CS 326
Programming Languages, Concepts and Implementation

Instructor: Mircea Nicolescu

Programming Language Syntax

A Simple Program

Compute greatest common divisor of 2 integers:

afbf0015

```
27bdffd0 afbf0014 0c1002a8 00000000 0c1002a8 afa2001c 8fa4001c
00401825 10820008 0064082a 10200003 00000000 10000002 00832023
00641823 1483fffa 0064082a 0c1002b2 00000000 8fbf0014 27bd0020
03e00008 00001025
```

- Machine language
  - Sequence of bits that directly controls the processor
  - Add, compare, move data, etc.
  - Computer's time more valuable than programmers' time

A Simple Program

```
    addiu   sp,sp,-32               b       C
    sw      ra,20(sp)               subu    a0,a0,v1
    jal     getint              B:  subu    v1,v1,a0
    nop                         C:  bne     a0,v1,A
    jal     getint                  slt     at,v1,a0
    sw      v0,28(sp)           D:  jal     putint
    lw      a0,28(sp)               nop
    move    v1,v0                   lw      ra,20(sp)
    beq     a0,v0,D                 addiu   sp,sp,32
    slt     at,v1,a0                jr      ra
A:  beq     at,zero,B               move    v0,zero
    nop
```

- Assembly language
  - Mnemonic abbreviations, 1-to-1 correspondence to machine language
  - Translated by an *assembler*
  - Still machine-dependent

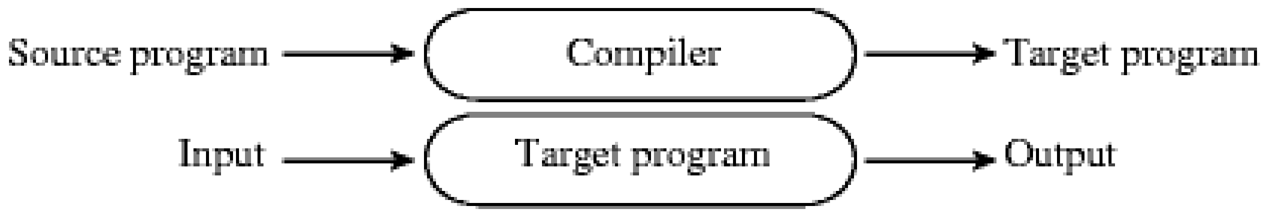
Compilation and Interpretation

- Need for:
  - Machine-independent language
  - Resemblance between language and mathematical computation

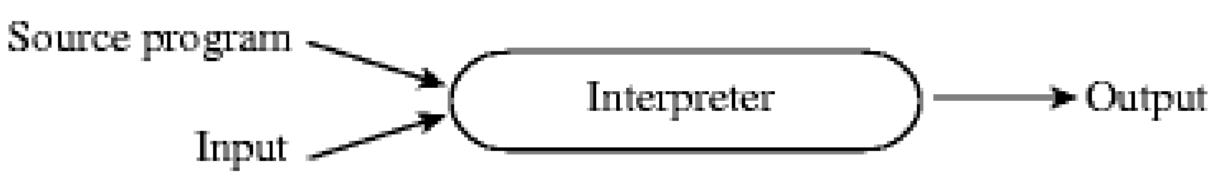
- High-level languages (the first: Fortran, Lisp)
  - Translated by a *compiler* or *interpreter*
  - No more 1-to-1 correspondence to machine language
  - Accepted when compilers could produce efficient code
  - Today: efficient compilers, large programs, high maintenance costs, rapid development requirements
  - Labor cost higher than hardware cost

Compilation and Interpretation

- *Compiler* translates into target language (machine language), then goes away
- The target program is the locus of control at execution time



- *Interpreter* stays around at execution time
- Implements a virtual machine whose machine language is the high-level language



