Programming Language Syntax
- Scanning -

Chapter 2, Sections 2.1-2.2

Regular Expressions

- *Token*: a shortest string of characters with meaning
- Tokens – specified by regular expressions
- An *alphabet* Σ is any finite nonempty set
  - Examples:
  - English: {a, b, …, z},
  - binary: {0, 1}
  - $\{a, b, \ldots, z, 0, 1, \ldots, 9, \cdot, |, {}^*, \varepsilon\}$
- The set of all finite strings over Σ is denoted Σ^*
- The *empty* string: $\varepsilon \in \Sigma^*$ (has zero characters)

Regular Expressions

- Regular expressions
- Regular expressions over an alphabet Σ are all strings obtained as follows:
  - ε is a regular expression
  - any character $a \in \Sigma$ is a regular expression
  - For reg. exp. α, β, the following are reg. exp.:
    - $\alpha \cdot \beta$ - *concatenation* ('$\cdot$' omitted: $\alpha\beta$)
    - $\alpha \mid \beta$ - *union* ('$|$' = or) (sometimes denoted $\alpha + \beta$)
    - α^* - *Kleene star* (0 or more repetitions)
      - $\alpha^+ = \alpha\, \alpha^*$ (1 or more repetitions)

Regular Expressions

- Example: Signed integers:

$$sign\_int \rightarrow (\texttt{+} \mid \texttt{-} \mid \varepsilon)(\texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9})(\texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9})^*$$

- Example: Numerical constants:

$$number \rightarrow integer \mid real$$

$$integer \rightarrow digit\ digit^*$$

$$real \rightarrow integer\ exponent \mid decimal\ (\ exponent \mid \varepsilon\ )$$

$$decimal \rightarrow digit^*\ (\ \texttt{.}\ digit \mid digit\ \texttt{.}\ )\ digit^*$$

$$exponent \rightarrow (\ \texttt{e} \mid \texttt{E}\ )\ (\ \texttt{+} \mid \texttt{-} \mid \varepsilon\ )\ integer$$

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

- ‘$\rightarrow$’ means “can be”
- Precedence order: ‘*’ > ‘·’ > ‘|’

Regular Expressions

- Other applications:
  - `grep` family of tools in Unix
  - many editors
  - scripting languages:
    - Perl
    - Python
    - Ruby
    - `awk`
    - `sed`

Formatting issues

- Upper vs. lower case
  - distinct in some languages: C, Python, Perl
  - same in others: Fortran, Lisp, Ada
- Identifiers: letters, digits, underscore (most languages)
  - camel case: someIdentifierName
  - underscore: some_identifier_name
- Unicode
  - non-Latin characters have become important
- White spaces
  - usually ignored
  - separate statements: Python, Haskell, Go, Swift
  - indentation important: Python, Haskell

Context-Free Grammars

Context Free Grammar (CFG)

- CFG consists of:
  - A set of *terminals, T*
  - A set of *non-terminals, N*
  - A *start symbol, S* ∈ *N*
  - A set of *productions;* subset of $N \times (N \cup T)^*$

- Example: Balanced parentheses:

$$S \to \varepsilon$$
$$S \to SS$$
$$S \to (S)$$

Context-Free Grammars

- Example: CFG for arithmetic expressions:

  $expr \rightarrow \texttt{id} \mid \texttt{number} \mid \texttt{-}\ expr \mid \texttt{(}\ expr\ \texttt{)} \mid expr\ op\ expr$

  $op \rightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid$

- *Derivation*: start with *S*, continue with productions
  - replace LHS nonterminal by the RHS
- Example: generate the string: `slope * x + intercept`

$$
\begin{array}{lll}
\underline{expr} & \Rightarrow expr\ op\ \underline{expr} & (S = expr) \\
 & \Rightarrow expr\ \underline{op}\ \texttt{id} & (`\Rightarrow\text{'} = \text{“derives”}) \\
 & \Rightarrow \underline{expr}\ \texttt{+}\ \texttt{id} & (`\Rightarrow^{*}\text{'} = 0 \text{ or more steps }) \\
 & \Rightarrow expr\ op\ \underline{expr}\ \texttt{+}\ \texttt{id} & \\
 & \Rightarrow expr\ \underline{op}\ \texttt{id}\ \texttt{+}\ \texttt{id} & \\
 & \Rightarrow \underline{expr}\ \texttt{*}\ \texttt{id}\ \texttt{+}\ \texttt{id} & \\
 & \Rightarrow \texttt{id}\quad \texttt{*}\quad \texttt{id}\ \texttt{+}\quad \texttt{id} & \\
 & \quad \texttt{(slope)}\quad \texttt{(x)}\quad \texttt{(intercept)} &
\end{array}
$$

- Sentential form: any string along the way

Context-Free Grammars

- *Right-most derivation*: the rightmost nonterminal is replaced

$$
\begin{aligned}
\underline{\mathit{expr}} &\Longrightarrow \mathit{expr}\ \mathit{op}\ \underline{\mathit{expr}} \\
&\Longrightarrow \mathit{expr}\ \underline{\mathit{op}}\ \texttt{id} \\
&\Longrightarrow \underline{\mathit{expr}} + \texttt{id} \\
&\Longrightarrow \mathit{expr}\ \mathit{op}\ \underline{\mathit{expr}} + \texttt{id} \\
&\Longrightarrow \mathit{expr}\ \underline{\mathit{op}}\ \texttt{id} + \texttt{id} \\
&\Longrightarrow \underline{\mathit{expr}} * \texttt{id} + \texttt{id} \\
&\Longrightarrow \texttt{id * id + id}
\end{aligned}
$$

