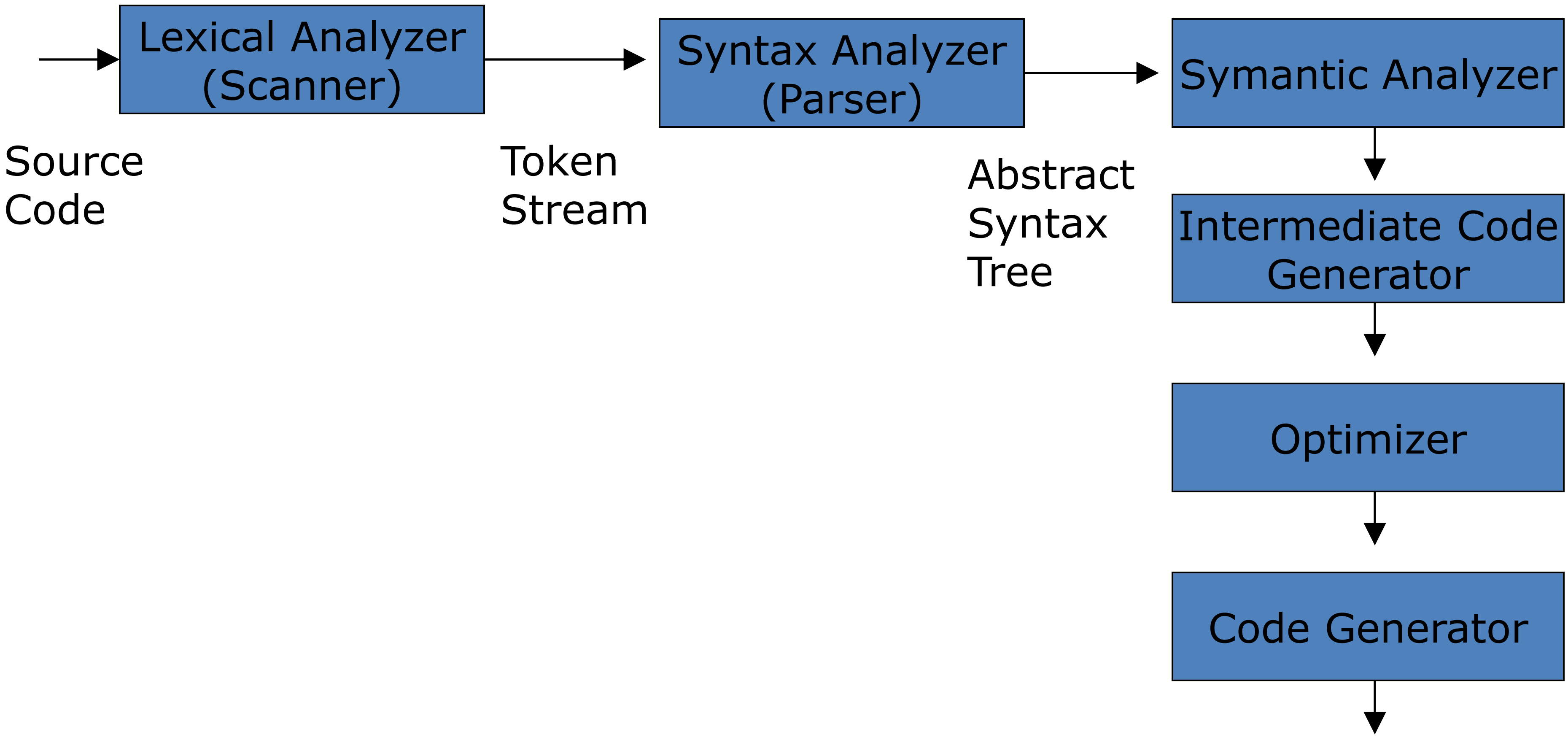
Introduction to Compiler Design

Lesson 7:

Parsers – Context Free Grammars

Compilers



Lexical Analyzer (Scanner)
Syntax Analyzer (Parser)
Symantic Analyzer
Source Code
Token Stream
Abstract Syntax Tree
Intermediate Code Generator
Optimizer
Code Generator

Scanner

Source Code:

```
position = initial + rate * 60 ;
```

Corresponding Tokens:

IDENT(position)
ASSIGN
IDENT(position)
PLUS
IDENT(rate)
TIMES
INT-LIT(60)
SEMI-COLON

Limitations of RE and FSA

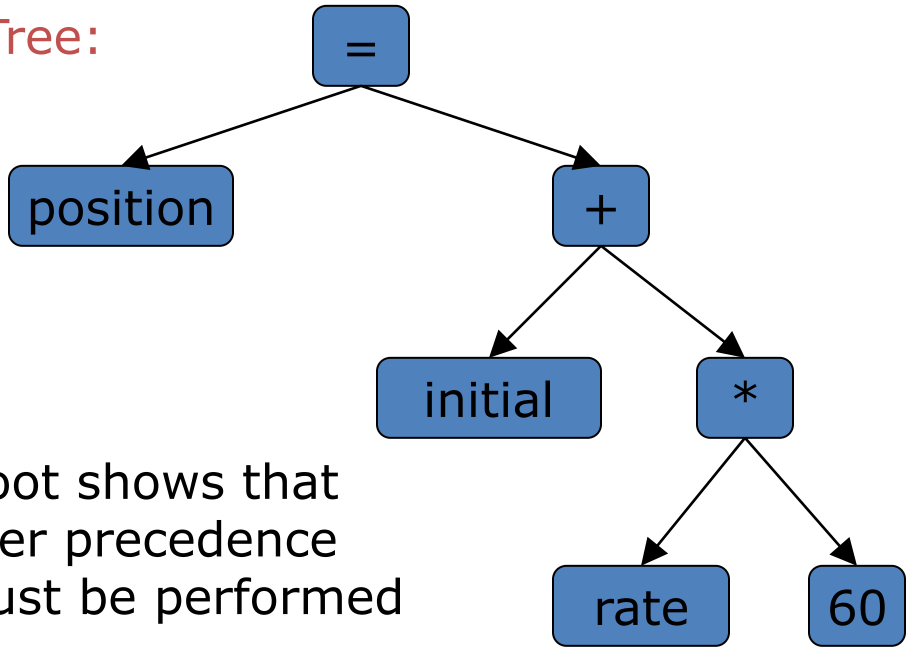
- Regular Expressions and Finite State Automata cannot express all languages
- For Example the language that consists of all balanced parenthesis: () and ((())) and ((((((()))))))
- Parsers can recognize more types of languages than RE or FSA

Parsing Example

Source Code:

position = initial + rate * 60 ;

Abstract-Syntax Tree:



- Interior nodes are **operators**.
- A node's children are **operands**.
- Each subtree forms "logical unit"

  e.g., the subtree with * at its root shows that because multiplication has higher precedence than addition, this operation must be performed as a unit (*not* initial+rate).

Parsers

- Input: Sequence of Tokens
- Output: a representation of program
  - Often AST, but could be other things
- Also find syntax errors
- CFGs used to define Parser
  (Context Free Grammar)

Context Free Grammars

stmt → **if (** expr **)** stmt **else** stmt

terminals **Rule or Production** **non-terminals**

Context Free Grammars (CFGs)

A CFG is a 4-tuple (N,Σ,P,S)

- N is a set of non-terminals, e.g., A, B, S, …
- Σ is the set of terminals
- P is a set of production rules
- S∈N is the initial non-terminal symbol (“start symbol”)

Context Free Grammars (CFGs)

A CFG is a 4-tuple (N,Σ,P,S)

- N is a set of non-terminals, e.g., A, B, S…
- Σ is the set of terminals
- P is a set of production rules
- S (in N) is the initial non-terminal symbol

Placeholder / interior nod
in the parse tree

Tokens from scanner

Rules for deriving string

If not otherwise specified, use the non-terminal that appears on the LHS of the first production as the start

Production

$$LHS \rightarrow RHS$$

Single nonterminal symbol

Expression: Sequence of terminals and nonterminals

Examples:

S → '(' S ')'

S → ε

Production

$$\text{stmt} \rightarrow \underbrace{\textbf{if (}\ \text{expr}\ \textbf{)}\ \text{stmt}\ \textbf{else}\ \text{stmt}}_{\text{Sequence of zero or more terminals and non-terminals}}$$

Sequence of **zero** or more terminals and non-terminals

Single non-terminal

An Example Grammar

An Example Grammar

<u>Terminals</u>
begin
end
semicolon
assign
id
plus

An Example Grammar

For readability, bold and lowercase

Terminals
begin
end
semicolon
assign
id
plus

An Example Grammar

For readability, bold and lowercase

Terminals

begin
end
Program boundary
semicolon
assign
id
plus

An Example Grammar

For readability, bold and lowercase

<u>**Terminals**</u>

begin
end
semicolon
assign
id
plus

Program boundary

Represents “;”
Separates statements

An Example Grammar

For readability, bold and lowercase

<u>Terminals</u>

begin
end
Program boundary
semicolon
Represents ";"
Separates statements
assign
id
Represents "=" in an assignment statement
plus

