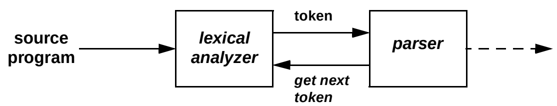
Lexical Analysis

- *Read source program and produce a list of **tokens** ("linear" analysis)*



- *The lexical structure is specified using **regular expressions***
- *Other secondary tasks:*

  (1) get rid of white spaces (e.g., `\t`, `\n`, `\sp`) and comments

  (2) line numbering

Example: Source Code

A Sample Toy Program:

```
(* define valid mutually recursive procedures *)
let

function do_nothing1(a: int, b: string)=
                do_nothing2(a+1)

function do_nothing2(d: int) =
                do_nothing1(d, "str")

in
        do_nothing1(0, "str2")
end
```

What do we really care here ?

The Lexical Structure

Output after the Lexical Analysis ----- <u>token + associated value</u>

LET 51	**FUNCTION** 56	**ID**(do_nothing1) 65
LPAREN 76	**ID**(a) 77	**COLON** 78
ID(int) 80	**COMMA** 83	**ID**(b) 85
COLON 86	**ID**(string) 88	**RPAREN** 94
EQ 95	**ID**(do_nothing2) 99	
LPAREN 110	**ID**(a) 111	**PLUS** 112
INT(1) 113	**RPAREN** 114	**FUNCTION** 117
ID(do_nothing2) 126		**LPAREN** 137
ID(d) 138	**COLON** 139	**ID**(int) 141
RPAREN 144	**EQ** 146	
ID(do_nothing1) 150		**LPAREN** 161
ID(d) 162	**COMMA** 163	**STRING**(str) 165
RPAREN 170	**IN** 173	
ID(do_nothing1) 177		**LPAREN** 188
INT(0) 189	**COMMA** 190	**STRING**(str2) 192
RPAREN 198	**END** 200	**EOF** 203

Tokens

- ***Tokens*** *are the <u>atomic unit</u> of a language, and are usually <u>specific strings</u> or <u>instances</u> of <u>classes</u> of strings.*

Tokens	Sample Values	Informal Description
LET	`let`	***keyword* LET**
END	`end`	***keyword* END**
PLUS	`+`	
LPAREN	`(`	
COLON	`:`	
STRING	`"str"`	
RPAREN	`)`	
INT	`49, 48`	***integer constants***
ID	`do_nothing1, a, int, string`	***letter followed by letters, digits, and under-scores***
EQ	`=`	
EOF		***end of file***

Lexical Analysis, How?

- *First, write down the **lexical specification** (how each token is defined?)*

  using **regular expression** to specify the lexical structure:

```
identifier = letter (letter | digit | underscore)*
letter = a | ... | z | A | ... | Z
digit = 0 | 1 | ... | 9
```

- *Second, based on the above **lexical specification**, build the lexical analyzer (to recognize tokens) by hand,*

  Regular Expression Spec ==> NFA ==> DFA ==>Transition Table ==> Lexical Analyzer

- *Or just by using **lex** --- the lexical analyzer generator*

  Regular Expression Spec (in ***lex*** format) ==> feed to ***lex*** ==> Lexical Analyzer

Regular Expressions

- ***regular expressions** are concise, linguistic characterization of **regular languages** (regular sets)*

```
identifier = letter (letter | digit | underscore)*
```

"or" ***" 0 or more"***

- ***each regular expression** define a regular language --- a <u>set of strings</u> over some alphabet, such as ASCII characters; each member of this set is called a **sentence,** or a **word***

- *we use regular expressions to define each category of tokens*

  For example, the above `identifier` specifies a set of strings that are a sequence of letters, digits, and underscores, starting with a letter.