- *Left-most derivation*: the leftmost nonterminal is replaced

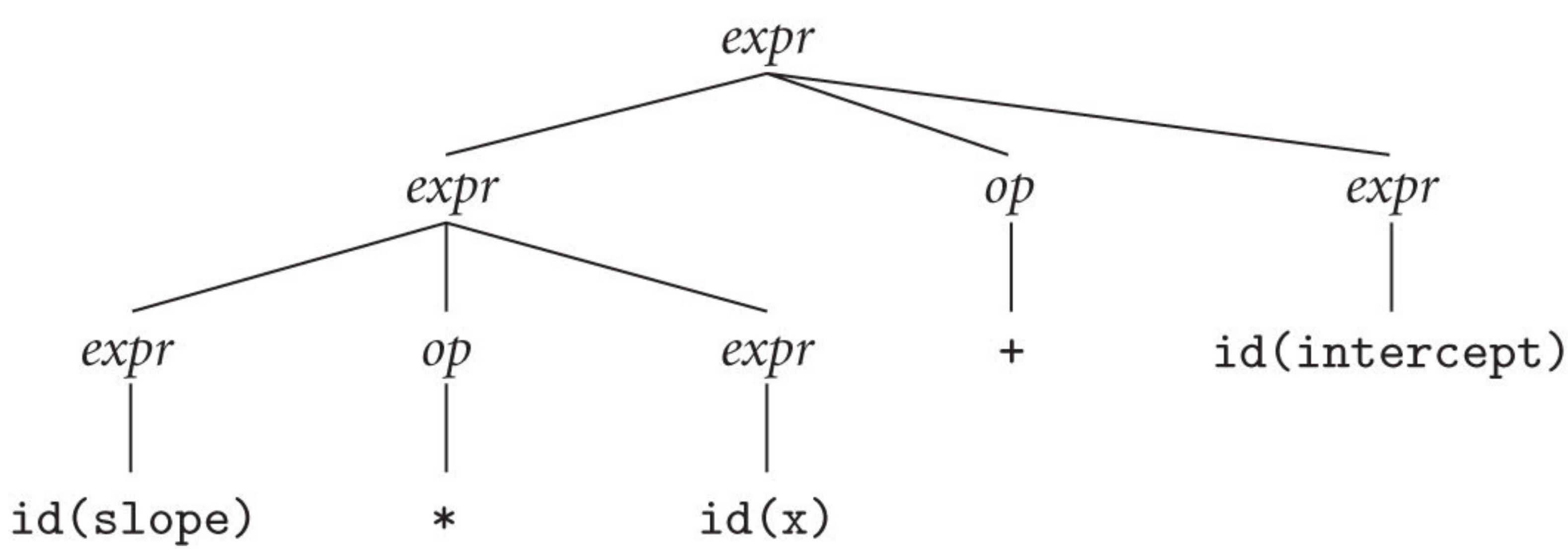
$$
\begin{aligned}
\underline{\mathit{expr}} &\Longrightarrow \underline{\mathit{expr}}\ \mathit{op}\ \mathit{expr} \\
&\Longrightarrow \underline{\mathit{expr}}\ \mathit{op}\ \mathit{expr}\ \mathit{op}\ \mathit{expr} \\
&\Longrightarrow \texttt{id}\ \underline{\mathit{op}}\ \mathit{expr}\ \mathit{op}\ \mathit{expr} \\
&\Longrightarrow \texttt{id *}\ \underline{\mathit{expr}}\ \mathit{op}\ \mathit{expr} \\
&\Longrightarrow \texttt{id * id}\ \underline{\mathit{op}}\ \mathit{expr} \\
&\Longrightarrow \texttt{id * id +}\ \underline{\mathit{expr}} \\
&\Longrightarrow \texttt{id * id + id}
\end{aligned}
$$

Context-Free Grammars

Parse Tree

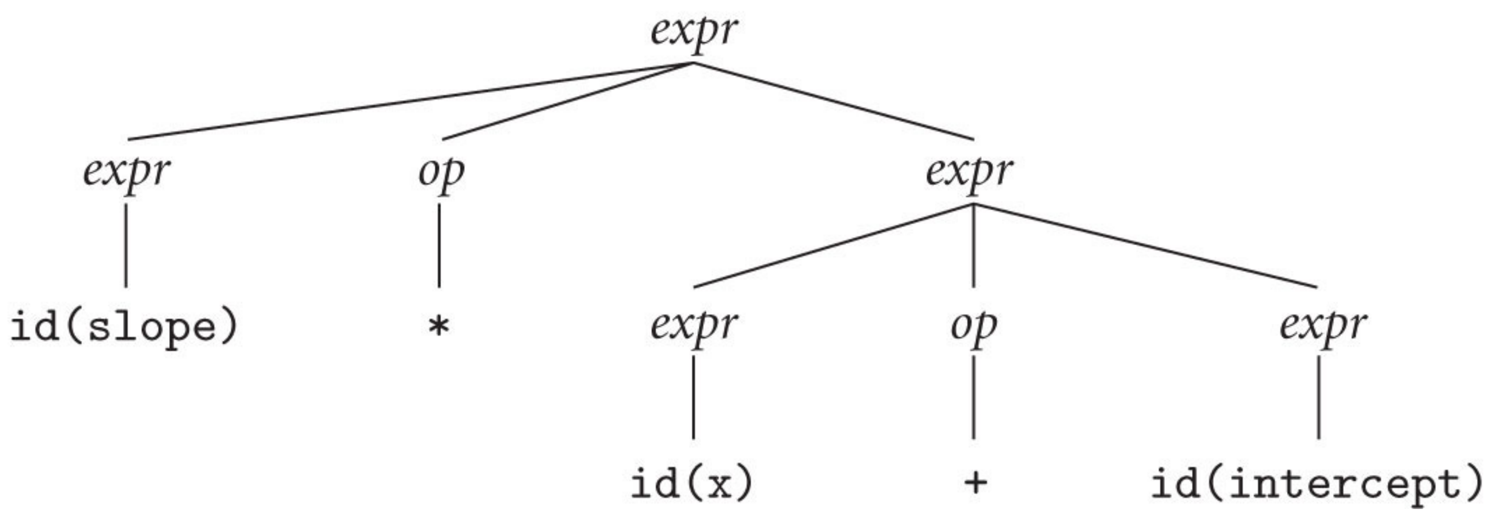
- Represents a derivation graphically
- Example: Parse tree for the string:

```
slope * x + intercept
```



