Principles of Programming Languages
Lecture 15

Wael Aboulsaadat

wael@cs.toronto.edu

http://portal.utoronto.ca/

Acknowledgment: parts of these slides are based on material by Diane Horton & Eric Joanis @ UoT

References: Scheme by Dybvig
PL Concepts and Constructs by Sethi
Concepts of PL by Sebesta
ML for the Working Prog. By Paulson
Prog. in Prolog by Clocksin and Mellish
PL Pragmatics by Scott

Grammar: introduction

- **Grammar:**
  - A Grammar is a formalism that describes which sequence of terminals are meaningful in a PL. Formally, it is defined as a quadruple (*N*, *T*, *P*, *S*) where:
    - *N* is the set of symbols called *Nonterminals*
    - *T* is the set of symbols called *Terminals*
    - *P* is the set of *productions*
    - *S* subsetof N is the nonterminal called the *starting symbol*
  - Example:

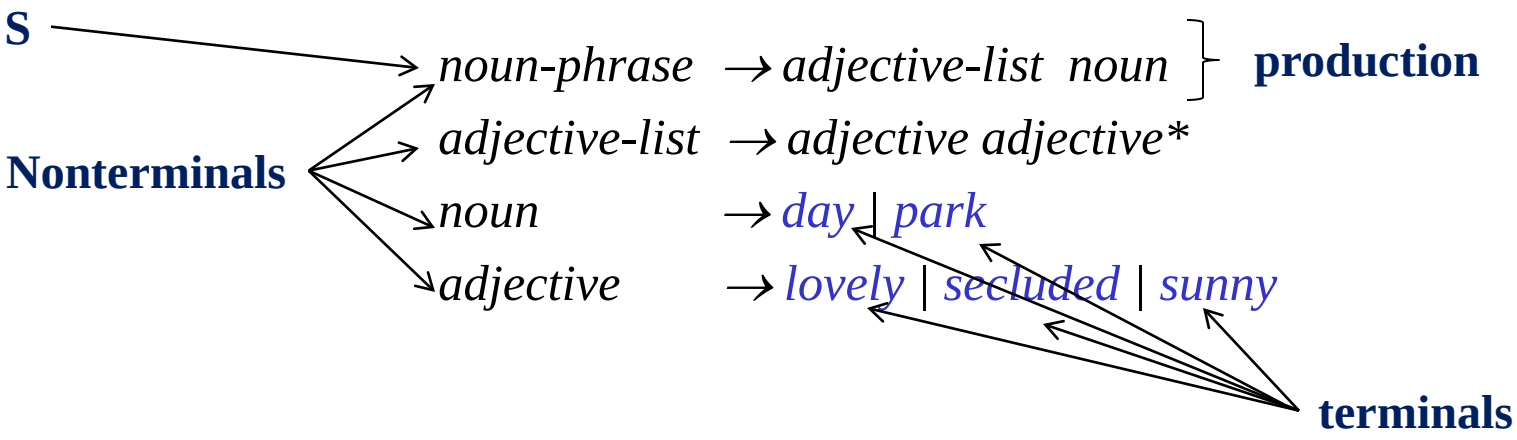
    G = (N,T,P,S) where N = {S} , T = {a,b},

    P={S → aS, S → bS, S → }

- **Production:**
  - A ***production*** is a rule of the form X → Y where X is a string of symbols (*terminals or nonterminals*) containing at least one nonterminal, and Y is a string of symbols (*terminals or nonterminals*)

Grammar: example

- Naive Grammar for noun phrases!



Grammar: definitions

noun-phrase	→	*adjective-list noun*
adjective-list	→	*adjective adjective**
noun	→	*day* | *park*
adjective	→	*lovely* | *secluded* | *sunny*

- **Language**
  - The collection of all valid strings

    e.g. the language of noun phrases

    = { lovely park, sunny lovely park, lovely day, …..}

- **Sentence**
  - A finite sequence of terminals, constructed according to the rules of the grammar for that PL

    e.g. lovely park

Grammar: definitions

- A *derivation* is a sequence of *productions* that begin with the *start* symbol *(noun-phrase in this case)* and derives a valid string in the language called the *yield*

- **E.g.** Is **sunny day** a valid sentence in the language of noun phrases?

