Context-Free Grammars

Using grammars in parsers

Jaruloj Chongstitvatana
Department of Mathametics and Computer Science
Chulalongkorn University

Outline

- Parsing Process
- Grammars
  - Context-free grammar
  - Backus-Naur Form (BNF)
- Parse Tree and Abstract Syntax Tree
- Ambiguous Grammar
- Extended Backus-Naur Form (EBNF)

Parsing Process

- Call the scanner to get tokens
- Build a parse tree from the stream of tokens
  - A parse tree shows the syntactic structure of the source program.
- Add information about identifiers in the symbol table
- Report error, when found, and recover from thee error

Grammar

a quintuple (V, T, P, S) where

V is a finite set of nonterminals, containing S,

T is a finite set of terminals,

P is a set of production rules in the form of $\alpha \rightarrow \beta$ where α and β are strings over $V \cup T$, and

S is the start symbol.

Example

G= ({S, A, B, C}, {a, b, c}, P, S)

P= { $S \rightarrow SABC$, $BA \rightarrow AB$, $CB \rightarrow BC$, $CA \rightarrow AC$,

$SA \rightarrow a$, $aA \rightarrow aa$, $aB \rightarrow ab$, $bB \rightarrow bb$, $bC \rightarrow bc$,

$cC \rightarrow cc$}

Context-Free Grammar

a quintuple (*V, T, P, S*) where

V is a finite set of nonterminals, containing *S*,

T is a finite set of terminals,

P is a set of production rules in the form of $\alpha \rightarrow \beta$ where α is in *V* and β is in $(V \cup T)^*$, and

S is the start symbol.

Any string in (V U T)* is called a *sentential form*.

Examples

E → E O E

E → (E)

E → id

O → +

O → -

O → *

O → /

S → SS

S → (S)S

S → ()

S → ε

Backus-Naur Form (BNF)

- Nonterminals are in < >.
- Terminals are any other symbols.
- ::= means □.
- | means or.
- Examples:

<E> ::= <E><O><E>| (<E>) | ID

<O> ::= + | - | * | /

<S> ::= <S><S> | (<S>)<S> | () | □

Derivation

A sequence of replacement of a substring in a sentential form.

Definition

Let $G = (V, T, P, S)$ be a CFG, α, β, γ be strings in $(V \cup T)^*$ and A is in V.

$\alpha A \beta \Rightarrow_G \alpha\gamma\beta$ if $A \to \gamma$ is in P.

$\Rightarrow^*_G$ denotes a derivation in zero step or more.

Examples

S → SS | (S)S | () | ε

S

→ S

→ (S)SS

→ (S)S(S)S

→ (S)S(())S

→ ((S)S(())S

→ ((S)())S(())S

→ ((())())S(())S

→ ((())()) (())S

→ ((())())(())

E → E O E | (E) | id

O → + | - | * | /

E

→ E O E

→ (E) O E

→ (E O E) O E

→ * ((E O E) O E) O E

→ ((id O E) O E) O E

→ ((id + E) O E) O E

→ ((id + id)) O E) O E

→ * ((id + id)) * id) + id

Leftmost Derivation Rightmost Derivation

Each step of the derivation is a replacement of the *leftmost* nonterminals in a sentential form.

Each step of the derivation is a replacement of the *rightmost* nonterminals in a sentential form.

E → E O E | (E) | id

O → + | - | * | /

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

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

Language Derived from Grammar

- Let $G = (V, T, P, S)$ be a CFG.
- A string w in T^* is derived from G if $S \Rightarrow_G^* w$.
- A language generated by G, denoted by $L(G)$, is a set of strings derived from G.

  $L(G) = \{w \mid S \Rightarrow_G^* w\}$.

Right/Left Recursive

A grammar is a *left recursive* if its production rules can generate a derivation of the form A $\Rightarrow^*$ A X.

Examples:

E $\Rightarrow$ E O id | (E) | id

E $\Rightarrow$ F + id | (E) | id

F $\Rightarrow$ E * id | id

E $\Rightarrow$ F + id

$\Rightarrow$ E * id + id

A grammar is a *right recursive* if its production rules can generate a derivation of the form A $\Rightarrow^*$ X A.