An Example Grammar

For readability, bold and lowercase

Terminals

begin Program
end boundary
semicolon Represents ";" Separates statements
assign Represents "=" in an assignment statement
id Identifier / variable name
plus

An Example Grammar

For readability, bold and lowercase

Terminals

begin
end
Program boundary

semicolon
Represents “;”
Separates statements

assign
Represents “=“ in an assignment statement

id
Identifier / variable name

plus
Represents “+“ operator in an expression

An Example Grammar

For readability, bold and lowercase

Terminals

begin
end
semicolon
assign
id
plus

Nonterminals

Prog
Stmts
Stmt
Expr

An Example Grammar

For readability, bold and lowercase

<u>Terminals</u>
begin
end
semicolon
assign
id
plus

For readability, Italics and UpperCamelCase

<u>Nonterminals</u>
Prog
Stmts
Stmt
Expr

An Example Grammar

For readability, bold and lowercase

Terminals

begin
end
semicolon
assign
id
plus

For readability, Italics and UpperCamelCase

Nonterminals

Prog Root of the parse tree
Stmts
Stmt
Expr

An Example Grammar

For readability, bold and lowercase

Terminals

begin
end
semicolon
assign
id
plus

For readability, Italics and UpperCamelCase

Nonterminals

Prog **Root of the parse tree**
Stmts **List of statements**
Stmt
Expr

An Example Grammar

For readability, bold and lowercase

Terminals

begin
end
semicolon
assign
id
plus

For readability, Italics and UpperCamelCase

Nonterminals

Prog Root of the parse tree
Stmts List of statements
Stmt A single statement
Expr

An Example Grammar

For readability, bold and lowercase

Terminals

begin
end
semicolon
assign
id
plus

For readability in italics

Nonterminals

Prog Root of the parse tree
Stmts List of statements
Stmt A single statement
Expr A mathematical expression

An Example Grammar

For readability, bold and lowercase

Terminals

begin
end
semicolon
assign
id
plus

Defines the syntax of legal programs

Productions

Prog	→ **begin** Stmts **end**
Stmts	→ Stmts **semicolon** Stmt
	| Stmt
Stmt	→ **id assign** Expr
Expr	→ **id**
	| Expr **plus id**

For readability in italics

Nonterminals

Prog
Stmts
Stmt
Expr

Productions

1. Prog → **begin** *Stmts* **end**

2. Stmts → *Stmts* **semicolon** *Stmt*

3. | *Stmt*

4. Stmt → **id assign** *Expr*

5. Expr → **id**

6. | *Expr* **plus id**

Productions

1. *Prog* → **begin** *Stmts* **end**
2. *Stmts* → *Stmts* **semicolon** *Stmt*
3. | *Stmt*
4. *Stmt* → **id assign** *Expr*
5. *Expr* → **id**
6. | *Expr* **plus id**

Derivation Sequence

Productions

Parse Tree

1. *Prog* → **begin** *Stmts* **end**
2. *Stmts* → *Stmts* **semicolon** *Stmt*
3. | *Stmt*
4. *Stmt* → **id assign** *Expr*
5. *Expr* → **id**
6. | *Expr* **plus id**

Derivation Sequence

Productions

1. *Prog* → **begin** *Stmts* **end**
2. *Stmts* → *Stmts* **semicolon** *Stmt*
3. | *Stmt*
4. *Stmt* → **id assign** *Expr*
5. *Expr* → **id**
6. | *Expr* **plus id**

Parse Tree

Derivation Sequence

Key

terminal

Nonterminal

Rule used

Productions

1. *Prog* → **begin** *Stmts* **end**
2. *Stmts* → *Stmts* **semicolon** *Stmt*
3. | *Stmt*
4. *Stmt* → **id assign** *Expr*
5. *Expr* → **id**
6. | *Expr* **plus id**

Parse Tree

Prog

Derivation Sequence

Prog

Key

terminal

Nonterminal

Rule used

Productions

1. *Prog* → **begin** *Stmts* **end**
2. *Stmts* → *Stmts* **semicolon** *Stmt*
3. | *Stmt*
4. *Stmt* → **id assign** *Expr*
5. *Expr* → **id**
6. | *Expr* **plus id**

Parse Tree

Prog

Derivation Sequence

1 *Prog* ⇒ **begin** Stmts **end**

Key

terminal

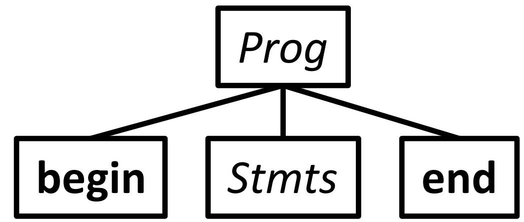
Nonterminal

Rule used

Productions

1. *Prog* → **begin** *Stmts* **end**
2. *Stmts* → *Stmts* **semicolon** *Stmt*
3. | *Stmt*
4. *Stmt* → **id assign** *Expr*
5. *Expr* → **id**
6. | *Expr* **plus id**

Parse Tree



Derivation Sequence

1 *Prog* ⇒ **begin** Stmts **end**

Key

terminal

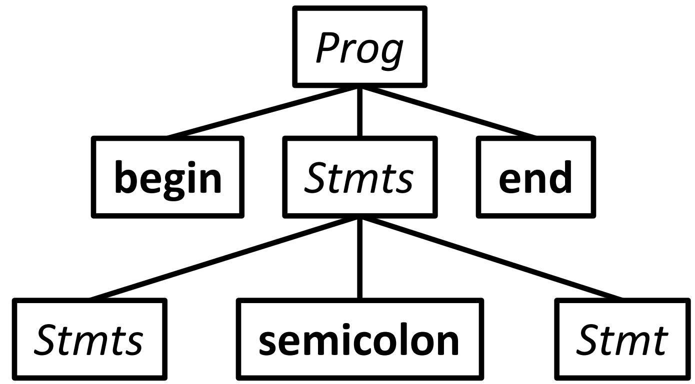
Nonterminal

Rule used

Productions

1. *Prog* → **begin** *Stmts* **end**
2. *Stmts* → *Stmts* **semicolon** *Stmt*
3. | *Stmt*
4. *Stmt* → **id assign** *Expr*
5. *Expr* → **id**
6. | *Expr* **plus id**

Parse Tree



Derivation Sequence

1 *Prog* ⇒ **begin** Stmts **end**

2 ⇒ **begin** Stmts **semicolon** Stmt **end**

Key

terminal

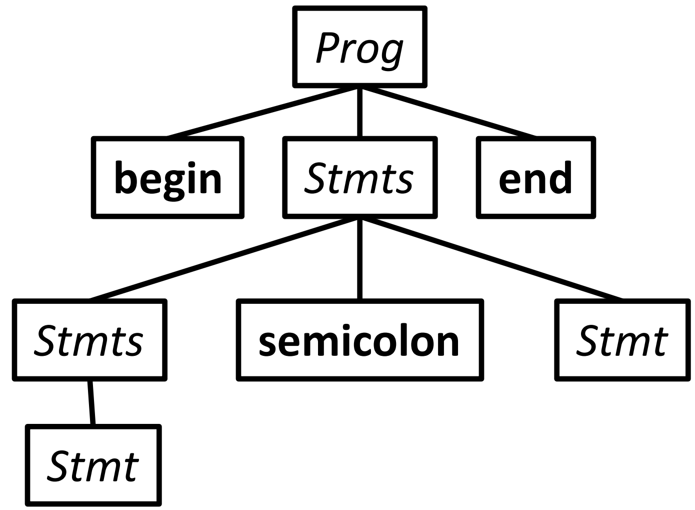
Nonterminal

Rule used

Productions

1. *Prog* → **begin** *Stmts* **end**
2. *Stmts* → *Stmts* **semicolon** *Stmt*
3. | *Stmt*
4. *Stmt* → **id assign** *Expr*
5. *Expr* → **id**
6. | *Expr* **plus id**

