightmost derivation: always pick *rightmost* nonterminal

From Derivations to Parse Trees

- A parse tree is a visualization of derivations

$$-\,(\mathbf{id} + \mathbf{id})$$

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id})$$



Ambiguity

- Grammars that produce more than one parse tree for some sentence are said to be *ambiguous*.

$$E \rightarrow E + E \mid E * E \mid (\,E\,) \mid \mathbf{id}$$

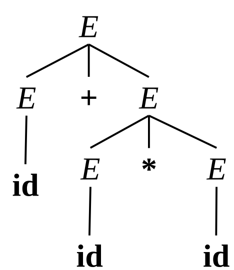
$$\mathbf{id} + \mathbf{id} * \mathbf{id}$$

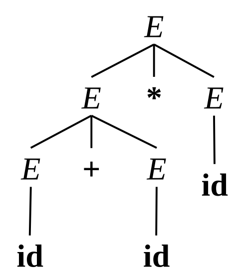
has two possible leftmost derivations*:

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow \mathbf{id} + E \\ &\Rightarrow \mathbf{id} + E * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id} \end{aligned}$$

$$\begin{aligned} E &\Rightarrow E * E \\ &\Rightarrow E + E * E \\ &\Rightarrow \mathbf{id} + E * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id} \end{aligned}$$

* also two possible rightmost derivations





Eliminating Ambiguity

- Ambiguity is bad
  - decision which production to pick is up to the compiler
  - may lead to semantically different programs

- Idea: rewrite grammar to eliminate ambiguity

$$E \rightarrow E + E \mid E \mathbin{\mathbf{*}} E \mid \mathbf{(}\ \mathrm{E}\ \mathbf{)} \mid \mathbf{id}$$

$$\Downarrow$$

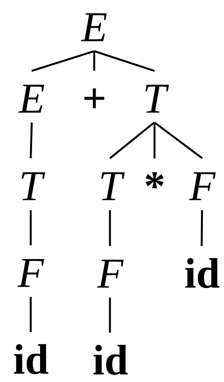
$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow \mathbf{(}\ E\ \mathbf{)} \mid \mathbf{id} \end{aligned}$$

Eliminating Ambiguity

- Parse **id** + **id** * **id**

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (\, E\,) \mid \mathbf{id}
\end{aligned}
$$

$$
\begin{aligned}
E &\Rightarrow E + T \\
&\Rightarrow T + T \\
&\Rightarrow F + T \\
&\Rightarrow \mathbf{id} + T \\
&\Rightarrow \mathbf{id} + T * F \\
&\Rightarrow \mathbf{id} + F * F \\
&\Rightarrow \mathbf{id} + \mathbf{id} * F \\
&\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{aligned}
$$

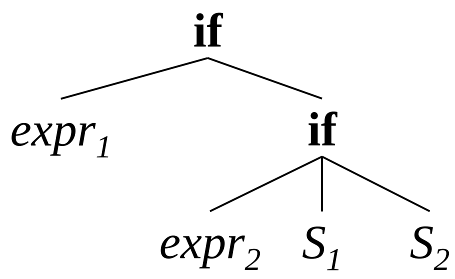


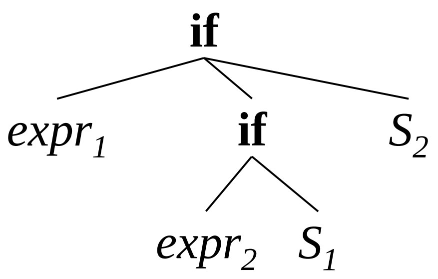
Eliminating Ambiguity

- Dangling else

$$\begin{array}{lll} \textit{stmt} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \\ & | & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \textbf{ else } \textit{stmt} \\ & | & \textbf{other} \end{array}$$