Examples:

E $\Rightarrow$ id O E | (E) | id

E $\Rightarrow$ id + F | (E) | id

F $\Rightarrow$ id * E | id

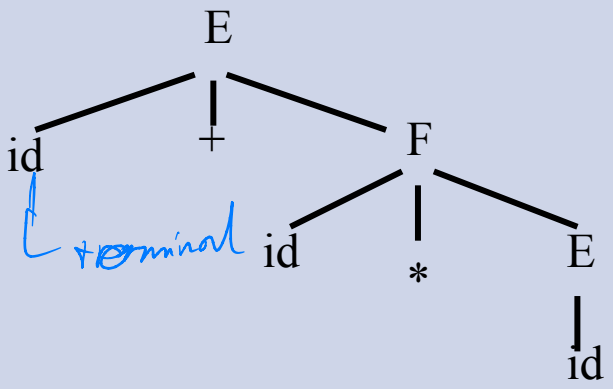
E $\Rightarrow$ id + F

$\Rightarrow$ id + id * E

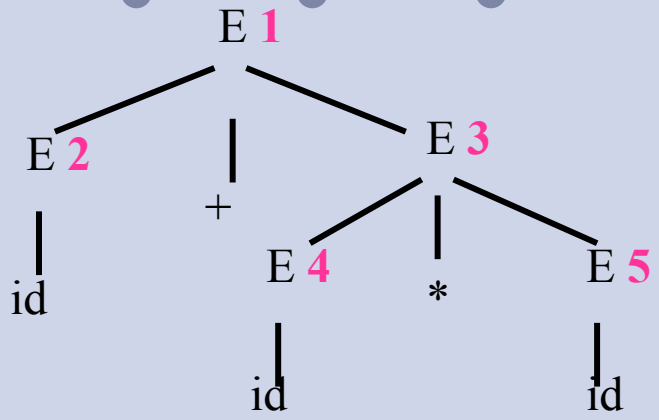
Parse Tree

A labeled tree in which

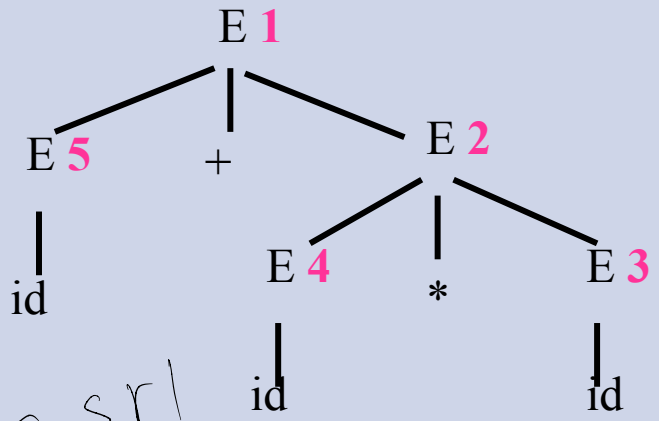
- the interior nodes are labeled by nonterminals
- leaf nodes are labeled by terminals
- the children of an interior node represent a replacement of the associated nonterminal in a derivation
- corresponding to a derivation



Parse Trees and Derivations



Preorder numbering



Reverse of postorder numbering

E ⇒ E + E (1)
⇒ id + E (2)
⇒ id + E * E (3)
⇒ id + id * E (4)
⇒ id + id * id (5)

E ⇒ E + E (1)
⇒ E + E * E (2)
⇒ E + E * id (3)
⇒ E + id * id (4)
⇒ id + id * id (5)

Grammar: Example

List of parameters in:

- Function definition
  - function sub(a,b,c)
- Function call
  - sub(a,1,2)

<argList>
 id , <arglist>
Þ **id, id** , <arglist>
Þ … **id** (, **id**)*

<argList>
 <arglist> , **id**
Þ<arglist> , **id, id**
Þ … **id** (, **id**)*

<Fdef> **function id (**
<argList> **)**
<argList> **id** , <arglist> | **id**
<Fcall> **id (** <parList> **)**
<parList> <par> ,<parlist>|
<par>
<par> **id** | **const**

<Fdef> **function id (**
<argList> **)**
<argList> <arglist> , **id** | **id**
<Fcall> **id (** <parList> **)**
<parList> <parlist> ,<par>|

Grammar: Example

List of parameters

If zero parameter is allowed, then ?

<Fdef> → **function id (** <argList> **)** |
function id ( )

<argList> → **id ,** <arglist> | **id**

<Fcall> → **id (** <parList> **) | id ( )**

<parList> → <par> **,**<parlist>|
<par>

<par> → **id | const**

Work ?

NO!
Generate
id , id , id
, ลงท้ายด้วย , ไม่ได้

<Fdef> → **function id (**
<argList> **)**

<argList> → **id ,** <arglist> | **id** | ε

<Fcall> → **id (** <parList> **)**

Grammar: Example

List of parameters

If zero parameter is allowed, then ?