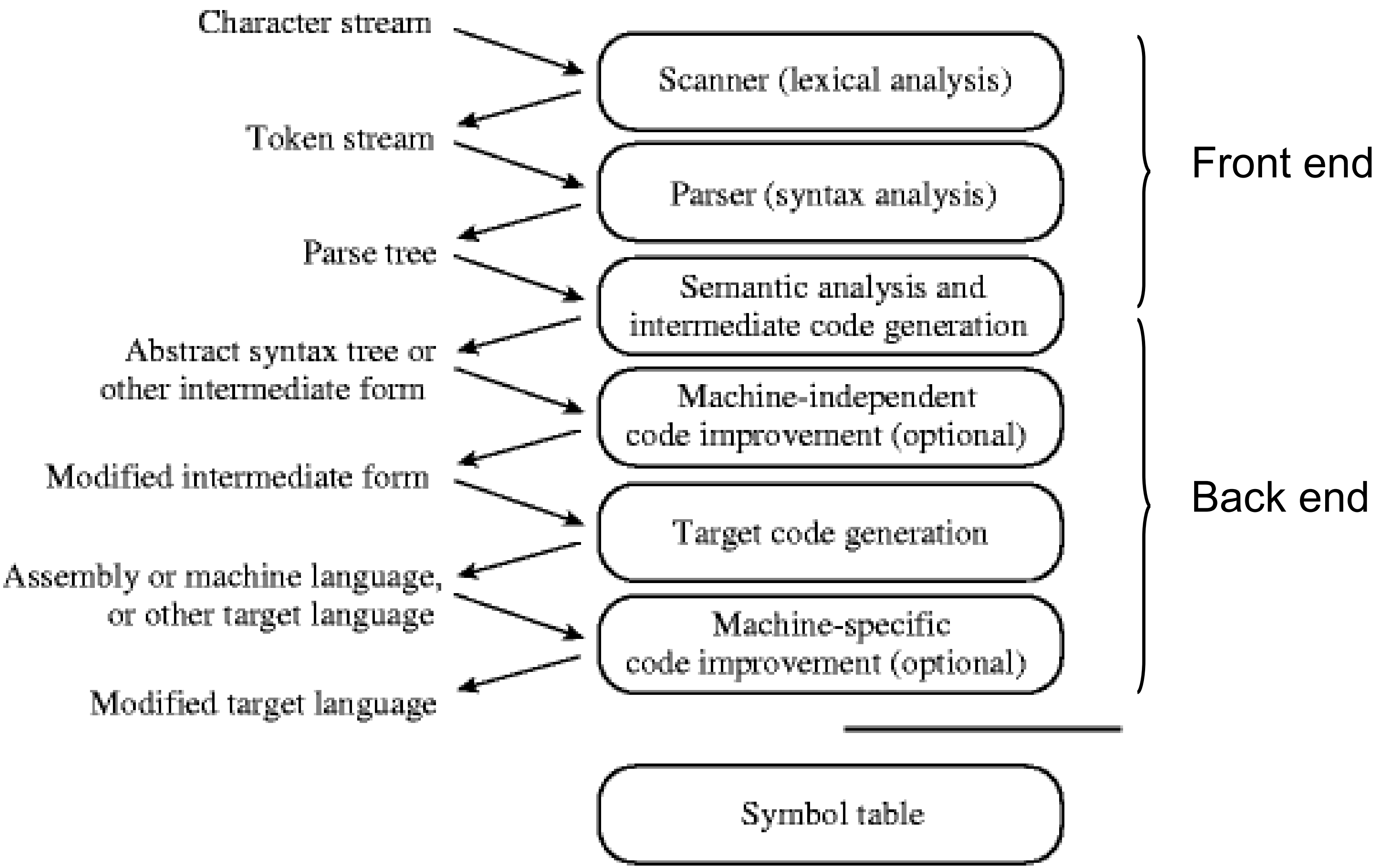
Compilation and Interpretation

- Interpretation
  - Greater flexibility, better diagnostics
  - Allow program features (data types or sizes) to depend on the input
  - Lisp, Prolog: program can write new pieces of itself and execute them on the fly
  - Java: compilation, then interpretation
- Compilation
  - Better performance
  - Many decisions are made only once, at compile time, not at every run time
  - Fortran, C

Programming Environments

- Set of tools:
  - Editors
  - Pretty printers
  - Style checkers
  - Preprocessors
  - Debuggers
  - Linkers
  - Module management
  - Browsers (cross-references)
  - Profilers
  - Assemblers
- Separate programs: Unix
- Integrated environment: Microsoft Visual C++, Visual Works for Smalltalk

Overview of Compilation



Lexical Analysis (Scanner)

- Compute greatest common divisor, in Pascal:

```
program gcd (input, output);
var i, j : integer;
begin
    read (i, j);
    while i <> j do
    if i > j then i := i – j
    else j := j – i;
    writeln (i)
end.
```

- The scanner extracts *tokens* (smallest meaningful units):

program	gcd	(	input	,	output	)	;
var	i	,	j	:	integer	;	begin
read	(	i	,	j	)	;	while

……………………………………………………………………………...……….

Syntax Analysis (Parser)

- The parser uses a context-free grammar, as a description of the language syntax:

program → PROGRAM *identifier* (*identifier more_identifiers*) ; *block* .

block → *labels constants types variables subroutines* BEGIN *statement*
more_statements END

more_identifiers → , *identifier more_identifiers*

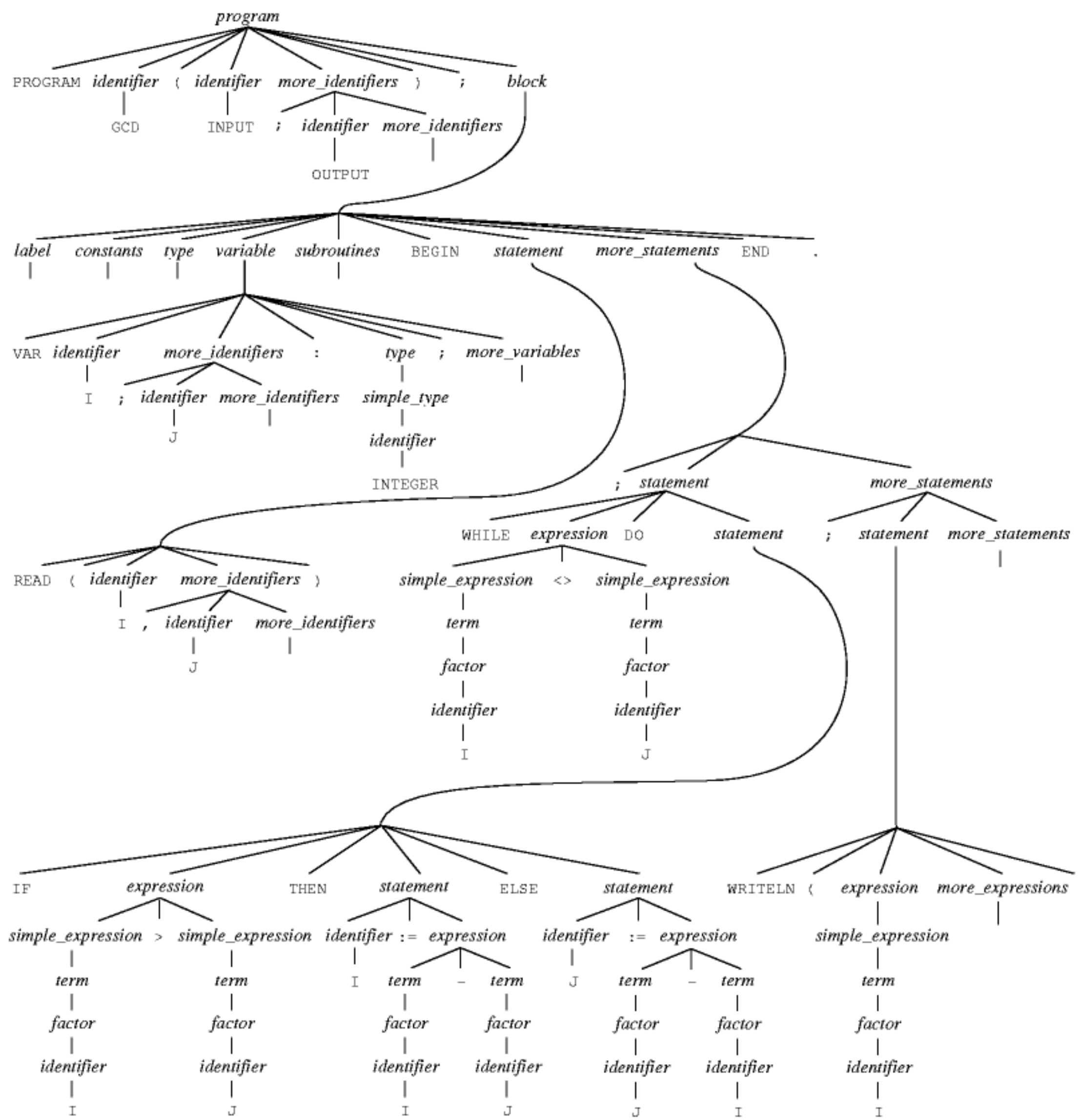
more_identifiers → ε

……………………..