Parse Tree



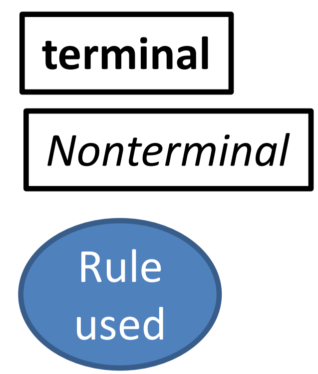
Derivation Sequence

1 *Prog* ⇒ **begin** Stmts **end**

2 ⇒ **begin** Stmts **semicolon** Stmt **end**

3 ⇒ **begin** Stmt **semicolon** Stmt **end**

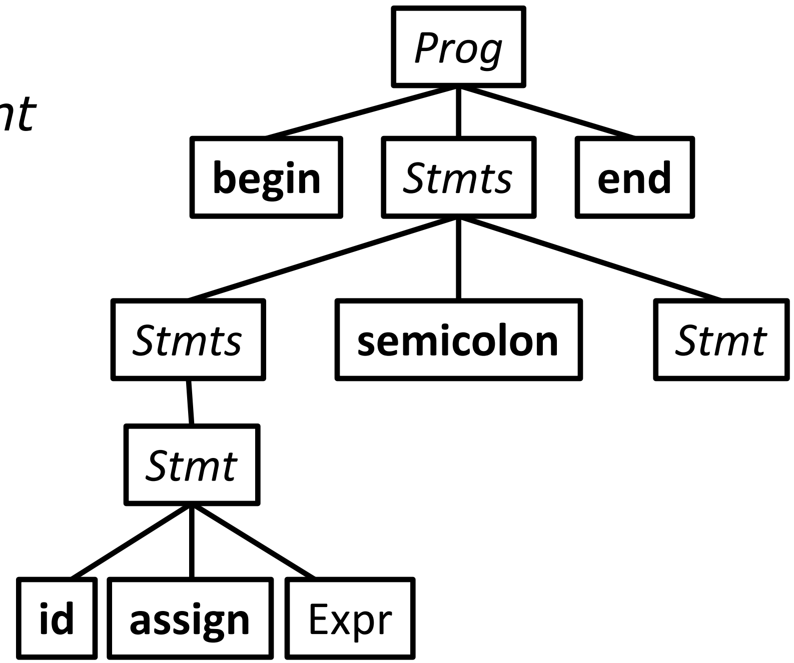
Key



Productions

1. *Prog* → **begin** *Stmts* **end**
2. *Stmts* → *Stmts* **semicolon** *Stmt*
3. | *Stmt*
4. *Stmt* → **id assign** *Expr*
5. *Expr* → **id**
6. | *Expr* **plus id**

Parse Tree



Derivation Sequence

1 *Prog* ⇒ **begin** Stmts **end**

2 ⇒ **begin** Stmts **semicolon** Stmt **end**

3 ⇒ **begin** Stmt **semicolon** Stmt **end**

4 ⇒ **begin id assign** Expr **semicolon** Stmt **end**

Key

terminal

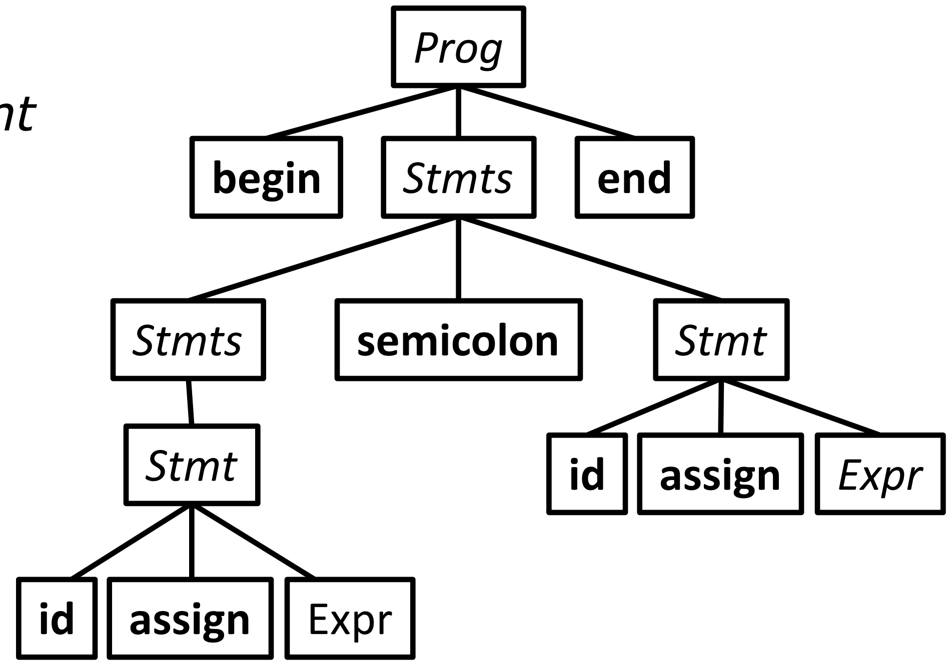
Nonterminal

Rule used

Productions

1. *Prog* → **begin** *Stmts* **end**
2. *Stmts* → *Stmts* **semicolon** *Stmt*
3. | *Stmt*
4. *Stmt* → **id assign** *Expr*
5. *Expr* → **id**
6. | *Expr* **plus id**

Parse Tree



Derivation Sequence

1 *Prog* ⇒ **begin** Stmts **end**

2 ⇒ **begin** Stmts **semicolon** Stmt **end**

3 ⇒ **begin** Stmt **semicolon** Stmt **end**

4 ⇒ **begin id assign** Expr **semicolon** Stmt **end**

4 ⇒ **begin id assign** Expr **semicolon id assign** Expr **end**

Key

terminal

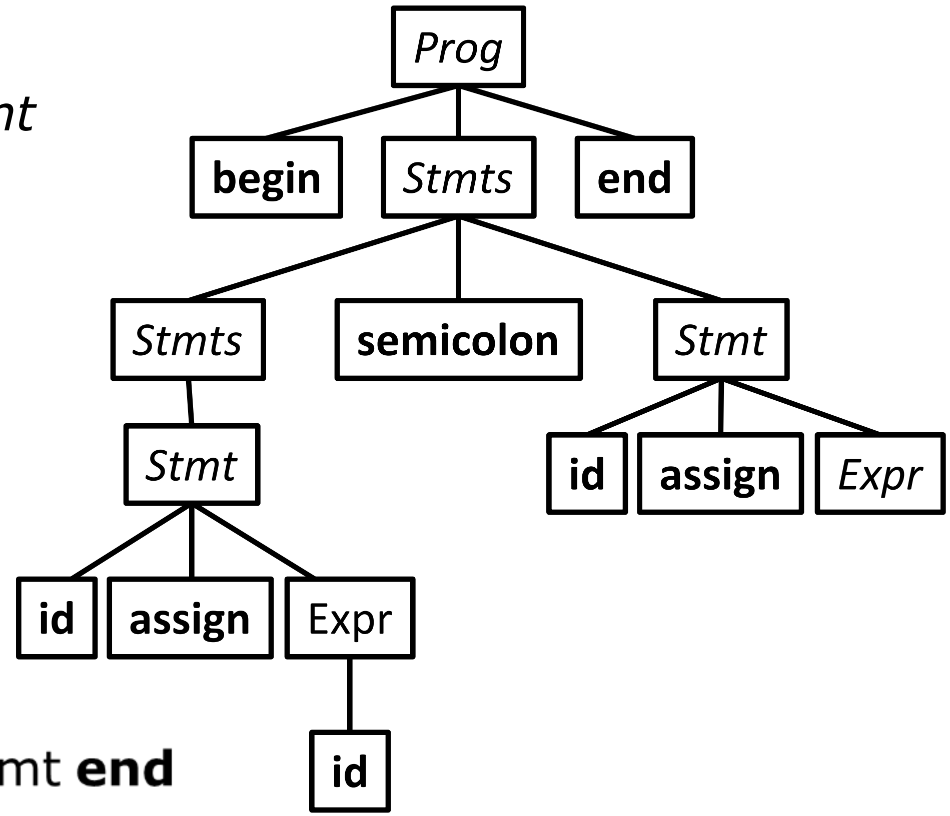
Nonterminal

Rule used

Productions

1. *Prog* → **begin** *Stmts* **end**
2. *Stmts* → *Stmts* **semicolon** *Stmt*
3. | *Stmt*
4. *Stmt* → **id assign** *Expr*
5. *Expr* → **id**
6. | *Expr* **plus id**