Regular Expressions and Regular Languages

- *Given an alphabet Σ, the **regular expressions** over Σ and their corresponding **regular languages** are*

  a) $\emptyset$ denotes $\emptyset$; ε ,the empty string, denotes the language $\{ \varepsilon \}$.

  b) for each a in Σ, a denotes { a } --- a language with one string.

  c) if *R* denotes L_R and *S* denotes L_S then *R* | *S* denotes the language $L_R \cup L_S$, i.e, $\{ x \mid x \in L_R \text{ or } x \in L_S \}$.

  d) if *R* denotes L_R and *S* denotes L_S then *RS* denotes the language $L_R L_S$, that is, $\{ xy \mid x \in L_R \text{ and } y \in L_S \}$.

  e) if *R* denotes L_R then R^* denotes the language L_R^* where L^* is the union of all L^i (i=0,...,∞) and L^i is just $\{x_1 x_2 ... x_i \mid x_1 \in L, ..., x_i \in L\}$.

  f) if *R* denotes L_R then *(R)* denotes the same language L_R.

Example

Regular Expression	Explanation
`a`*	*0 or more a's*
`a`+	*1 or more a's*
`(a|b)`*	*all strings of* a*'s and* b*'s (including ε)*
`(aa|ab|ba|bb)`*	*all strings of* a*'s and* b*'s of even length*
`[a-zA-Z]`	*shorthand for* `"a|b|...|z|A|...|Z"`
`[0-9]`	*shorthand for* `"0|1|2|...|9"`
`0([0-9])*0`	*numbers that start and end with 0*
`(ab|aab|b)*(a|aa|ε)`	*?*
?	*all strings that contain* `foo` *as substring*

- *the following is **not** a regular expression:* $a^n b^n \quad (n > 0)$

Lexical Specification

- *Using **regular expressions** to specify **tokens***

```
keyword = begin | end | if | then | else
identifier = letter (letter | digit | underscore)*
integer = digit+
relop = < | <= | = | <> | > | >=
letter = a | b | ... | z | A | B | ... | Z
digit = 0 | 1 | 2 | ... | 9
```

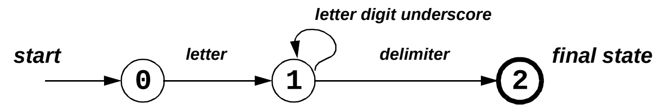
- ***Ambiguity** : is "`begin`" a keyword or an identifier ?*
- ***Next step:** to construct a token recognizer for languages given by regular expressions --- by using **finite automata** !*

given a string *x*, the token recognizer says "yes" if *x* is a sentence of the specified language and says "no" otherwise

CS421 COMPILERS AND INTERPRETERS

Transition Diagrams

- *Flowchart with **states** and **edges;** each edge is labelled with characters; certain subset of states are marked as "**final states**"*
- *Transition from state to state proceeds along edges according to the next **input character***



- *Every string that ends up at a **final state** is accepted*
- *If get "stuck", there is no transition for a given character, it is an error*
- *Transition diagrams can be easily translated to programs using **case** statements (in C).*

CS421 COMPILERS AND INTERPRETERS

Transition Diagrams (cont'd)

The token recognizer (for identifiers) based on transition diagrams:

```
state0:   c = getchar();
          if (isalpha(c)) goto state1;
          error();
          ...
state1:   c = getchar();
          if (isalpha(c) || isdigit(c) ||
                 isunderscore(c)) goto state1;
          if (c == ‘,’ || ... || c == ‘)’) goto state2;
          error();
          ...
state2:   ungetc(c,stdin); /* retract current char */
          return(ID, ... the current identifier ...);
```

Next: ***1. finite automata are generalized transition diagrams !***
2. how to build finite automata from regular expressions?