  noun-phrase $\Rightarrow$ *adjective-list* day

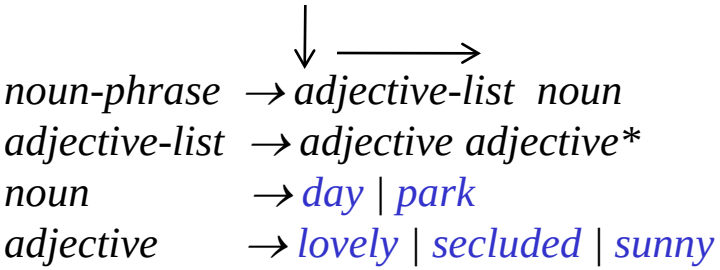
  adjective-list $\Rightarrow$ sunny

- Two *productions* were used to generate the yield sunny day

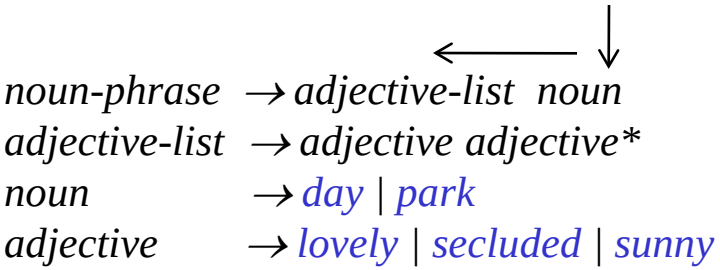
- *Sentential form*
  - A finite sequence of terminals and non-terminals constructed according to the rules of the grammar for that PL

Grammar: derivation types

- **Leftmost Derivation**
  - In a leftmost derivation, the replaced nonterminal is always the leftmost nonterminal.



- **Rightmost Derivation**
  - In a rightmost derivation, the replaced nonterminal is always the rightmost nonterminal.



Grammar: parsers

- **Top-down Parsers**
  1. Start from the grammar
  2. Expand the top most rule
  3. Keep expanding till you get a yield similar to input or error
  - E.g. LL parsers (**L**eft-to-right scan, **Left** most derivation)

- **Bottom-up Parsers**
  1. Start from the input code, read a token
  2. Find the most basic rule that matches that token,
  3. Read one more token, going up the rules till you reach the start symbol
  4. If you cannot reach the start symbol, then it is an error
  - E.g. LR parsers (**L**eft-to-right scan, **R**ight most derivation)

Grammar: example 2

noun-phrase → *adjective-list noun*
adjective-list → *adjective adjective**
noun → *day* | *park*
adjective → *lovely* | *secluded* | *sunny*

- **lovely secluded park**
  - Top-down parsing
  - Scanning input from left to right
  - Left to right derivation

noun-phrase ⇒ *adjective-list noun*
adjective-list ⇒ *adjective adjective**
adjective ⇒ *lovely*
adjective-list ⇒ *lovely adjective**
adjective ⇒ *secluded*
noun-phrase ⇒ *lovely secluded noun*
noun ⇒ *park*
noun-phrase ⇒ *lovely secluded park*

Grammar: example 2

noun-phrase	$\rightarrow$	*adjective-list noun*
adjective-list	$\rightarrow$	*adjective adjective**
noun	$\rightarrow$	*day* | *park*
adjective	$\rightarrow$	*lovely* | *secluded* | *sunny*

- **lovely secluded park**
  - Bottom-up parsing
  - Scanning input from left to right
  - Left to right derivation

lovely $\Rightarrow$ *adjective*

secluded $\Rightarrow$ *adjective*

lovely secluded $\Rightarrow$ *adjective adjective**

park $\Rightarrow$ *noun*

adjective adjective noun* $\Rightarrow$ *noun-phrase*

Grammar: example 3

- **A grammar for expressions**

expression	→ *identifier*
	→ *number*
	→ - *expression*
	→ (*expression*)
	→ *expression operator expression*
operator	→ +
	→ -
	→ *
	→ /

For example:

x+y
Index/5
(x+(m*t))
-10
x
20

Grammar: example 3

- **A grammar for expressions**

expression	→ *identifier*	[1]
	→ *number*	[2]
	→ - *expression*	[3]
	→ (*expression*)	[4]
	→ *expression operator expression*	[5]
operator	→ +	[6]
	→ -	[7]
	→ *	[8]
	→ /	[9]

- **m*x + b** *expression*

Grammar: example 3

- **A grammar for expressions**

expression	→ *identifier*	[1]
	→ *number*	[2]
	→ - *expression*	[3]
	→ (*expression*)	[4]
	→ *expression operator expression*	[5]
operator	→ +	[6]
	→ -	[7]
	→ *	[8]
	→ /	[9]

- **m*x + b** *expression*

Grammar: example 3

- **A grammar for expressions**

expression	→ *identifier*	[1]
	→ *number*	[2]
	→ - *expression*	[3]
	→ (*expression*)	[4]
	→ *expression operator expression*	[5]
operator	→ +	[6]
	→ -	[7]
	→ *	[8]
	→ /	[9]

- **m*x + b** *expression* ⇒ *expression operator expression* *(using 5)*

Grammar: example 3

- **A grammar for expressions**

expression	→ *identifier*	[1]
	→ *number*	[2]
	→ - *expression*	[3]
	→ (*expression*)	[4]
	→ *expression operator expression*	[5]
operator	→ +	[6]
	→ -	[7]
	→ *	[8]
	→ /	[9]

- **m*x + b** *expression* ⇒ *expression operator expression* *(5)*
  ⇒ *expression operator identifier* *(1)*

