CIT3136 - Lecture 5
Context-Free Grammars and Parsing

Definition of a Context-free Grammar:

- **An alphabet or set of basic symbols (like regular expressions, only now the symbols are whole tokens, not chars), including ε. (*Terminals*)**
- **A set of *names* for structures (like *statement, expression, definition*). (*Non-terminals*)**
- **A set of grammar *rules* expressing the structure of each name. (*Productions*)**
- **A *start* symbol (the name of the most general structure** — *compilation_unit* **in C).**

Basic Example: Simple integer arithmetic expressions

2 non-terminals

exp → *exp op exp* | **(** *exp* **)** | ***number***

op → **+** | **-** | *****

6 terminals

6 productions (3 on each line)

In what way does such a CFG differ from a regular expression?

```
digit = 0|1|…|9
number = digit digit*
```

Recursion!

Recursive rules

"Base" rule

CFGs are designed to represent recursive (i.e. nested) structures

But consequences are huge:

- **The structure of a matched string is no longer given by just a sequence of symbols (lexeme), but by a tree (parse tree)**
- **Recognizers are no longer finite, but may have arbitrary data size, and must have some notion of stack.**

Recognition Process is much more complex:

- **Algorithms can use stacks in many different ways.**
- **Nondeterminism is much harder to eliminate.**
- **Even the number of states can vary with the algorithm (only 2 states necessary if stack is used for “state”structure.**

Major Consequence: Many parsing algorithms, not just one

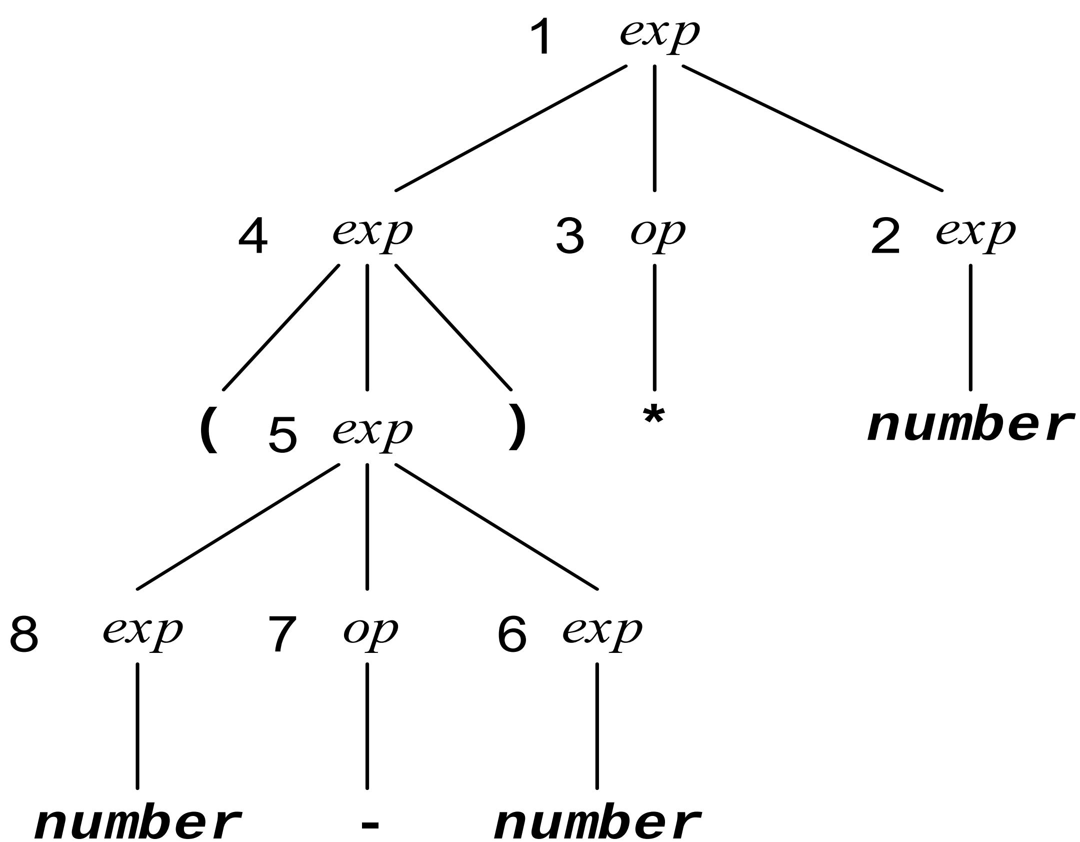
- Top down
  - Recursive descent (hand choice)
  - “Predictive” table-driven, “LL” (outdated)
- Bottom up
  - “LR” and its cousin “LALR” (machine-generated choice [Yacc/Bison])
  - Operator-precedence (outdated)

Structural Issues First!