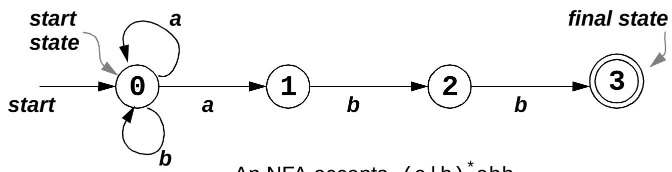
CS421 COMPILERS AND INTERPRETERS

Finite Automata

- ***Finite Automata** are similar to transition diagrams; they have **states** and **labelled edges**; there are one unique **start state** and one or more than one **final states***
- ***Nondeterministic Finite Automata (NFA) :***
  - *a) ε can label edges (these edges are called **ε-transitions**)*
  - *b) some character can label 2 or more edges out of the same state*
- ***Deterministic Finite Automata (DFA) :***
  - *a) no edges are labelled with ε*
  - *b) each charcter can label at most **one** edge out of the same state*
- ***NFA** and **DFA** accepts string **x** if there exists a path from the start state to a final state labeled with characters in **x***

NFA: multiple paths   **DFA:** one unique path

Example: NFA



An NFA accepts `(a|b)*abb`

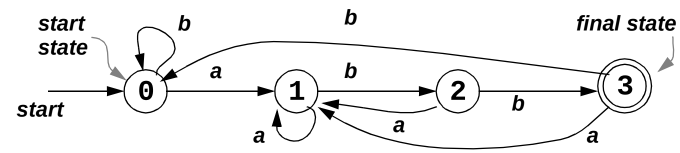
There are many possible moves --- to accept a string, we only need one sequence of moves that lead to a final state.

input string: `aabb`

One sucessful sequence: $0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$

Another unsuccessful sequence: $0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$

Example: DFA



A DFA accepts `(a|b)*abb`

There is only one possible sequence of moves --- either lead to a final state and accept or the input string is rejected

input string: `aabb`

The sucessful sequence: $0 \xrightarrow{a} 1 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$

Transition Table

- *Finite Automata can also be represented using* ***transition tables***

For NFA, each entry is a set of states:

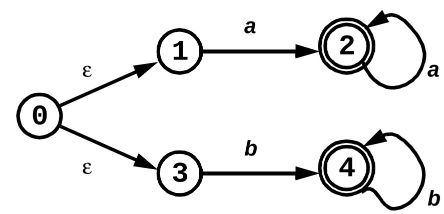
STATE	a	b
0	{0,1}	{0}
1	-	{2}
2	-	{3}
3	-	-

For DFA, each entry is a unique state:

STATE	a	b
0	1	0
1	1	2
2	1	3
3	1	0

NFA with ε-transitions

1. NFA can have ε-transitions --- edges labelled with ε



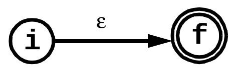
accepts the regular language denoted by `(aa*|bb*)`

Regular Expressions -> NFA

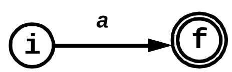
- *How to construct NFA (with ε-transitions) from a regular expression ?*
- ***Algorithm** : apply the following **construction rules** , use unique names for all the states. (**important invariant**: always **one final state !**)*

1. Basic Construction

- ε



- $a \in \Sigma$

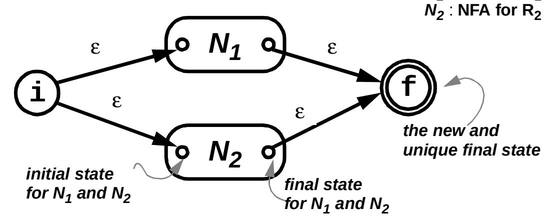


RE -> NFA (cont'd)

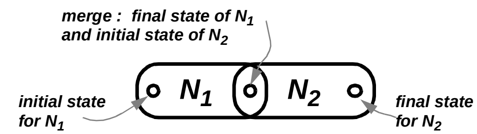
2. "Inductive" Construction

N_1 : NFA for R_1
N_2 : NFA for R_2

- $R_1 \mid R_2$



- $R_1 R_2$

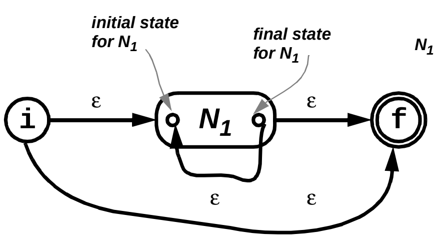


RE -> NFA (cont'd)