- Generates a parse tree, that reflects the structure of the program
- Shows how the sequence of tokens can be derived under the rules of the grammar

Syntax Analysis (Parser)

Parse tree:

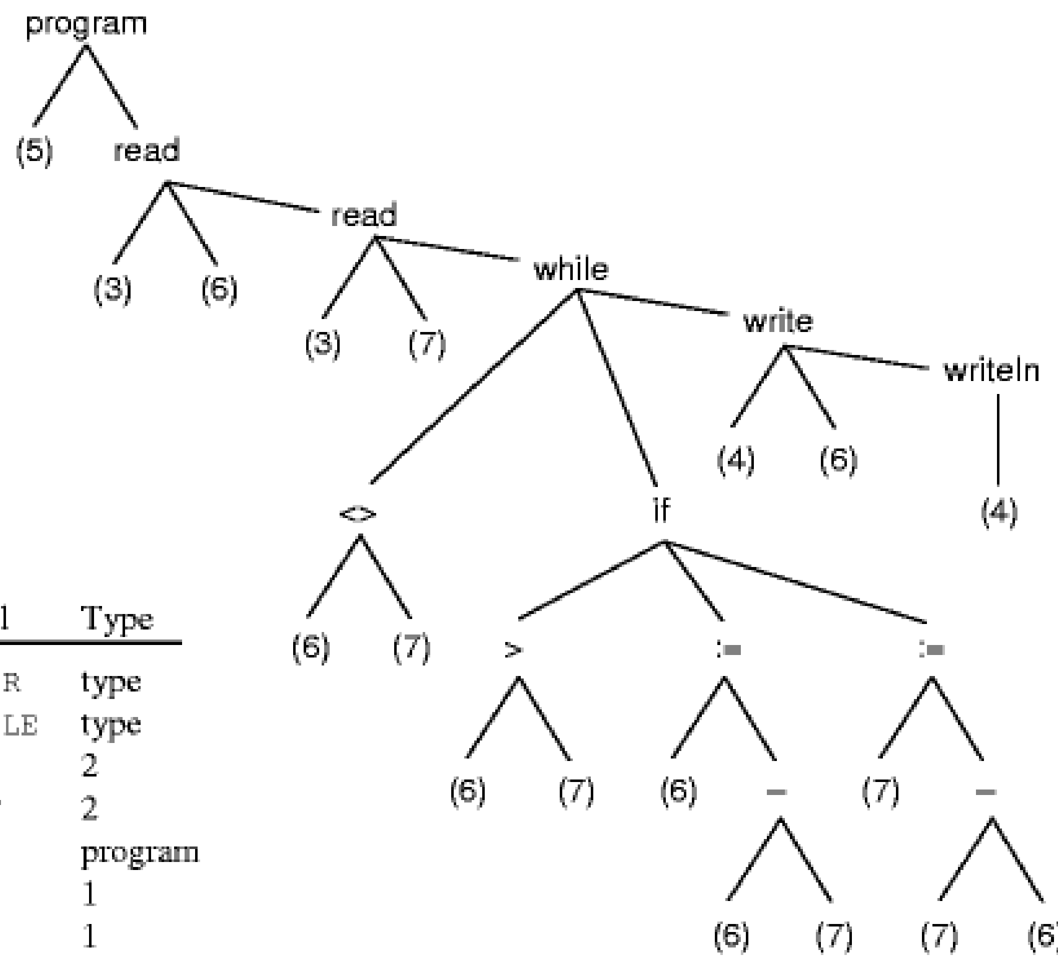


Semantic Analysis

- Discovers meaning in a program
- Static semantic analysis (at compile time):
  - Identifiers declared before use
  - Subroutine calls provide correct number and type of arguments
- Dynamic semantics (cannot be checked at compile time, so generate code to check at run time):
  - Pointers dereferenced only when refer to a valid object
  - Array subscript expressions lie within bounds
- Simplifies the parse tree → syntax tree
- Maintains symbol table, attaches attributes

Semantic Analysis

Syntax tree:



Index	Symbol	Type
1	INTEGER	type
2	TEXTFILE	type
3	INPUT	2
4	OUTPUT	2
5	GCD	program
6	I	1
7	J	1

Back end

- Target code generation
  - Generate assembly or machine language
  - Traverses the syntax tree to generate elementary operations: loads and stores, arithmetic operations, tests and branches
- Code improvement (optional)
  - A.k.a “optimization”
  - Transform program into a new version with same functionality, but more efficient (faster, uses less memory)

Compilation Review

- Lexical analysis (scanner)
  - Break program into primitive components, called tokens (identifiers, numbers, keywords, ...)
  - Formal models: regular grammars and finite automata

- Syntactic analysis (parser)
  - Create parse tree of the program
  - Formal models: context free grammars and pushdown automata

- Symbol table
  - Store information about declared objects (identifiers, procedure names, ...)

Compilation Review

- Semantic analysis
  - Understand relationships between tokens in the program
- Code improvement (machine independent)
  - Rewrite syntax tree to create a more efficient program
- Code generation
  - Convert parsed program into executable form in target (machine) language
- Code improvement (machine dependent)
  - Rewrite target code to create a more efficient program

Will discuss scanning and parsing

Formal Translation Models

- Formal language: a set of strings of symbols drawn from a finite alphabet
- Examples:
  - L_1 = { black, white }
  - L_2 = { 1, 01, 001, 0001, ... }
  - L_3 = { ab, aabb, aaabbb, ... }
  - L_4 = English

Grammars

- Nonterminals: a finite set of symbols:
  - <sentence> <subject> <predicate> <verb> <article> <noun>
- Terminals: a finite set of symbols:
  - the, boy, girl, ran, ate, cake
- Start symbol: one of the nonterminals:
  - <sentence>

Grammars

- Rules (productions): A finite set of replacement rules:

  <sentence> → <subject> <predicate>

  <subject> → <article> <noun>

  <predicate> → <verb> <article> <noun>

  <verb> → ran | ate

  <article> → the

  <noun> → boy | girl | cake

Grammars

- Replacement operator (written =>): Replace a nonterminal by the right hand side of a rule

- Derivation:

<sentence> => <subject> <predicate>	First rule
=> <article> <noun> <predicate>	Second rule
=> the <noun> <predicate>	Fifth rule
... => the boy ate the cake	

- Can also derive:

  <sentence> => ... => the cake ate the boy

  Syntax does not imply correct semantics

Grammars

- Concepts:

- Derivation: series of replacements to obtain string of terminals from start symbol
- Sentential form: any intermediate string of symbols during derivation
- Yield: the final sentential form, containing only terminals

- Language generated by grammar G: the set of all possible yields

Parse Trees

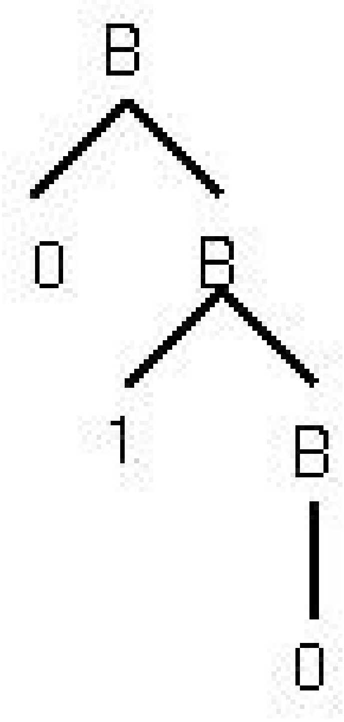
- Grammar:

