

- Programming Assingment #1 is out
- read Chapters 1 and 2
- may want to write a warm-up problem (e.g., read in integers into binary tree, traverse tree and output ints in increasing order)
- work in pairs (alternating with assignments is not a good idea!)

## Lexical Analysis

- lexical analysis: breaking up the input into individual words = tokens
- ```
begin print ( "HelloWorld" ) end /* output */
```
- token: sequence of characters treated as a unit in the grammar of the language
    - begin, print, end, (, ), identifier "HelloWorld"
    - some tokens have semantic values
      - (e.g., identifiers and literals)
    - not tokens: comments, white spaces (blank, tab, newline), preprocessor directives

## Regular Expressions

- most appropriate for describing tokens
- finite descriptions for specifying possibly infinite languages
- core operations:
  - symbol a
  - alternation  $M | N$  (e.g.  $a | b = \{a, b\}$ )
  - concatenation  $M \cdot N$  (e.g.,  $(a | b) \cdot a = \{aa, ba\}$ )
  - epsilon  $\epsilon$  (e.g.,  $(a \cdot b) | \epsilon = \{\epsilon, "ab"\}$ )
  - repetition  $M^*$  (e.g.,  $((a|b) \cdot a)^* = \{\epsilon, "aa", "ba", "aaaa", "baaa", \dots\}$ )

- abbreviations for convenience:
- $[abcd] = (a | b | c | d)$   $[a-kM-Q0-9]=\dots$
- $a \cdot b = ab$   $\cdot =$  any char except newline
- $M^+ = MM^*$   $\dots$  string
- $M? = M | \epsilon$

```

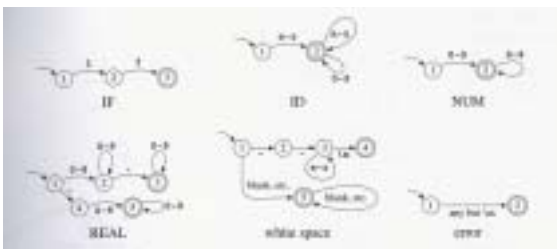
if                                     (IF);
[a-z][a-z0-9]*                         (ID);
[0-9]+                                  (NUM);
([0-9]+|"."[0.9]*)|([0-9]*"."[0.9]+) (REAL);
("--"[a-z]*"\n")|(" "|\n"|\t")+ (continue());
.                                       (error(); continue());

```

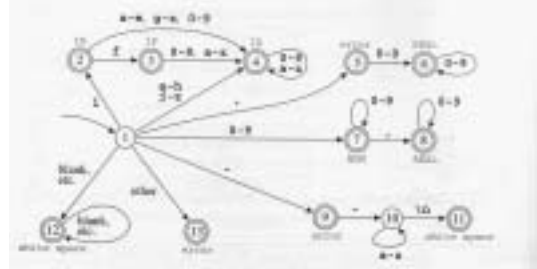
- Complete specification – handle errors.
- Longest match (greedy)
- Rule Priority (for the longest match, first rule in spec).
- cannot use regexp to parse everything!
  - (e.g., HTML/XML better of with parser modules)

## Finite Automata

- if regexp are the language, finite automata are the machine
- finite set of states (including start and final states);
- edges which lead from one state to another;
- edges labeled with a symbol.



- Deterministic Finite Automata – no two edges leaving from the same state are labeled with the same symbol
- alphabet, set of states, starting state, transition matrix, set of final states



- need to combine automata for separate regexps into a single machine – the lexer.
  - for now ad-hoc
- many possible final states – labeled with token type.
- Translation matrix – state\_in[] x input\_chars -> state\_out[]
- Finality array – map final states into actions

- Track progress through input by counting position
- To recognize the longest match additional state needed:
  - variable Last-Final – state number of most recent final state
  - variable Input-Position-at-Last-Final – so that we can roll back

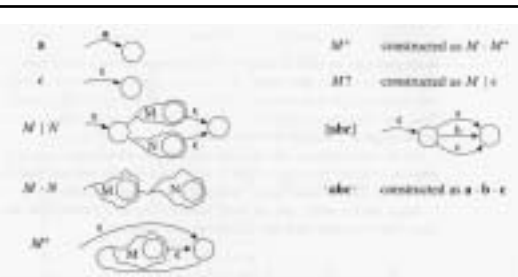
e.g.,

```

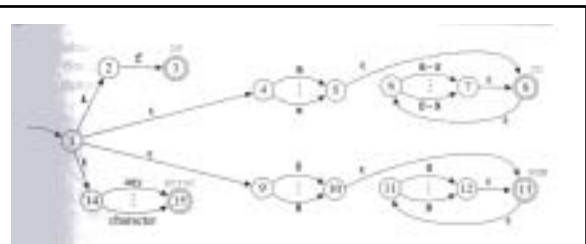
if --not-a-comment
return IF, discard white space, try to do longest match on
the comment, but error ('-' is illegal within the
comment)
  
```

## Non-Deterministic FA

- have a choice of edges labeled with the same symbol from the same state
- may have special edges labeled with  $\epsilon$  –
  - can transition into a new state without consuming the input
- why? – easy target for regular expressions



- tail (start edge) and a head (ending state)



- states IF, ID, NUM, error
- how do we tell machine how to 'guess' which edge should it take?
- regexp  $\Rightarrow$  NFA, NFA  $\Rightarrow$  DFA
  - (also NFA  $\Rightarrow$  regexp)

- with NFA – need to guess correctly in order to recognize correct token (longest, priority...)
  - but NFAs are still well suited for regexps
- instead of guessing – explore all possibilities at once!
- make a DFA out of NFA – sets of possible states which can be reached along the way in the NFA will correspond to a single state in the new DFA

### $\epsilon$ -Closure of {set of states}

- $\epsilon$ -Closure(S) – set of states that can be reached from a state  $s$  in  $S$  without consuming any of the input (just going through  $\epsilon$  edges)
  - starting state in figure – (1,4,9,14)

$\epsilon$ Closure(S) = all states already in S +  
for each state in S – all states that can be reached through  $\epsilon$  edges from s

### DFAedge(d,c)

- d – initial set of states (from NFA)
- c – input char
- DFAedge(d,c) – set of states that can be reached from all the states in d through c or  $\epsilon$  edges

DFAedge(d,c) = closure(U edge(s,c)) for each s in d

- NFA  $\Rightarrow$  DFA by making each set of states in NFA correspond to a state in DFA

- In the new DFA
  - start state d1 is closure of (s1)
  - for each symbol in alphabet: DFAedge
  - ...
- Final state in new DFA
  - any of NFA states final
  - must be marked by token type
  - if more than one token recognized – mark with one that occurs first in spec.



## Implementing a lexer

- Large % part of compiler cycles are used at front-end – reading input and discovering tokens
- Lexer construction - by hand or automatic
- Reading the input
  - char by char (variable length tokens)
  - buffer (end of buffer test)
  - end-of-line/newline character
    - representation OS dependent
    - end of last line
  - entire file in memory

- Interpreting the input
  - precompute tables for function values for ASCII chars
  - then index into table
  - issue: are values 0-256 or -128 – 127
  - can merge tables for multiple functions
- Searching for longest match – repeated traversals of input
  - token1 - a
  - token2 - a\*b
- Translation table – large and sparse
  - compression methods
  - need to be able to access Next-State efficiently

- Error-handling:
  - on unmatched char – discard or pass to parser so that it knows that something was there?
  - specify regexps for typical errors
  - include some error recovery?