2. "Inductive" Construction (cont'd)

- R_1^*

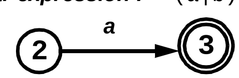
N_1 : NFA for R_1



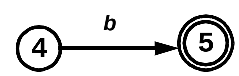
Example : RE -> NFA

Converting the regular expression : `(a|b)*abb`

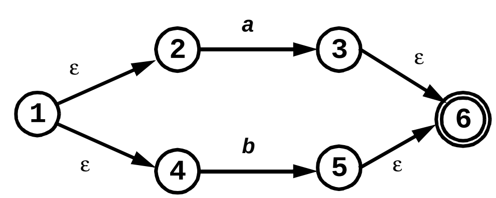
`a (in a|b)===>`



`b (in a|b)===>`



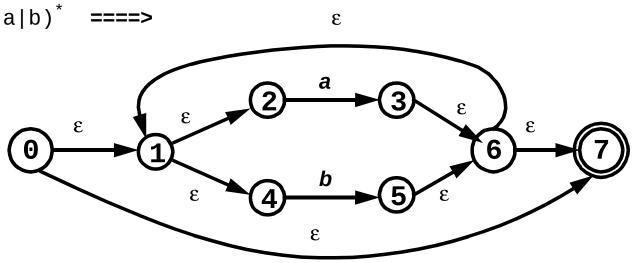
`a|b` **====>**



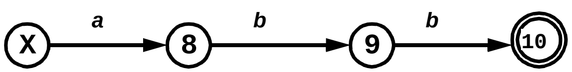
Example : RE -> NFA (cont'd)

Converting the regular expression : `(a|b)*abb`

`(a|b)*` **====>**



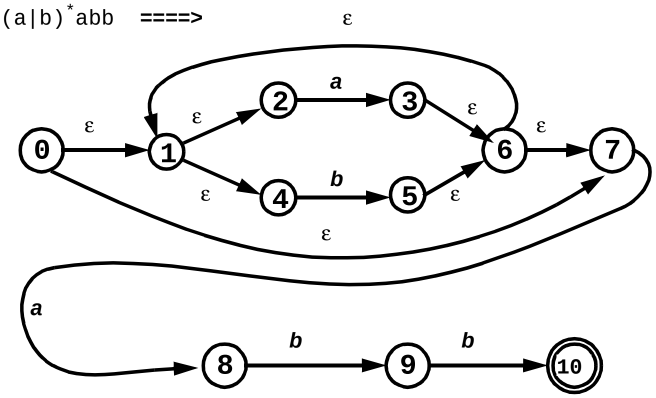
`abb` **====>** (several steps are omitted)



Example : RE -> NFA (cont'd)

Converting the regular expression : `(a|b)*abb`

`(a|b)*abb` **====>**



NFA -> DFA

- *NFA are non-deterministic; need DFA in order to write a deterministic prorgam !*
- *There exists an algorithm ("subset construction") to convert any NFA to a DFA that accepts the same language*
- *States in DFA are **sets of states** from NFA; DFA simulates "in parallel" all possible moves of NFA on given input.*
- ***Definition:** for each state **s** in NFA,*

  ε-CLOSURE(**s**) = { **s** } ∪ { **t** | **s** can reach **t** via ε-transitions }
- ***Definition:** for each set of states **S** in NFA,*

  ε-CLOSURE(**S**) = $\cup_i$ ε-CLOSURE(**s**) for all $\mathbf{s}_i$ in **S**

NFA -> DFA (cont'd)