Context-Free Grammars

- Different parse tree for: `slope * x + intercept`
- Tree allowed by the grammar but incorrect for the expression



- *Ambiguous* grammar: two different parse trees for one string
  - Ambiguity is a problem for parsers
  - We want unambiguous grammars

Context-Free Grammars

- Better version – unambiguous
- Captures *associativity* and *precedence*

$$expr \to term \mid expr\ add\_op\ term$$

$$term \to factor \mid term\ mult\_op\ factor$$

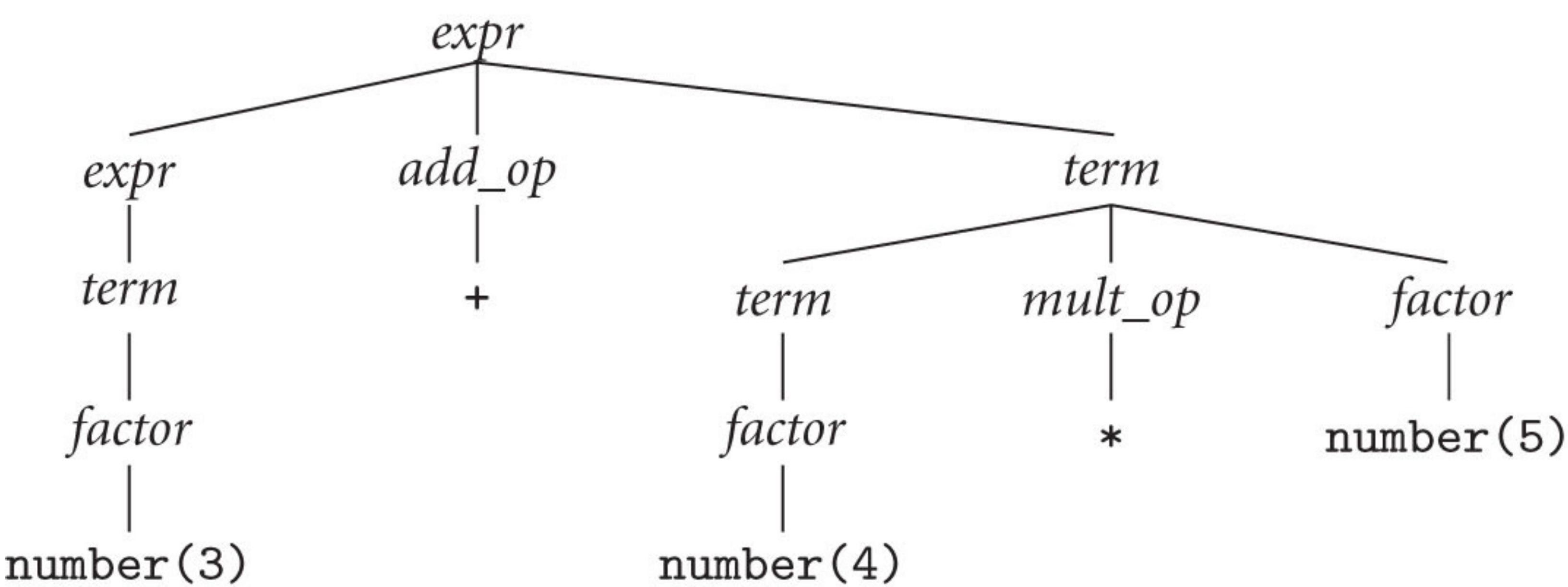
$$factor \to \texttt{id} \mid \texttt{number} \mid \texttt{-}\ factor \mid \texttt{(}\ expr\ \texttt{)}$$