- Consider **if** $expr_1$ **then if** *expr2* **then** *S1* **else** S2





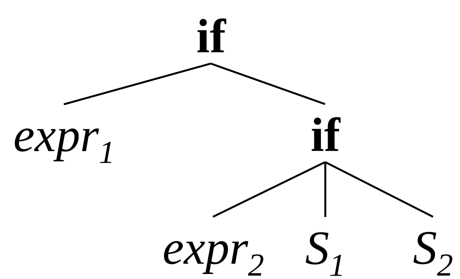
CSE 컴퓨터공학부
Department of Computer Science & Engineering

Eliminating Ambiguity

- Dangling else: **else** matches closest **then**

$$
\begin{array}{lcl}
\mathit{stmt} & \rightarrow & \mathit{mif} \\
 & | & \mathit{uif} \\
\mathit{mif} & \rightarrow & \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{mif}\ \textbf{else}\ \mathit{mif} \\
 & | & \textbf{other} \\
\mathit{uif} & \rightarrow & \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt} \\
 & | & \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{mif}\ \textbf{else}\ \mathit{uif}
\end{array}
$$

- Consider **if** $expr_1$ **then if** *expr2* **then** *S1* **else** S2



Eliminating Ambiguity

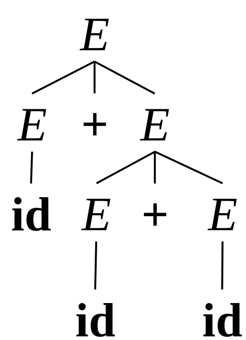
- There is no automatic way to eliminate ambiguity
  - eliminate by hand
    - lead to significantly more complex grammars
  - allow some ambiguity (use with care!)
    - allows for more natural definitions
    - need some disambiguation mechanism
- Typically, some ambiguity is allowed

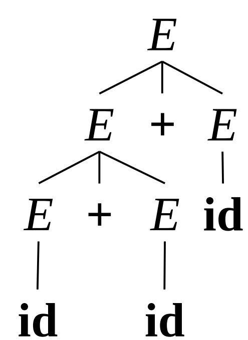
Ambiguous Grammars for PLs

- Provide additional means to define the semantics of the program even in the presence of ambiguity
  - precedence
  - association

- Example $$E \rightarrow E + E \mid \mathbf{id}$$

$$\mathbf{id} + \mathbf{id} + \mathbf{id}$$





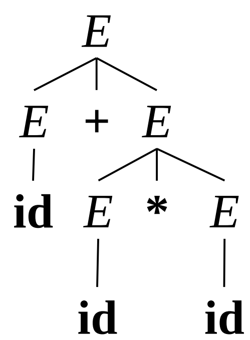
  - declare left associativity: (Yacc/bison) `%left +`

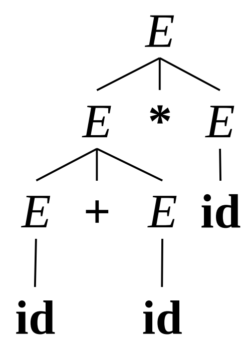
Ambiguous Grammars for PLs

- Another Example

$$E \rightarrow E + E \mid E - E \mid E * E \mid E \,/\, E \mid \mathbf{id}$$

$$\mathbf{id} + \mathbf{id} * \mathbf{id}$$





- declare left associativity and precedence of operators: (Yacc/bison)

```
%left + -
%left * /
```

Left Recursive Grammars

- A grammar is left recursive if it has a production rule where the nonterminal on the LHS is identical to the first symbol of the production body, i.e.

$$X \rightarrow Y_1 Y_2 \ldots Y_n \quad \text{where} \quad X = Y_1$$

- This can lead to endless recursion in a predictive (top-down) parser

$$E \quad \rightarrow \quad E + T \mid \mathbf{id}$$

$$1 + 2$$
$$\uparrow$$

$$\underline{E} \rightarrow \underline{E} + T \rightarrow \underline{E} + T + T \rightarrow \underline{E} + T + T + T \rightarrow \underline{E} + T + T + T + T \rightarrow \ldots$$

Eliminating Left-Recursion

- Rewrite left-recursive productions

$$A \rightarrow A\alpha \mid \beta \quad \Longrightarrow \quad \begin{array}{lll} A & \rightarrow & \beta A' \\ A' & \rightarrow & \alpha A' \mid \varepsilon \end{array}$$