- *each DFA-state is a **set** of NFA-states*
- *suppose the **start state** of the NFA is **s,** then the **start state** for its DFA is* ε-CLOSURE(s) ; *the **final states** of the DFA are those that include a **NFA-final-state***
- ***Algorithm** : converting an NFA **N** into a DFA **D ----***

```
Dstates = {ε-CLOSURE(s0),s0 is N's start state}
Dstates are initially "unmarked"
while there is an unmarked D-state X do {
   mark X
   for each a ∈ Σ do {
    T = {states reached from any si in X via a}
    Y = ε-CLOSURE(T)
    if Y ∉ Dstates then add Y to Dstates "unmarked"
    add transition from X to Y, labelled with a
   }
}
```

Example : NFA -> DFA

- *converting NFA for* `(a|b)*abb` *to a DFA* -------------

The start state A = ε-CLOSURE(0) = {0, 1, 2, 4, 7}; **Dstates**={A}

1st iteration: A is unmarked; mark A now;
a-transitions: T = {3, 8}
a new state B= ε-CLOSURE(3) ∪ ε-CLOSURE(8)
= {3, 6, 1, 2, 4, 7} ∪ {8} = {1, 2, 3, 4, 6, 7, 8}
add a transition from A to B labelled with a

b-transitions: T = {5}
a new state C = ε-CLOSURE(5) = {1, 2, 4, 5, 6, 7}
add a transition from A to C labelled with b
Dstates = {A, B, C}

2nd iteration: B, C are unmarked; we pick B and mark B first;
B = {1, 2, 3, 4, 6, 7, 8}
B's a-transitions: T = {3, 8}; T's ε-CLOSURE is B itself.
add a transition from B to B labelled with a

Example : NFA -> DFA (cont'd)

B's b-transitions: T = {5, 9};
a new state D = ε-CLOSURE({5, 9}) = {1, 2, 4, 5, 6, 7, 9}
add a transition from B to D labelled with b
Dstates = {A, B, C, D}

then we pick C, and mark C
C's a-transitions: T = {3, 8}; its ε-CLOSURE is B.
add a transition from C to B labelled with a
C's b-transitions: T = {5}; its ε-CLOSURE is C itself.
add a transition from C to C labelled with b

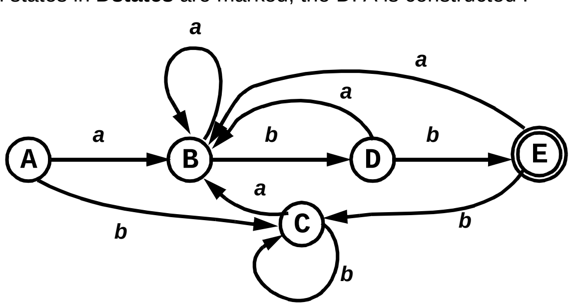
next we pick D, and mark D
D's a-transitions: T = {3, 8}; its ε-CLOSURE is B.
add a transition from D to B labelled with a
D's b-transitions: T = {5, 10};
a new state E = ε-CLOSURE({5, 10}) = {1, 2, 4, 5, 6, 7, 10}
Dstates = {A, B, C, D, E}; E is a **final state** since it has 10;

next we pick E, and mark E

Example : NFA -> DFA (cont'd)

E's a-transitions: T = {3, 8}; its ε-CLOSURE is B.
add a transition from E to B labelled with a
E's b-transitions: T = {5}; its ε-CLOSURE is C itself.
add a transition from E to C labelled with b

all states in **Dstates** are marked, the DFA is constructed !