  $B \rightarrow 0B \mid 1B \mid 0 \mid 1$

- Generate 010

- Derivation:

  B => 0B => 01B => 010



Parse tree

Parse Trees

- But derivations may not be unique:
- Grammar:

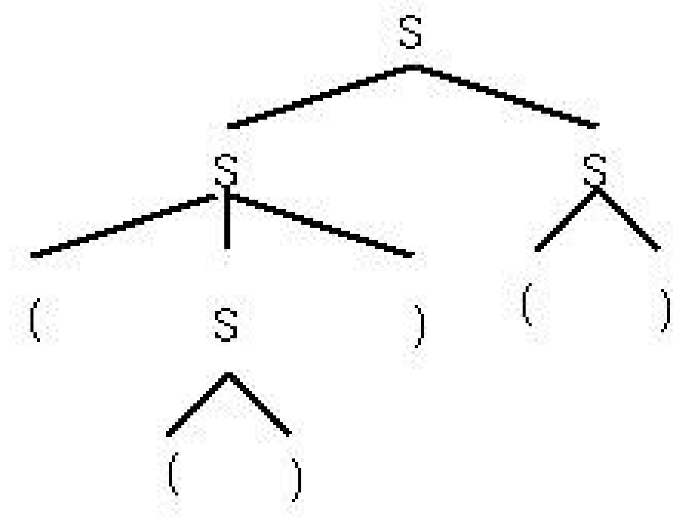
  $S \rightarrow SS \mid (S) \mid ()$

- Generate (())()
- Derivation1:

  S => SS =>(S)S =>(())S =>(()

- Derivation2:

  S => SS => S() =>(S)() =>(())()



Parse tree

Different derivations but get the same parse tree

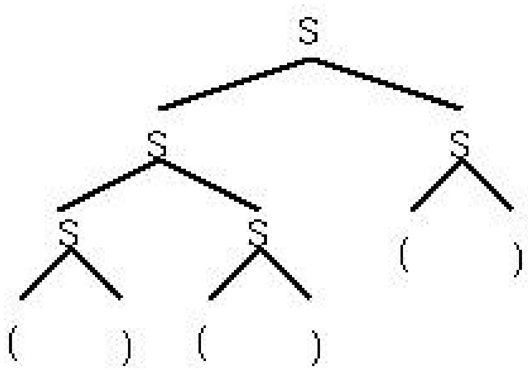
Parse Trees

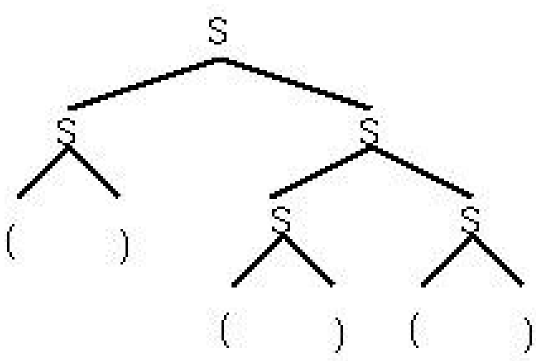
- Ambiguity
- Grammar: S -> SS | (S) | ()

- Generate ()()()
- Unique derivation:

  S => SS => SSS => ()SS => ()()S => ()()()

- Ambiguous grammar ⇔ one string has multiple parse trees





Readings

- Chapter 2, up to 2.2.3

Classification

- Chomsky hierarchy (incomplete):

Language	**Generator**	**Acceptor**	**Compile phase**
Regular	Regular grammar	Finite automaton	Lexical analysis (scanning)
Context-free	Context-free grammar	Push-down automaton	Syntax analysis (parsing)

- Regular languages are a subset of context-free ones

Specifying Syntax

- Define the structure of a language
- Interest for programmers, to write syntactically valid programs

- Will discuss:
  - regular expressions (equivalent to regular grammars)
  - context-free grammars

Regular Expressions

- Define what tokens are valid in a programming language
- The set of all valid tokens ⇔ a formal language that is regular
  - described using regular expressions

- Tokens (Pascal):
  - symbols: + - ; :=
  - keywords: begin end while if
  - integer literals: 141
  - real literals: 5.38e25
  - string literals: 'Tom'
  - identifiers: myVariable

Regular Expressions

- Variations:
  - uppercase / lowercase distinction (C vs Pascal)
  - keywords must be lowercase (C vs Modula-3)
  - free format: white space is ignored, only the relative order of tokens counts, not position in page
  - exceptions:
    - fixed format, 72 characters per line, special purpose columns (Fortran < 90)
    - line-breaks separate statements (Basic)
    - indentation matters (Haskell, Occam)

Regular Expressions

- Example: natural numbers

  digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

  non_zero_digit = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

  natural_number = *non_zero_digit digit**

- Or better:

  non_zero_digit = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

  digit = 0 | *non_zero_digit*

  natural_number = *non_zero_digit digit**

Regular Expressions

- Example: numeric literals (Pascal)

digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

unsigned_int = *digit digit**

unsigned_number = *unsigned_int* ((. *unsigned_int*) | ε)
((e (+ | - | ε) *unsigned_int*) | ε)

number = (+ | - | ε) *unsigned_number*

Regular Expressions

- Definition:

  - A regular expression R is either:
    - a character
    - the empty string ε
    - R_1 R_2 (concatenation)
    - R_1 | R_2 (alternation)
    - R_1* (repetition zero or more times - Kleene closure)

- Also used: R+ (repetition one or more times) ↔ R R*
- Note: no recursion allowed, if it has recursion it is not regular

Regular Expressions

- Language:

  set of strings over alphabet {*a*,*b*} that contain at least one *b*

- Regular expression:

  $(a \mid b)^* \, b \, (a \mid b)^*$

- Language:

  set of all Social Security Numbers, including the separator –

- Regular expression:

  $(0|1|2|3|4|5|6|7|8|9)^3 - (0|1|2|3|4|5|6|7|8|9)^2 - (0|1|2|3|4|5|6|7|8|9)^4$

Regular Expressions

- Regular expression:

  $(0 \mid 1)^* 0\ 0$

- Language:

  set of all strings over alphabet {*0*,*1*} that end in *00*

- Regular expression:

  $(a \mid b)^* a (a \mid b)^* a (a \mid b)^*$

- Language:

  set of all strings over alphabet {*a*,*b*} that contain at least two *a*’s

Context-Free Grammars

- Language:

  set of strings over alphabet {*a*,*b*} that read the same from left to right as from right to left (palindromes)

- Grammar:

  $S \rightarrow a\ S\ a \mid b\ S\ b \mid a \mid b \mid \varepsilon$

Context-Free Grammars

- Example: arithmetic expression

expression → identifier | number | - *expression* | (*expression*)
| *expression operator expression*

operator → + | - | * | /

  - nonterminals: *expression*, *operator*
  - terminals: identifier, number, +, -, *, /, (,)
  - start symbol: *expression*

Derivation and Parse Trees

- Generate "slope * x + intercept"

expression => *expression operator* <u>*expression*</u>

=> *expression* <u>*operator*</u> identifier

=> <u>*expression*</u> + identifier

=> *expression operator* <u>*expression*</u> + identifier

=> *expression* <u>*operator*</u> identifier + identifier

=> <u>*expression*</u> * identifier + identifier

=> identifier * identifier + identifier

(slope) (x) (intercept)

Derivation and Parse Trees

expression => *expression operator* <u>*expression*</u>

=> *expression* <u>*operator*</u> identifier

=> <u>*expression*</u> + identifier

=> *expression operator* <u>*expression*</u> + identifier

=> *expression* <u>*operator*</u> identifier + identifier

=> <u>*expression*</u> * identifier + identifier

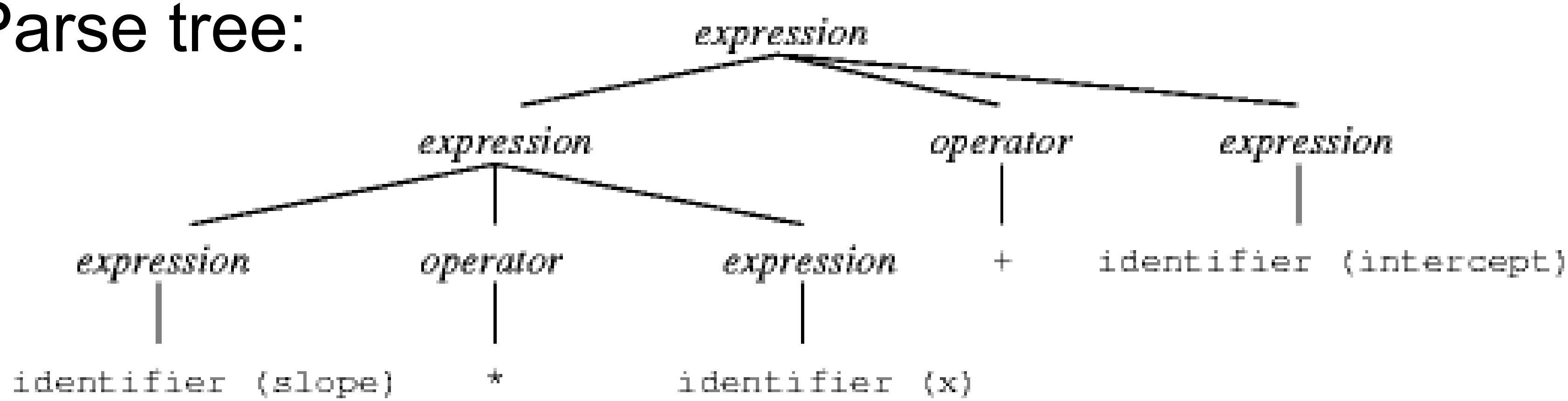
=> identifier * identifier + identifier

(slope) (x) (intercept)

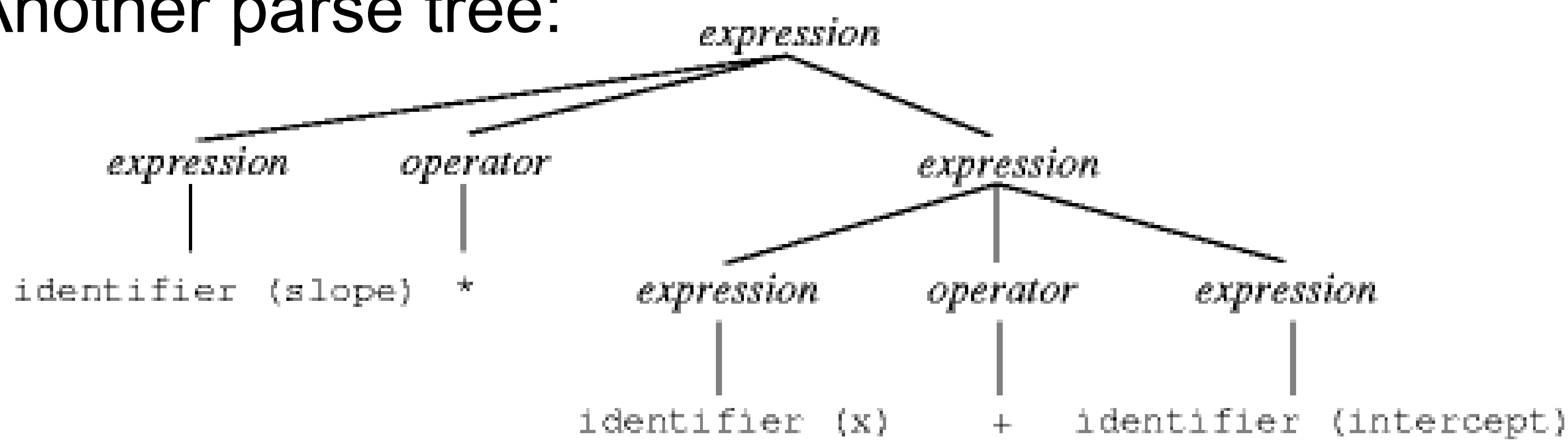
- Derivation – the series of replacements
- Sentential form – any intermediate string of symbols
- Yield – the final sentential form, with only terminals
- Right-most / left-most derivation – strategy on what nonterminal to expand

Derivation and Parse Trees

- Parse tree:



- Another parse tree:



Derivation and Parse Trees

- Issues:

- ambiguity: more than one parse tree
- for any given CF language - infinitely many CF grammars
- avoid ambiguous ones

- reflect useful properties:
  - associativity: 10-4-3 means (10-4)-3
  - precedence: 3+4*5 means 3+(4*5)