<Fdef> → **function id (** <argList> **)|** **function id ( )**

<argList> → **id ,** <arglist> | **id**

<Fcall> → **id (** <parList> **) | id ( )**

<parList> → <par> **,**<parlist>| <par>

<par> → **id** | **const**

Work ?

NO!
Generate
id , id , id
,

<Fdef> → **function id (** <argList> **)**

<argList> → **id ,** <arglist> | **id** | ∅

<Fcall> → **id (** <parList> **)**

Grammar: Example

List of statements:

- No statement
- One statement:
  - s;
- More than one statement:
  - s; s; s;
- A statement can be a block of statements.
  - {s; s; s;}

Is the following correct?

{ {s; {s; s;} s; {}} s; }

<St> ::= □ | **s;** | **s;** <St> | **{** <St> **}**

<St>

<St>

⇒ **{** <St> **}** <St>

⇒ **{** <St> **}**

⇒ **{ {** <St> **}** <St>**}**

{ { <St> **} s;** <St>**}**

{ { <St> **} s; }**

{ { s; <St> **} s;}**

{ { s; { <St> **}** <St> **} s;}**

{ { s; { <St> **} s;** <St> **} s;}**

{ { s; { <St> **} s; {** <St> **}** <St> **} s;}**

{ { s; { <St> **} s; { }** <St> **}**

{ { s; { <St> **} s; {} } s;}**

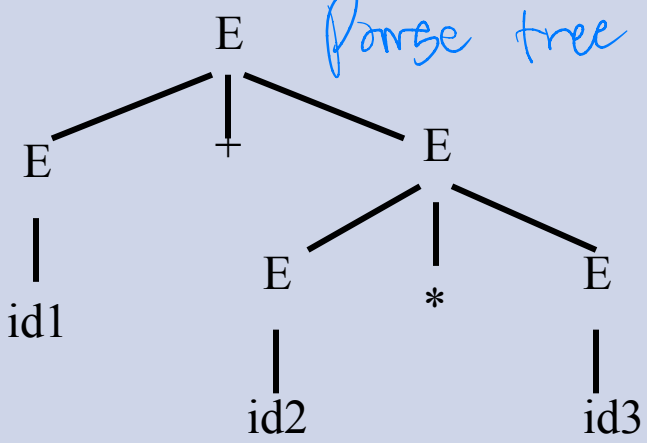
{ { s; { s; <St> **} s; {} } s;}**

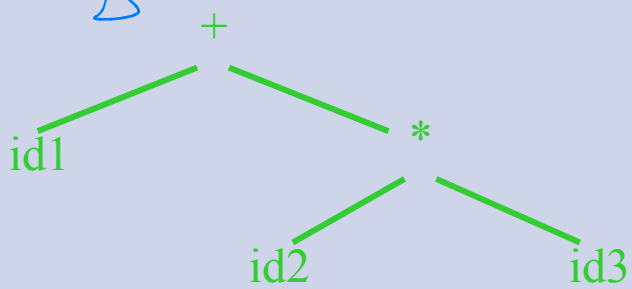
⇒ **{ { s; { s; s;} s; {} } s;}**

Abstract Syntax Tree

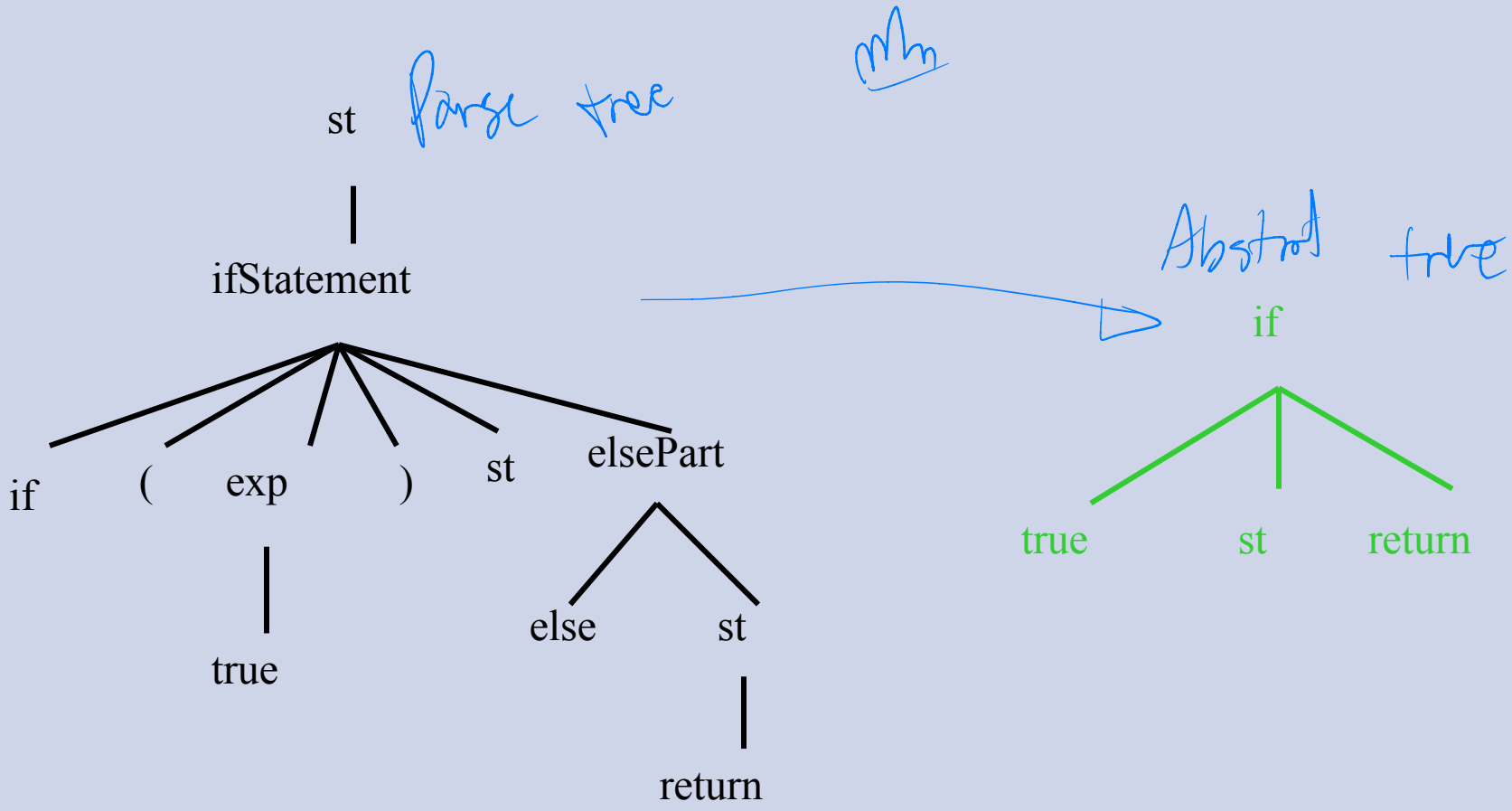
- Representation of actual source tokens
- Interior nodes represent operators.
- Leaf nodes represent operands.

Abstract Syntax Tree for Expression





Abstract Syntax Tree for If Statement



Ambiguous Grammar

- A grammar is ambiguous if it can generate two different parse trees for one string.
- Ambiguous grammars can cause inconsistency in parsing.