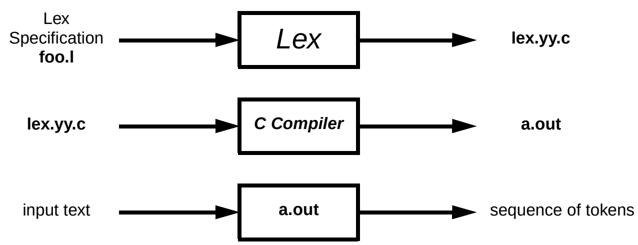


Other Algorithms

- ***How to minimize a DFA ? (see Dragon Book 3.9, pp141)***
- ***How to convert RE to DFA directly ? (see Dragon Book 3.9, pp135)***
- ***How to prove two Regular Expressions are equivalent ? (see Dragon Book pp150, Exercise 3.22)***

Lex

- ***Lex*** *is a program generator ---------- it takes* ***lexical specification*** *as input, and produces a* ***lexical processor*** *written in C.*



- ***Implementation of Lex****:*

  Lex Spec -> NFA -> DFA -> Transition Tables + Actions -> yylex()

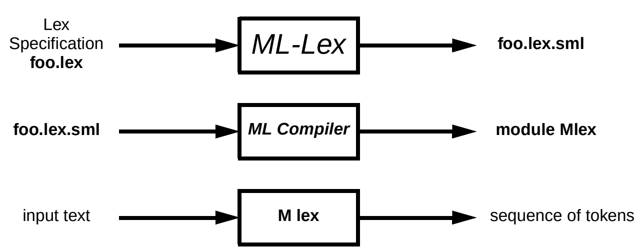
Lex Specification

```
DIGITS [0-9]                              } lex definition
......

%%
expression     action                     } translation rules
integer        printf("INT");

......
%%
.....                                     } user's C functions
char getc() { ......                         (optional)
}
```

- ***expression*** *is a regular expression ;* ***action*** *is a piece of C program;*
- *for details, read the* ***Lesk&Schmidt*** *paper*

ML-Lex

- ***ML-Lex*** *is like* ***Lex*** *---------- it takes* ***lexical specification*** *as input, and produces a* ***lexical processor*** *written in Standard ML.*



- *Implementation of* ***ML-Lex*** *is similar to implementation of* ***Lex***

ML-Lex Specification

```
type pos = int                            } user's ML declarations
val lineNum = ...
val lexresult = ....
....
%%
%s COMMENT STRING;                        } ml-lex definitions
SPACE=[ \t\n\012];
DIGITS=[0-9];
.....
%%
expression => (action);                   } translation rules
integer    => (print("INT"));               can call the above
......     => (...lineNum...);              ML declarations
```

- ***expression*** *is a regular expression ;* ***action*** *is a piece of ML program; when the input matches the* **`expression`***, the* **`action`** *is executed, the text matched is placed in the variable* **`yytext.`**

What does ML-Lex generate?

foo.lex → *ML-Lex* → foo.lex.sml

sample foo.lex.sml:

everything in part 1 of `foo.lex`

```
structure Mlex =
  struct
    structure UserDeclarations = struct ... end
    ......
    fun makeLexer yyinput = ....
  end
```

To use the generated lexical processor:

```
val lexer =
       Mlex.makeLexer(fn _ => input (openIn "toy"));
val nextToken = lexer()
```

each call returns one token !

input filename

ML-Lex Definitions

- *Things you can write inside the "**ml-lex definitions**" section (2nd part):*

`%s COMMENT STRING`	*define new **start states***
`%reject`	**`REJECT()`** *to reject a match*
`%count`	*count the line number*
`%structure {identifier}`	*the resulting structure name* (the default is `Mlex`)

(hint: you probably don't need use %reject, %count,or %structure for assignment 2.)

