The University of North Carolina at Chapel Hill

COMP 144 Programming Language Concepts

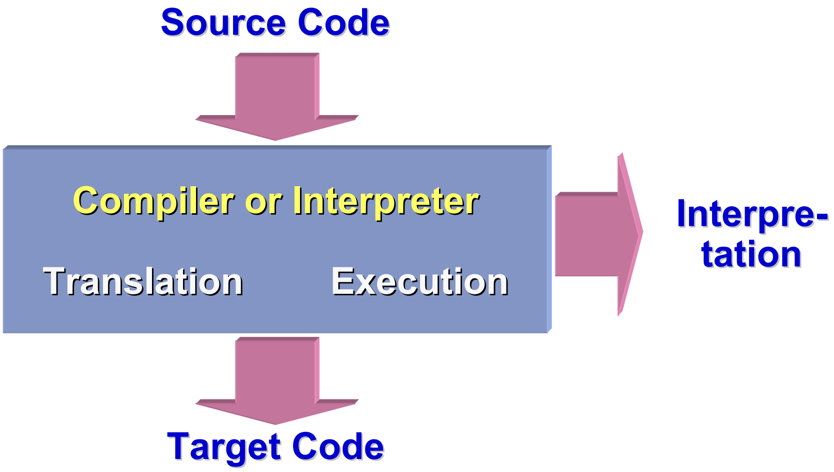
Spring 2002

Lecture 5:
Syntax Analysis

Felix Hernandez-Campos

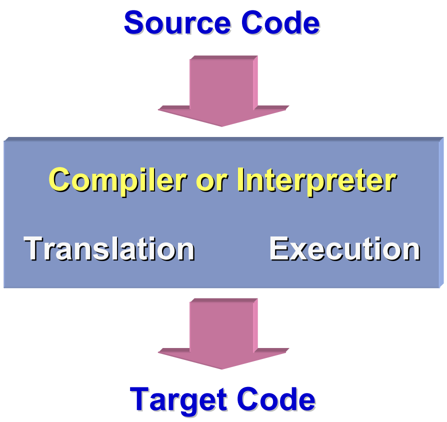
Jan 18

Review: Compilation/Interpretation



Review: Syntax Analysis

- Specifying the **form** of a programming language
  - Tokens
    - Regular Expression
  - Syntax
    - Context-Free Grammars