Example: Ambiguous Grammar

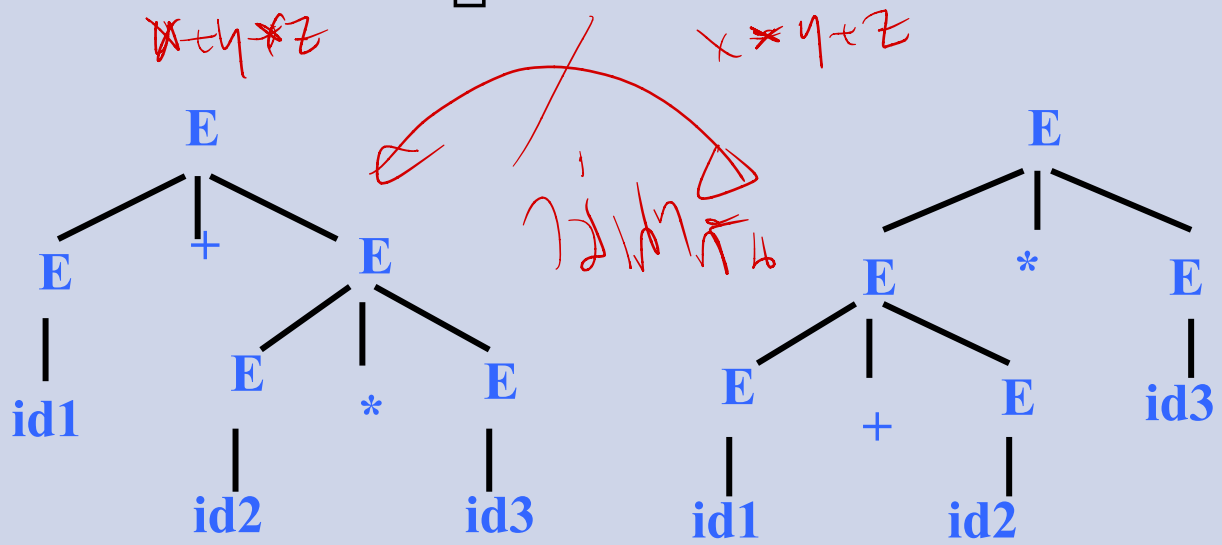
E → E + E

E → E - E

E → E * E

E → E / E

E → id



Ambiguity in Expressions

- Which operation is to be done first?
  - solved by precedence
    - An operator with higher precedence is done before one with lower precedence.
    - An operator with higher precedence is placed in a rule (logically) further from the start symbol.
  - solved by associativity
    - If an operator is right-associative (or left-associative), an operand in between 2 operators is associated to the operator to the right (left).
    - Right-associated : W + (X + (Y + Z))
    - Left-associated : ((W + X) + Y) + Z

