The University of North Carolina at Chapel Hill

COMP 144 Programming Language Concepts

Spring 2002

# Lecture 4: Syntax Specification

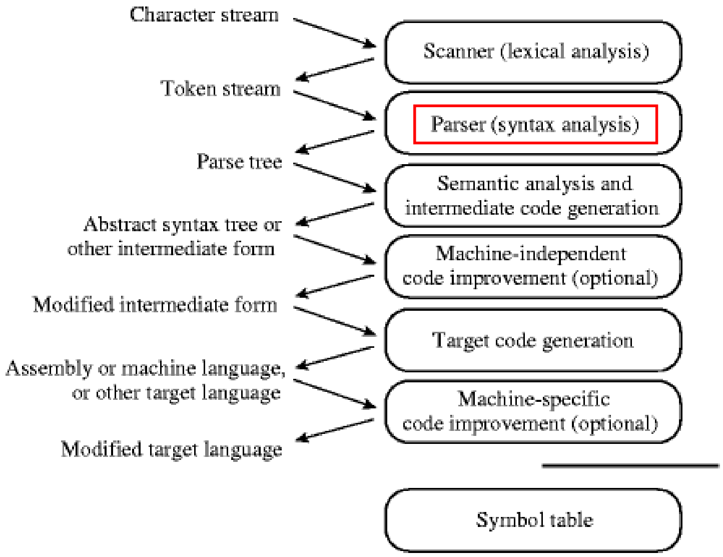
Felix Hernandez-Campos

Jan 16

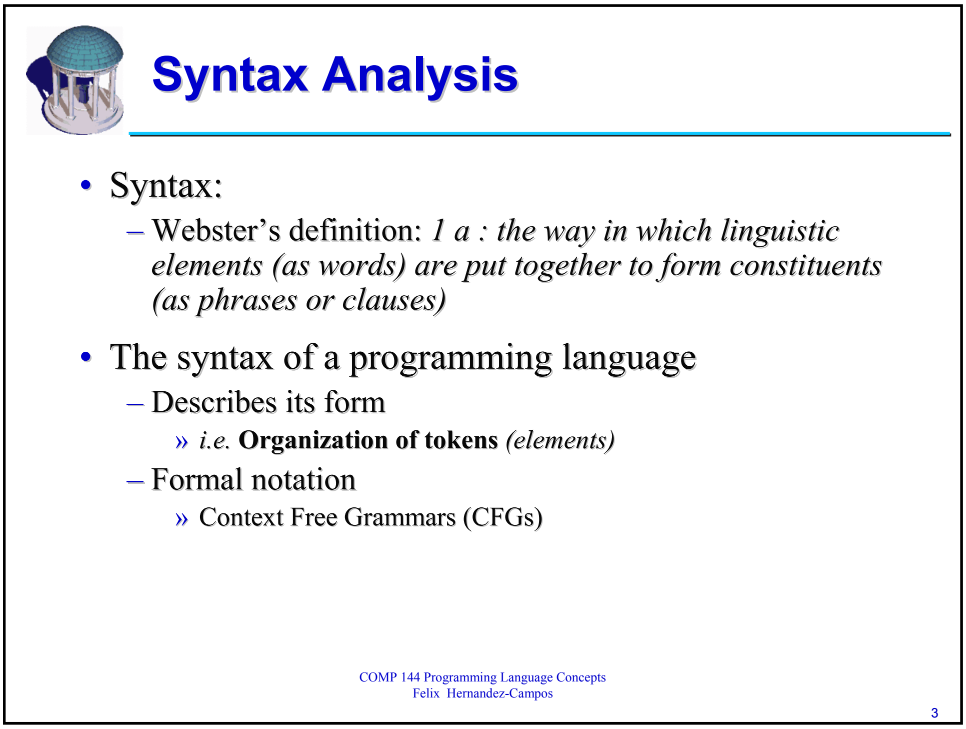
COMP 144 Programming Language Concepts
Felix Hernandez-Campos