$$add\_op \to \texttt{+} \mid \texttt{-}$$

$$mult\_op \to \texttt{*} \mid \texttt{/}$$

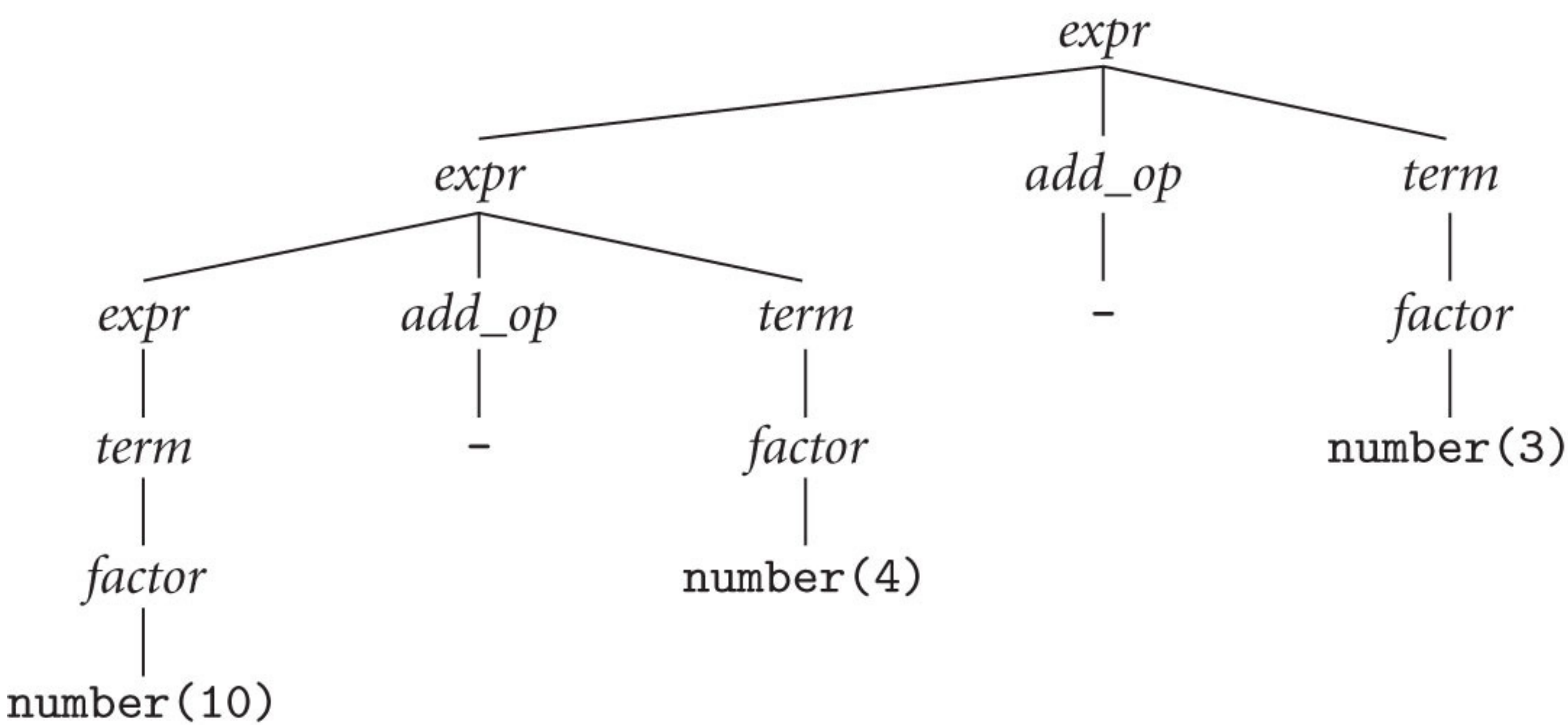
Context-Free Grammars

- Parse tree for: `3 + 4 * 5`
  - Precedence rules



Context-Free Grammars

- Parse tree for: `10 - 4 - 3`
  - Left-associativity rules



Scanning

Scanning = Lexical Analysis

- tokenizing source
- removing comments
- saving text of identifiers, numbers, strings
- saving source locations (file, line, column) for error messages

Scanning

- Example: simple calculator language

$assign \rightarrow$ `:=` (Algol style; C has ‘`=`’)
$plus \rightarrow$ `+`
$minus \rightarrow$ `-`
$times \rightarrow$ `*`
$div \rightarrow$ `/`
$lparen \rightarrow$ `(`
$rparen \rightarrow$ `)`
$id \rightarrow letter\ (\ letter \mid digit\ )^*$ (except for `read` and `write`)
$number \rightarrow digit\ digit^* \mid digit^*\ (\ \texttt{.}\ digit \mid digit\ \texttt{.}\ )\ digit^*$
$comment \rightarrow \texttt{/*}\ (\ non\text{-}\texttt{*} \mid \texttt{*}\ non\text{-}\texttt{/}\ )^*\ \texttt{*}^+\texttt{/}$
$\mid \texttt{//}\ (non\text{-}newline)^*\ newline$