Parse Tree



Derivation Sequence

1 *Prog* ⇒ **begin** Stmts **end**

2 ⇒ **begin** Stmts **semicolon** Stmt **end**

3 ⇒ **begin** Stmt **semicolon** Stmt **end**

4 ⇒ **begin id assign** Expr **semicolon** Stmt **end**

4 ⇒ **begin id assign** Expr **semicolon id assign** Expr **end**

5 ⇒ **begin id assign id semicolon id assign** Expr **end**

Key

terminal

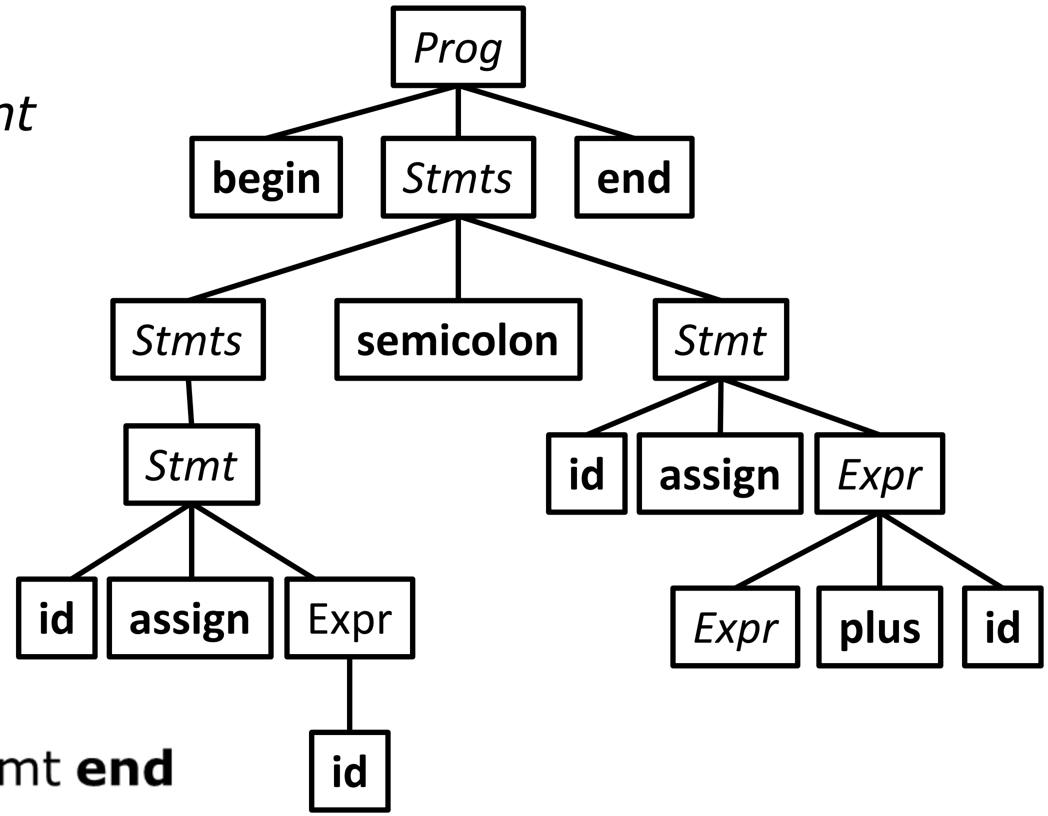
Nonterminal

Rule used

Productions

1. *Prog* → **begin** *Stmts* **end**
2. *Stmts* → *Stmts* **semicolon** *Stmt*
3. | *Stmt*
4. *Stmt* → **id assign** *Expr*
5. *Expr* → **id**
6. | *Expr* **plus id**

Parse Tree



Derivation Sequence

1 *Prog* ⇒ **begin** Stmts **end**

2 ⇒ **begin** Stmts **semicolon** Stmt **end**

3 ⇒ **begin** Stmt **semicolon** Stmt **end**

4 ⇒ **begin id assign** Expr **semicolon** Stmt **end**

4 ⇒ **begin id assign** Expr **semicolon id assign** Expr **end**

5 ⇒ **begin id assign id semicolon id assign** Expr **end**

6 ⇒ **begin id assign id semicolon id assign** Expr **plus id end**

Key

terminal

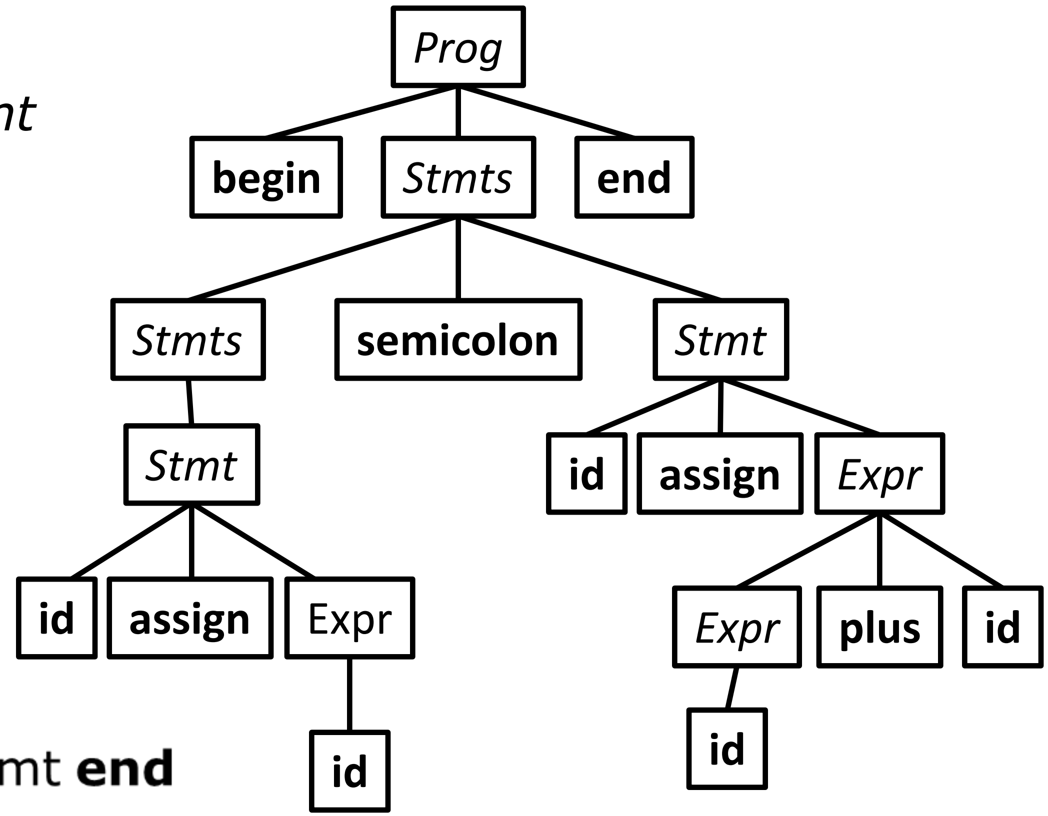
Nonterminal

Rule used

Productions

1. *Prog* → **begin** *Stmts* **end**
2. *Stmts* → *Stmts* **semicolon** *Stmt*
3. | *Stmt*
4. *Stmt* → **id assign** *Expr*
5. *Expr* → **id**
6. | *Expr* **plus id**

Parse Tree



Derivation Sequence

1 *Prog* ⇒ **begin** Stmts **end**

2 ⇒ **begin** Stmts **semicolon** Stmt **end**

3 ⇒ **begin** Stmt **semicolon** Stmt **end**

4 ⇒ **begin id assign** Expr **semicolon** Stmt **end**

4 ⇒ **begin id assign** Expr **semicolon id assign** Expr **end**

5 ⇒ **begin id assign id semicolon id assign** Expr **end**

6 ⇒ **begin id assign id semicolon id assign** Expr **plus id end**

5 ⇒ **begin id assign id semicolon id assign id plus id end**

Key

terminal

Nonterminal

Rule used

Example with Boolean Expressions

- “true” and “false” are boolean expressions
- If exp_1 and exp_2 are boolean expressions, then so are:
  - exp_1 || exp_2
  - exp_1 && exp_2
  - ! exp_1
  - (exp_1)

Corresponding CFG

bexp → TRUE

bexp → FALSE

bexp → bexp OR bexp

bexp → bexp AND bexp

bexp → NOT bexp

bexp → LPAREN bexp RPAREN

UPPERCASE represent tokens (thus terminals)
lowercase represent non-terminals

CFG for Assignments

- Here is CFG for simple assignment statements (Can only assign boolean expressions to identifiers)

stmt → ID ASSIGN *bexp* SEMICOLON

CFG for simple IF statements

Combine these CFGs and add 2 more rules for simple IF statements:

1. stmt → IF LPAREN bexp RPAREN stmt
2. stmt → IF LPAREN bexp RPAREN stmt ELSE stmt
3. stmt → ID ASSIGN bexp SEMICOLON
4. bexp → TRUE
5. bexp → FALSE
6. bexp → bexp OR bexp
7. bexp → bexp AND bexp
8. bexp → NOT bexp
9. bexp → LPAREN bexp RPAREN

Example

Write a context-free grammar for the language of very simple while loops (in which the loop body only contains one statement) by adding a new production with nonterminal *stmt* on the left-hand side.

CFG Languages

- The language defined by a context-free grammar is the set of strings (sequences of terminals) that can be **derived** from the start nonterminal.

- Think of productions as rewriting rules

```
Set cur_seq = starting non-terminal
While (non-terminal, X, exists in cur_seq):
   Select production with X on left of “→”
   Replace X with right portion of selected
    production
```

- Try it with given CFG

What Strings are in Language

1. stmt → IF LPAREN bexp RPAREN stmt
2. stmt → IF LPAREN bexp RPAREN stmt ELSE stmt
3. stmt → ID ASSIGN bexp SEMICOLON
4. bexp → TRUE
5. bexp → FALSE
6. bexp → bexp OR bexp
7. bexp → bexp AND bexp
8. bexp → NOT bexp
9. bexp → LPAREN bexp RPAREN

```
Set cur_seq = starting non-terminal
While (non-terminal, X, exists in cur_seq):
    Select production with X on left of "→"
    Replace X with right portion of selected production
```

Example

exp → exp PLUS term
exp → exp MINUS term
exp → term
term → term TIMES factor
term → term DIVIDE factor
term → factor
factor → LPAREN exp RPAREN
factor → ID

What is the language?