Source Code

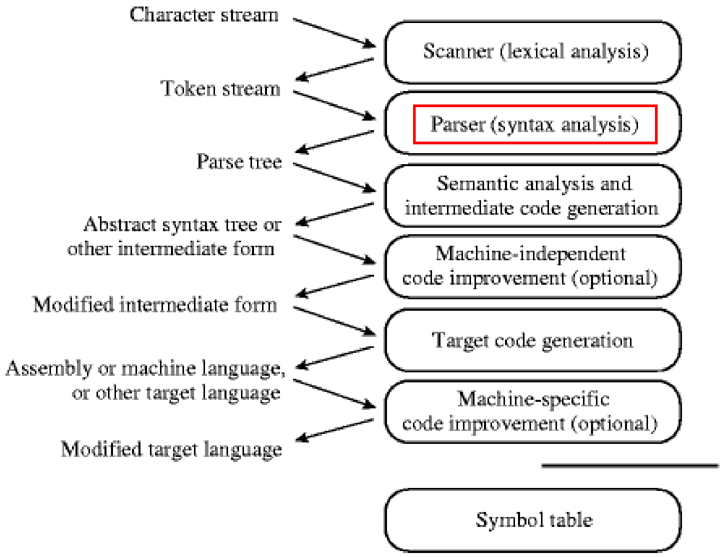
Compiler or Interpreter

Translation

Execution

Target Code

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

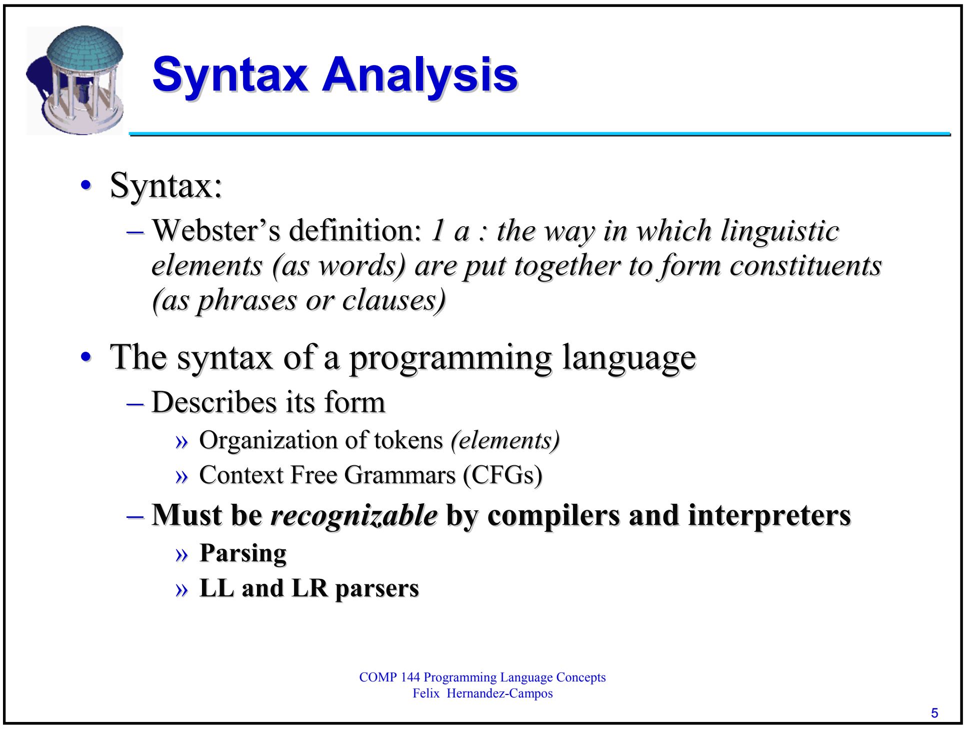
Target code generation

Assembly or machine language, or other target language

Machine-specific code improvement (optional)

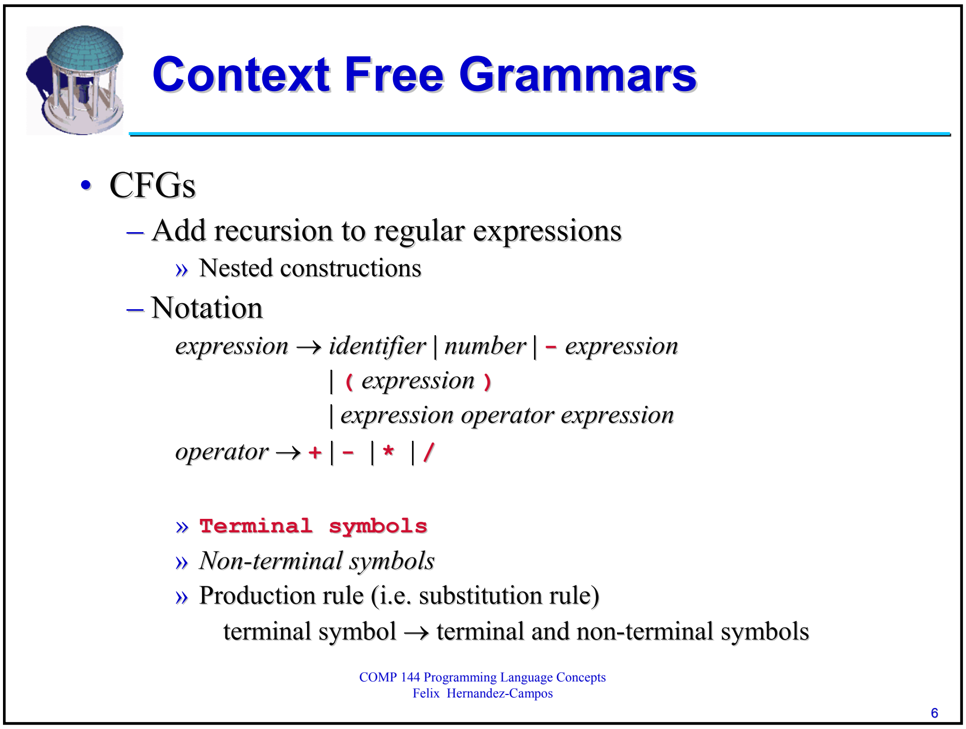
Modified target language

Symbol table



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
    - » Organization of tokens *(elements)*
    - » Context Free Grammars (CFGs)
  - **Must be *recognizable* by compilers and interpreters**
    - » **Parsing**
    - » **LL and LR parsers**



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)

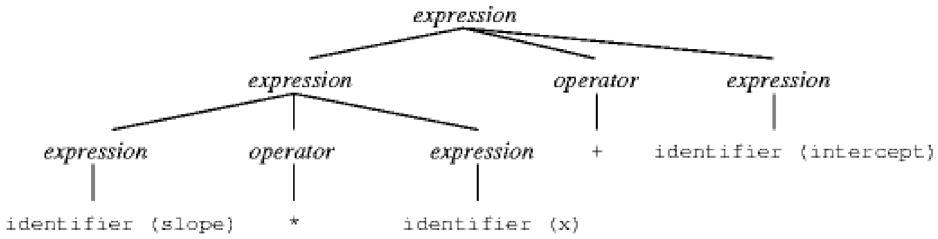
terminal symbol → terminal and non-terminal symbols

Parsers

- Scanners
  - Task: recognize language tokens
  - Implementation: Deterministic Finite Automaton
    - » Transition based on the next character
- Parsers
  - Task: recognize language syntax (organization of tokens)
  - Implementation:
    - » Top-down parsing
    - » Bottom-up parsing

Parse Trees

- A parse is graphical representation of a derivation
- Example



Parsing example

- Example: comma-separated list of identifier
  - CFG

    id_list → **id** *id_list_tail*
    id_list_tail → **,** *id_list_tail*
    id_list_tail → **;**

  - Parsing

    A, B, C;