Scanning

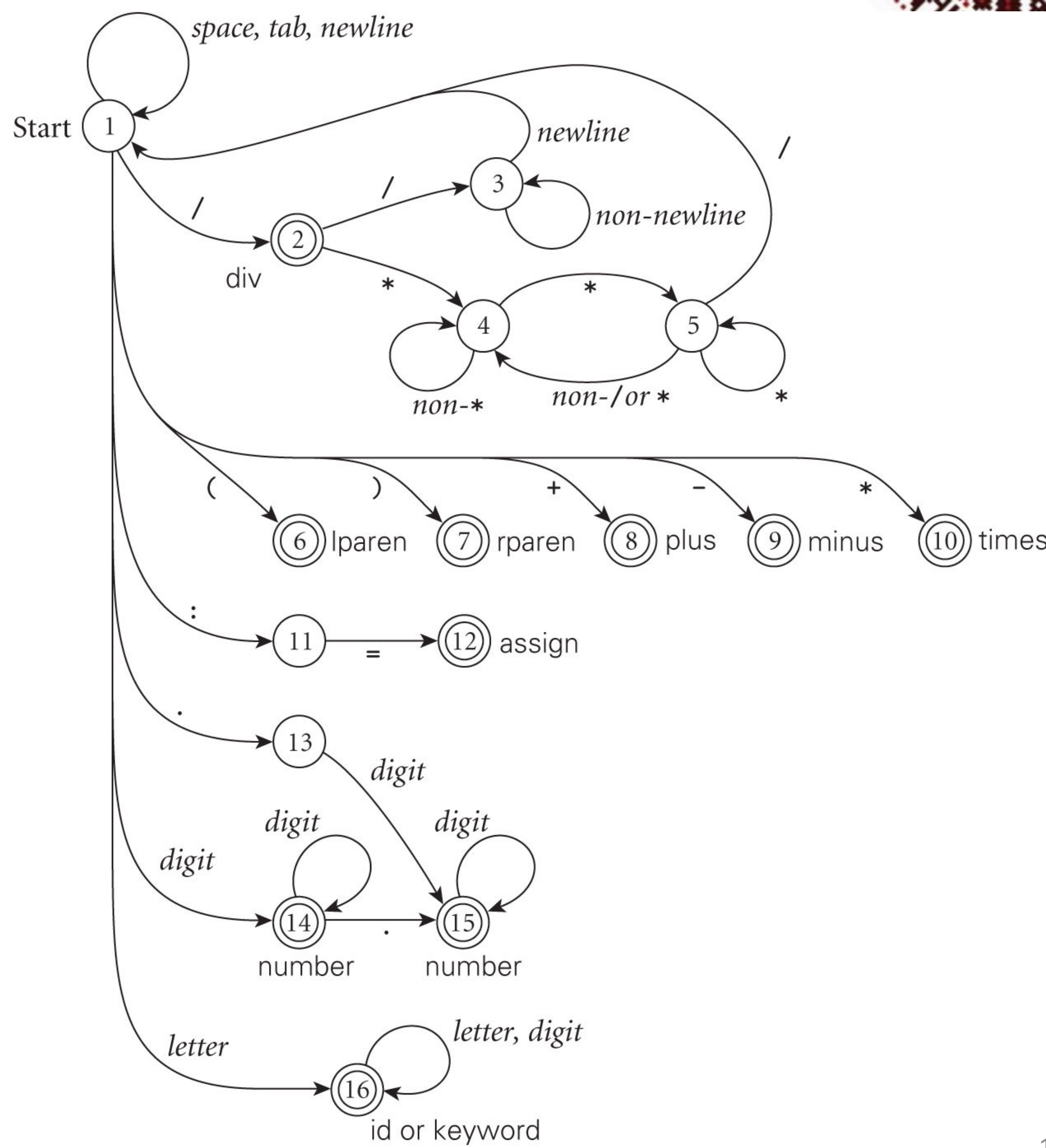
Ad-hoc scanner

- Longest possible token extracted
- White spaces are delimiters

```
skip any initial white space (spaces, tabs, and newlines)
if cur_char ∈ {'(', ')', '+', '-', '*'}
    return the corresponding single-character token
if cur_char = ':'
    read the next character
    if it is '=' then return assign else announce an error
if cur_char = '/'
    peek at the next character
    if it is '*' or '/'
        read additional characters until "*/" or newline is seen, respectively
        jump back to top of code
    else return div
if cur_char = .
    read the next character
    if it is a digit
        read any additional digits
        return number
    else announce an error
if cur_char is a digit
    read any additional digits and at most one decimal point
    return number
if cur_char is a letter
    read any additional letters and digits
    check to see whether the resulting string is read or write
    if so then return the corresponding token
    else return id
else announce an error
```

Scanning

- Structured scanner
- DFA – Deterministic Finite Automaton
- Separate final state for each token category

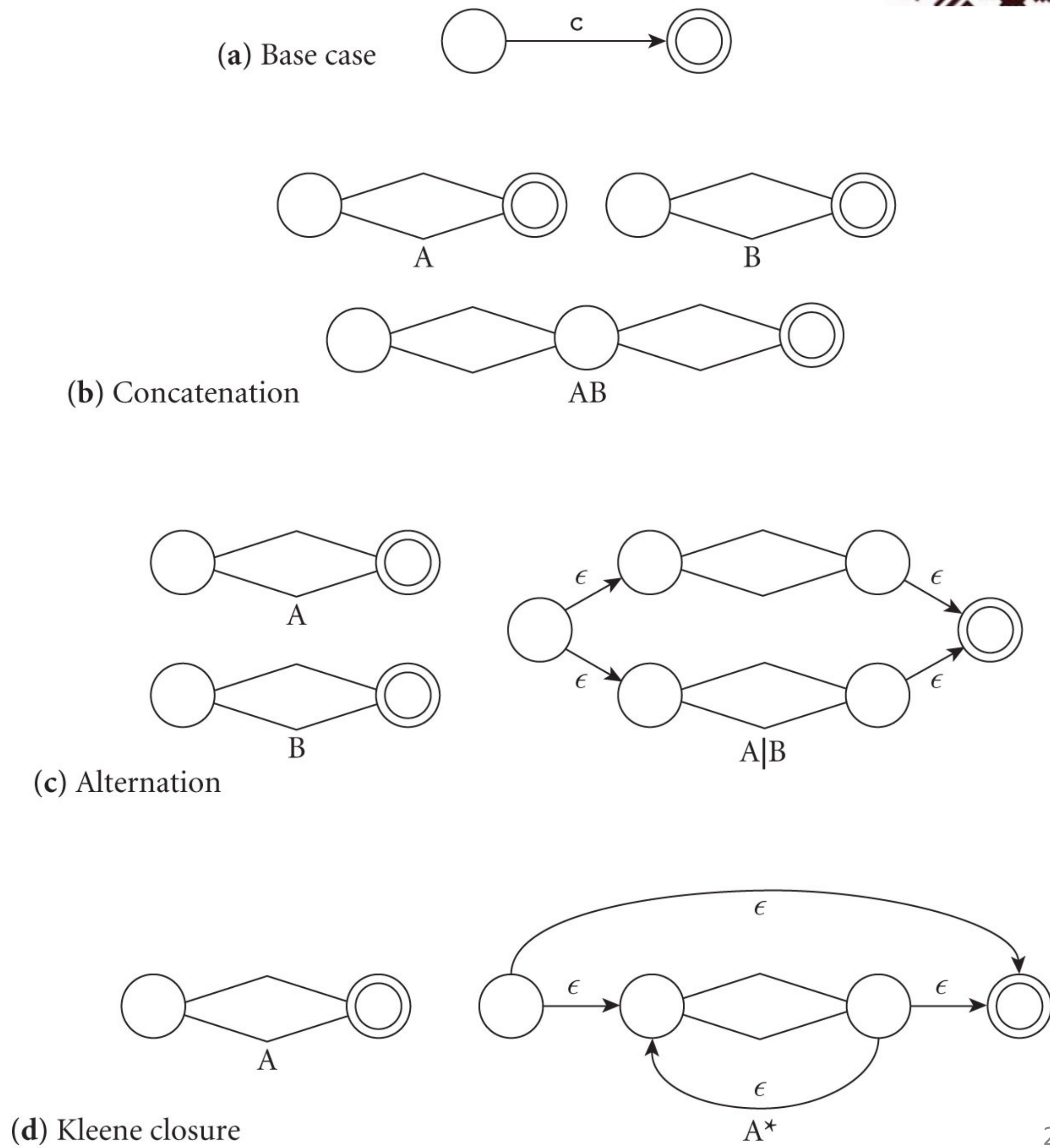


Scanning

- DFA
  - Built automatically from regular expressions
  - Tools: `lex`, `flex`, `scagen`
  - Difficult to build directly
    - build first an NFA – Nondeterministic FA
    - convert to DFA
    - minimize DFA (smallest number of states)