Derivation and Parse Trees

- A new grammar for arithmetic expressions:

expression → *term* | *expression add_op term*

term → *factor* | *term mult_op factor*

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

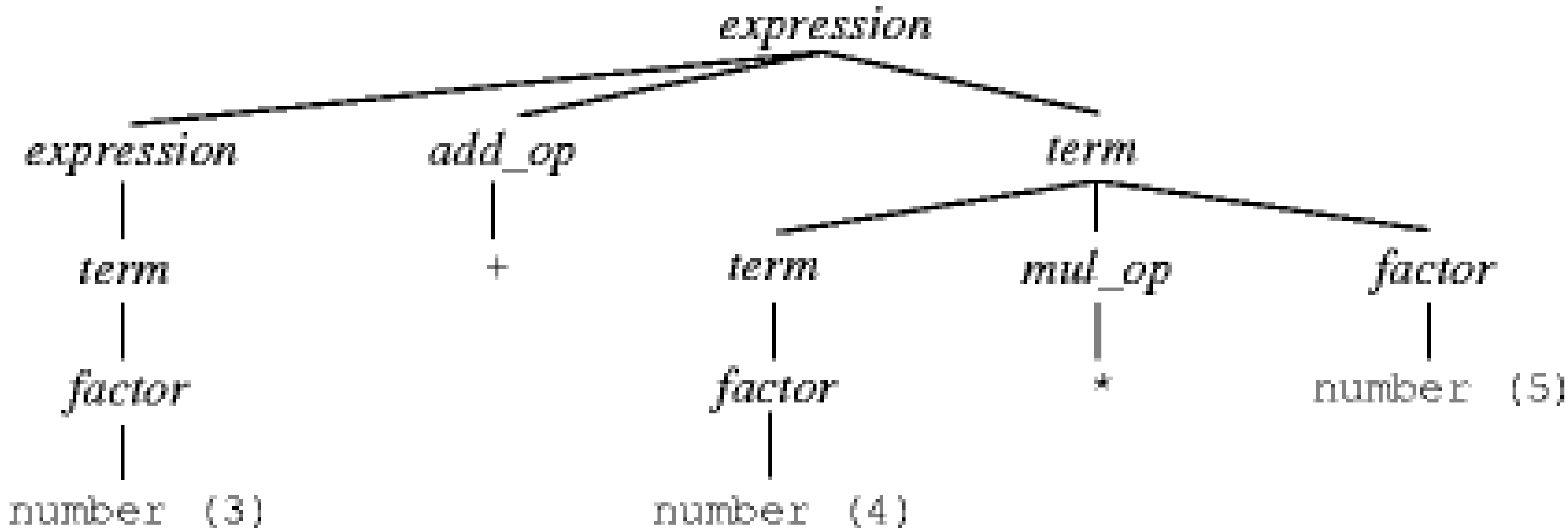
add_op → + | -

mult_op → * | /

- Unambiguous, also captures associativity and precedence

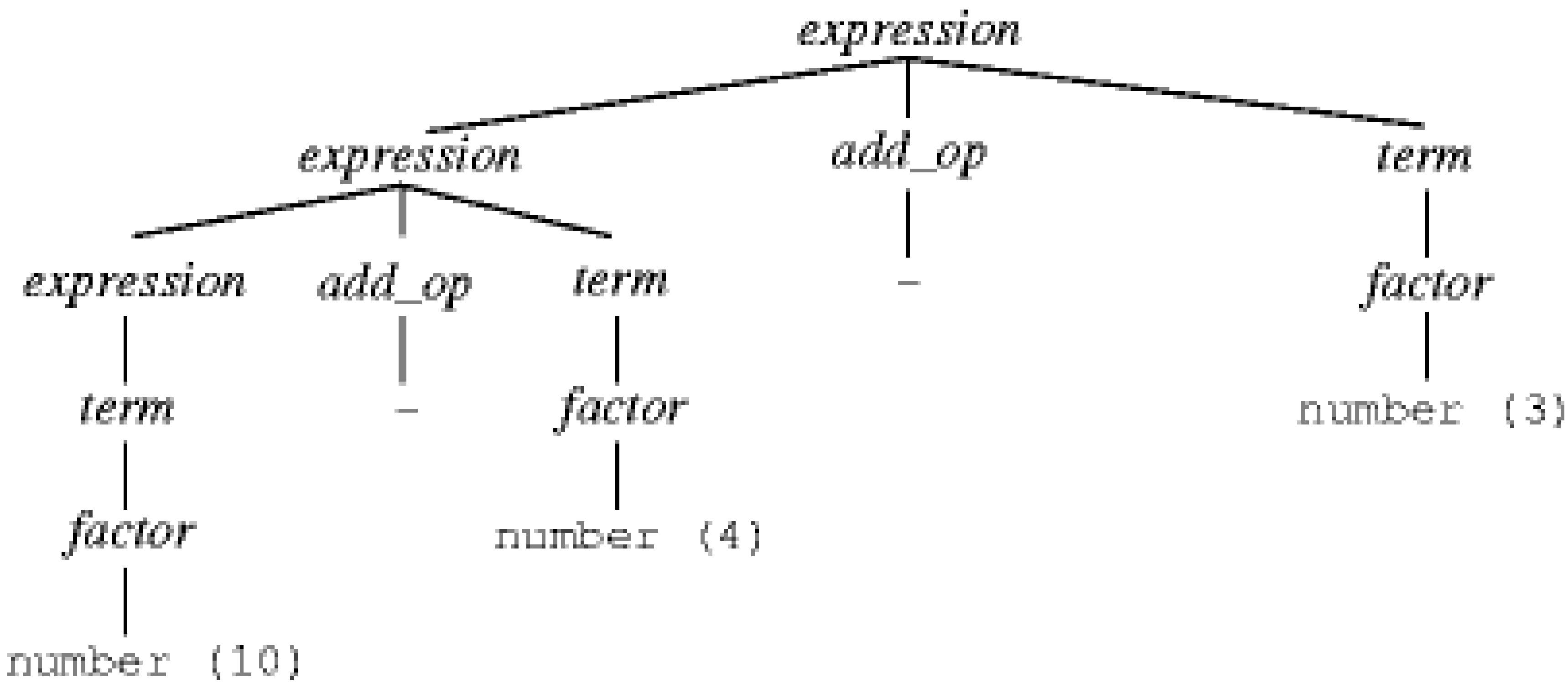
Derivation and Parse Trees

- Precedence: 3 + 4 * 5



Derivation and Parse Trees

- Associativity: 10 - 4 - 3



Announcements

- Readings
  - Rest of Chapter 2, up to (and including) 2.2.3
- Homework
  - HW 1 out – due on February 8
  - Submission
    - at the beginning of class
    - with a title page: Name, Class, Assignment #, Date
    - preferably typed

Recognizing Syntax

- Verify if a given program conforms to the syntax of the language
- Discover the syntactic structure of the program

- Corresponds to language implementation (writing compilers)