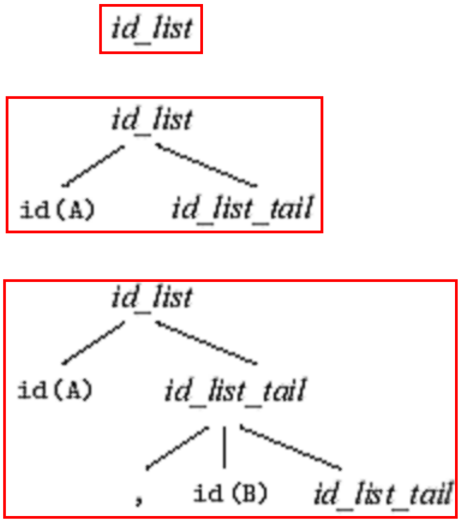
Top-down derivation of A, B, C;

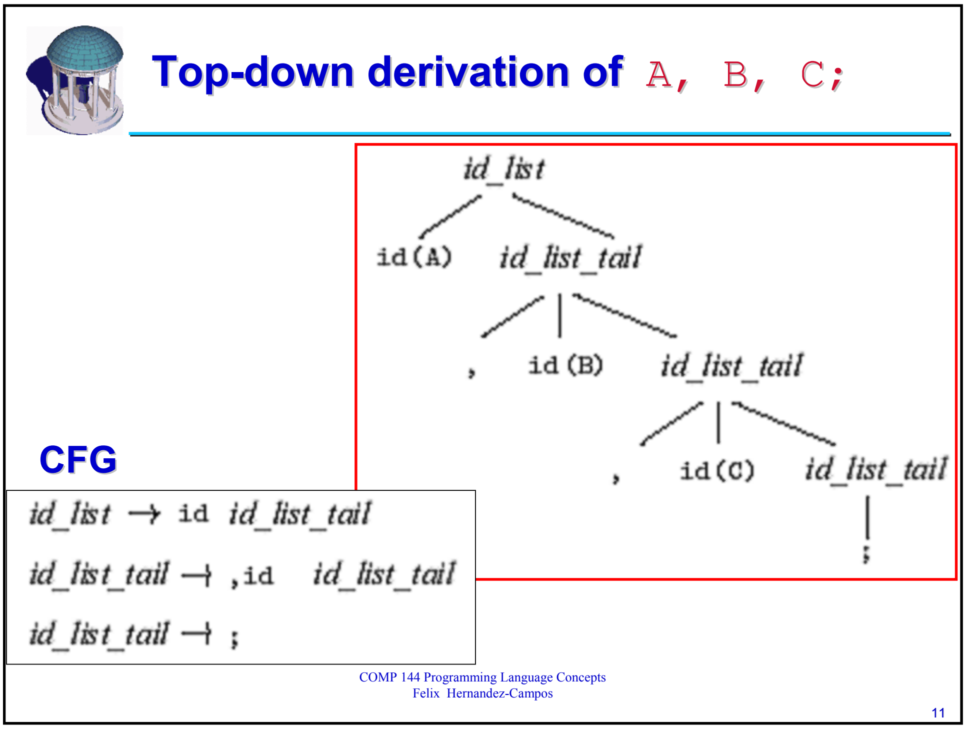
CFG

id_list → id *id_list_tail*
id_list_tail → , id *id_list_tail*
id_list_tail → ;

↑

Left-to-right,
Left-most derivation
LL(1) parsing





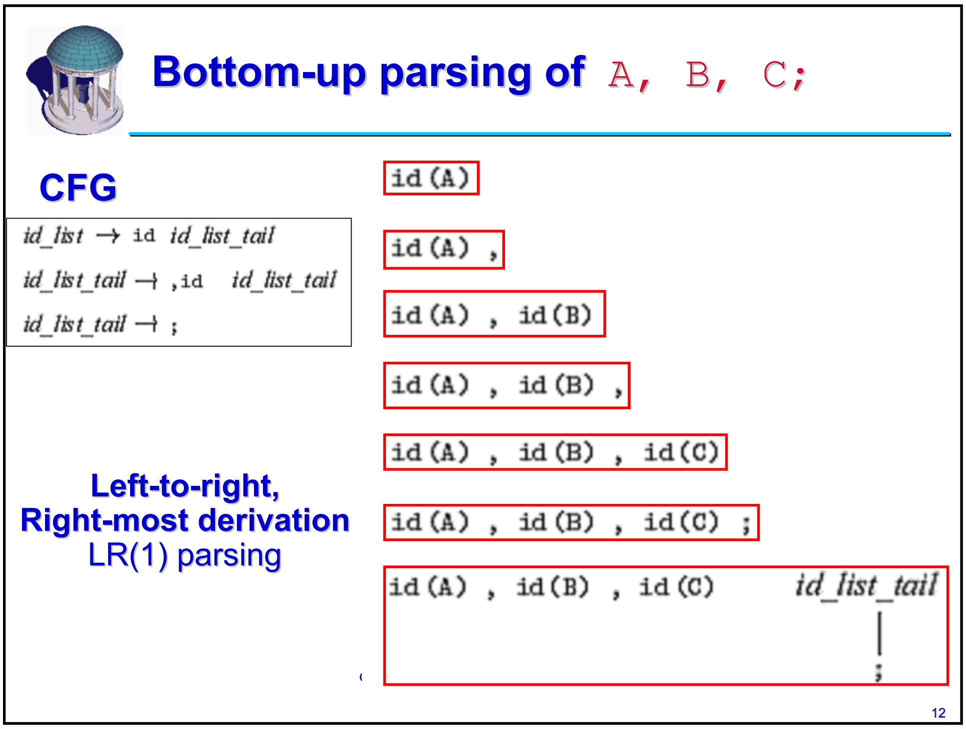
Top-down derivation of `A, B, C;`

CFG

id_list → id *id_list_tail*

id_list_tail → ,id *id_list_tail*

id_list_tail → ;