1

## Phases of Compilation

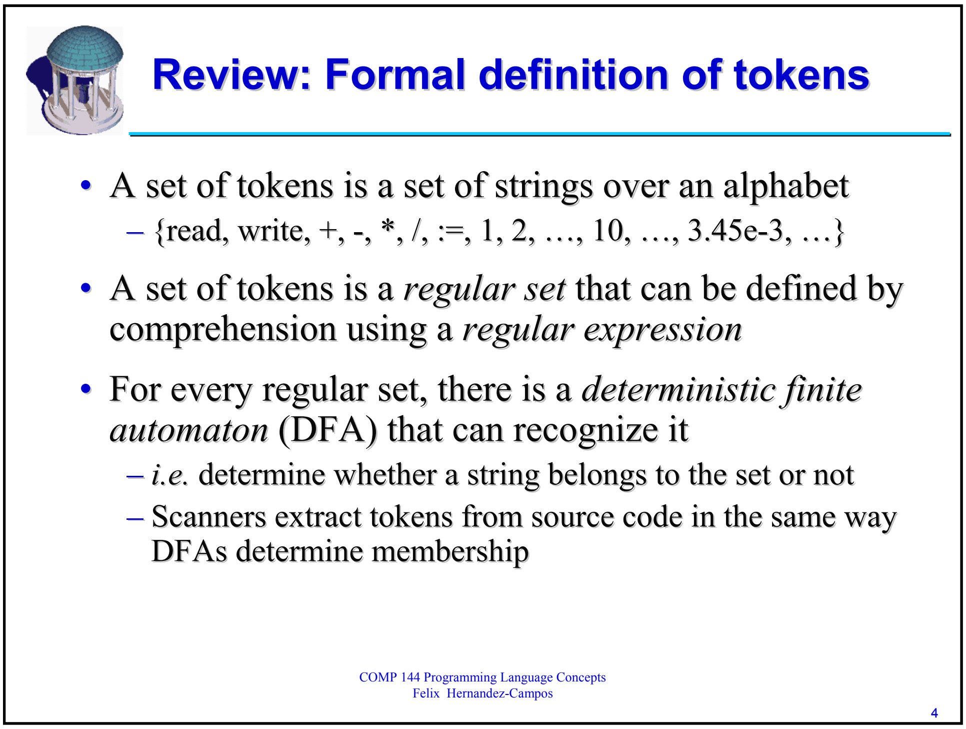

Character stream
Scanner (lexical analysis)
Token stream
Parser (syntax analysis)
Parse tree
Semantic analysis and intermediate code generation
Abstract syntax tree or other intermediate form
Machine-independent code improvement (optional)
Modified intermediate form
Target code generation
Assembly or machine language, or other target language
Machine-specific code improvement (optional)
Modified target language
Symbol table

2



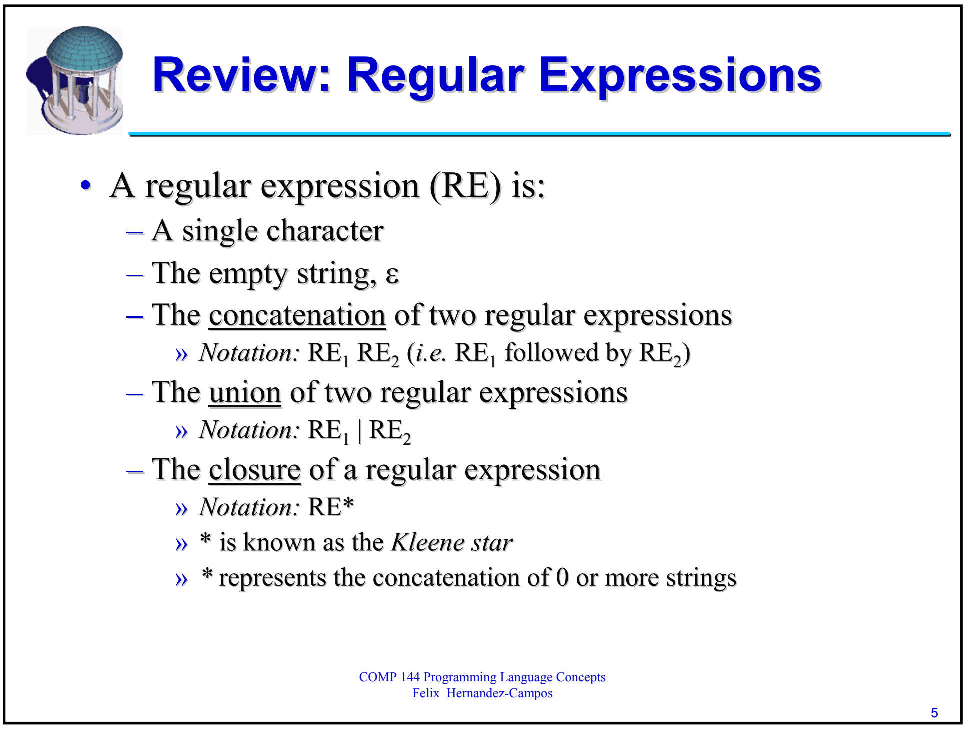
## Syntax Analysis

- Syntax:
  - Webster's definition: *1 a : the way in which linguistic elements (as words) are put together to form constituents (as phrases or clauses)*
- The syntax of a programming language
  - Describes its form
    - » *i.e.* **Organization of tokens** *(elements)*
  - Formal notation
    - » Context Free Grammars (CFGs)



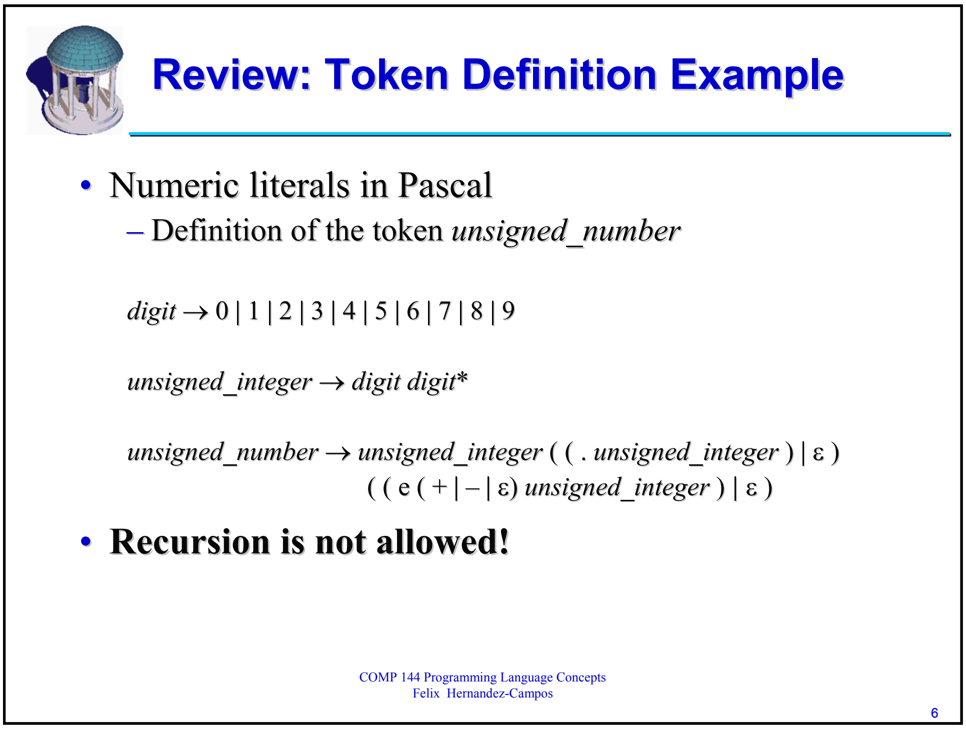
## Review: Formal definition of tokens

- A set of tokens is a set of strings over an alphabet
  - {read, write, +, -, *, /, :=, 1, 2, …, 10, …, 3.45e-3, …}
- A set of tokens is a *regular set* that can be defined by comprehension using a *regular expression*
- For every regular set, there is a *deterministic finite automaton* (DFA) that can recognize it
  - *i.e.* determine whether a string belongs to the set or not
  - Scanners extract tokens from source code in the same way DFAs determine membership