- This simple rule suffices for most grammars

$$\begin{array}{lll} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (\, E\,) \mid \mathbf{id} \end{array}$$

Eliminating Left-Recursion

- More elaborate rule:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n \qquad \text{no } \beta_i \text{ begins with } A$$

$$\Downarrow$$

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

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$$

- However, this rule still fails on

$$S \rightarrow A\mathbf{a} \mid \mathbf{b}$$

$$A \rightarrow A\mathbf{c} \mid S\mathbf{d} \mid \varepsilon$$

Systematically Eliminating Left-Recursion

- Systematically eliminate left-recursion
  - input: grammar G (no cycles, no ε-productions)
  - output: equivalent grammar with no left recursion (may include ε-productions)
  - algorithm

```
arrange nonterminals in some order A_1, A_2, …, A_n
for i := 1 to n do
  for j := 1 to i-1 do
    replace each production of the form A_i → A_j γ
      by the production A_i → δ_1γ|δ_2γ|…|δ_kγ, where
      A_j → δ_1|δ_2|…|δ_k are all current A_j-productions
  od
  eliminate immediate left recursion in A_i
od
```

- Example

$$S \rightarrow A\mathbf{a} \mid \mathbf{b} \qquad\qquad S \rightarrow A\mathbf{a} \mid \mathbf{b} \qquad\qquad S \rightarrow A\mathbf{a} \mid \mathbf{b}$$

$$A \rightarrow A\mathbf{c} \mid S\mathbf{d} \mid \varepsilon \quad \Rightarrow \quad A \rightarrow A\mathbf{c} \mid A\mathbf{ad} \mid \mathbf{bd} \mid \varepsilon \quad \Rightarrow \quad A \rightarrow \mathbf{bd}A' \mid A'$$

$$A' \rightarrow \mathbf{c}A' \mid \mathbf{ad}A' \mid \varepsilon$$

Left Factoring

- Left factoring transforms a grammar into a form that is suitable for top-down parsing

stmt → **if** *expr* **then** *stmt* **else** *stmt*

| **if** *expr* **then** *stmt*

if (a < b) then …

<if> <**(**> <id,“a”> <relop,“**<**“> <id,“b”> <**)**> <**then**> …

↑

which production should the parser choose?

Left Factoring

- Left factoring transforms a grammar into a form that is suitable for top-down parsing
  - idea: defer decision until the common prefix has been consumed

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \quad \Rightarrow \quad A \rightarrow \alpha A' \qquad A' \rightarrow \beta_1 \mid \beta_2$$

  - left factoring
    - input: grammar G
    - output: left-factored equivalent grammar
    - algorithm

```
do
  for each nonterminal A, find the longest prefix α common to two
    or more of its alternatives. If α ≠ ε replace all A-productions
    A → αβ1 | αβ2 | … | αβn | γ with where γ represents all alternatives
    that do not begin with α by
      A → αA' | γ
      A → β1 | β2 | … | βn
until no change occurs
```

Limits of Context-Free Grammars

- Variable declaration before use

```
int i;
if (a > b) then a := a-b;
else a := b-a;
i = a + 7;
```

$L = \{ wcw \mid w \text{ in } (a|b)^* \}$

- Number of actual parameters match with formal parameters

```
int foo(int p1, int p2) {…}

int main(void) {
  foo(1, 2, 3, 4, 5);
}
```

$L = \{ a^n b^m c^n d^m \mid n \geq 1 \text{ and } m \geq 1 \}$

Context-Free Grammars vs. Regular Expressions

- Which are more powerful?
  - We can easily construct a grammar from a regular expression (details see textbook 4.2.7)

(a|b)*abb

$$
\begin{aligned}
A &\rightarrow aA \mid bA \mid aB \\
B &\rightarrow bC \\
C &\rightarrow bD \\
D &\rightarrow \varepsilon
\end{aligned}
$$

  - The opposite is not true:

$$A \rightarrow aAb \mid \varepsilon$$

- "regular expressions can count one thing, but not two"
  "context-free grammars can count two things, but not three"