Leftmost and Rightmost Derivations

- A derivation is a **leftmost** derivation if it is always the leftmost nonterminal that is chosen to be replaced.
- It is a **rightmost** derivation if it is always the rightmost one.

Derivation Notation

- E =>   a
- E =>* a
- E =>+ a

Parse Trees

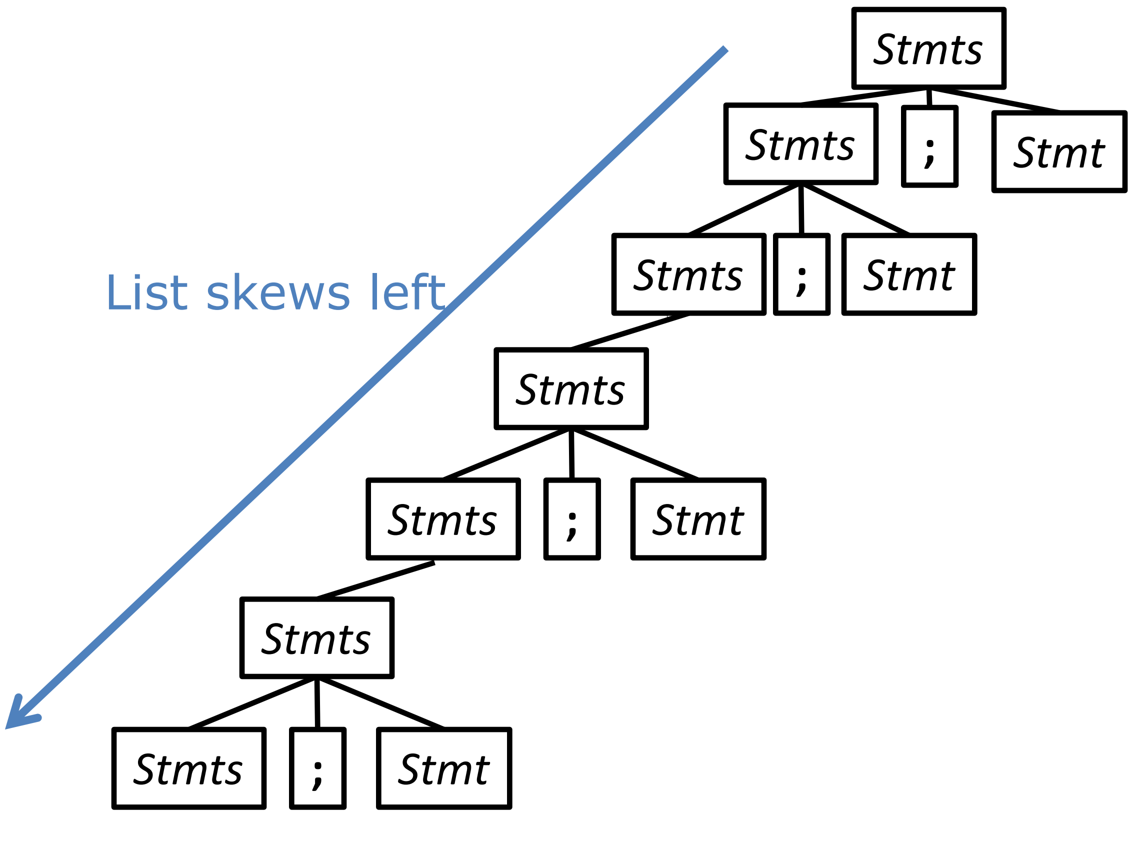
```
Start with the start nonterminal.
Repeat:
   choose a leaf nonterminal X
   choose a production X --> alpha
   the symbols in alpha become the children of X
     in the tree
until there are no more leaf non-
  terminals left.
```

The derived string is formed by reading the leaf nodes from left to right.

List Grammars

- Useful to repeat a structure arbitrarily often

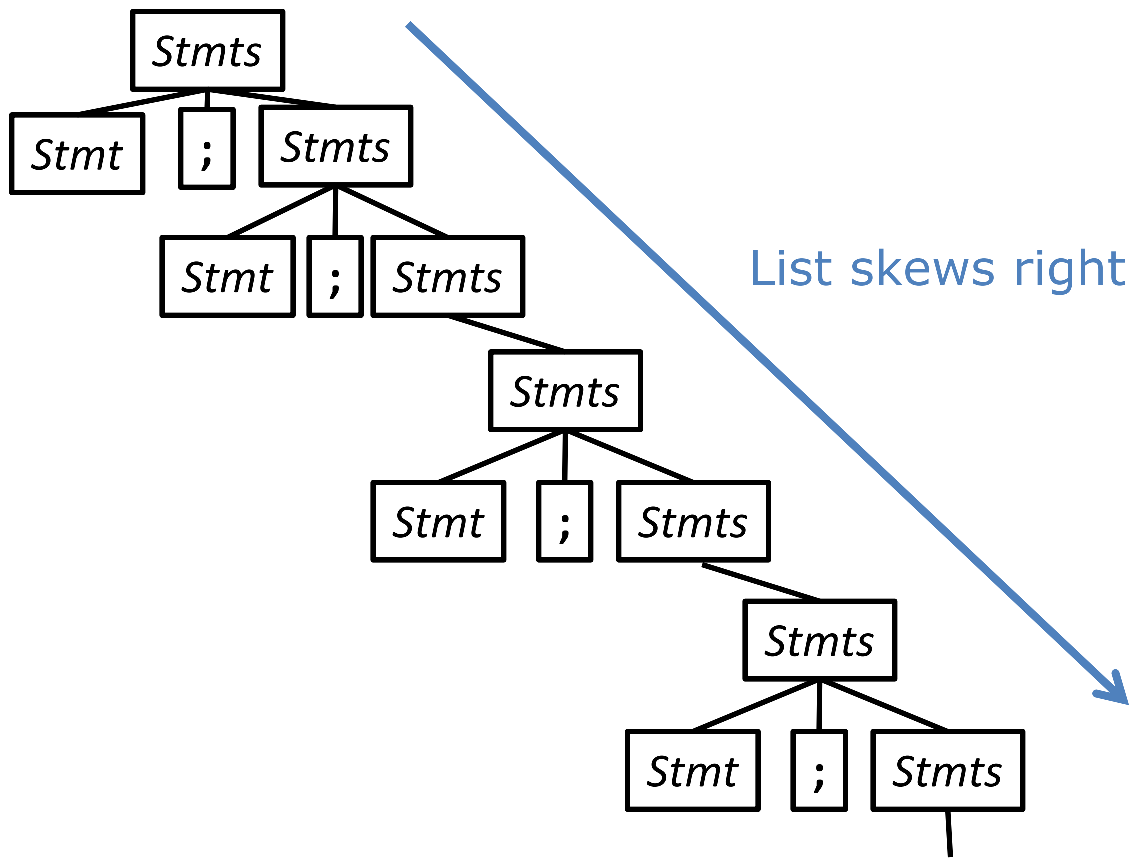
$$Stmts \rightarrow Stmts\ \textbf{semicolon}\ Stmt \mid Stmt$$



List Grammars

- Useful to repeat a structure arbitrarily often

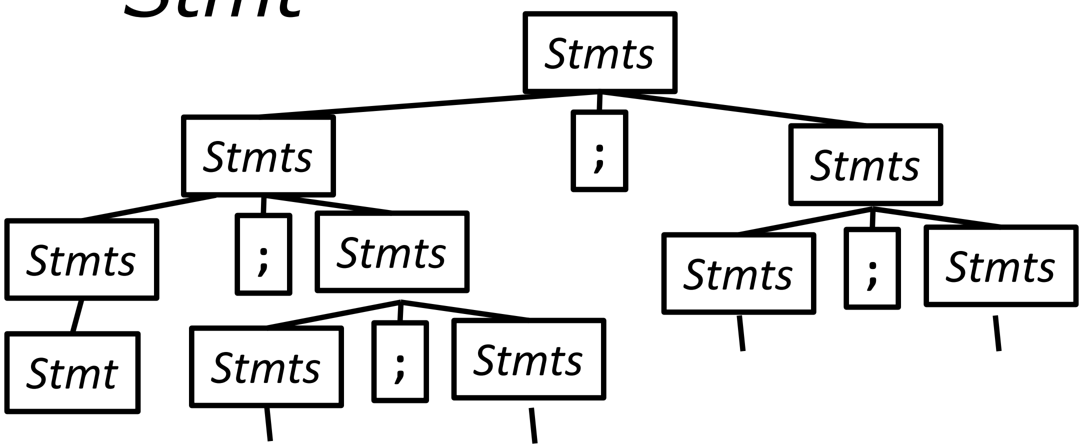
$$Stmts \rightarrow Stmt\ \textbf{semicolon}\ Stmts \mid Stmt$$



List Grammars

- What if we allowed both "skews"?