- Will discuss:
  - scanners
  - parsers

Architecture

- Scanner
  - ignores white space (blanks, tabs, new-lines)
  - ignores comments
  - recognizes tokens
  - implemented as a function that returns next token every time it is called
- Parser
  - calls the scanner to obtain tokens
  - builds parse tree
  - passes it to the later phases (semantic analysis, code generation and improvement)

- Parser controls the compilation process - "syntax-directed translation"

Scanning

- The scanner is usually implemented as either:
  - an ad-hoc program
  - an explicit finite automaton
  - a table-driven finite automaton

- General rule - always accept longest possible token:

  foobar is foobar, not f or foo or foob

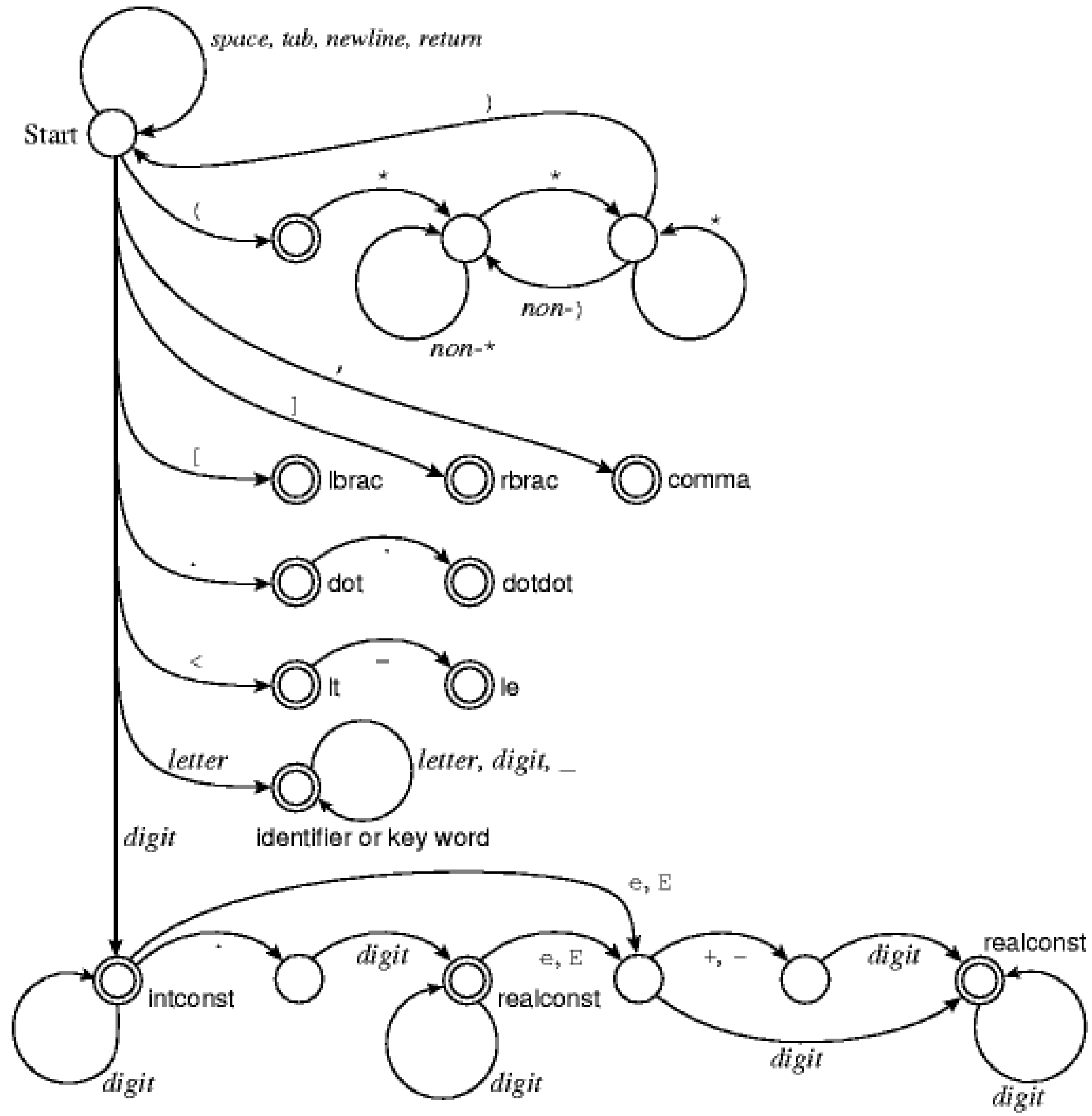
  3.14 is 3.14, not 3 then . then 14

Ad-hoc Scanner

- Hand-written scanner for Pascal (incomplete):

```
we skip any initial white space (spaces, tabs, and newlines)
we read the next character
if it is a ( we look at the next character
    if that is a * we have a comment;
        we skip forward through the terminating *)
    otherwise we return a left parenthesis and reuse the look-ahead
if it is one of the one-character tokens ([ ] , ; = + - etc.)
    we return that token
if it is a . we look at the next character
    if that is a . we return ..
    otherwise we return . and reuse the look-ahead
if it is a < we look at the next character
    if that is a = we return <=
    otherwise we return < and reuse the look-ahead
etc.
```

Finite Automaton (FA)



Scanner - Explicit FA

```
state := start
loop
    case state of
        start :
            erase text of current token
            case input_char of
                ' ', '\t', '\n', '\r' : no_op
                '[' : state := got_lbrac
                ']' : state := got_rbrac
                ',' : state := got_comma
                . . .
                '(' : state := saw_lparen
                '.' : state := saw_dot
                '<' : state := saw_lthan
                . . .
                'a'..'z', 'A'..'Z' :
                    state := in_ident
                '0'..'9' : state := in_int
                . . .
                else error
        . . .
        saw_lparen: case input_char of
            '*' : state := in_comment
            else return lparen
        in_comment: case input_char of
            '*' : state := leaving_comment
            else no_op
        leaving_comment: case input_char of
            ')' : state := start
            else state := in_comment
        . . .
        saw_dot : case input_char of
            '.' : state := got_dotdot
            else return dot
        . . .
        saw_lthan : case input_char of
            '=' : state := got_le
            else return lt
        . . .
        in_ident : case input_char of
            'a'..'z', 'A'..'Z', '0'..'9', '_' : no_op
            else
                look up accumulated token
                    in keyword table
                if found, return keyword
                else return identifier
        . . .
        in_int : case input_char of
            '0'..'9' : no_op
            '.' :
                peek at character beyond input_char;
                    if '0'..'9', state := saw_real_dot
                    else
                        unread peeked-at character
                        return intconst
            'a'..'z', 'A'..'Z', '_' : error
            else return intconst
        . . .
        saw_real_dot : . . .
        . . .
        got_lbrac : return lbrac
        got_rbrac : return rbrac
        got_comma : return comma
        got_dotdot : return dotdot
        got_le : return le
        . . .
    append input_char to text of current token
    read new input_char
```