Recap: Context-Free Grammars

- A context-free grammar $\boldsymbol{G}$ consists of
  - $\boldsymbol{T}$, the set of terminal symbols
  - $\boldsymbol{N}$, the set of nonterminals
  - $S \in \boldsymbol{N}$, the start symbol
  - $\boldsymbol{P}$, the set of productions of the form

$$X \rightarrow Y_1 Y_2 \ldots Y_k \qquad \text{where } X \in \boldsymbol{N},\ Y_i \in \boldsymbol{N} \cup \boldsymbol{T}$$

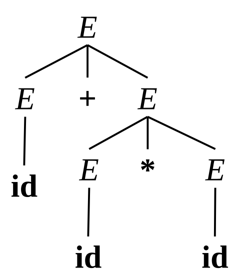
- Representation of choice for most programming languages
  - ideal to represent nested constructs
    - balanced brackets, if…else, begin…end
  - however, CFGs cannot express context
    - declaration before use of variables
    - # of actual parameters = # of formal parameters to a function

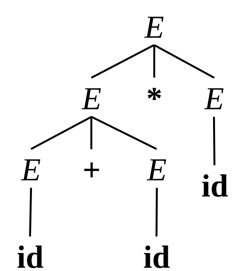
Recap: Context-Free Grammars

- Ambiguous Grammars
  - a grammar that allows different parse trees for the same input

$$E \rightarrow E + E \mid E * E \mid (\mathrm{E}) \mid \mathbf{id}$$

id + id * id





  - resolutions
    - rewrite grammar
      - cleanest solution, but: more complex grammar, cannot be automated
    - allow some ambiguity, use other means to clarify which parse tree should be generated
      - associativity, precedence

Recap: Context-Free Grammars

- Recursive Grammars
  - a grammar is left-recursive if there exists a derivation
    $A \Rightarrow^{+} A\alpha$
    for some nonterminal A. (right-recursion analogous)
  - resolutions
    - none necessary for bottom-up (LR) parsers, however, recursive-descent parsers may end up in an endless loop if the grammar is left-recursive
      - illustration for immediate left-recursion:

$$A \rightarrow A\alpha \mid \beta \quad \Longrightarrow \quad \begin{array}{lcl} A & \rightarrow & \beta A' \\ A' & \rightarrow & \alpha A' \mid \varepsilon \end{array}$$

      - eliminate left-recursion using the algorithm on slide 31

Recap: Context-Free Grammars

- Left factoring
  - eliminates common nontrivial prefixes from productions

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \quad \Rightarrow \quad \begin{array}{lll} A & \rightarrow & \alpha A' \\ A' & \rightarrow & \beta_1 \mid \beta_2 \end{array}$$

  - useful for top-down parsers because it defers the decision whether to select production $A \rightarrow \alpha\beta_1$ or $A \rightarrow \alpha\beta_2$ until after α has been seen
  - use algorithm discussed on slide 33

Extended Backus-Naur Form

Extended Backus-Naur Form

- Notation for context free grammars
  - Backus-Naur Form (BNF)
    - a formal way to specify context-free grammars
    - developed by John Backus for ALGOL

```
<symbol>    ::=  __expression__
```

meta symbols: ::= (definition), < > (nonterminal), | (alternation)

```
<digit>     ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<number>    ::=  <digit> | <number> <digit>
```

  - Extended Backus-Naur Form
    - originally developed by Niklaus Wirth as an extension of his Wirth syntax notation (link to paper)
      - introduced [] (option), {} (repetition)
    - later standardized (ISO/IEC 14977)

Extended Backus-Naur Form

- Notation: Extended Backus-Naur Form
  - standardized notation (ISO/IEC 14977:1996)
    - meta symbols: “=” (definition), “,” (concatenation), “;” (end of production)

```
module   =   "module", ident, ";",
             { [typeDecl], [varDecl], [funcDecl] },
             "begin", stmtSeq, "end", ident, ".";
```

  - we use no symbol for concatenation, and “.” to indicate the end of the production

```
module   =   "module" ident ";"
             { [typeDecl] [varDecl] [funcDecl] }
             "begin" stmtSeq "end" ident ".".
```