Express matching of a string [“`(34-3)*42`”] by a *derivation:*

(1)	$exp \Rightarrow exp\ op\ exp$	[$exp \rightarrow exp\ op\ exp$]
(2)	$\Rightarrow exp\ op$ ***number***	[$exp \rightarrow$ ***number***]
(3)	$\Rightarrow exp$ ***** ***number***	[$op \rightarrow$ *****]
(4)	$\Rightarrow$ **(** *exp* **)** ***** ***number***	[$exp \rightarrow$ **(** *exp* **)**]
(5)	$\Rightarrow$ **(** *exp op exp* **)** ***** ***number***	[$exp \rightarrow exp\ op\ exp$]
(6)	$\Rightarrow$ **(***exp op* ***number*****)** ***** ***number***	[$exp \rightarrow$ ***number***]
(7)	$\Rightarrow$ **(***exp* **-** ***number*****)** ***** ***number***	[$op \rightarrow$ **-**]
(8)	$\Rightarrow$ **(*number* - *number*)*number**	[$exp \rightarrow$ ***number***]

Abstract the structure of a derivation to a parse tree:



Derivations can vary, even when the parse tree doesn't:

A *leftmost* derivation (Slide 8 was a *rightmost*):

(1) exp	$\Rightarrow$ exp op exp	[exp $\rightarrow$ exp op exp]
(2)	$\Rightarrow$ (exp) op exp	[exp $\rightarrow$ (exp)]
(3)	$\Rightarrow$ (exp op exp) op exp	[exp $\rightarrow$ exp op exp]
(4)	$\Rightarrow$ (number op exp) op exp	[exp $\rightarrow$ number]
(5)	$\Rightarrow$ (number - exp) op exp	[op $\rightarrow$ -]
(6)	$\Rightarrow$ (number - number) op exp	[exp $\rightarrow$ number]
(7)	$\Rightarrow$ (number - number) * exp	[op $\rightarrow$ *]
(8)	$\Rightarrow$ (number - number) * number	[exp $\rightarrow$ number]

A leftmost derivation corresponds to a (top-down) preorder traversal of the parse tree.

A rightmost derivation corresponds to a (bottom-up) postorder traversal, but in reverse.

Top-down parsers construct leftmost derivations.