Precedence

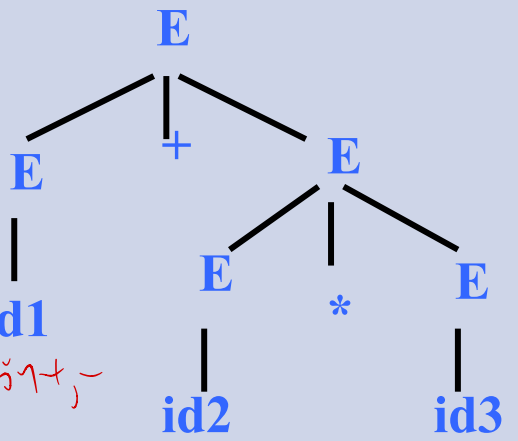
E → E + E

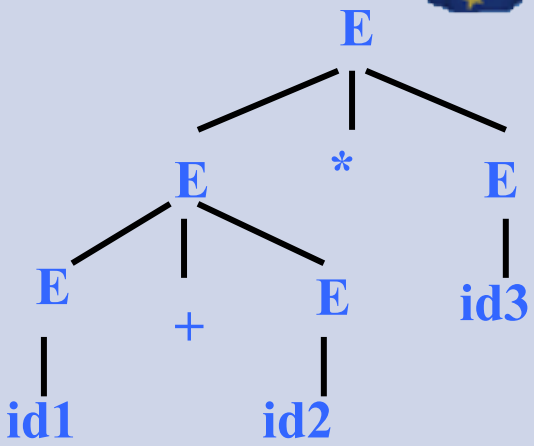
E → E - E

E → E * E

E → E / E

E → id





E → E + E

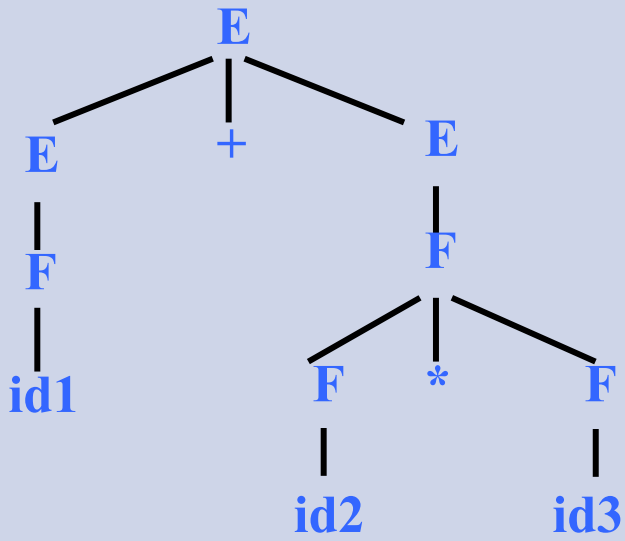
E → E - E

E → F

F → F * F

F → F / F

F → id



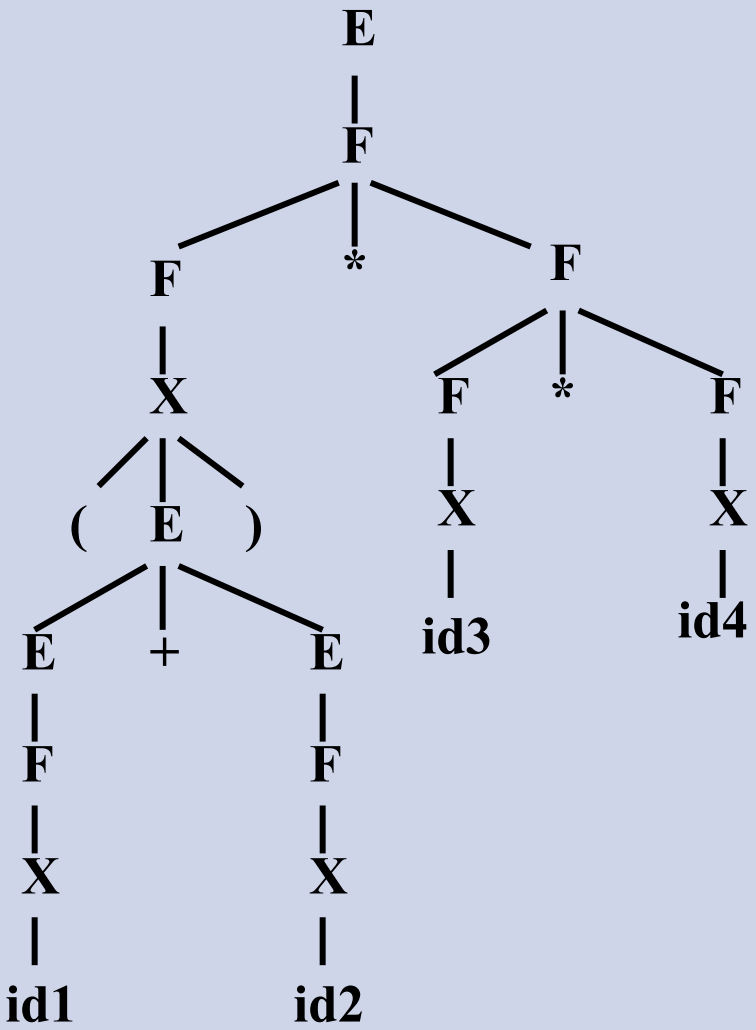
Precedence (cont'd)