Extended Backus-Naur Form

- Notation: Extended Backus-Naur Form

Notation	Usage	Example
`=`	definition	`letter = "A".."Z".`
`.`	termination	`letter = "A".."Z".`
`|`	alternation	`letter = "A".."Z" | "a".."z".`
`[ … ]`	option	`number = ["-"] digit.`
`{ … }`	repetition (≥ 0)	`number = ["-"] digit {digit}.`
`( … )`	grouping	`factor = [unaryOp] (ident | number).`
`"…", '…'`	terminal symbol	`"module", '"'`

Example: SnuPL/1

- EBNF of SnuPL/1

```
module      =  "module" ident ";" varDeclaration { subroutineDecl } "begin"
               statSequence "end" ident ".".

letter      =  "A”.."Z" | "a".."z" | "_".
digit       =  "0".."9".
character   =  ASCIIchar | "\n" | "\t" | "\"" | "\'" | "\\" | "\0"
char        =  "'" character "'"
string      =  '"' {character} '"'.

ident       =  letter { letter | digit }.
number      =  digit { digit }.
boolean     =  "true" | "false".
type        =  basetype | type "[" [ number ] "]".
basetype    =  "boolean" | "char" | "integer".
```

Example: SnuPL/1

- EBNF of SnuPL/1 (cont'd)

```
qualident      =   ident { "[" expression "]" }.

factOp         =   "*" | "/" | "&&".

termOp         =   "+" | "-" | "||".

relOp          =   "=" | "#" | "<" | "<=" | ">" | ">=".


factor         =   qualident | number | boolean | char| string |
                   "(" expression ")" | subroutineCall | "!" factor.
term           =   factor { factOp factor }.

simpleexpr     =   ["+"|"-"] term { termOp term }.

expression     =   simpleexpr [ relOp simplexpr ].
```

Example: SnuPL/1

- EBNF of SnuPL/1 (cont'd)

```
assignment         =  qualident ":=" expression.

subroutineCall     =  ident "(" [ expression {"," expression} ] ")".

ifStatement        =  "if" "(" expression ")" "then" statSequence
                      [ "else" statSequence ] "end".
whileStatement     =  "while" "(" expression ")" "do" statSequence "end".

returnStatement    =  "return" [ expression ].


statement          =  assignment | subroutineCall | ifStatement | whileStatement |
                      returnStatement.
statSequence       =  [ statement { ";" statement } ].

varDeclaration     =  [ "var" varDeclSequence ";" ].

varDeclSequence    =  varDecl { ";" varDecl }.

varDecl            =  ident { "," ident } ":" type.
```


CSE 컴퓨터공학부 Department of Computer Science & Engineering

Example: SnuPL/1

- EBNF of SnuPL/1 (cont'd)

```
subroutineDecl      =   (procedureDecl | functionDecl)
                        subroutineBody ident ";".
procedureDecl       =   "procedure" ident [ formalParam ] ";".

functionDecl        =   "function" ident [ formalParam ] ":" type ";".

formalParam         =   "(" [ varDeclSequence ] ")".

subroutineBody      =   varDeclaration "begin" statSequence "end".



comment             =   "//" {[^\n]} \n

whitespace          =   { " " | \t | \n }
```

Error Handling

CSE 컴퓨터공학부
Department of Computer Science & Engineering

Error Handling

- Error handling is an important part of the parser

```
$ snuplc fibonacci.mod
Segmentation fault

$ snuplc fibonacci.mod
snuplc: parser.c:184: module: Assertion '!strcmp(tmpid,
token.value)' failed.
Aborted.
```

  - not very helpful
  - we want something like

```
$ snuplc fibonacci.mod
syntax error in fibonacci.mod:8:5 :
module identifier mismatch (expected 'fibonacci', got 'huga').
```

Error Types

- Possible types of errors
  - lexical errors

```
module test;@
var a: integer
```

  - syntax errors

```
  a := b +/ c
```

  - semantic errors

```
var a, b: integer;
    d: boolean;
begin
  a := b + c + d
```