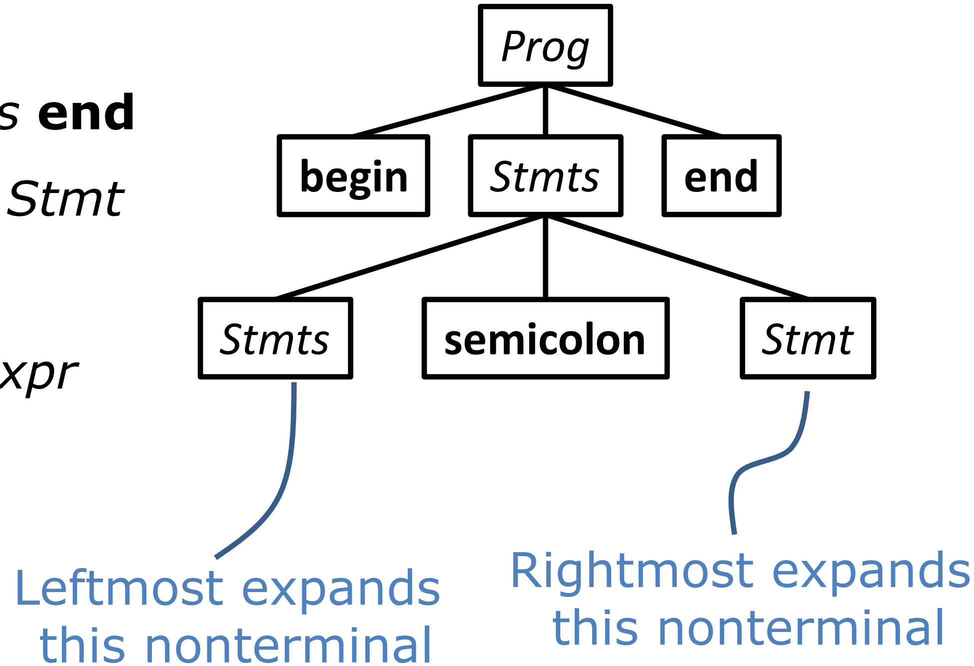
Stmts → *Stmts* **semicolon** *Stmts* | *Stmt*



Derivation Order

- Leftmost Derivation: always expand the leftmost nonterminal
- Rightmost Derivation: always expand the rightmost nonterminal

1. *Prog* → **begin** *Stmts* **end**
2. Stmts → *Stmts* **semicolon** *Stmt*
3. | *Stmt*
4. *Stmt* → **id assign** *Expr*
5. *Expr* → **id**
6. | *Expr* **plus id**



Ambiguity

Even with a fixed derivation order, it is possible to derive the same string in multiple ways

For Grammar G and string w

- *G* is ambiguous if
  - >1 leftmost derivation of w
  - >1 rightmost derivation of w
  - > 1 parse tree for w

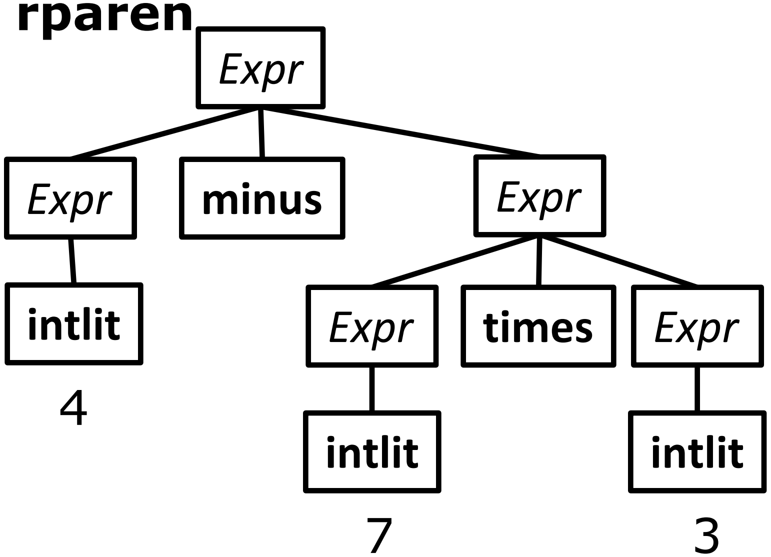
Example: Ambiguous Grammars

Expr → **intlit**
| *Expr* **minus** *Expr*
| *Expr* **times** *Expr*
| **lparen** *Expr* **rparen**

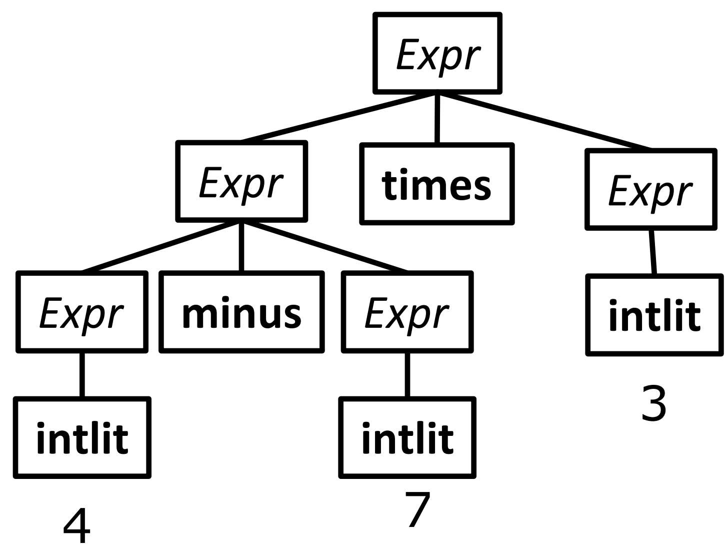
Derive the string 4 - 7 * 3

(assume tokenization)

Parse Tree 1



Parse Tree 2



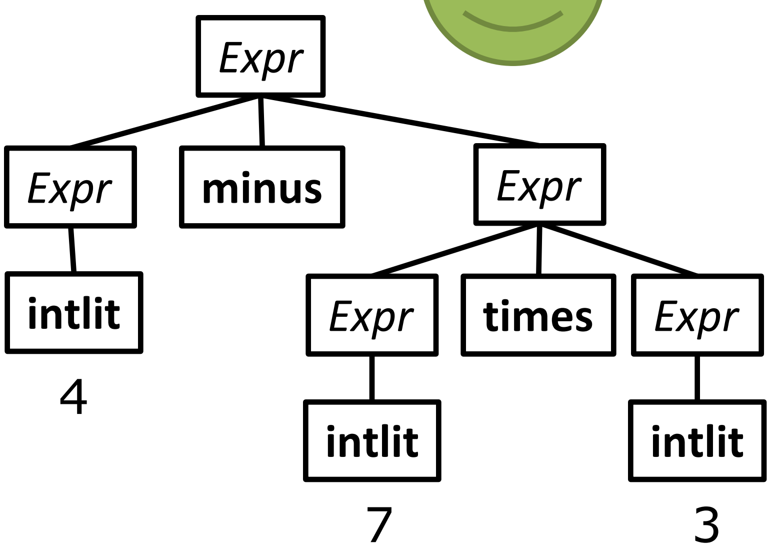
Why is Ambiguity Bad?

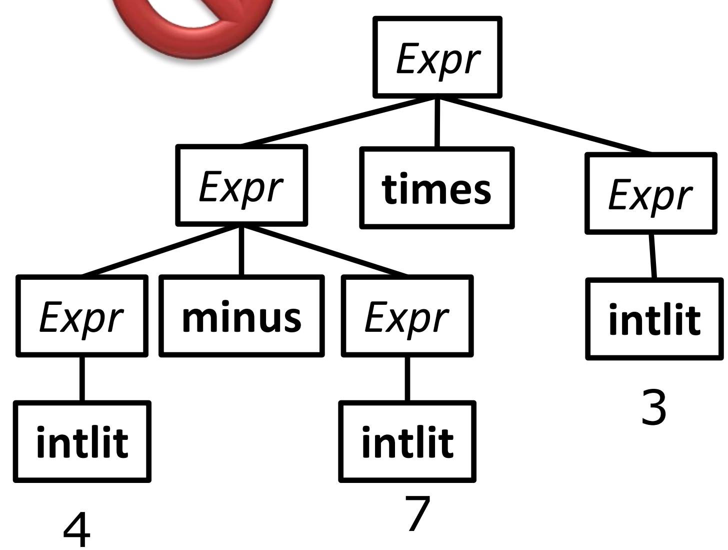
Eventually, we'll be using CFGs as the basis for our parser

- Parsing is much easier when there is no ambiguity in the grammar
- The parse tree may mismatch user understanding!