Bottom-up parsing of `A, B, C;`

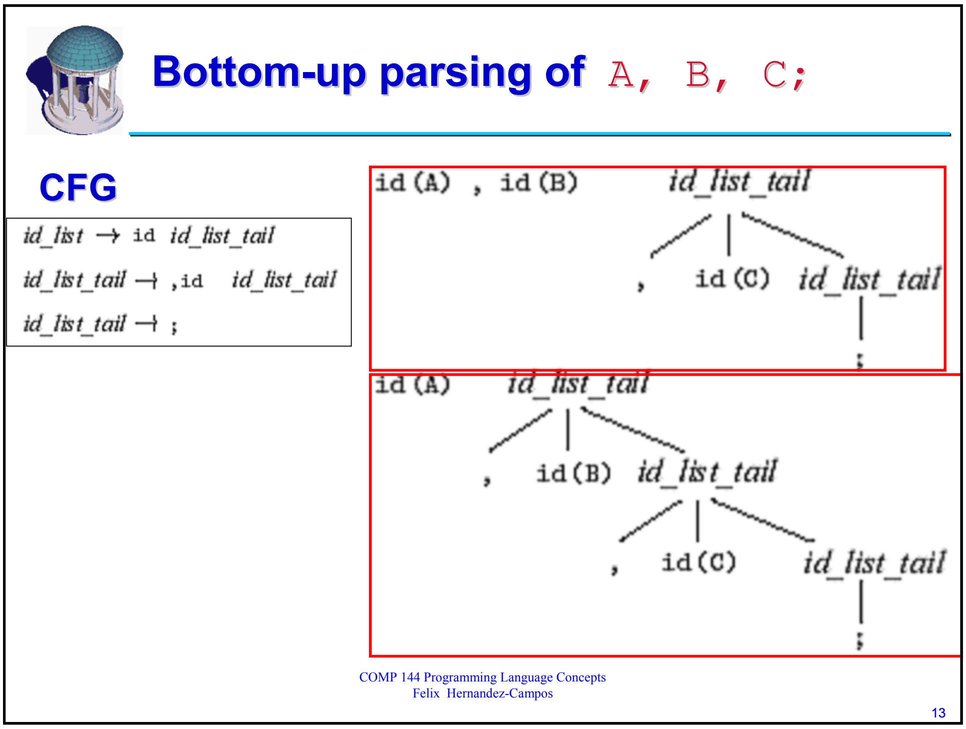
CFG

id_list → id *id_list_tail*

id_list_tail → ,id *id_list_tail*

id_list_tail → ;

Left-to-right,
Right-most derivation
LR(1) parsing



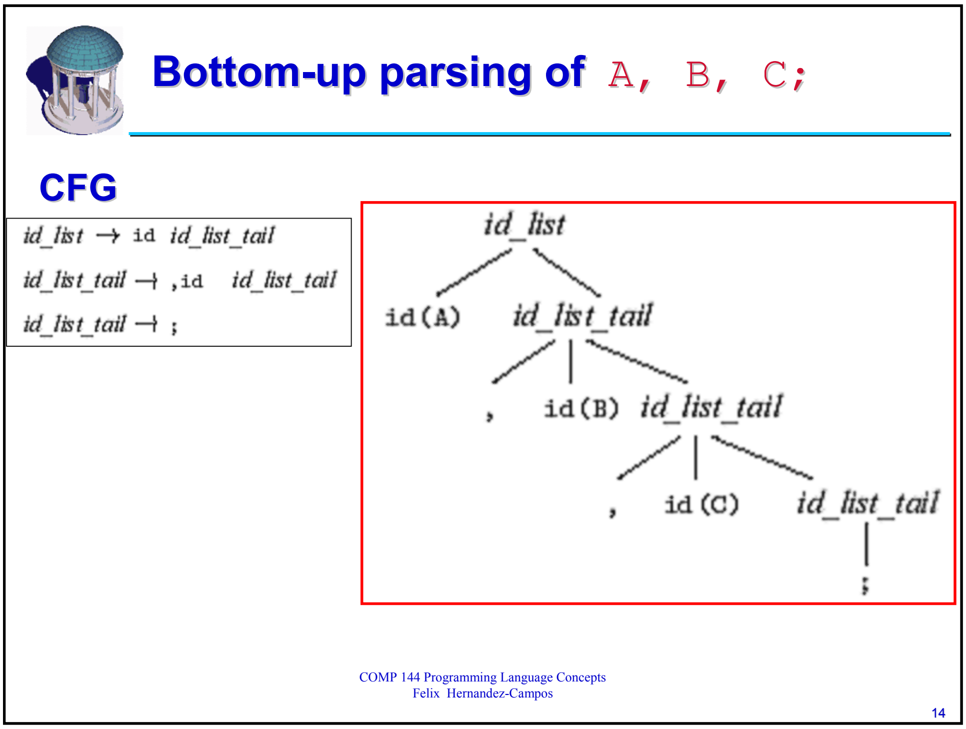
Bottom-up parsing of `A, B, C;`

CFG

id_list → id *id_list_tail*

id_list_tail → ,id *id_list_tail*

id_list_tail → ;