Scanning

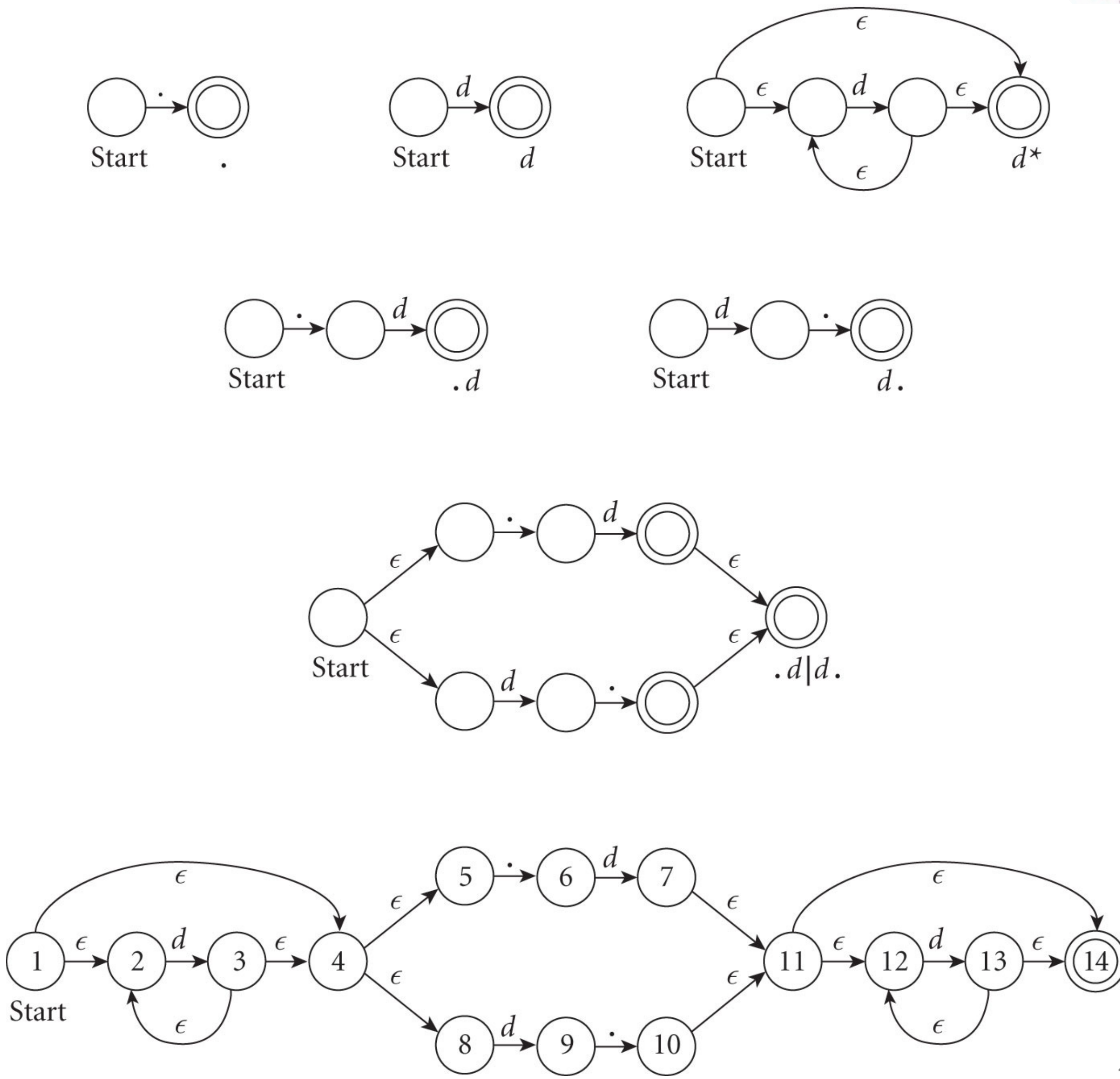
- Reg.exp. to NFA
- Follows the structural definition of regular expressions



Scanning

- Reg.exp. to NFA
- Example:

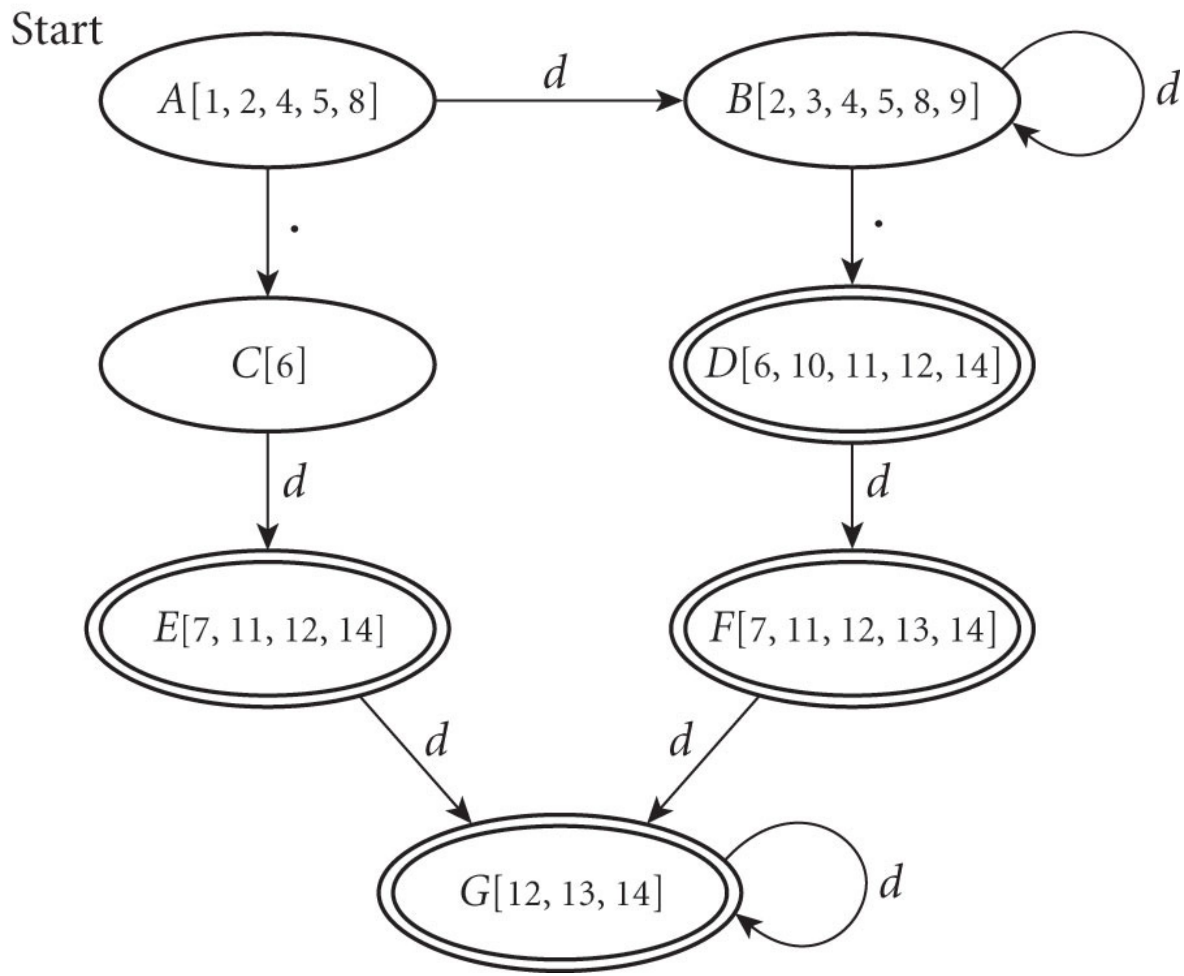
$d^* (\, . \, d \mid d \, . \,) \, d^*$



Scanning

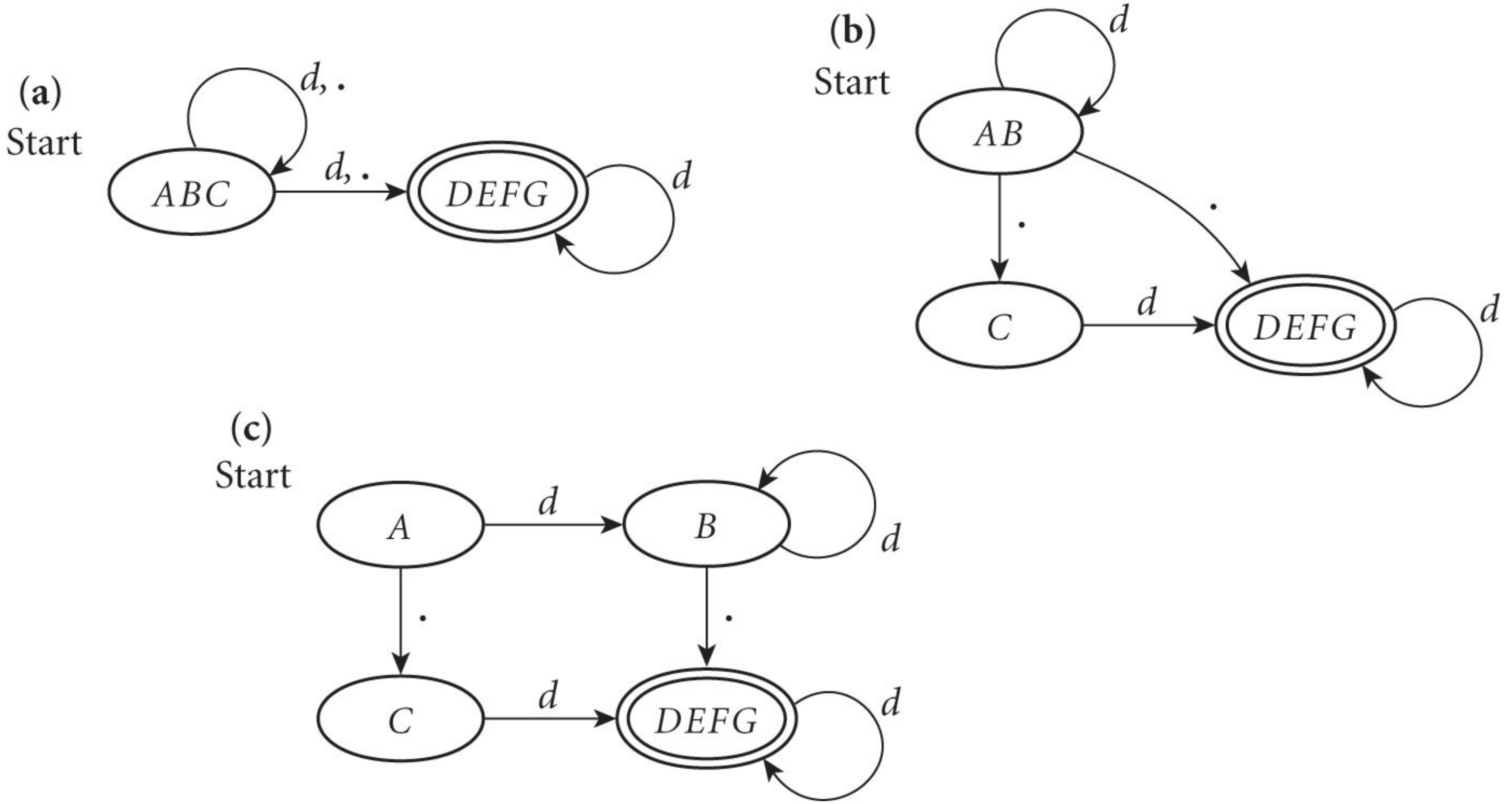
- NFA to DFA
- Example:

$d^* (. d | d .) d^*$



Scanning

- DFA minimization
- Example: $d^* (\, . \, d \mid d \, . \,) \, d^*$



Scanning

- Scanners are built three ways:
  - ad-hoc:
    - fastest, most compact code
  - semi-mechanical pure DFA
    - nested `case` statements
  - table-driven DFA
    - automatically-generated scanners
- "Longest-possible token" rule
  - return only when next character cannot continue current token
  - the next character needs to be saved for the next token
- Keywords
  - DFA would need many states to identify
  - Better treat keywords as exceptions to the identifier rule

Scanning

- Nested `case` statement DFA

```
state := 1
loop
   read cur_char
   case state of
      1: case cur_char of
            ' ', '\t', '\n':        ...
            'a' ... 'z':            ...
            '0' ... '9':            ...
             '>' :                  ...
             ...
      2: case cur_char of
             ...
      ⋮
      n: case cur_char of
             ...
```