E → E + E | E - E | F

F → F * F | F / F | X

X → (E) | id

(id1 + id2) * id3 * id4



Associativity

Left-associative operators

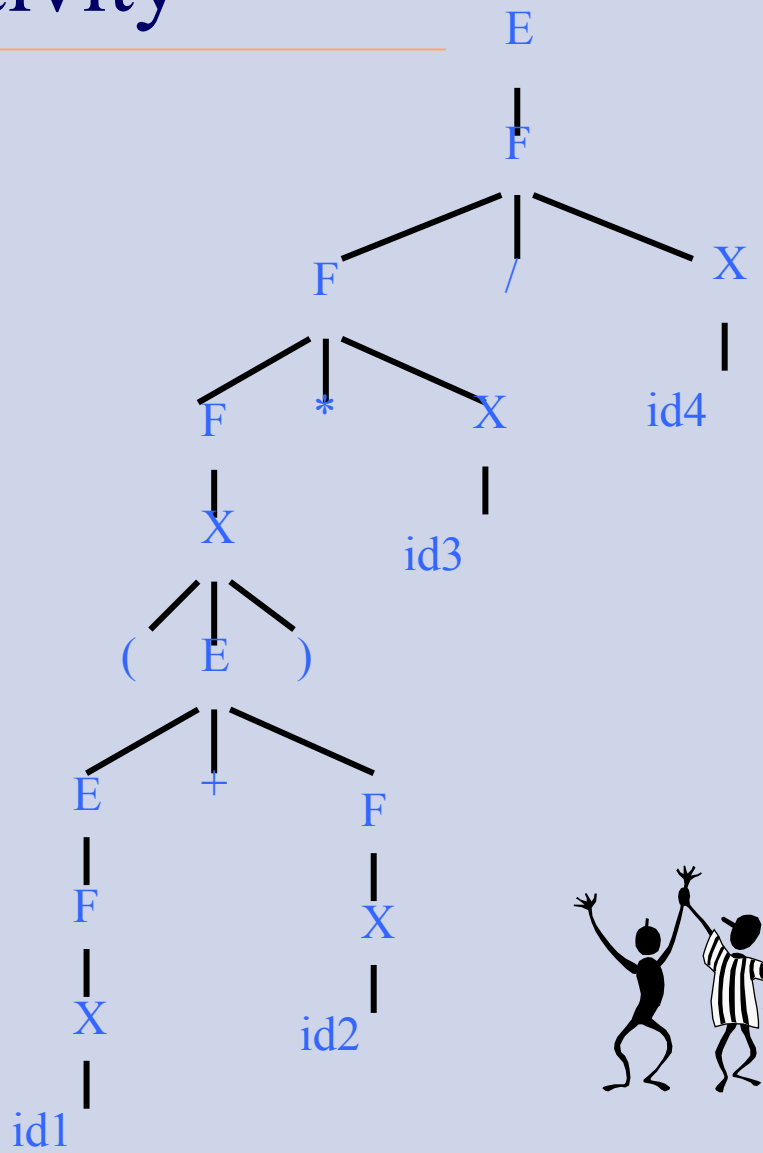
E → E + F | E - F | F

F → F * X | F / X | X

X → (E) | id

(id1 + id2) * id3 / id4

= (((id1 + id2) * id3) / id4)

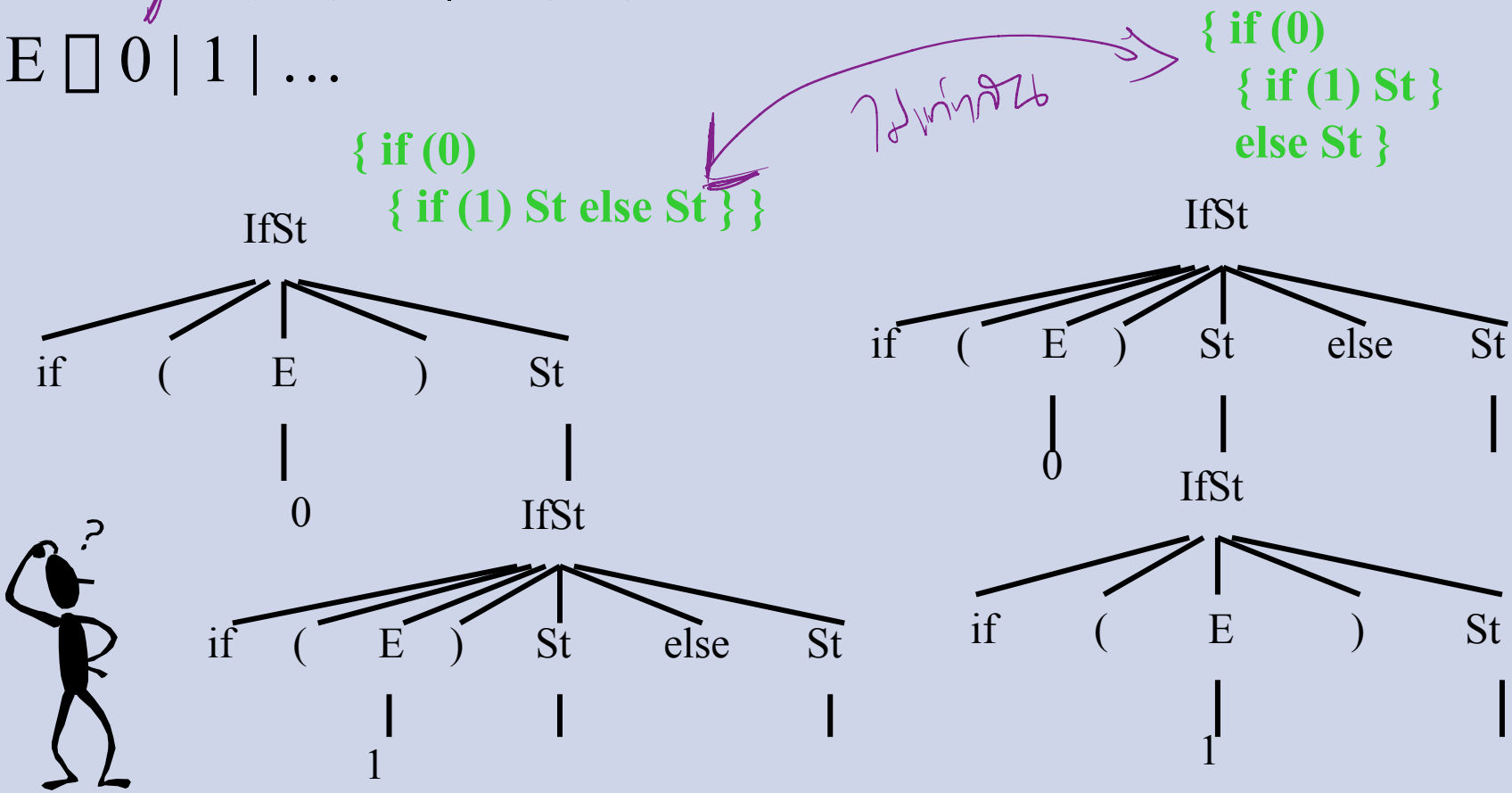


Ambiguity in Dangling Else

St → IfSt | ...