Operator precedence

4 - 7 * 3





Resolving Grammar Ambiguity: Precedence

Expr → **intlit**
| *Expr* **minus** *Expr*
| *Expr* **times** *Expr*
| **lparen** *Expr* **rparen**

Intuitive problem

- "Context-freeness"
- Nonterminals are the same for both operators

To fix precedence

- 1 nonterminal per precedence level
- Parse lowest level first

Resolving Grammar Ambiguity: Precedence

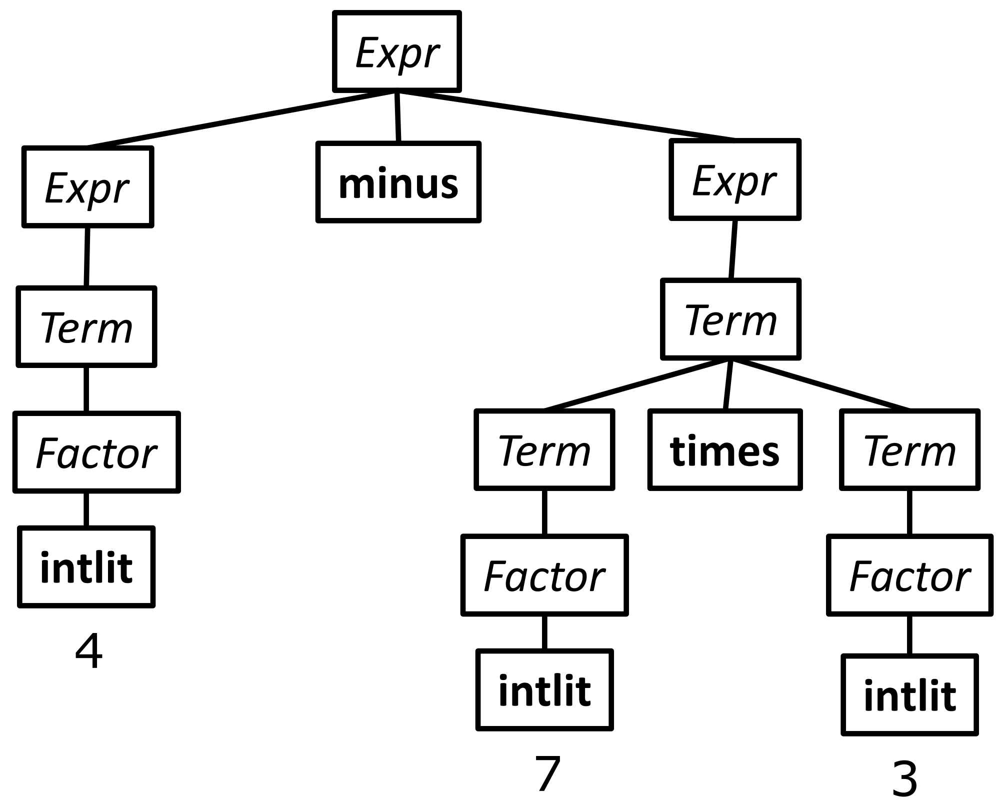
Expr $\rightarrow$ **intlit**
| *Expr* **minus** *Expr*
| *Expr* **times** *Expr*
| **lparen** *Expr* **rparen**

Expr $\rightarrow$ *Expr* **minus** *Expr*
| *Term*
Term $\rightarrow$ *Term* **times** *Term*
| *Factor*
Factor $\rightarrow$ **intlit**
| **lparen** *Expr* **rparen**

lowest precedence level first

1 nonterminal per precedence level

Derive the string 4 - 7 * 3



Resolving Grammar Ambiguity: Precedence

Fixed Grammar

Expr → **expr minus expr**

| *Term*

Term → *Term* **times** *Term*

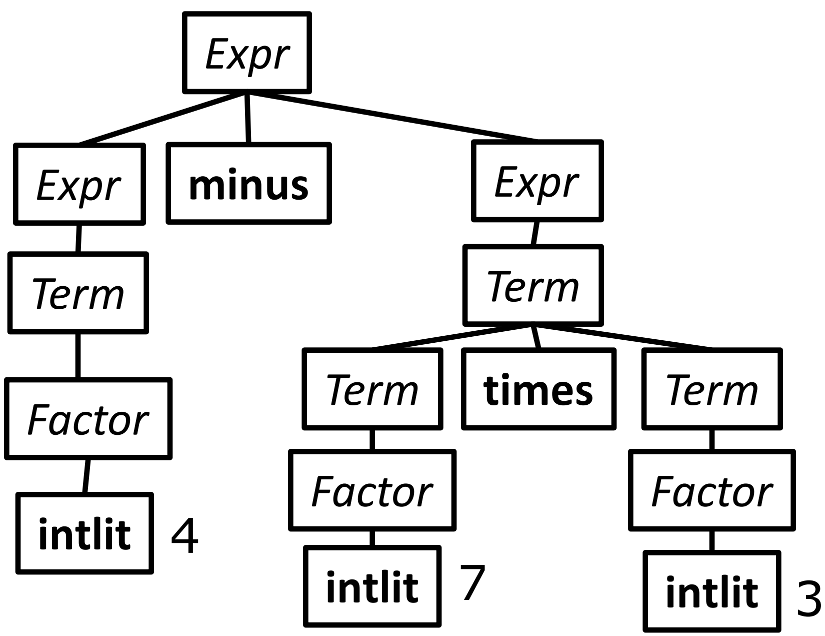
| *Factor*

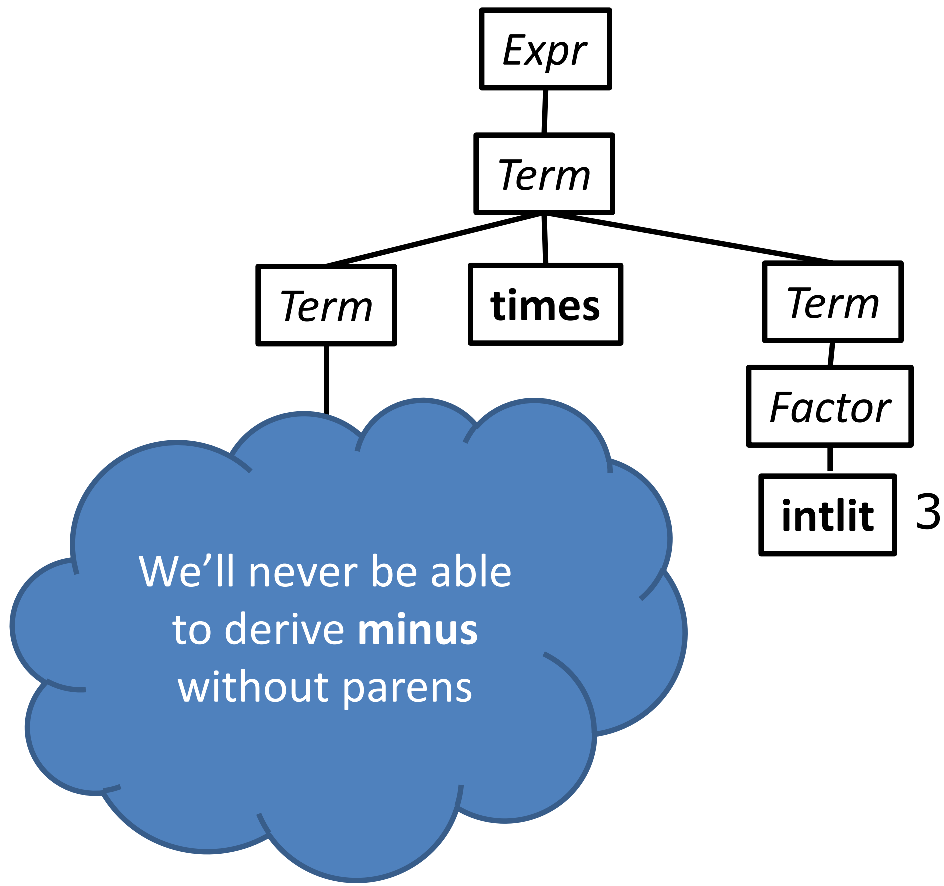
Factor → **intlit**

| **lparen** *Expr* **rparen**

Derive the string 4 - 7 * 3

Let's try to re-build the wrong parse tree





Did we fix all ambiguity?