Scanning

- Look-ahead
- May need to peek at more than one character
- *look-ahead* – characters necessary to decide
- Example: Pascal
  - have 3 so far and see '.'
  - 3.14 or 3..5 may follow
- Example: Fortran
  - arbitrarily long look-ahead
  - `DO 5 I = 1,25`
    - execute statements up to 5 for I from 1 to 25
  - `DO 5 I = 1.25`
    - assign 1.25 to the variable `DO5I`
  - NASA's Mariner 1 may have been lost due to '`.`' i.o '`,`'
  - Fortran 77 has better syntax: `DO 5,I = 1,25`

Scanning

- Table-driven scanning

```
state = 0 . . number_of_states
token = 0 . . number_of_tokens
scan_tab : array [char, state] of record
      action : (move, recognize, error)
      new_state : state
token_tab : array [state] of token          -- what to recognize
keyword_tab : set of record
      k_image : string
      k_token : token
-- these three tables are created by a scanner generator tool
```

(continued on next slide)

Scanning

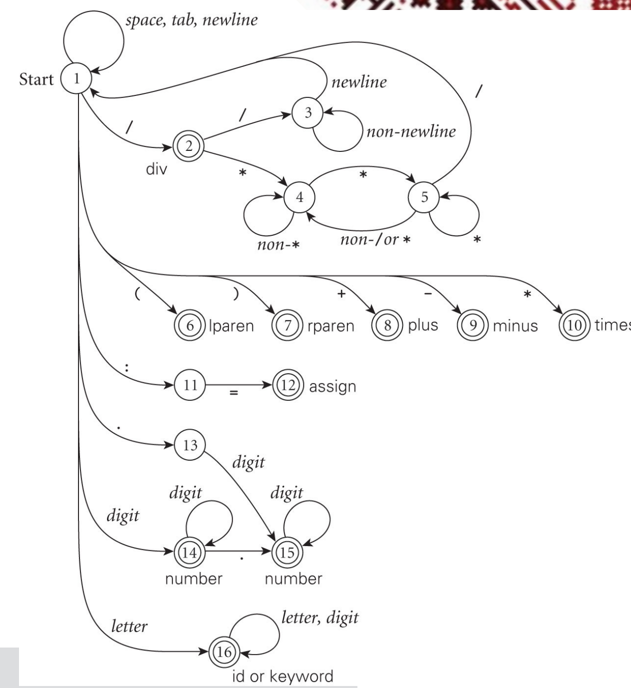
Table-driven scanning (cont'd)

```
state = 0 . . number_of_states
token = 0 . . number_of_tokens
scan_tab : array [char, state] of record
    action : (move, recognize, error)
    new_state : state
token_tab : array [state] of token
keyword_tab : set of record
    k_image : string
    k_token : token
```

```
tok : token
cur_char : char
remembered_chars : list of char
repeat
    cur_state : state := start_state
    image : string := null
    remembered_state : state := 0        -- none
    loop
        read cur_char
        case scan_tab[cur_char, cur_state].action
            move:
                if token_tab[cur_state] ≠ 0
                    -- this could be a final state
                    remembered_state := cur_state
                    remembered_chars := ϵ
                add cur_char to remembered_chars
                cur_state := scan_tab[cur_char, cur_state].new_state
            recognize:
                tok := token_tab[cur_state]
                unread cur_char        -- push back into input stream
                exit inner loop
            error:
                if remembered_state ≠ 0
                    tok := token_tab[remembered_state]
                    unread remembered_chars
                    remove remembered_chars from image
                    exit inner loop
                -- else print error message and recover; probably start over
        append cur_char to image
    -- end inner loop
until tok ∉ {white_space, comment}
look image up in keyword_tab and replace tok with appropriate keyword if found
return ⟨tok, image⟩
```

Scanning

- Scanner table used by previous code
  - state 17: white spaces; state 18: comments
  - `scan_tab`: entire table but last column
  - `token_tab`: last column
  - `keyword_tab` = {**`read`**, **`write`**}



State	*space, tab*	*newline*	/	∗	(	)	+	−	:	=	.	*digit*	*letter*	*other*	
1	17	17	2	10	6	7	8	9	11	–	13	14	16	–	
2	–	–	3	4	–	–	–	–	–	–	–	–	–	–	div
3	3	18	3	3	3	3	3	3	3	3	3	3	3	3	
4	4	4	4	5	4	4	4	4	4	4	4	4	4	4	
5	4	4	18	5	4	4	4	4	4	4	4	4	4	4	
6	–	–	–	–	–	–	–	–	–	–	–	–	–	–	lparen
7	–	–	–	–	–	–	–	–	–	–	–	–	–	–	rparen
8	–	–	–	–	–	–	–	–	–	–	–	–	–	–	plus
9	–	–	–	–	–	–	–	–	–	–	–	–	–	–	minus
10	–	–	–	–	–	–	–	–	–	–	–	–	–	–	times
11	–	–	–	–	–	–	–	–	–	12	–	–	–	–	
12	–	–	–	–	–	–	–	–	–	–	–	–	–	–	assign
13	–	–	–	–	–	–	–	–	–	–	–	15	–	–	
14	–	–	–	–	–	–	–	–	–	–	15	14	–	–	number
15	–	–	–	–	–	–	–	–	–	–	–	15	–	–	number
16	–	–	–	–	–	–	–	–	–	–	–	16	16	–	identifier
17	17	17	–	–	–	–	–	–	–	–	–	–	–	–	white_space
18	–	–	–	–	–	–	–	–	–	–	–	–	–	–	comment

Scanning

- Lexical errors
- Very few – most strings correspond to some token
- Should recover to enable the compiler to detect more errors
  - throw away the current, invalid, token
  - skip forward to the next possible beginning of a new token
  - restart the scanning algorithm
  - count on the error-recovery mechanism of the parser to cope with a syntactically invalid sequence of tokens