Bottom-up parsing of `A, B, C;`

CFG

id_list → id *id_list_tail*

id_list_tail → ,id *id_list_tail*

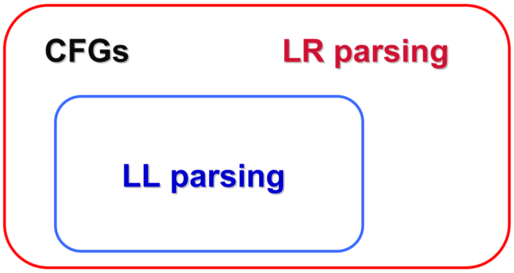
id_list_tail → ;

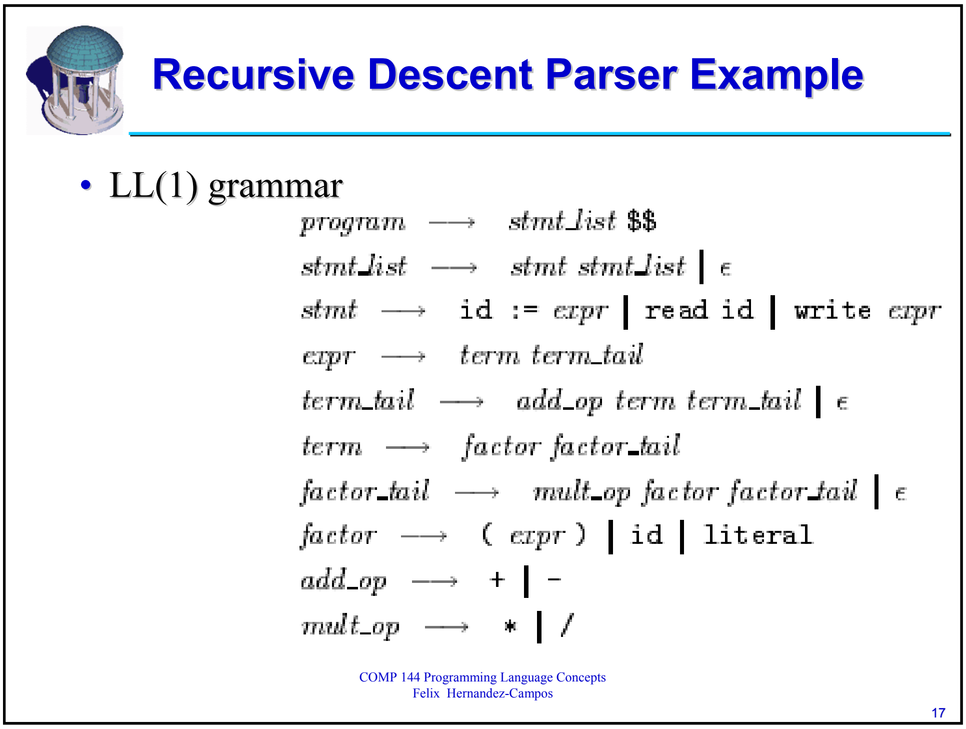
Parsing

- Parsing an arbitrary Context Free Grammar
  - $O(n^3)$
  - Too slow for large programs
- Linear-time parsing
  - LL parsers
    - Recognize LL grammar
    - Use a top-down strategy
  - LR parsers
    - Recognize LR grammar
    - Use a bottom-up strategy

Hierarchy of Linear Parsers

- Basic containment relationship
  - All CFGs can be recognized by LR parser
  - Only a subset of all the CFGs can be recognized by LL parsers