## Review: Regular Expressions

- A regular expression (RE) is:
  - A single character
  - The empty string, ε
  - The concatenation of two regular expressions
    - *Notation:* $RE_1\ RE_2$ (*i.e.* $RE_1$ followed by $RE_2$)
  - The union of two regular expressions
    - *Notation:* $RE_1 \mid RE_2$
  - The closure of a regular expression
    - *Notation:* RE*
    - * is known as the *Kleene star*
    - * represents the concatenation of 0 or more strings



## Review: Token Definition Example

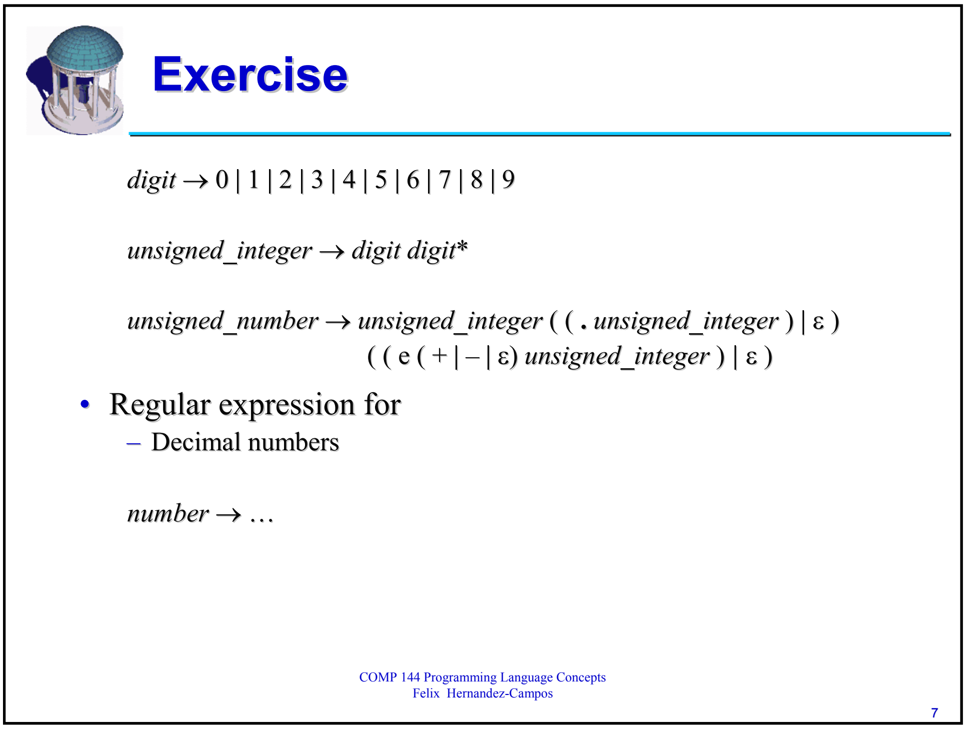
- Numeric literals in Pascal
  - Definition of the token *unsigned_number*

$$\mathit{digit} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

$$\mathit{unsigned\_integer} \rightarrow \mathit{digit}\ \mathit{digit}^*$$

$$\mathit{unsigned\_number} \rightarrow \mathit{unsigned\_integer}\ (\ (\ .\ \mathit{unsigned\_integer}\ ) \mid \varepsilon\ )\ (\ (\ \mathrm{e}\ (\ + \mid - \mid \varepsilon)\ \mathit{unsigned\_integer}\ ) \mid \varepsilon\ )$$

- **Recursion is not allowed!**



## Exercise

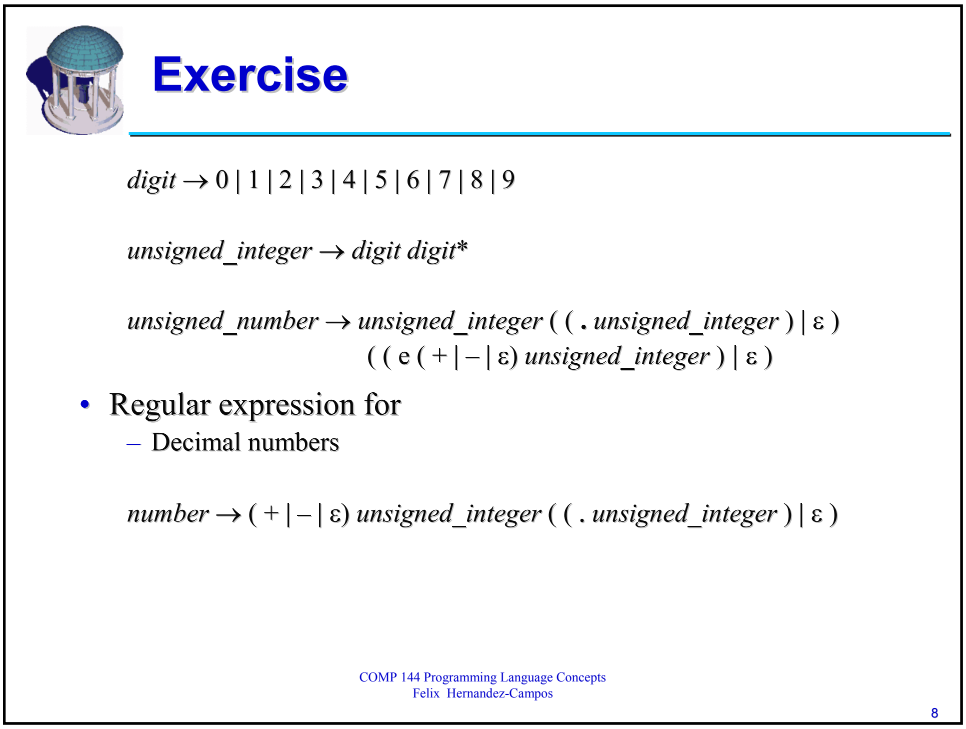
$$\mathit{digit} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