Grammar: example 3

- **A grammar for expressions**

expression	→ *identifier*	[1]
	→ *number*	[2]
	→ - *expression*	[3]
	→ (*expression*)	[4]
	→ *expression operator expression*	[5]
operator	→ +	[6]
	→ -	[7]
	→ *	[8]
	→ /	[9]

- **m*x + b** *expression* ⇒ *expression operator expression* *(5)*
  ⇒ *expression operator identifier* *(1)*
  ⇒ *expression + b* *(6)*

Grammar: example 3

- **A grammar for expressions**

expression	→ *identifier*	[1]
	→ *number*	[2]
	→ - *expression*	[3]
	→ (*expression*)	[4]
	→ *expression operator expression*	[5]
operator	→ +	[6]
	→ -	[7]
	→ *	[8]
	→ /	[9]

- **m*x + b** *expression* ⇒ *expression operator expression* *(5)*
  ⇒ *expression operator identifier* *(1)*
  ⇒ *expression + b* *(6)*
  ⇒ *expression operator expression + b* *(5)*

Grammar: example 3

- **A grammar for expressions**

expression	→ *identifier*	[1]
	→ *number*	[2]
	→ - *expression*	[3]
	→ (*expression*)	[4]
	→ *expression operator expression*	[5]
operator	→ +	[6]
	→ -	[7]
	→ *	[8]
	→ /	[9]

- **m*x + b**

expression	⇒ *expression operator expression*	*(5)*
	⇒ *expression operator identifier*	*(1)*
	⇒ *expression* + *b*	*(6)*
	⇒ *expression operator expression* + *b*	*(5)*
	⇒ *expression operator x* + *b*	*(1)*

Grammar: example 3

- **A grammar for expressions**

expression	→ *identifier*	[1]
	→ *number*	[2]
	→ - *expression*	[3]
	→ (*expression*)	[4]
	→ *expression operator expression*	[5]
operator	→ +	[6]
	→ -	[7]
	→ *	[8]
	→ /	[9]

- **m*x + b** *expression* ⇒ *expression operator expression* (5)
  ⇒ *expression operator identifier* (1)
  ⇒ *expression + b* (6)
  ⇒ *expression operator expression + b* (5)
  ⇒ *expression operator x + b* (1)
  ⇒ *expression * x + b* (8)

Grammar: example 3

- **A grammar for expressions**

expression	→ *identifier*	[1]
	→ *number*	[2]
	→ - *expression*	[3]
	→ (*expression*)	[4]
	→ *expression operator expression*	[5]
operator	→ +	[6]
	→ -	[7]
	→ *	[8]
	→ /	[9]

- **m*x + b** *expression* ⇒ *expression operator expression* (5)
  ⇒ *expression operator identifier* (1)
  ⇒ *expression* + *b* (6)
  ⇒ *expression operator expression* + *b* (5)
  ⇒ *expression operator x* + *b* (1)
  ⇒ *expression* * *x* + *b* (8)
  ⇒ *identifier* * *x* + *b* (1)
  m * *x* + *b*

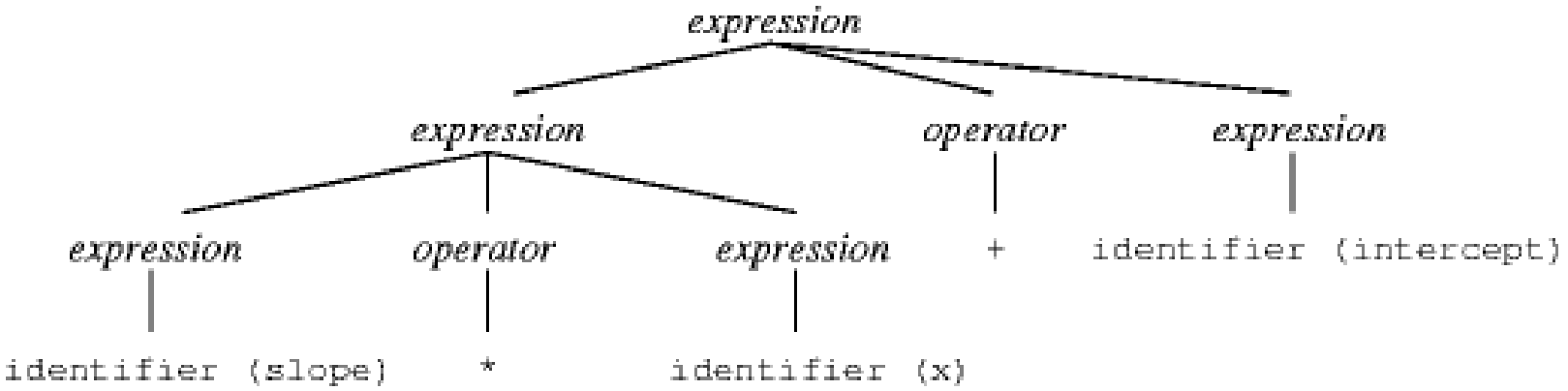
Grammar: parse tree

- **A *parse tree* describes the hierarchical syntactic structure of the sentence based on a given language**