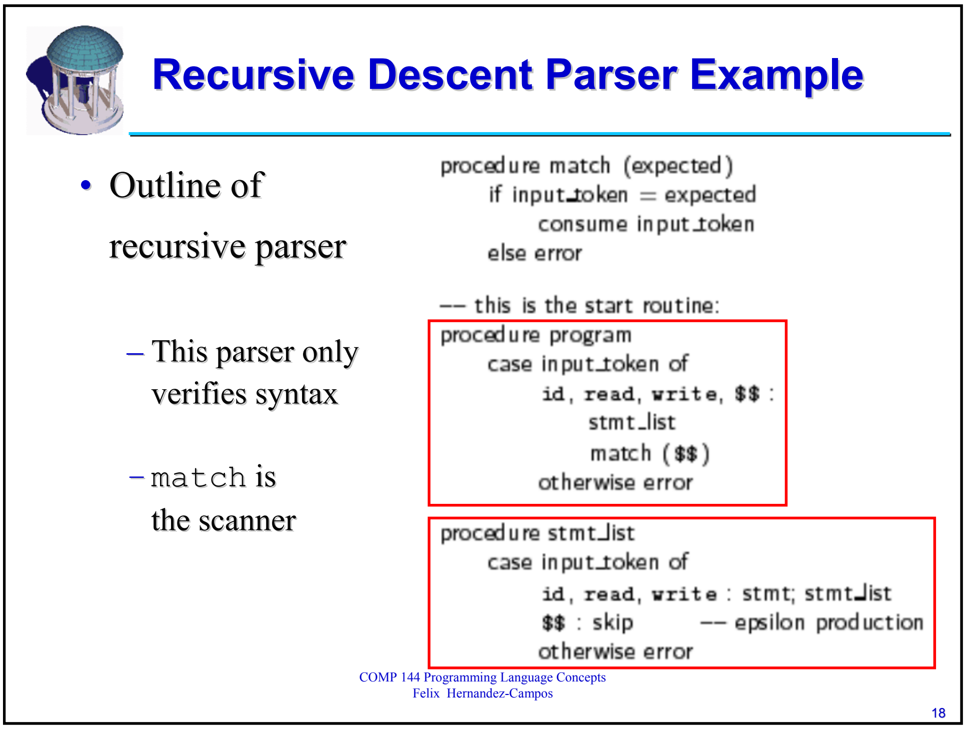




Recursive Descent Parser Example

- LL(1) grammar

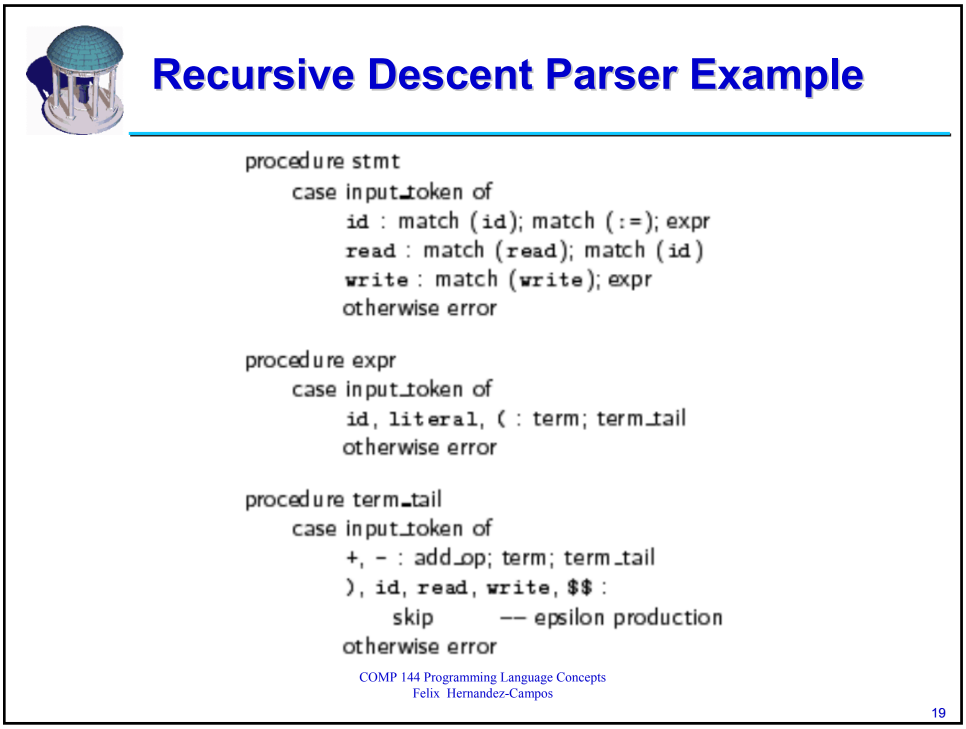
```
program      →  stmt_list $$
stmt_list    →  stmt stmt_list | ε
stmt         →  id := expr | read id | write expr
expr         →  term term_tail
term_tail    →  add_op term term_tail | ε
term         →  factor factor_tail
factor_tail  →  mult_op factor factor_tail | ε
factor       →  ( expr ) | id | literal
add_op       →  + | -
mult_op      →  * | /
```