$$\mathit{unsigned\_integer} \rightarrow \mathit{digit}\ \mathit{digit}^*$$

$$\mathit{unsigned\_number} \rightarrow \mathit{unsigned\_integer}\ (\ (\ .\ \mathit{unsigned\_integer}\ ) \mid \varepsilon\ )\ (\ (\ \mathrm{e}\ (\ + \mid - \mid \varepsilon)\ \mathit{unsigned\_integer}\ ) \mid \varepsilon\ )$$

- Regular expression for
  - Decimal numbers

$\mathit{number} \rightarrow \ldots$



## Exercise

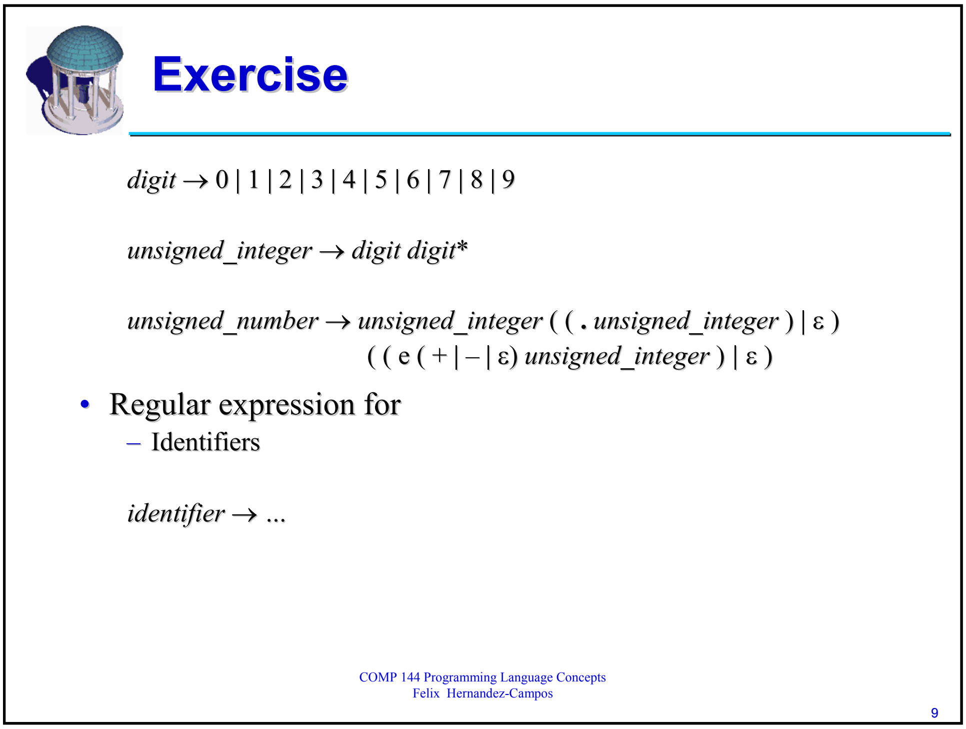
$$\mathit{digit} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

$$\mathit{unsigned\_integer} \rightarrow \mathit{digit}\ \mathit{digit}^*$$

$$\mathit{unsigned\_number} \rightarrow \mathit{unsigned\_integer}\ (\ (\ .\ \mathit{unsigned\_integer}\ ) \mid \varepsilon\ )\ (\ (\ \mathrm{e}\ (\ + \mid - \mid \varepsilon)\ \mathit{unsigned\_integer}\ ) \mid \varepsilon\ )$$

- Regular expression for
  - Decimal numbers

$$\mathit{number} \rightarrow (\ + \mid - \mid \varepsilon)\ \mathit{unsigned\_integer}\ (\ (\ .\ \mathit{unsigned\_integer}\ ) \mid \varepsilon\ )$$



## Exercise

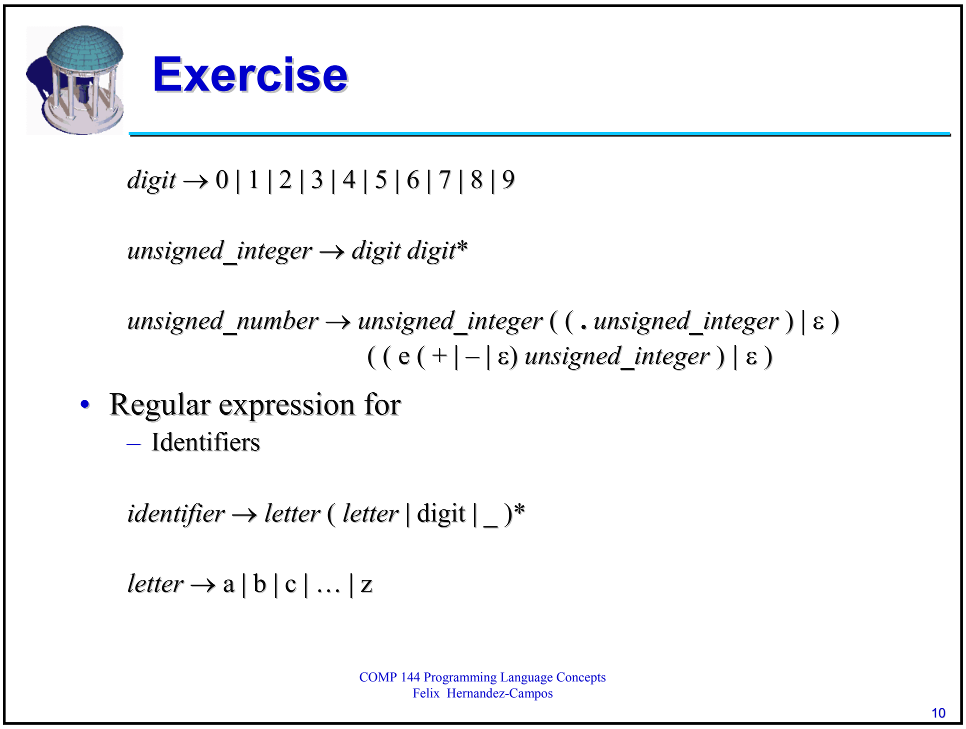
$\mathit{digit} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