- **In a parse tree**
  - Each internal node is a ***non-terminal***, its children are the rhs of a rule for that non-terminal.
  - All leafs are ***terminals***

Grammar: parse tree example

- **m*x + b** *expression* ⇒ *expression operator expression* (5)
  ⇒ *expression operator identifier* (1)
  ⇒ *expression* + *b* (6)
  ⇒ *expression operator expression* + *b* (5)
  ⇒ *expression operator* *x* + *b* (1)
  ⇒ *expression* * *x* + *b* (8)
  ⇒ *identifier* * *x* + *b* (1)
  m * *x* + *b*



Grammar: example 3 – second derivation

- **A grammar for expressions**

expression	→ *identifier*	[1]
	→ *number*	[2]
	→ - *expression*	[3]
	→ (*expression*)	[4]
	→ *expression operator expression*	[5]
operator	→ +	[6]
	→ -	[7]
	→ *	[8]
	→ /	[9]

- **m*x + b** *expression* ⇒ *expression operator* ~~*expression*~~ *(5)*

Grammar: example 3 – second derivation

- **A grammar for expressions**

expression	→ *identifier*	[1]
	→ *number*	[2]
	→ - *expression*	[3]
	→ (*expression*)	[4]
	→ *expression operator expression*	[5]
operator	→ +	[6]
	→ -	[7]
	→ *	[8]
	→ /	[9]

- **m*x + b** *expression* ⇒ *expression operator* ~~*expression*~~ *(5)*
  ⇒ *expression operator expression operator expression (5)*

Grammar: example 3 – second derivation

- **A grammar for expressions**

expression	→ *identifier*	[1]
	→ *number*	[2]
	→ - *expression*	[3]
	→ (*expression*)	[4]
	→ *expression operator expression*	[5]
operator	→ +	[6]
	→ -	[7]
	→ *	[8]
	→ /	[9]

- **m*x + b** *expression* ⇒ *expression operator* ~~*expression*~~ *(5)*
  ⇒ *expression operator expression operator expression (5)*
  ⇒ *expression operator expression operator identifier (1)*

Grammar: example 3 – second derivation

- **A grammar for expressions**

expression	→ *identifier*	[1]
	→ *number*	[2]
	→ - *expression*	[3]
	→ (*expression*)	[4]
	→ *expression operator expression*	[5]
operator	→ +	[6]
	→ -	[7]
	→ *	[8]
	→ /	[9]

- **m*x + b** *expression* ⇒ *expression operator* $\overleftarrow{\text{expression}}$ *(5)*
  ⇒ *expression operator expression operator expression (5)*
  ⇒ *expression operator expression operator identifier (1)*
  ⇒ *expression operator expression + b (6)*

Grammar: example 3 – second derivation

- **A grammar for expressions**

expression	→ *identifier*	[1]
	→ *number*	[2]
	→ - *expression*	[3]
	→ (*expression*)	[4]
	→ *expression operator expression*	[5]
operator	→ +	[6]
	→ -	[7]
	→ *	[8]
	→ /	[9]

- **m*x + b** *expression* ⇒ *expression operator* $\overleftarrow{\textit{expression}}$ *(5)*
  ⇒ *expression operator expression operator expression (5)*
  ⇒ *expression operator expression operator identifier (1)*
  ⇒ *expression operator expression + b (6)*
  ⇒ *expression operator identifier + b (1)*

Grammar: example 3 – second derivation

- **A grammar for expressions**

expression	→ *identifier*	[1]
	→ *number*	[2]
	→ - *expression*	[3]
	→ (*expression*)	[4]
	→ *expression operator expression*	[5]
operator	→ +	[6]
	→ -	[7]
	→ *	[8]
	→ /	[9]

- **m*x + b** *expression* ⇒ *expression operator* $\overleftarrow{\textit{expression}}$ *(5)*
  ⇒ *expression operator expression operator expression* *(5)*
  ⇒ *expression operator expression operator identifier* *(1)*
  ⇒ *expression operator expression + b* *(6)*
  ⇒ *expression operator identifier + b* *(1)*
  ⇒ *expression * x + b* *(8)*

Grammar: example 3 – second derivation

- **A grammar for expressions**

expression	→ *identifier*	[1]
	→ *number*	[2]
	→ - *expression*	[3]
	→ (*expression*)	[4]
	→ *expression operator expression*	[5]
operator	→ +	[6]
	→ -	[7]
	→ *	[8]
	→ /	[9]

- **m*x + b** *expression*

⇒ *expression operator* ←*expression* (5)
⇒ *expression operator expression operator expression* (5)
⇒ *expression operator expression operator identifier* (1)
⇒ *expression operator expression + b* (6)
⇒ *expression operator identifier + b* (1)
⇒ *expression * x + b* (8)
⇒ *identifier * x + b* (1)