CSE 컴퓨터공학부
Department of Computer Science & Engineering

Error Handling

- Error handling should be
  - precise
  - quick recovery
  - efficient

- Methods to handle errors
  - panic mode
  - error productions
  - automatic error correction

Panic Mode

- Idea: when an error is detected skip tokens until a synchronizing token is found, and continue from there
  - synchronizing token: tokens that have a well-known role in the language
    - statement separator (";"), end of current block/function ("end")
  - pros: detects unrelated errors
  - cons: may swamp the user with follow-up errors caused by the initial error

- Example: `( 1 + * 2) + 3`
  - synchronizing token in an expression: next integer
  - Bison: indicate synchronizing tokens using the terminal error

```
E → int | E + E | ( E ) | error int | ( error )
```

Error Productions

- Idea: specify known common mistakes as part of the grammar
  - promote common errors to alternative syntax
  - cons: complicates grammar, especially if used extensively
  - also used to warn user about special/deprecated syntax

- Example: `3x` instead of `3*x`
  - modify original production

  E → int | E + E | E * E | (E)

  to

  E → int | E + E | E * E | **EE** | (E)

Error Correction

- Idea: try to fix the error by inserting/deleting some tokens from the original program
  - try to be as close to the original program as possible
    - edit distance
    - exhaustive search within a certain scope (e.g. within a begin..end block)
  - disadvantages
    - fixing the error does not guarantee that the intended meaning of the programmer is maintained
    - hard to implement
    - slows down the compiler
  - PL/C (Cornell, 1975)

Intermediate Representations: Abstract Syntax Trees

Abstract Syntax Trees

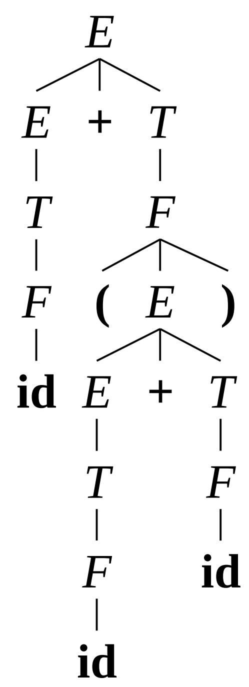
- ASTs (Abstract Syntax Trees) are a common form of an intermediate representation in a parser
  - represent the same information as the parse tree
  - do not preserve the derivations

Abstract Syntax Trees

- Consider the grammar

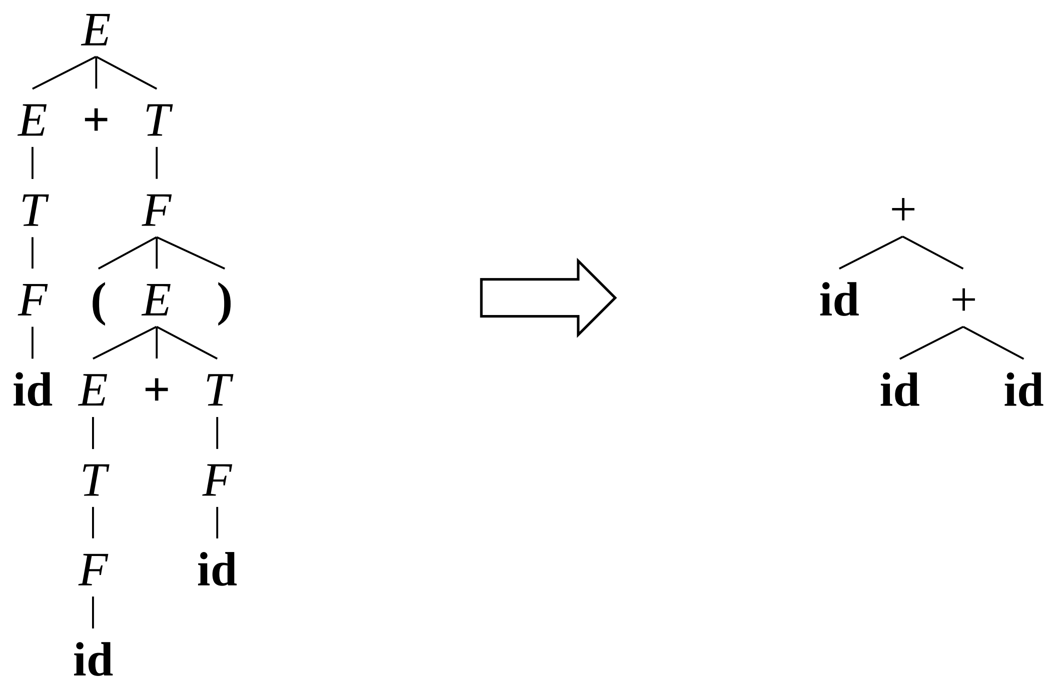
$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (\, E\,) \mid \mathbf{id}$$