<u>Definition of named regular expressions :</u>

<u>identifier</u> = <u>regular</u> <u>expression</u>

```
SPACE=[ \t\n\012]
IDCHAR=[_a-zA-Z0-9]
```


ML-Lex Translation Rules

- *Each translation rule (3rd part) are in the form*

  `<start-state-list>` **regular expression** `=>` (**action**);

- *Valid ML-Lex regular expressions: (see ML-Lex-manual pp 4-6)*

  a character stands for itself except for the reserved chars:
  `? * + | ( ) ^ $ / ; . = < > [ { " \`
  to use these chars, use backslash! for example, `\\\"` represents the string `\"`

  using square brackets to enclose a set of characters
  (`\ - ^` are reserved)

`[abc]`	**char** a, **or** b, **or** c
`[^abc]`	**all chars except** a, b, c
`[a-z]`	**all chars from** a **to** z
`[\n\t\b]`	**new line, tab, or backspace**
`[-abc]`	**char** - **or** a **or** b **or** c

ML-Lex Translation Rules (cont'd)

- *Valid ML-Lex regular expressions: (cont'd)*

escape sequences: (can be used inside or outside square brackets)

`\b`	**backspace**
`\n`	**newline**
`\t`	**tab**
`\ddd`	**any ascii char (**`ddd` **is 3 digit decimal)**

`.`	any char except newline (equivalent to `[^\n]`)
`"x"`	match string x exactly even if it contains reserved chars
`x?`	an optional x
`x*`	0 or more x's
`x+`	1 or more x's
`x|y`	x or y
`^x`	if at the beginning, match at the beginning of a line only
`{x}`	substitute definition x (defined in the lex definition section)
`(x)`	same as regular expression x
`x{n}`	repeating x for n times
`x{m-n}`	repeating x from m to n times

ML-Lex Translation Rules (cont'd)

what are valid actions ?

- *Actions are basically ML code (with the following extensions)*
- *All actions in a lex file must return values of the same type*
- *Use* **`yytext`** *to refer to the current string*

```
[a-z]+ => (print yytext);
[0-9]{3} => (print (Char.ord(sub(yytext,0))));
```

- *Can refer to anything defined in the ML-Declaration section (1st part)*
- **`YYBEGIN`** <u>**`start-state`**</u> ----- *enter into another start state*
- `lex()` *and* `continue()` *to reinvoking the lexing function*
- `yypos` --- *refer to the current position*

Ambiguity

- *what if* ***more than one translation rules*** *matches ?*

  A. *<u>longest</u> match is preferred*
  B. *among rules which matched the same number of characters, the rule given <u>first</u> is preferred*

```
  %%
  %%
1 while                    => (Tokens.WHILE(...));
2 [a-zA-Z][a-zA-Z0-9_]* => (Tokens.ID(yytext,...));
3 "<"                      => (Tokens.LESS(...));
4 "<="                     => (Tokens.LE(yypos,...));
```

input "`while`" matches rule 1 according B above

input "`<=`" matches rule 4 according A above

Start States (or Start Conditions)

- *<u>start states</u> permit multiple lexical analyzers to run together.*
- *each <u>translation rule</u> can be prefixed with* **`<start-state>`**
- *the lexer is initially in a predefined start stae called* **`INITIAL`**
- *define new start states* (in <u>**ml-lex-definitions**</u>): **`%s COMMENT STRING`**
- *to switch to another start states (in <u>**action**</u>):* **`YYBEGIN COMMENT`**
- ***example****: multi-line comments in C*

```
%%
%s COMMENT
%%
<INITIAL>"/*"    => (YYBEGIN COMMENT; continue());
<COMMENT>"*/"    => (YYBEGIN INITIAL; continue());
<COMMENT>.|"\n"  => (continue());
<INITIAL> ........
```


Implementation of Lex

- *construct NFA for <u>sum</u> of Lex translation rules (regexp/action);*
- *convert NFA to DFA, then minimize the DFA*
- *to recognize the input, simulate DFA to* ***termination****; find the <u>last</u> DFA state that includes NFA final state, execute associated action (this pickes* ***longest*** *match). If the last DFA state has >1 NFA final states, pick one for rule that appears* ***first***
- *how to represent DFA, the transition table:*

  2D array indexed by state and input-character too big !

  each state has a linked list of (char, next-state) pairs too slow!

  hybrid scheme is the best ------- see Dragon Book page 144-146