Grammar: example 3 – second derivation

- **A grammar for expressions**

expression	→ *identifier*	[1]
	→ *number*	[2]
	→ - *expression*	[3]
	→ (*expression*)	[4]
	→ *expression operator expression*	[5]
operator	→ +	[6]
	→ -	[7]
	→ *	[8]
	→ /	[9]

- **m*x + b** *expression* ⇒ *expression operator* ~~*expression*~~ *(5)*

⇒ *expression operator expression operator expression* *(5)*

⇒ *expression operator expression operator identifier* *(1)*

⇒ *expression operator expression + b* *(6)*

⇒ *expression operator identifier + b* *(1)*

⇒ *expression * x + b* *(8)*

⇒ *identifier * x + b* *(1)*

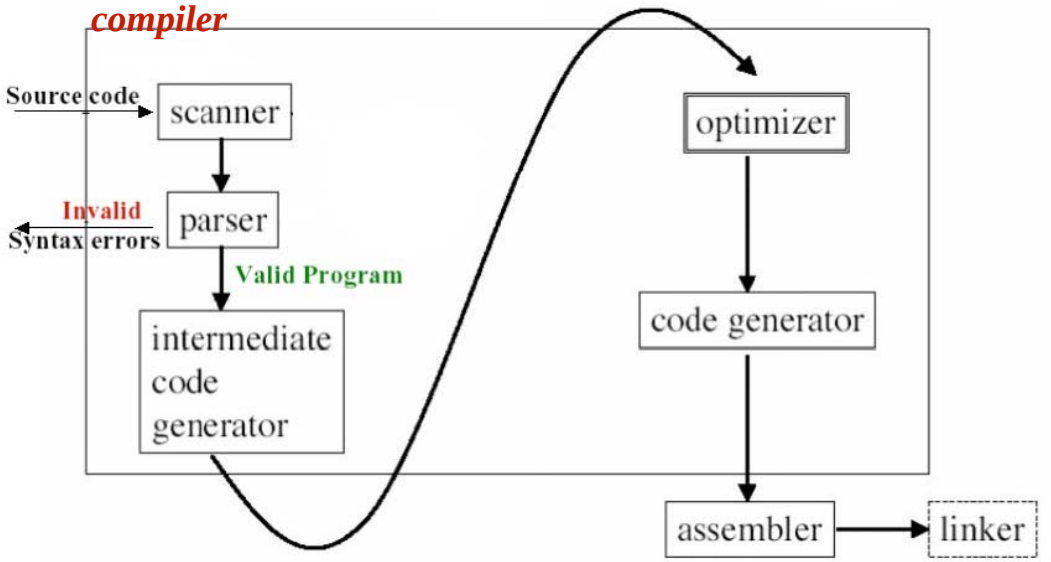
⇒ m * *x + b*

Grammar: ambiguity

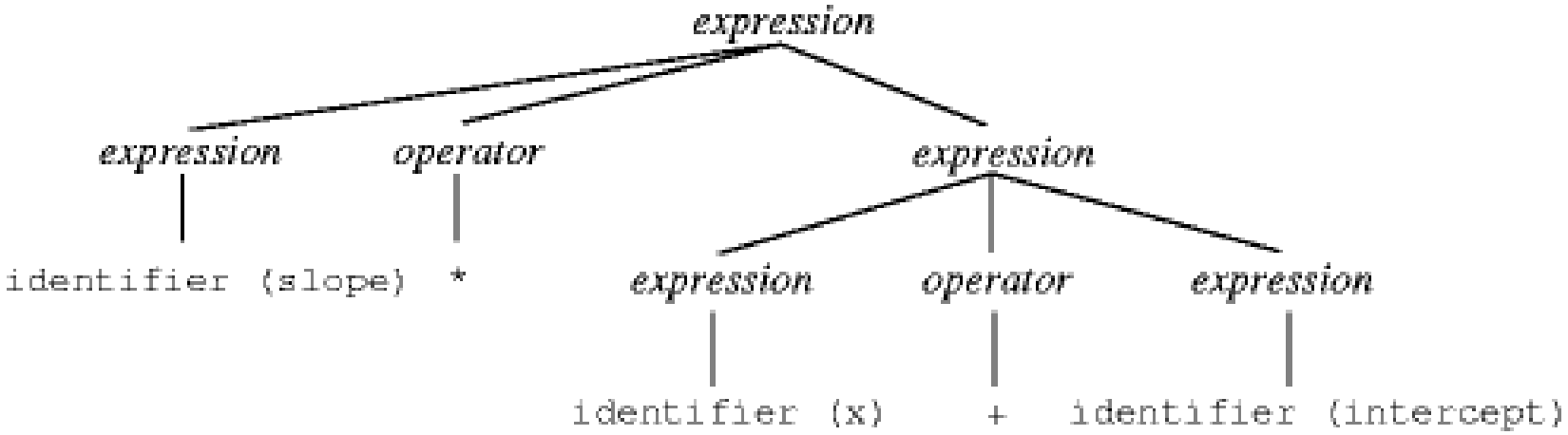
- A grammar that allows multiple parses of a single input string is termed *ambiguous*.