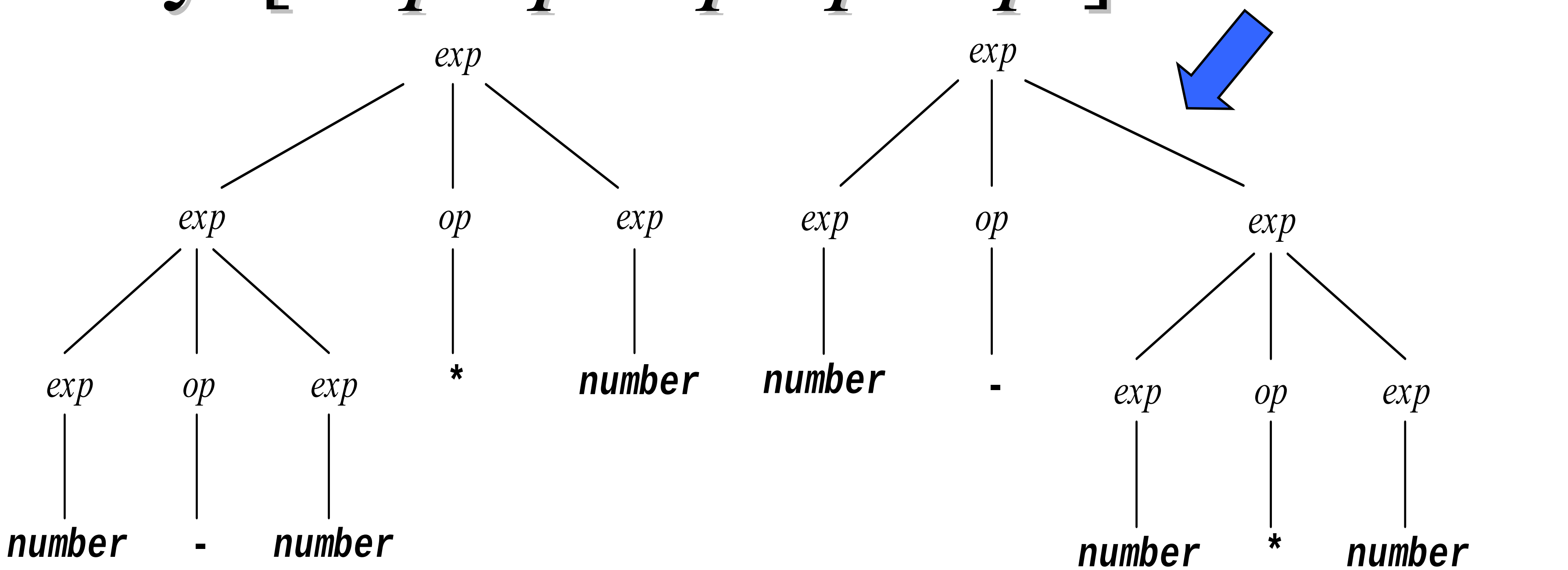
(LL = Left-to-right traversal of input, constructing a Leftmost derivation)

Bottom-up parsers construct rightmost derivations in reverse order.

(LR = Left-to-right traversal of input, constructing a Rightmost derivation)

But what if the parse tree *does* vary?[*exp op exp op exp*]

Correct one



The grammar is ambiguous, but why should we care? *Semantics!*

Principle of Syntax-directed Semantics

The parse tree will be used as the basic model; semantic content will be attached to the tree; thus the tree should reflect the structure of the eventual semantics (*semantics-based syntax* would be a better term)

Sources of Ambiguity:

- **Associativity and precedence of operators**
- **Sequencing**
- **Extent of a substructure (dangling else)**
- **“Obscure” recursion (unusual)**
  – ***exp*** $\rightarrow$ ***exp exp***

Dealing with ambiguity

- **Disambiguating rules**
- **Change the grammar (but not the language!)**
- **Can all ambiguity be removed?**
  - **Backtracking can handle it, but the expense is great**