$\mathit{unsigned\_integer} \rightarrow \mathit{digit}\ \mathit{digit}^{*}$

$$\mathit{unsigned\_number} \rightarrow \mathit{unsigned\_integer}\ (\ (\ .\ \mathit{unsigned\_integer}\ ) \mid \varepsilon\ )$$
$$(\ (\ \mathrm{e}\ (\ + \mid - \mid \varepsilon)\ \mathit{unsigned\_integer}\ ) \mid \varepsilon\ )$$

- Regular expression for
  - Identifiers

$\mathit{identifier} \rightarrow \ldots$



## Exercise

$\mathit{digit} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

$\mathit{unsigned\_integer} \rightarrow \mathit{digit}\ \mathit{digit}^{*}$

$$\mathit{unsigned\_number} \rightarrow \mathit{unsigned\_integer}\ (\ (\ .\ \mathit{unsigned\_integer}\ ) \mid \varepsilon\ )$$
$$(\ (\ \mathrm{e}\ (\ + \mid - \mid \varepsilon)\ \mathit{unsigned\_integer}\ ) \mid \varepsilon\ )$$

- Regular expression for
  - Identifiers

$\mathit{identifier} \rightarrow \mathit{letter}\ (\ \mathit{letter} \mid \mathrm{digit} \mid \_\ )^{*}$

$\mathit{letter} \rightarrow \mathrm{a} \mid \mathrm{b} \mid \mathrm{c} \mid \ldots \mid \mathrm{z}$

## Context Free Grammars

- CFGs
  - Add recursion to regular expressions
    - » Nested constructions
  - Notation

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

| **(** *expression* **)**

| *expression operator expression*

*operator* → **+** | **-** | ***** | **/**

- » **`Terminal symbols`**
- » *Non-terminal symbols*
- » Production rule (i.e. substitution rule)

terminal symbol → terminal and non-terminal symbols

## Backus-Naur Form

- Backus-Naur Form (BNF)
  - Equivalent to CFGs in power
  - CFG

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

| ( *expression* )

| *expression operator expression*

*operator* → + | - | * | /

- BNF

⟨expression⟩ → ⟨identifier⟩ | ⟨number⟩ | - ⟨expression⟩

| ( ⟨expression⟩ )

| ⟨expression⟩ ⟨operator⟩ ⟨expression⟩

⟨operator⟩ → + | - | * | /

## Extended Backus-Naur Form

- Extended Backus-Naur Form (EBNF)
  - Adds some convenient symbols
    - » Union |
    - » Kleene star *
    - » Meta-level parentheses ( )
  - It has the same expressive power

## Extended Backus-Naur Form

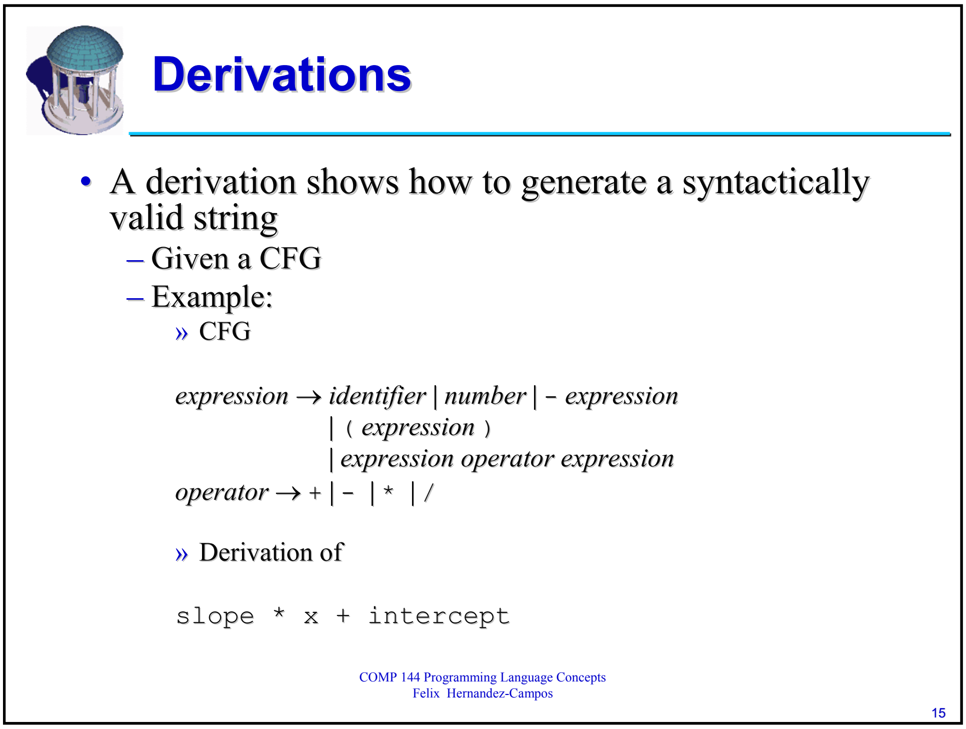
- Extended Backus-Naur Form (EBNF)
  - It has the same expressive power