IfSt → if (E) St | if (E) St else St

E → 0 | 1 | …



Disambiguating Rules for Dangling Else

St →
MatchedSt | UnmatchedSt

UnmatchedSt →
if (E) St |
if (E) MatchedSt else UnmatchedSt

MatchedSt →
if (E) MatchedSt else MatchedSt|
...

E →
0 | 1

if (0) if (1) St else St
= if (0)
if (1) St else St

Extended Backus-Naur Form (EBNF)

Kleene's Star/ Kleene's Closure

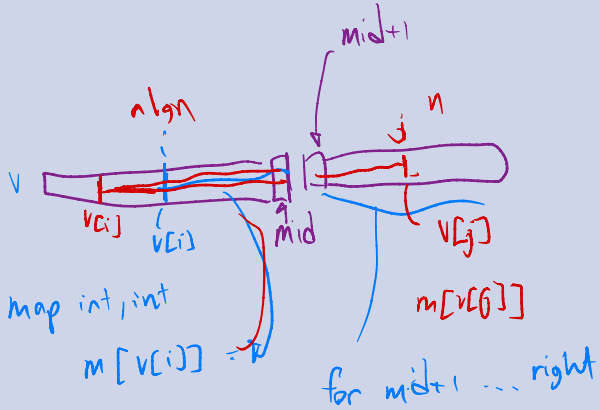
Seq ::= St {; St}

Seq ::= {St ;} St

Optional Part

IfSt ::= if (E) St [else St]

E ::= F [+ E] | F [- E]



Syntax Diagrams

Graphical representation of EBNF rules

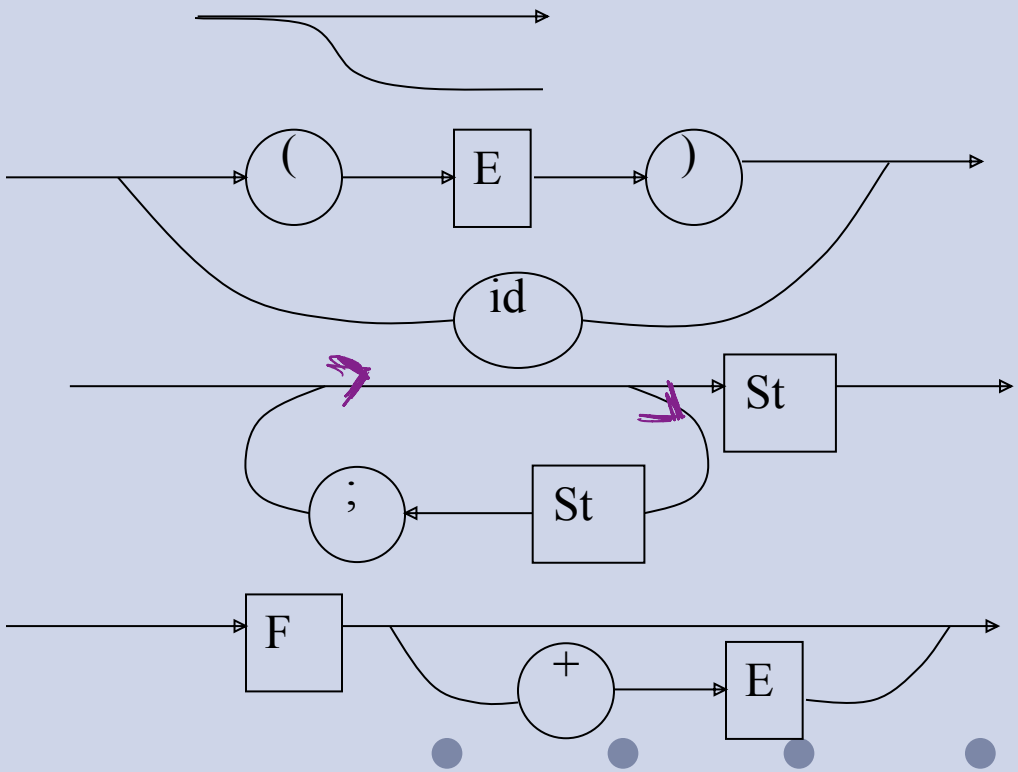
- nonterminals: IfSt
- terminals: id
- sequences and choices:

Examples

X ::= (E) | id

Seq ::= {St ;} St

E ::= F [+ E]



Reading Assignment

- Louden, Compiler construction
  - Chapter 3