Example: integer arithmetic

exp → ***exp addop term*** **|** ***term***

addop → **+ | -**

term → ***term mulop factor*** **|** ***factor***

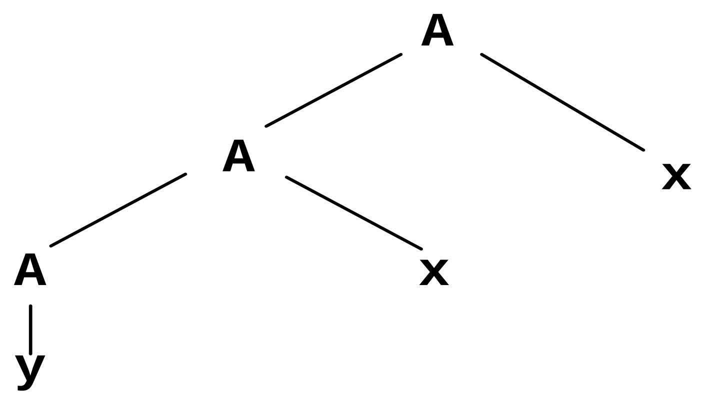
mulop → *****

factor → **(** ***exp*** **)** **|** ***number***

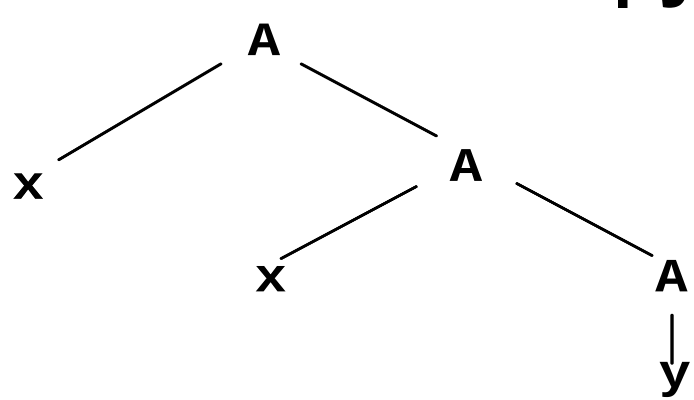
Precedence "cascade"

Repetition and Recursion

- **Left recursion: $A \rightarrow A\ x \mid y$**
  - **yxx:**



- **Right recursion: $A \rightarrow x\ A \mid y$**
  - **xxy:**



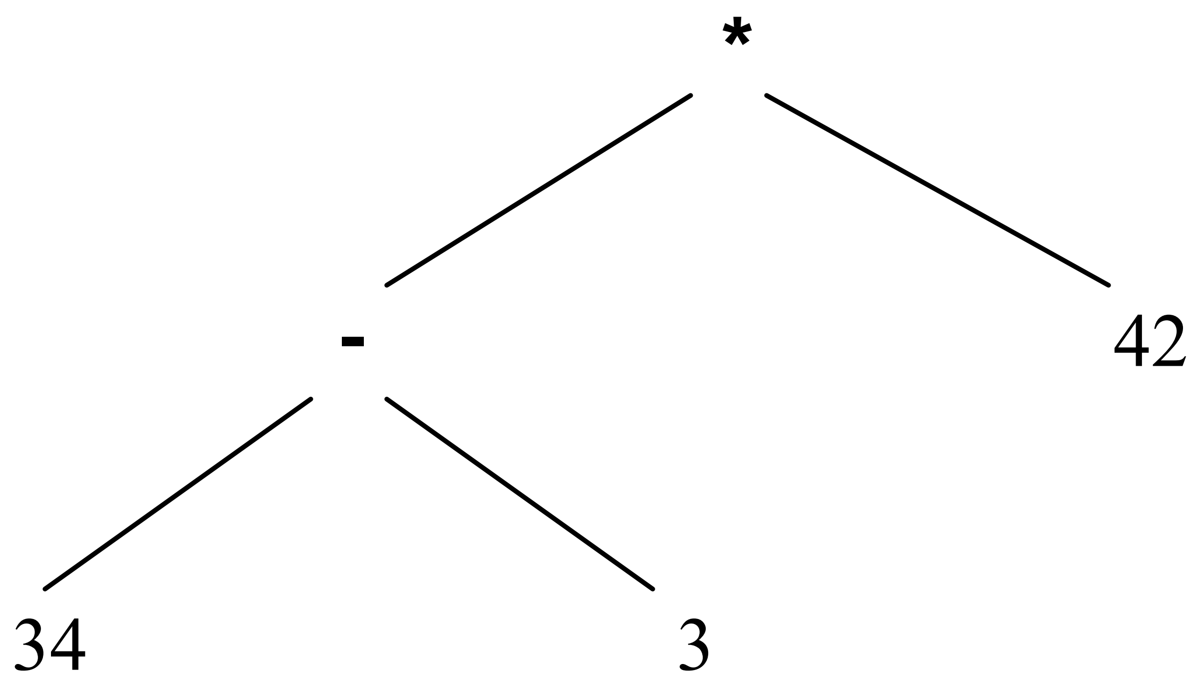
Repetition & Recursion, cont.