Recursive Descent Parser Example

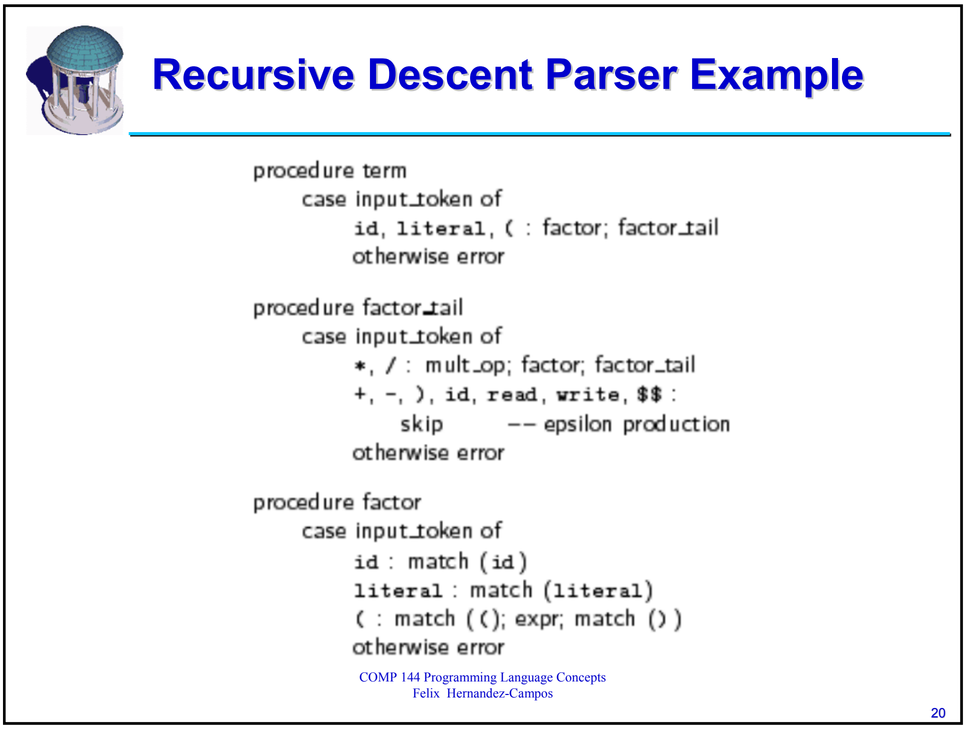
- Outline of recursive parser
  - This parser only verifies syntax
  - `match` is the scanner

```
procedure match (expected)
    if input_token = expected
        consume input_token
    else error

-- this is the start routine:
procedure program
    case input_token of
        id, read, write, $$ :
            stmt_list
            match ($$)
        otherwise error

procedure stmt_list
    case input_token of
        id, read, write : stmt; stmt_list
        $$ : skip          -- epsilon production
        otherwise error
```



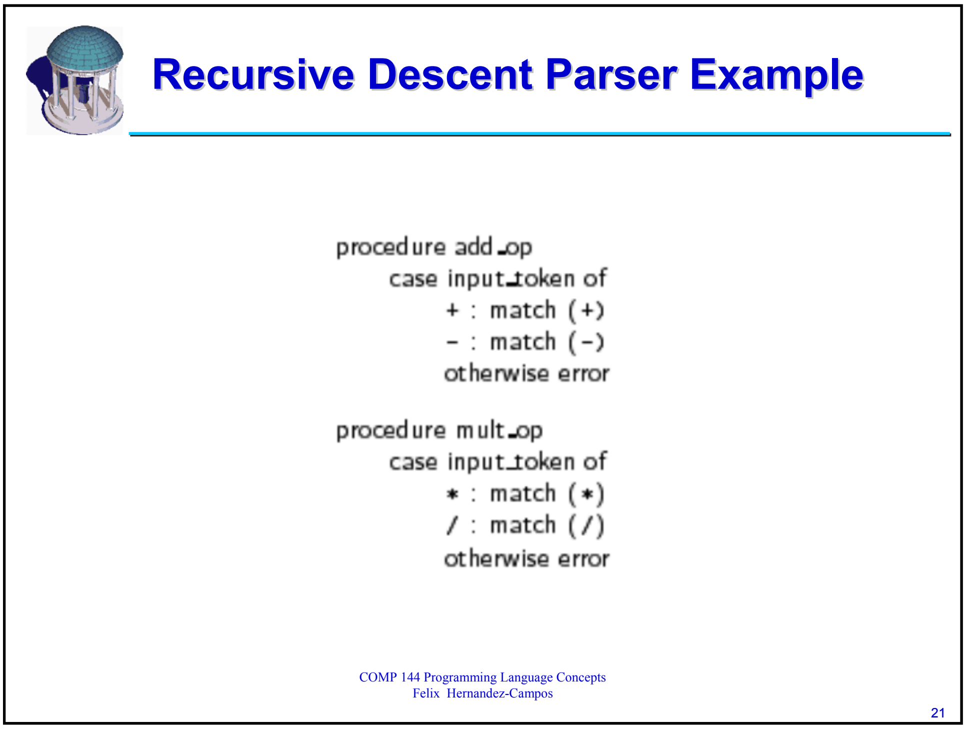
Recursive Descent Parser Example

```
procedure stmt
    case input_token of
        id : match (id); match (:=); expr
        read : match (read); match (id)
        write : match (write); expr
        otherwise error

procedure expr
    case input_token of
        id, literal, ( : term; term_tail
        otherwise error

procedure term_tail
    case input_token of
        +, - : add_op; term; term_tail
        ), id, read, write, $$ :
            skip        -- epsilon production
        otherwise error
```