**BNF**

⟨digit⟩ → 0
⟨digit⟩ → 1
…
⟨digit⟩ → 9
⟨unsigned_integer⟩ → ⟨digit⟩
⟨unsigned_integer⟩ → ⟨digit⟩ ⟨unsigned_integer⟩

**EBNF**

⟨digit⟩ → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
⟨unsigned_integer⟩ → ⟨digit⟩ ⟨digit⟩*



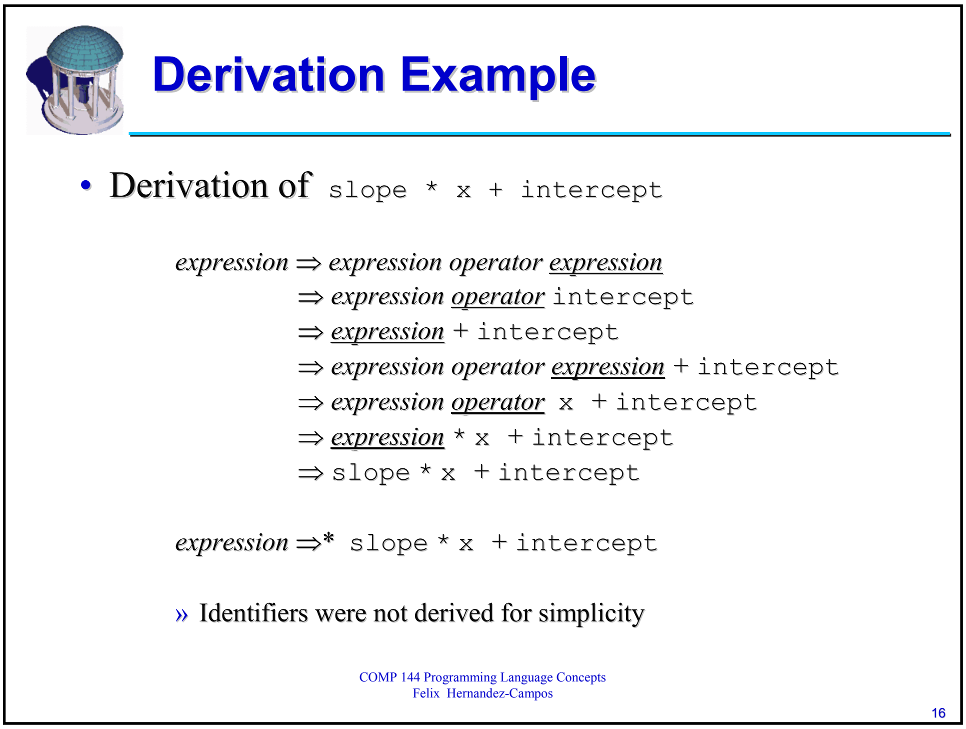
## Derivations

- A derivation shows how to generate a syntactically valid string
  - Given a CFG
  - Example:
    - » CFG

$$\begin{aligned} \mathit{expression} &\rightarrow \mathit{identifier} \mid \mathit{number} \mid -\ \mathit{expression} \\ &\mid (\ \mathit{expression}\ ) \\ &\mid \mathit{expression}\ \mathit{operator}\ \mathit{expression} \\ \mathit{operator} &\rightarrow + \mid - \mid * \mid / \end{aligned}$$

    - » Derivation of

```
slope * x + intercept
```



## Derivation Example

- Derivation of `slope * x + intercept`

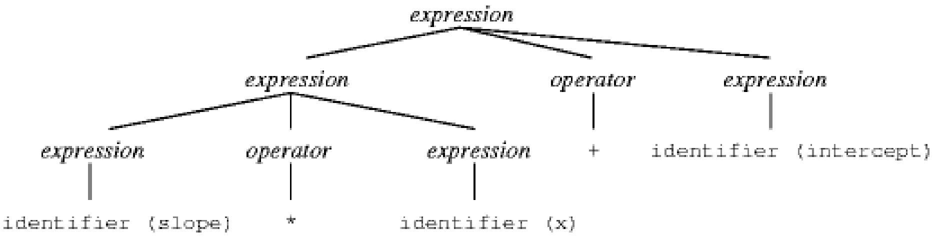
$$\begin{aligned} \mathit{expression} &\Rightarrow \mathit{expression}\ \mathit{operator}\ \underline{\mathit{expression}} \\ &\Rightarrow \mathit{expression}\ \underline{\mathit{operator}}\ \texttt{intercept} \\ &\Rightarrow \underline{\mathit{expression}} + \texttt{intercept} \\ &\Rightarrow \mathit{expression}\ \mathit{operator}\ \underline{\mathit{expression}} + \texttt{intercept} \\ &\Rightarrow \mathit{expression}\ \underline{\mathit{operator}}\ \texttt{x} + \texttt{intercept} \\ &\Rightarrow \underline{\mathit{expression}} * \texttt{x} + \texttt{intercept} \\ &\Rightarrow \texttt{slope} * \texttt{x} + \texttt{intercept} \end{aligned}$$

$$\mathit{expression} \Rightarrow^* \texttt{slope} * \texttt{x} + \texttt{intercept}$$