- **Sometimes we care which way recursion goes: operator associativity**
- **Sometimes we don’t: statement and expression sequences**
- **Parsing always has to pick a way!**
- **The tree may remove this information (see next slide)**

Abstract Syntax Trees

- **Express the essential structure of the parse tree only**
- **Leave out parens, cascades, and “don’t-care” repetitive associativity**
- **Corresponds to actual internal tree structure produced by parser**
- **Use sibling lists for “don’t care” repetition: s1 --- s2 --- s3**

Previous Example [(34-3)*42]



Data Structure

```
typedef enum {Plus,Minus,Times} OpKind;
typedef enum {OpK,ConstK} ExpKind;
typedef struct streenode
{ ExpKind kind;
  OpKind op;
  struct streenode *lchild,*rchild;
  int val;
} STreeNode;
typedef STreeNode *SyntaxTree;
```

Or (using a union):

```
typedef enum {Plus,Minus,Times} OpKind;
typedef enum {OpK,ConstK} ExpKind;
typedef struct streenode
{ ExpKind kind;
  struct streenode *lchild,*rchild;
  union {
    OpKind op;
    int val; } attribute;
} STreeNode;
typedef STreeNode *SyntaxTree;
```

Sequence Examples

- ***stmt-seq*** $\rightarrow$ ***stmt*** **;** ***stmt-seq*** | ***stmt***
  one **or more stmts** ***separated*** **by a ;**
- ***stmt-seq*** $\rightarrow$ ***stmt*** **;** ***stmt-seq*** | ε
  zero **or more stmts** ***terminated*** **by a ;**
- ***stmt-seq*** $\rightarrow$ ***stmt-seq*** **;** ***stmt*** | ***stmt***
  one **or more stmts** ***separated*** **by a ;**
- ***stmt-seq*** $\rightarrow$ ***stmt-seq*** **;** ***stmt*** | ε
  zero **or more stmts** ***preceded*** **by a ;**

“Obscure” Ambiguity Example

Incorrect attempt to add unary minus:

$exp \rightarrow exp\ addop\ term \mid term \mid$ **-** exp

$addop \rightarrow$ **+** | **-**

$term \rightarrow term\ mulop\ factor \mid factor$

$mulop \rightarrow$ *****

$factor \rightarrow$ **(** exp **)** | ***number***

Ambiguity Example, continued

- **Better: (only one at beg. of an exp)**

  $exp \rightarrow exp\ addop\ term \mid term \mid -\ term$

- **Or maybe: (many at beg. of term)**

  $term \rightarrow -\ term \mid term1$

  $term1 \rightarrow term1\ mulop\ factor \mid factor$

- **Or maybe: (many anywhere)**

  $factor \rightarrow (\ exp\ ) \mid \textbf{number} \mid -\ factor$

Dangling else ambiguity

$statement \rightarrow if\text{-}stmt \mid$ **`other`**

$if\text{-}stmt \rightarrow$ **`if (`** exp **`)`** $statement$

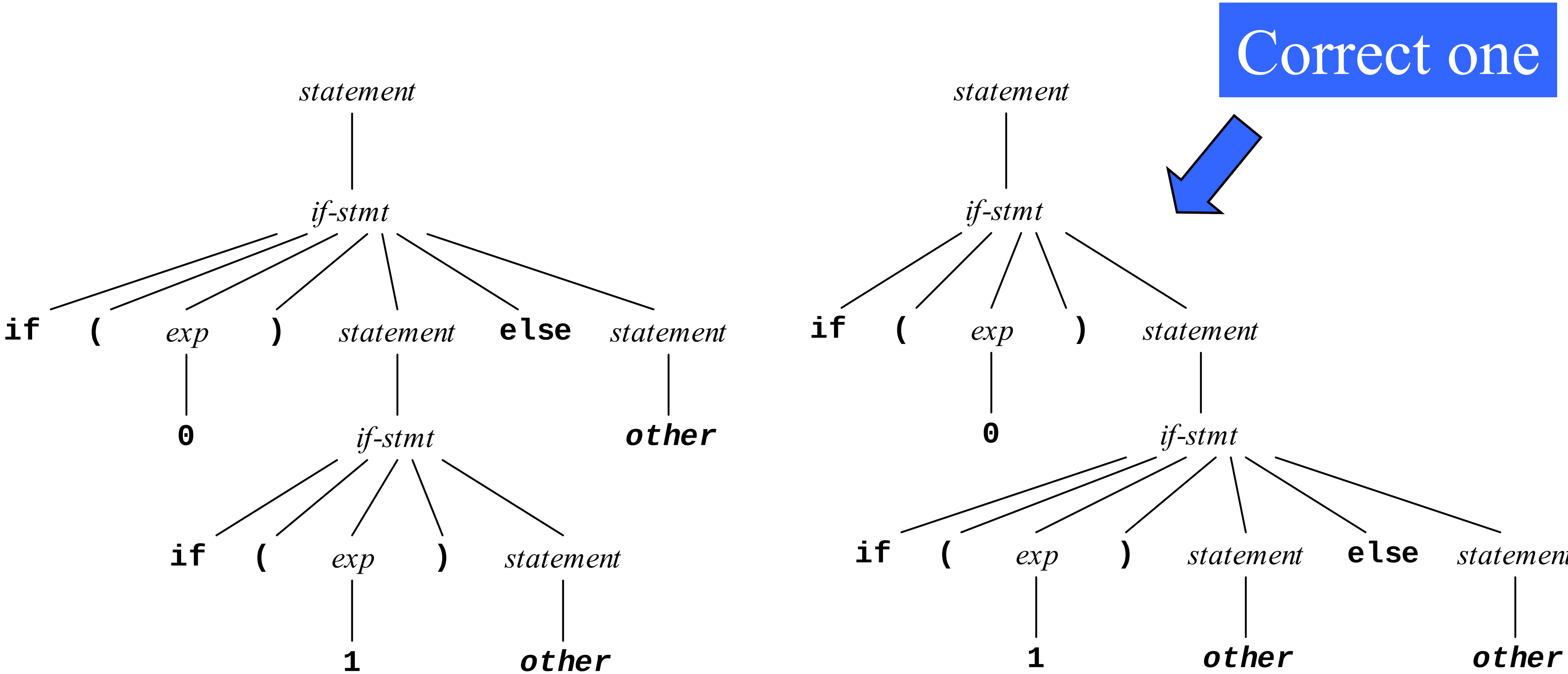
$\mid$ **`if (`** exp **`)`** $statement$ **`else`** $statement$