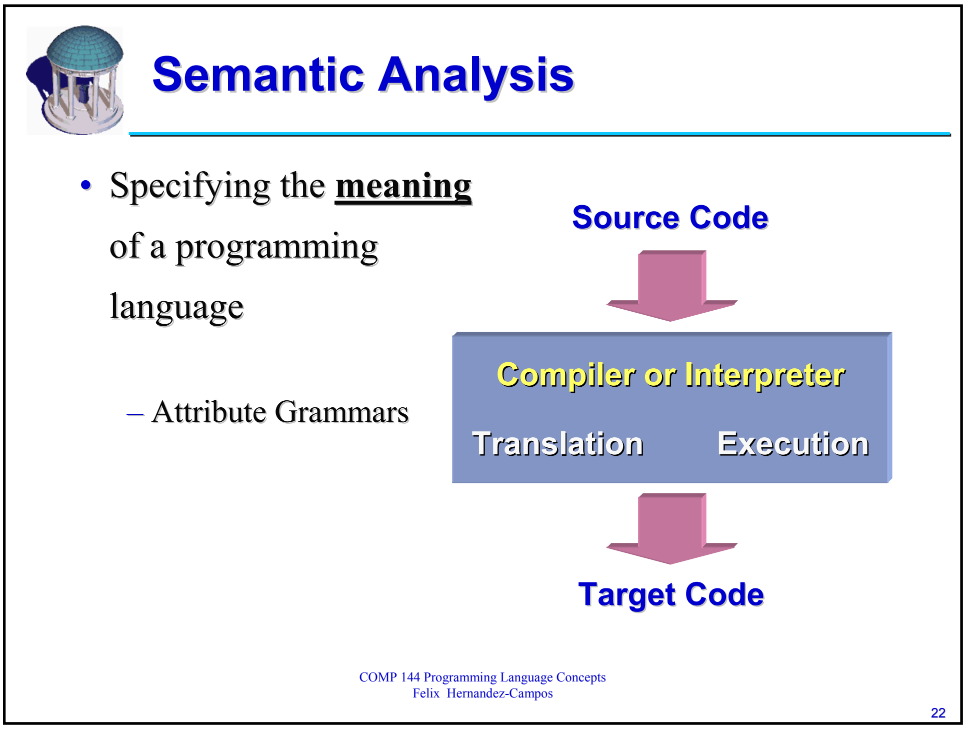
Recursive Descent Parser Example

```
procedure term
    case input_token of
        id, literal, ( : factor; factor_tail
        otherwise error

procedure factor_tail
    case input_token of
        *, / : mult_op; factor; factor_tail
        +, -, ), id, read, write, $$ :
            skip        -- epsilon production
        otherwise error

procedure factor
    case input_token of
        id : match (id)
        literal : match (literal)
        ( : match ((); expr; match ())
        otherwise error
```



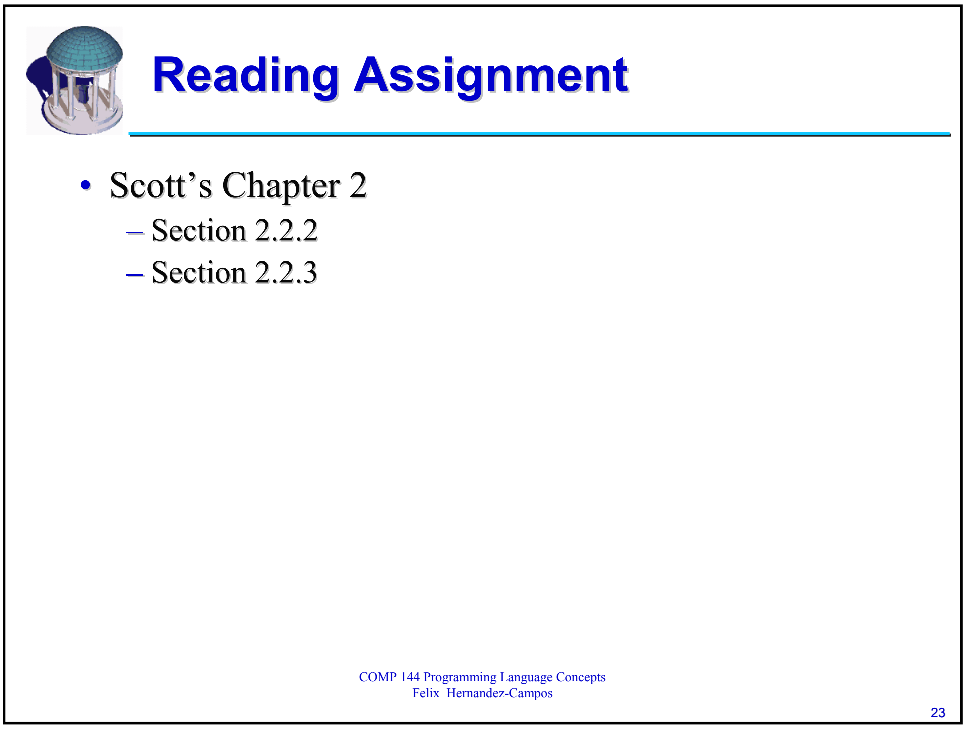
Recursive Descent Parser Example

```
procedure add_op
    case input_token of
        + : match (+)
        - : match (-)
        otherwise error

procedure mult_op
    case input_token of
        * : match (*)
        / : match (/)
        otherwise error
```



Semantic Analysis

- Specifying the **meaning** of a programming language
  - Attribute Grammars



Reading Assignment

- Scott’s Chapter 2
  - Section 2.2.2
  - Section 2.2.3