- » Identifiers were not derived for simplicity

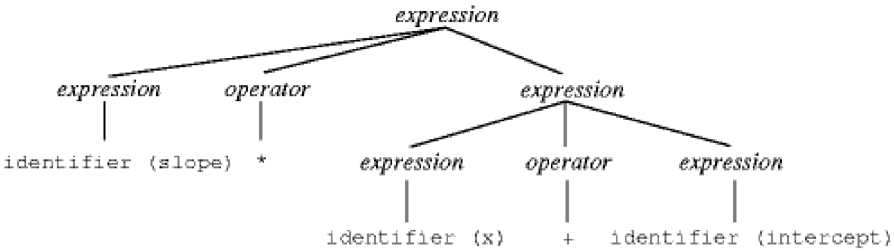
# Parse Trees

- A parse is graphical representation of a derivation
- Example

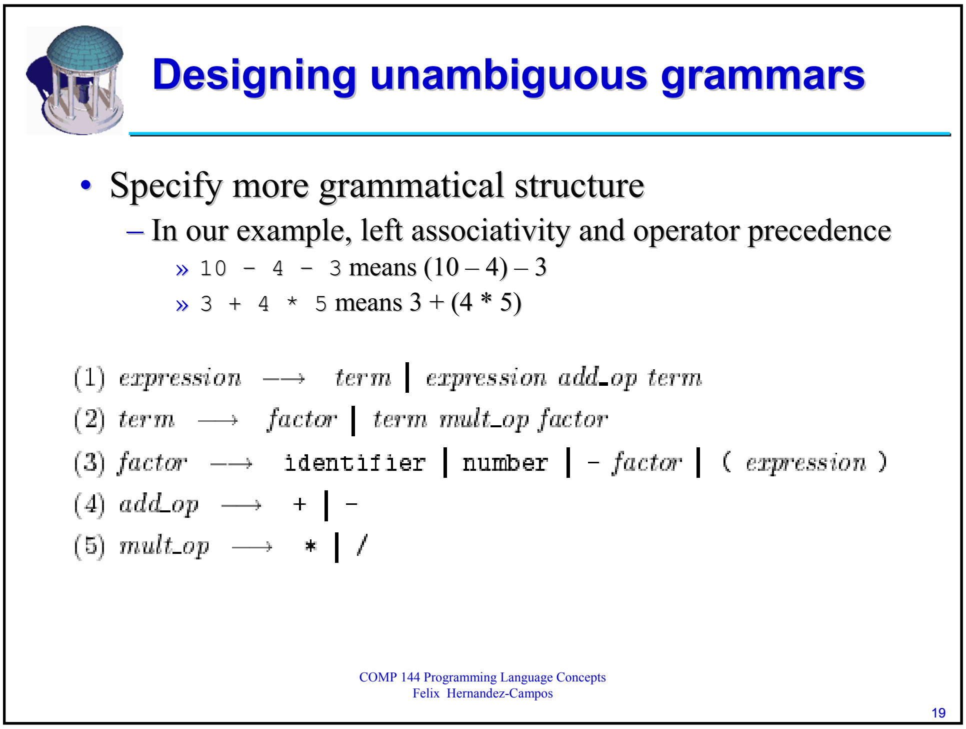


# Ambiguous Grammars

- Alternative parse tree
  - same expression
  - same grammar



- This grammar is ambiguous



## Designing unambiguous grammars

- Specify more grammatical structure
  - In our example, left associativity and operator precedence
    - » `10 - 4 - 3` means (10 – 4) – 3
    - » `3 + 4 * 5` means 3 + (4 * 5)

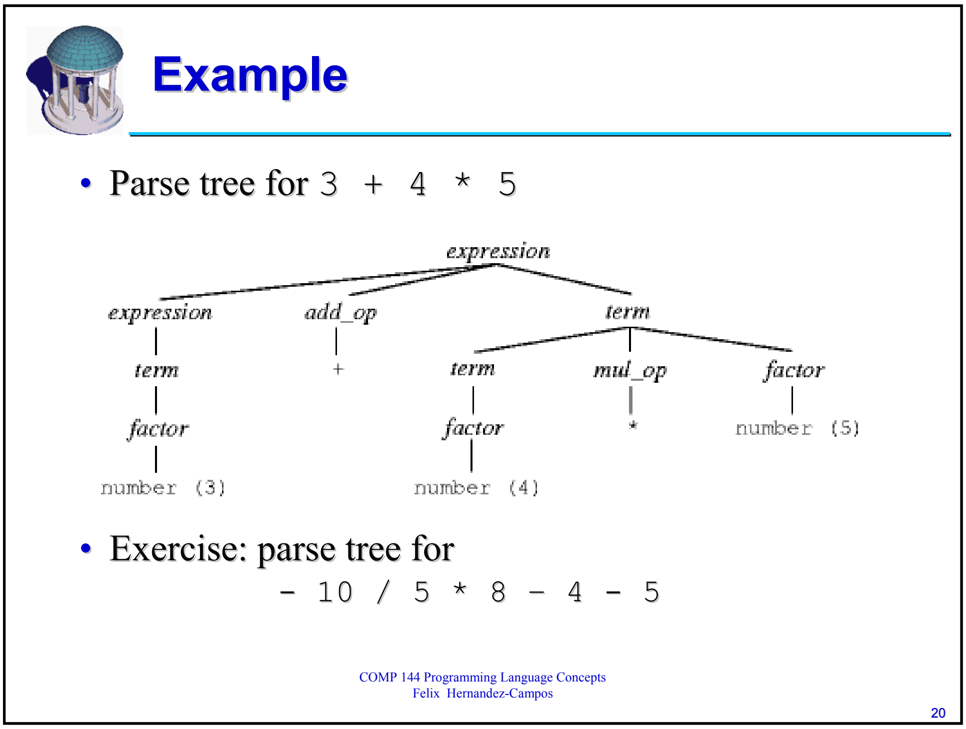
(1) *expression* ⟶ *term* | *expression add_op term*