$exp \rightarrow$ **`0`** $\mid$ **`1`**

The following string has two parse trees:

```
if(0) if(1) other else other
```

Parse trees for dangling else:



Disambiguating Rule:

An else part should always be associated with the nearest if-statement that does not yet have an associated else-part.

(*Most-closely nested rule*: easy to state, but hard to put into the grammar itself.)

Note that a "bracketing keyword" can remove the ambiguity:

Bracketing keyword

if-stmt → `if` (*exp*) *stmt* **end**

| `if` (*exp*) *stmt* `else` *stmt* ***end***

Extra Notation:

- **So far: Backus-Naur Form (BNF)**
  - **Metasymbols are |** $\rightarrow$ ε
- **Extended BNF (EBNF):**
  - **New metasymbols [...] and {...}**
  - ε **largely eliminated by these**
- **Parens? Maybe yes, maybe no:**
  - ***exp*** $\rightarrow$ ***exp* (+ | -) *term* | *term***
  - ***exp*** $\rightarrow$ ***exp* + *term* | *exp* - *term* | *term***

EBNF Metasymbols:

- **Brackets [...] mean "optional" (like ? in regular expressions):**
  - *exp* → *term* '**|**' *exp* | *term* **becomes:**
    exp → *term* ['**|**' *exp*]
  - *if-stmt* → `if` (*exp*) *stmt*
    | `if` (*exp*) *stmt* `else` *stmt*
    becomes:
    if-stmt → `if` (*exp*) *stmt* [`else` *stmt*]
- **Braces {...} mean "repetition" (like * in regexps - see next slide)**

Braces in EBNF

- **Replace *only* left-recursive repetition:**
  - $exp \rightarrow exp + term \mid term$ **becomes:**
    $exp \rightarrow term \{ + term \}$
- **Left associativity still implied**
- **Watch out for choices:**
  - $exp \rightarrow exp + term \mid exp - term \mid term$
    is not the same as
    $exp \rightarrow term \{ + term \} \mid term \{ - term \}$

Simple Expressions in EBNF

$exp \rightarrow term \{ \ addop \ term \ \}$

$addop \rightarrow$ **+** | **-**

$term \rightarrow factor \{ \ mulop \ factor \ \}$

$mulop \rightarrow$ *****

$factor \rightarrow$ **(** exp **)** | ***number***

Final Notational Option: Syntax Diagrams (from EBNF):