Fixed Grammar

Expr → *Expr* **minus** *Expr*

| *Term*

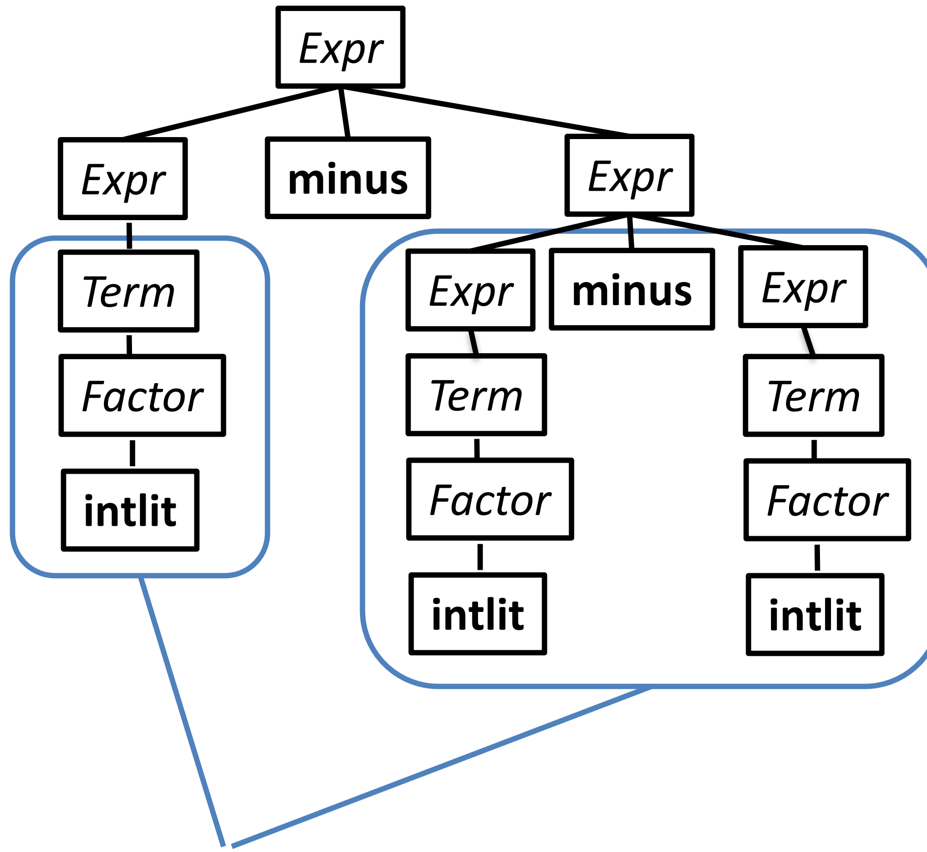
Term → *Term* **times** *Term*

| *Factor*

Factor → **intlit**

| **lparen** *Expr* **rparen**

Derive the string 4 - 7 - 3



These subtrees could have been swapped!

Where we are so far

Precedence

- We want correct behavior on 4 – 7 * 9
- A new nonterminal for each precedence level

Associativity

- We want correct behavior on 4 – 7 – 9
- Minus should be *left associative*: a – b – c = (a – b) – c
- Problem: the *recursion* in a rule like

$$\mathit{Expr} \rightarrow \mathit{Expr}\ \mathbf{minus}\ \mathit{Expr}$$

Definition: Recursion in Grammars

- **A grammar is *recursive* in (nonterminal) *X* if**

  $X \Rightarrow^{+} \alpha X \gamma$ for non-empty strings of symbols α and γ
- **A grammar is *left-recursive* in *X* if**

  $X \Rightarrow^{+} X \gamma$ for non-empty string of symbols γ
- **A grammar is *right-recursive* in *X* if**

  $X \Rightarrow^{+} \alpha X$ for non-empty string of symbols α

Resolving Grammar Ambiguity: Associativity

Recognize left-assoc operators with left-recursive productions

Recognize right-assoc operators with right-recursive productions

Example: 4 – 7 – 9

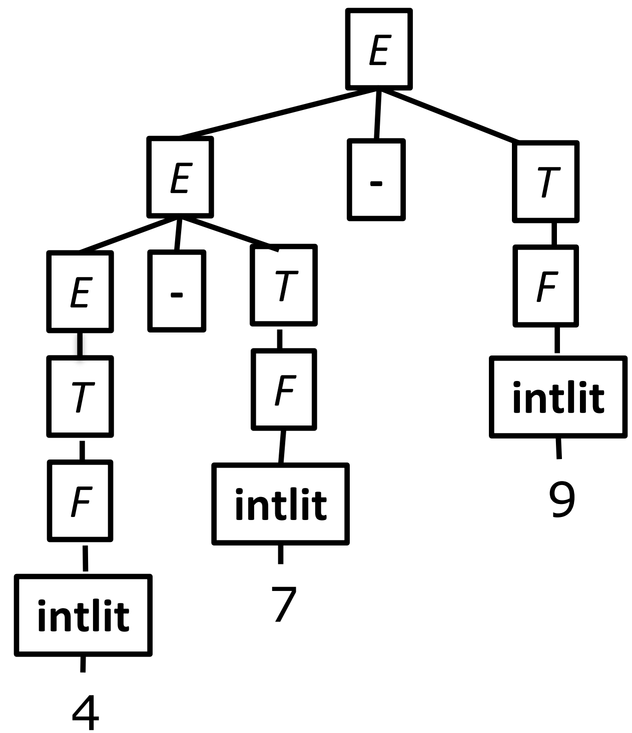
Expr → *Expr* **minus** ~~*Expr*~~ *Term*

  | *Term*

Term → *Term* **times** ~~*Term*~~ *Factor*

  | *Factor*

Factor → **intlit** | **lparen** *Expr* **rparen**



Resolving Grammar Ambiguity: Associativity

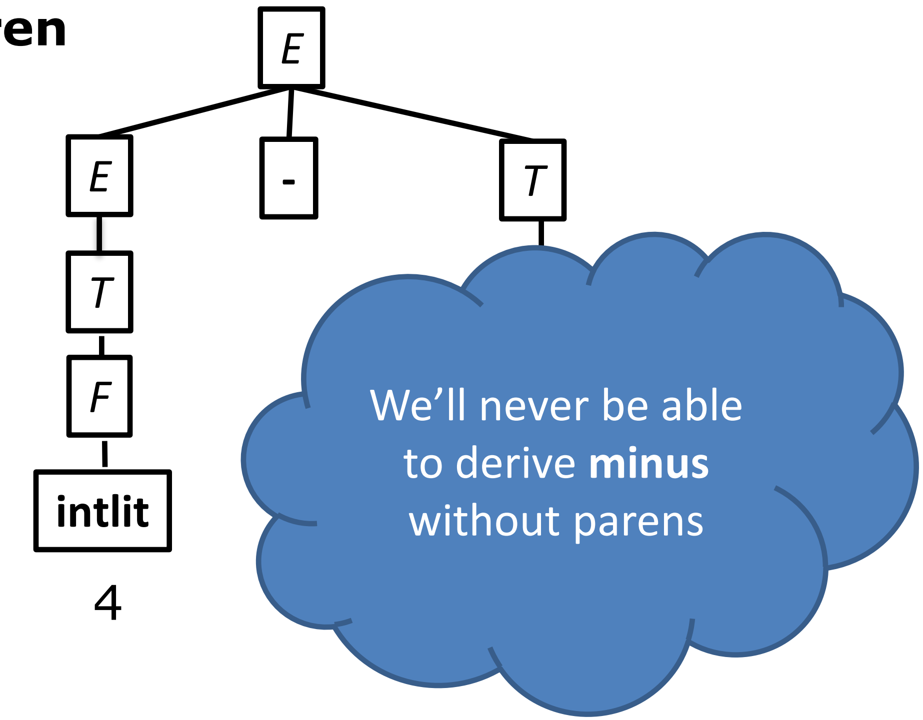
Expr $\rightarrow$ *Expr* **minus** *Term*
 | *Term*

Term $\rightarrow$ *Term* **times** *Factor*
 | *Factor*

Factor $\rightarrow$ **intlit** | **lparen** *Expr* **rparen**

Example: 4 – 7 – 9

Let's try to re-build the wrong parse tree again



Example

- Language of Boolean expressions
  - bexp → TRUE
    bexp → FALSE
    bexp → bexp OR bexp
    bexp → bexp AND bexp
    bexp → NOT bexp
    bexp → LPAREN bexp RPAREN
- Add nonterminals so that **OR** has lowest precedence, then **AND**, then **NOT**. Then change the grammar to reflect the fact that both **AND** and **OR** are left associative.
- Draw a parse tree for the expression:
  - true AND NOT true.

Another ambiguous example

Stmt $\rightarrow$

if Cond **then** Stmt |

if Cond **then** Stmt **else** Stmt | …

Consider this word in this grammar:

if a **then if** b **then** s **else** s2

How would you derive it?

CFGs for Whole Languages

- To write a grammar for a whole programming language, break down the problem into pieces. For example, think about a Java program: a program consists of one or more classes

program → classlist

classlist → class | class classlist

CFGs for Whole Languages

- A class is the word "class", optionally preceded by the word "public", followed by an identifier, followed by an open curly brace, followed by the class body, followed by a closing curly brace

class → PUBLIC CLASS ID LCURLY classbody RCURLY

| CLASS ID LCURLY classbody RCURLY

CFGs for Whole Languages

- A class body is a list of zero or more field and/or method definitions

$$\text{classbody} \rightarrow \varepsilon \mid \text{deflist}$$

$$\text{deflist} \rightarrow \text{def} \mid \text{def deflist}$$

And So On…