(2) *term* ⟶ *factor* | *term mult_op factor*

(3) *factor* ⟶ `identifier` | `number` | `-` *factor* | `(` *expression* `)`

(4) *add_op* ⟶ `+` | `-`

(5) *mult_op* ⟶ `*` | `/`

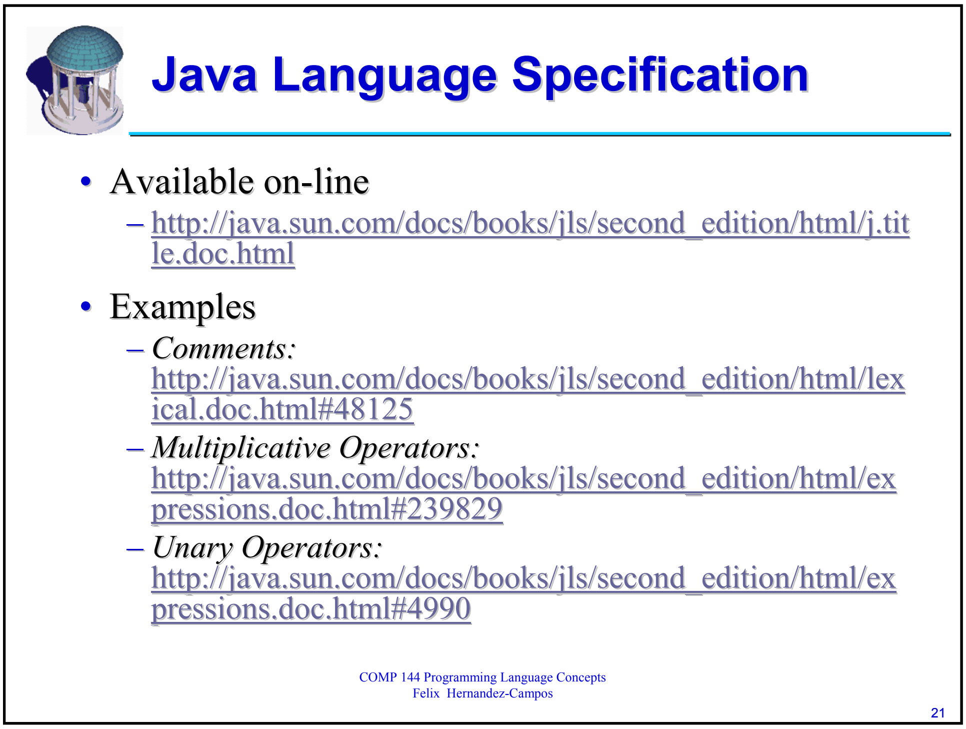


## Example

- Parse tree for `3 + 4 * 5`

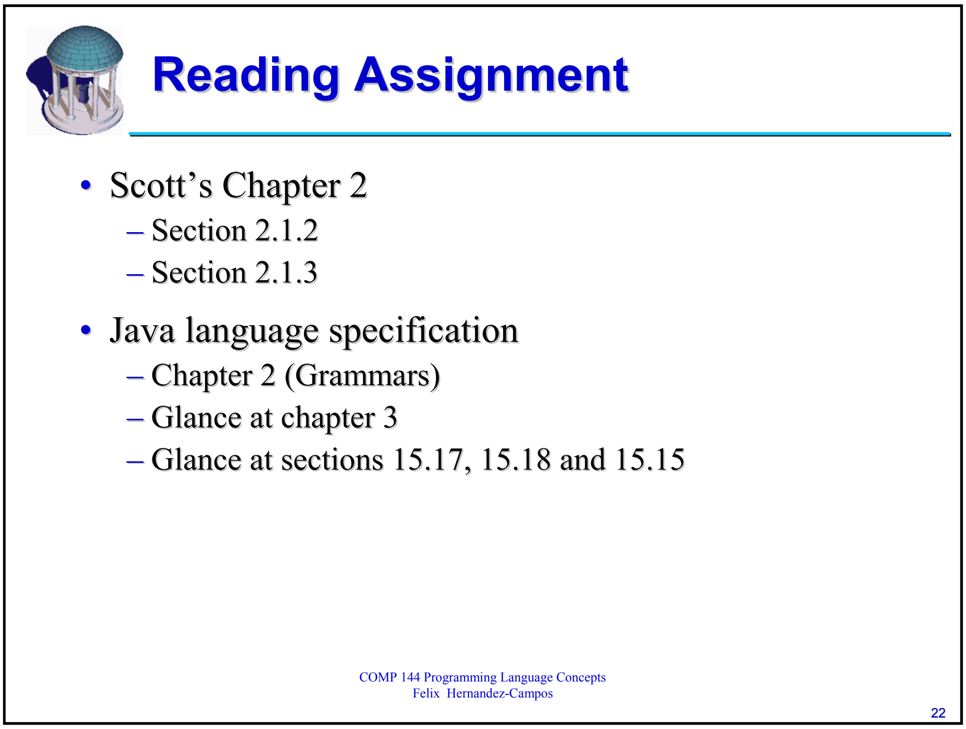
- Exercise: parse tree for

`- 10 / 5 * 8 - 4 - 5`



## Java Language Specification

- Available on-line
  - http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html
- Examples
  - *Comments:* http://java.sun.com/docs/books/jls/second_edition/html/lexical.doc.html#48125
  - *Multiplicative Operators:* http://java.sun.com/docs/books/jls/second_edition/html/expressions.doc.html#239829
  - *Unary Operators:* http://java.sun.com/docs/books/jls/second_edition/html/expressions.doc.html#4990



## Reading Assignment

- Scott's Chapter 2
  - Section 2.1.2
  - Section 2.1.3
- Java language specification
  - Chapter 2 (Grammars)
  - Glance at chapter 3
  - Glance at sections 15.17, 15.18 and 15.15