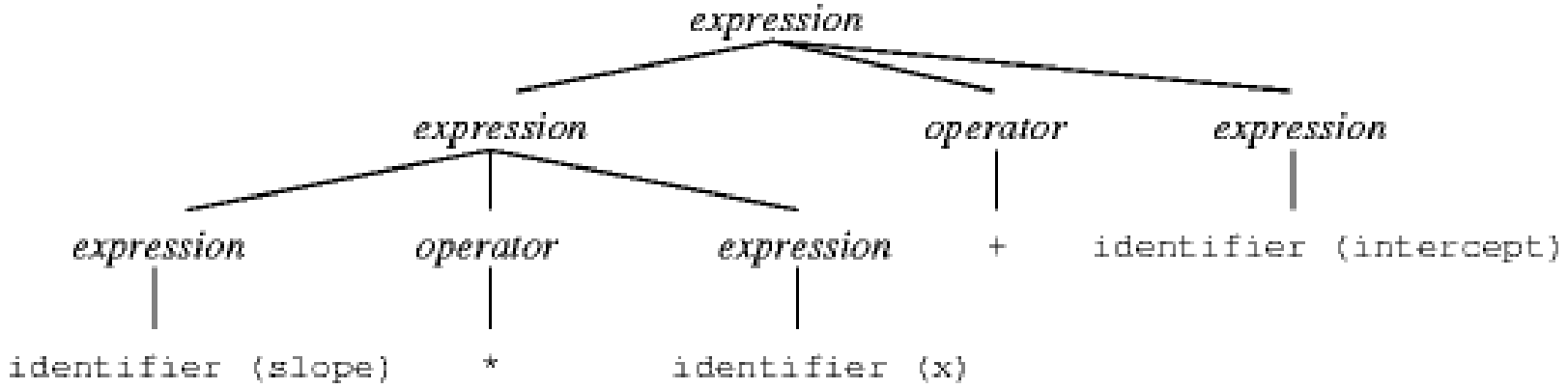
- Ambiguous grammars are really bad (& dangerous)

Grammar: how compiler works again…



Grammar: 2 parse trees for m * x + b





Language Specification: left-to-right will produce a different tree also but…

- **A grammar for expressions**

expression	→ *identifier*	[1]
	→ *number*	[2]
	→ - *expression*	[3]
	→ (*expression*)	[4]
	→ *expression operator expression*	[5]
operator	→ +	[6]
	→ -	[7]
	→ *	[8]
	→ /	[9]

- **m*x + b**

expression ⇒ *expression operator expression* *(using 5)*

⇒ *identifier operator expression* *(using 1)*

⇒ *m operator expression* *(using 8)*

⇒ *m * expression operator expression* *(using 5)*

⇒ *m * x operator expression* *(using 1)*

⇒ *m * x + expression* *(using 6)*

⇒ *m * x + identifier* *(using 1)*

⇒ *m * x + y* *(using 1)*

Grammar: fixing example 3

- **A modified grammar for expressions**

expression	→ *term*
	→ *expression add-op term*
term	→ *factor*
	→ *term mult-op factor*
factor	→ *identifier* | *number* | - *factor* | *expression*
add-op	→ + | -
mult-op	→ * | /

- | means or

Grammar: fixing example 3 – R-to-L derivation

m*x + b

- **A modified grammar for expressions**

expression → *term*

→ *expression add term*

term → *factor*

→ *term mult factor*

factor → *identifier* | *number* | - *factor* | *expression*

add → + | -

mult → * | /

expression ⇒

expression	*add*	*term*	
expression	*add*	*factor*	
expression	*add*	*identifier*	
expression	*add*	*b*	
expression	+	*b*	
term	+	*b*	
term	*mult factor*	+	*b*
term	*mult identifier*	+	*b*
term	*mult*	*x*	+ *b*
term	*	*x*	+ *b*
factor	*	*x*	+ *b*
identifier	*	*x*	+ *b*
m	*	*x*	+ *b*

Grammar: fixing example 3 – L-to-R derivation

↓
m*x + b

- **A modified grammar for expressions**

expression → *term*
→ *expression add term*
term → *factor*
→ *term mult factor*
factor→ *identifier* | *number* | - *factor* | *expression*
add→ + | -
mult→ * | /

expression ⇒
term mult factor
factor mult factor
identifier mult factor
m mult factor
*m * factor*
*m * expression*
*m * expression add term*
*m * term add term*
*m * factor add term*
*m * identifier add term*
*m * x add term*
*m * x + term*