and the parse tree for `1 + (2 + 3)`:



Abstract Syntax Tree

- An AST is a concise representation of the parse tree:



parse tree AST

Construction of the AST

- Represent each construct in the program with a node in the AST
  - operations (expressions)
  - while, if, for loops
  - assignments
  - functions
  - the entire program

- Base node:

```
class Node {
};
```

Construction of the AST

- Binary Operations

```
class BinaryOp : public Node {
  public:
    BinaryOp(Op op, Node *left, Node *right)
      : _op(op), _left(left), _right(right) {};

  private:
    Op _op;
    Node *_left, *_right;
};
```

- Unary Operations

```
class UnaryOp : public Node {
  public:
    UnaryOp(Op op, Node *node) : _op(op), _node(node) {};

  private:
   Op _op;
};
```

Construction of the AST

- Statement sequence

```
class StmtSeq : public Node {
  public:
    StmtSeq(Node *stmts, Node *stmt) : _stmts(stmts), _stmt(stmt) {};

  private:
    Node *_stmts, *_stmt;
};
```

- if-then-else statement

```
class If : public Node {
  public:
    If(Node *cond, Node *iftrue, Node *iffalse)
    : _cond(cond), _iftrue(iftrue), _iffalse(iffalse) {};

  private:
   Node *_cond, *_iftrue, *_iffalse;
};
```

Construction of the AST

- While

```
class While : public Node {
  public:
    While(Node *cond, Node *body) : _cond(cond), _body(body) {};

  private:
    Node *_cond, *_body;
};
```

- Assignment

```
class Assign : public Node {
  public:
    Assign(Node *lhs, Node *rhs) : _lhs(lhs), _rhs(rhs) {};

  private:
   Node *_lhs, *_rhs;
};
```

Construction of the AST

- Subroutine Call

```
class Call : public Node {
  public:
    Call(Node *params, Symbol *target) : _params(params), _target(target) {};

  private:
    Node *_params;
    Symbol *_target;
};
```

- Identifier

```
class Ident : public Node {
  public:
    Ident(Symbol *ident) : _ident(ident) {};

  private:
   Symbol *_ident;
};
```

AST Construction w/ Syntax-Directed Translation

- Annotate production rules with actions that build up the parse tree

E	$\rightarrow$	$E + T$	`{ return new BinaryOp('+', E(), T()); }`
	$\vert$	T	`{ return T(); }`
T	$\rightarrow$	$T * F$	`{ return new BinaryOp('*', T(), F()); }`
	$\vert$	F	`{ return F(); }`
F	$\rightarrow$	(E)	`{ return E(); }`
	$\vert$	**id**	`{ return new Ident(GetSymbol(id)); }`

AST Construction w/ Syntax-Directed Translation

- Example: `1 + (2 + 3)`

E	$\rightarrow$	$E + T$	`{ return new BinaryOp('+', E(), T()); }`
	$\mid$	T	`{ return T(); }`
T	$\rightarrow$	$T * F$	`{ return new BinaryOp('*', T(), F()); }`
	$\mid$	F	`{ return F(); }`
F	$\rightarrow$	(E)	`{ return E(); }`
	$\mid$	**id**	`{ return new Ident(GetSymbol(id)); }`