Look-Ahead

- In Pascal:

3.14	real number
3..5	the range of integers between 3 and 5

- If it has seen the 3 and the next character coming is a dot, cannot decide yet - needs to peek one more character ahead

Look-Ahead

- In Fortran IV:

```
DO 5 I = 1,25        loop (for I = 1 to 25)
DO 5 I = 1.25        assignment (DO5I = 1.25)
```

- After seeing DO, cannot decide until reaching the comma or dot

```
DO 5,I = 1,25        alternate syntax for loop, FORTRAN 77
```

Scanner – Table-Driven FA

```
state = 1..number of states
action_rec = record
    action : (move, recognize, error)
    new_state : state
    token_found : token

scan_tab : array [char, state] of action_rec
keyword_tab : set of record
    k_image : string
    k_token : token
-- these two tables are created by a scanner generator tool

tok : token
image : string
cur_state : state
cur_char : char

state := start_state
image := null
repeat
    loop
        read cur_char
        case scan_tab[cur_char, cur_state].action
            move:
                cur_state := scan_tab[cur_char, cur_state].new_state
            recognize:
                tok := scan_tab[cur_char, cur_state].token_found
                exit inner loop
            error:
                -- print error message and recover; probably start over
        append cur_char to image
    -- end inner loop
until tok not in [white_space, comment]
look image up in keyword_tab and replace tok with appropriate keyword if found
return (tok, image)
```

Automatic Scanner Generators

- Unix: lex / flex
  - Take regular expresions as input
  - Produce a C program as output, that implements the scanner (table-driven)

Parsing

- Given any context-free grammar, we can create a parser that runs in $O(n^3)$ time – too long

- Particular classes of grammars that run in $O(n)$ (linear) time:
  - LL (left-to-right, leftmost derivation)
  - LR (left-to-right, rightmost derivation)

  - LL parsers are also called “top-down” or “predictive”
  - LR parsers are also called “bottom-up” or “shift-reduce”

Parsing

- Example – consider the grammar:

id_list → id *id_list_tail*

id_list_tail → , id *id_list_tail*

id_list_tail → ;

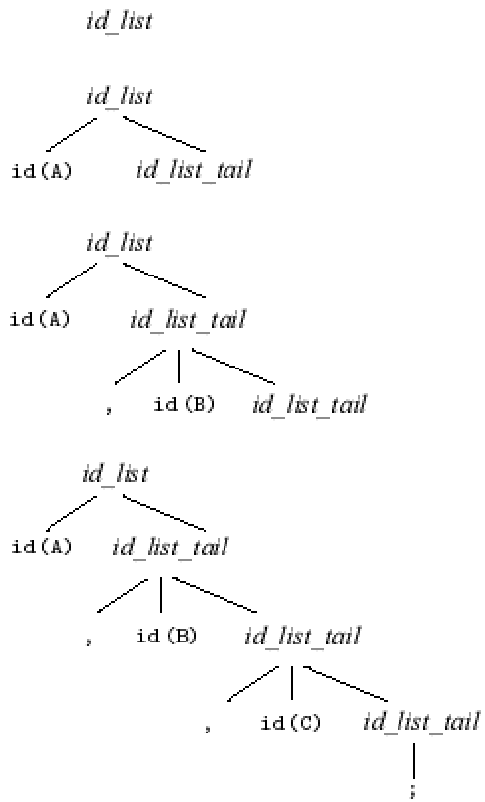
- Describes a comma-separated list of identifiers, such as:

A, B, C;

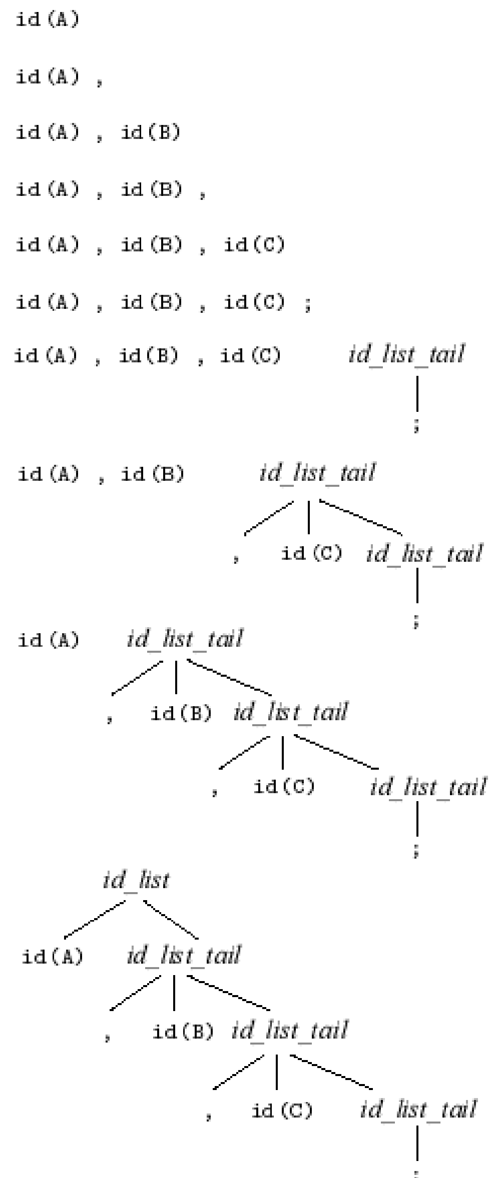
- Let’s parse the input above

Parsing

Top-down



Bottom-up



Input:

A, B, C;

Grammar:

id_list → id *id_list_tail*

id_list_tail → ,id *id_list_tail*

id_list_tail → ;

Parsing

- The example grammar is not very well suited for bottom-up parsing. Why?

- Because it needs to shift all input tokens until it finds the ;
- Here is a more suitable one:

$$
\begin{aligned}
\textit{id\_list} &\rightarrow \textit{id\_list\_prefix}\ ; \\
\textit{id\_list\_prefix} &\rightarrow \textit{id\_list\_prefix}\ ,\ \mathrm{id} \\
&\rightarrow \mathrm{id}
\end{aligned}
$$

- Other classes of grammars:
  - Subclasses of LR: SLR, LALR, …
  - Subclasses of LL: SLL, …
- Notation for number of tokens to look ahead:
  - LL(k), LR(k)
  - In general, only LL(1) and LR(1) are used

Parsing

- Implementing a parser:
  - Recursive descent parser (top-down) – section 2.2.3 from textbook
  - Automatic parser generators – in Unix: yacc / bison
    - Take a grammar as input
    - Produce a C program as output, that implements the parser

Announcements

- Readings
  - Chapter 2, up to (and including) 2.2.3