Grammar: L-to-R vs. R-to-L derivation

m*x + b

expression ⇒

term *mult* *factor*

factor *mult* *factor*

identifier *mult* *factor*

m *mult* *factor*

m * *factor*

m * *expression*

m * *expression add term*

m * *term* *add term*

m * *factor* *add term*

m * *identifier* *add term*

m * *x* *add term*

m * *x* + *term*

m * *x* + *b*

expression ⇒

expression *add* *term*

expression *add* *factor*

expression *add* *identifier*

expression *add* *b*

expression + *b*

term + *b*

term *mult factor* + *b*

term *mult identifier* + *b*

term *mult* *x* + *b*

term * *x* + *b*

factor * *x* + *b*

identifier * *x* + *b*

m * *x* + *b*

Grammar: sources of ambiguity

- **Operator Problem:** *associativity and precedence*
  - E.g.
    - Precedence of multiplication/subtraction/addition/…
  - Solution:
    - Change the grammar to reflect operator precedence

      X*Y-Z means ((X*Y) – Z)

- **Substructure Problem:** *extent of a substructure*
  - E.g.
    - Dangling else

Grammar: If-then-else

- **Consider:** if (logic-expression) then
  if (logic-expression) then
  statement 1
  else
  statement 2

- **Grammar:**

  < *if stmt* > :: = if < *logic expression* > then < *stmt* >
  | if < *logic expression* > then < *stmt* > else < *stmt* >
  < *stmt* > :: = < *if stmt* > | . . .

Grammar: If-then-else

- **Grammar:**

  < *if stmt* > :: = if < *logic expression* > then < *stmt* >

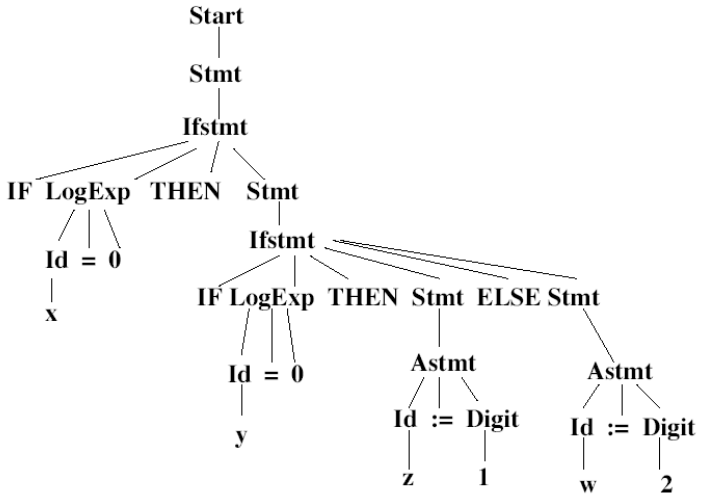
  | if < *logic expression* > then < *stmt* > else < *stmt* >

  < *stmt* > :: = < *if stmt* > | . . .

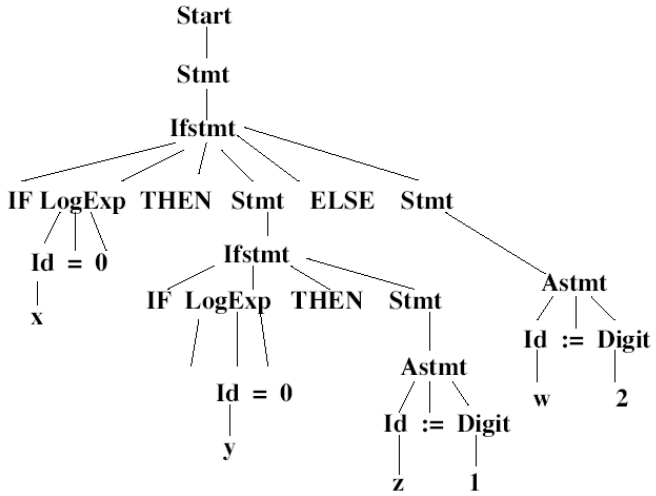
- **Example:**

```
if (x=0) then
    if (y = 0) then
        z := 1
    else
        w := 2
```

IF x = 0 THEN IF y = 0 THEN z := 1 ELSE w := 2;



IF x = 0 THEN IF y = 0 THEN z := 1 ELSE w := 2;



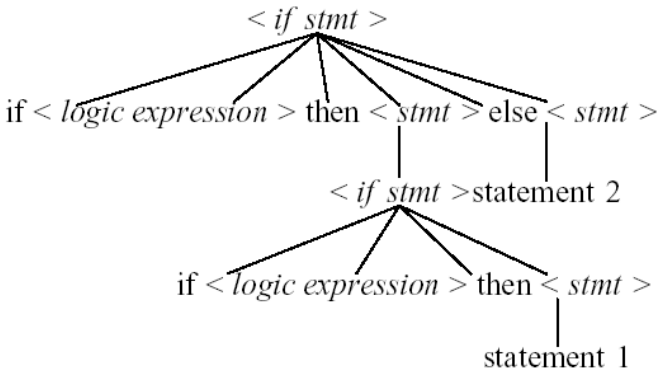
Grammar: If-then-else

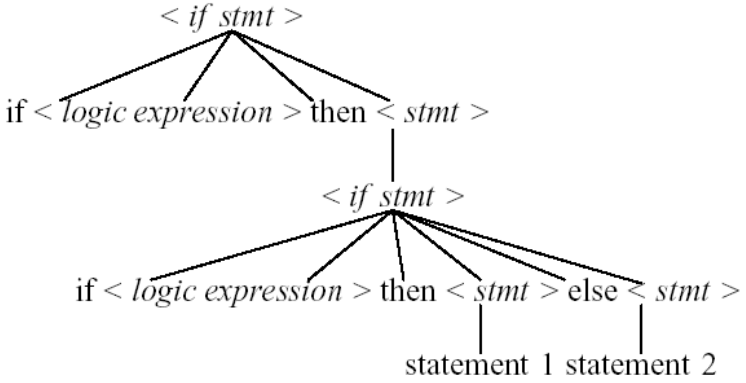
- **Consider:** if (logic-expression) then
  if (logic-expression) then
  statement 1
  else
  statement 2

- **Grammar:**

  < *if stmt* > :: = if < *logic expression* > then < *stmt* >
  | if < *logic expression* > then < *stmt* > else < *stmt* >

  < *stmt* > :: = < *if stmt* > | . . .





Grammar: substructure problem, how to solve ambiguity?

- **Use block structure to enclose if statement** *(e.g. Algol60)*
  - E.g.

```
if x = 0 then
begin
        if y = 0 then
          z:= 1
end
else
          w := 2
```

- **Use statement begin/end markers** *(e.g. Algol68)*
  - E.g.

```
if x = 0 then
    if y = 0 then
        z := 1
    fi
else
    w := 2
fi
```

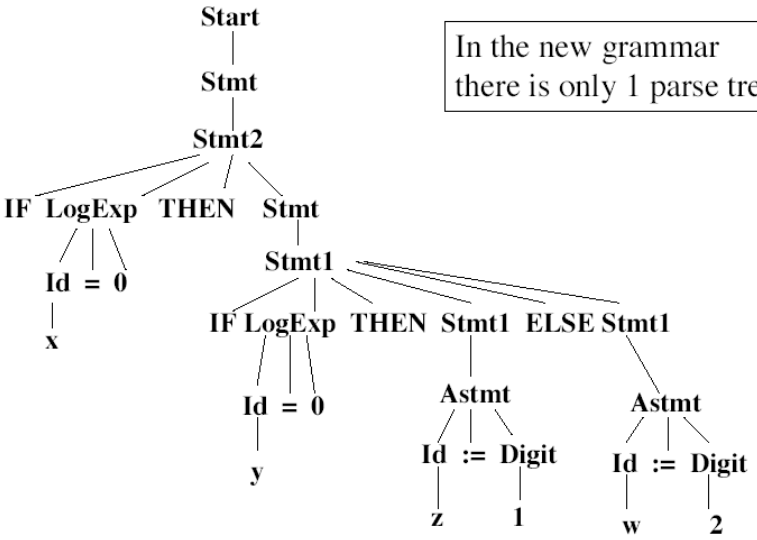
- **Change the if statement grammar to disallow parse tree 2; that is, always associate an else with the closet if** *(e.g. Pascal)*

Grammar: If-then-else , Pascal solution

Start ::=	Stmt
Stmt ::=	Stmt1 | Stmt2
Stmt1 ::=	IF LogExp THEN Stmt1 ELSE Stmt1 | Astmt
Stmt2 ::=	IF LogExp THEN Stmt | IF LogExp THEN Stmt1 ELSE Stmt2
Astmt ::=	Id := Digit
Digit ::=	0|1|2|3|4|5|6|7|8|9
LogExp ::=	Id = 0
Id ::=	a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

Note: only if statements with IF..THEN..ELSE are allowed after the THEN clause of an IF-THEN-ELSE statement.

IF x = 0 THEN IF y = 0 THEN z := 1 ELSE w := 2;



In the new grammar there is only 1 parse tree!

Grammar: backus Naur form

- **Backus Naur Form (BNF) is a metalanguage for describing programming languages**

- **A BNF grammar *is a context free grammar***

- **Notation:**
  - Nonterminals are enclosed in angle brackets, i.e. "<" and ">"
  - Uses "::=" instead of "→" in productions
  - Productions having the same left hand side can be grouped together using the alteration symbol "|"
    - e.g <S> ::= a <S> | b <s> |
  - Lists are described using recursive rules
    - e.g. <ident.list> ::= identifier | identifier, <ident.list>

Grammar: extended BNF

- **Notation:**
  - (…|…|…) Any one of the alterations
  - […] Optional part
  - (…)* or {…} or […]* repeat zero or more times
  - (…)⁻ or {…}⁻ or […]⁻ repeat one or more times
  - "x" or 'x' terminal symbol
  - Unquoted words non-terminal symbol

- **Example:**
  - Using the above notation

    < *expression* > : : = < *expression* > + < *term* >
    | < *expression* > - < *term* >
    | < *term* >

    could be written in the form of an iteration, as follows:

    < *expression* > : : = < *term* > [(+ | -) < *term* >]*