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 —— token + associated value

| 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 atomic unit of a language, and are usually specific strings or instances of classes of strings.

<table>
<thead>
<tr>
<th>Tokens</th>
<th>Sample Values</th>
<th>Informal Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LET</td>
<td>let</td>
<td>Keyword LET</td>
</tr>
<tr>
<td>END</td>
<td>end</td>
<td>Keyword END</td>
</tr>
<tr>
<td>PLUS</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>LPAREN</td>
<td>(</td>
<td></td>
</tr>
<tr>
<td>COLON</td>
<td>:</td>
<td></td>
</tr>
<tr>
<td>STRING</td>
<td>&quot;str&quot;</td>
<td></td>
</tr>
<tr>
<td>RPAREN</td>
<td>)</td>
<td></td>
</tr>
<tr>
<td>INT</td>
<td>49, 48</td>
<td>Integer constants</td>
</tr>
<tr>
<td>ID</td>
<td>do_nothing1, a, int, string</td>
<td>letter followed by letters, digits, and underscores</td>
</tr>
<tr>
<td>EQ</td>
<td>=</td>
<td></td>
</tr>
<tr>
<td>EOF</td>
<td></td>
<td>end of file</td>
</tr>
</tbody>
</table>
Lexical Analysis, How?

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

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

  \[
  \text{identifier} = \text{letter} \ (\text{letter} \ | \ \text{digit} \ | \ \text{underscore})^* \\
  \text{letter} = a \ | \ ... \ | \ z \ | \ A \ | \ ... \ | \ Z \\
  \text{digit} = 0 \ | \ 1 \ | \ ... \ | \ 9
  \]

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

  Regular Expression Spec \(\Rightarrow\) NFA \(\Rightarrow\) DFA \(\Rightarrow\) Transition Table \(\Rightarrow\) Lexical Analyzer

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

  Regular Expression Spec (in **lex** format) \(\Rightarrow\) feed to lex \(\Rightarrow\) Lexical Analyzer

Regular Expressions and Regular Languages

• Given an alphabet \(\Sigma\), the **regular expressions** over \(\Sigma\) and their corresponding regular languages are

  a) \(\emptyset\) denotes \(\emptyset\); \(\epsilon\), the empty string, denotes the language \(\{\epsilon\}\).

  b) for each \(a\) in \(\Sigma\), 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 \(L_i\) (\(i=0,\ldots,\infty\)) and \(L^1\) is just \(\{x_1 x_2 x_3 \mid x_1 \in L, ..., x_3 \in L\}\).

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

Regular Expressions

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

  \[
  \text{identifier} = \text{letter} \ (\text{letter} \ | \ \text{digit} \ | \ \text{underscore})^* \\
  \text{letter} = a \ | \ ... \ | \ z \ | \ A \ | \ ... \ | \ Z \\
  \text{digit} = 0 \ | \ 1 \ | \ ... \ | \ 9
  \]

• each regular expression define a regular language --- a set of strings 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.

<table>
<thead>
<tr>
<th>Example</th>
<th>Regular Expression</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a^+)</td>
<td>0 or more (a)</td>
<td></td>
</tr>
<tr>
<td>(a^*)</td>
<td>1 or more (a)</td>
<td></td>
</tr>
<tr>
<td>((a</td>
<td>b)^*)</td>
<td>all strings of (a) and (b) (including (\epsilon))</td>
</tr>
<tr>
<td>((aa</td>
<td>ab</td>
<td>ba</td>
</tr>
<tr>
<td>([a-zA-Z])</td>
<td>shorthand for “a</td>
<td>b...</td>
</tr>
<tr>
<td>([0-9])</td>
<td>shorthand for “0</td>
<td>1</td>
</tr>
<tr>
<td>(0([0-9])^*0)</td>
<td>numbers that start and end with 0</td>
<td></td>
</tr>
<tr>
<td>((ab</td>
<td>a+b)^*a</td>
<td>a+a</td>
</tr>
</tbody>
</table>

• the following is not a regular expression: \(a^nb^n\) (\(n > 0\))
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.

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).

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?

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 character 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 successful sequence: \(0 \rightarrow 0 \rightarrow 1 \rightarrow 2 \rightarrow 3\)

Another unsuccessful sequence: \(0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0\)

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 successful sequence: \(0 \rightarrow 1 \rightarrow 1 \rightarrow 2 \rightarrow 3\)

Transition Table

- Finite Automata can also be represented using transition tables

For NFA, each entry is a set of states:

<table>
<thead>
<tr>
<th>STATE</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{0,1}</td>
<td>{0}</td>
</tr>
<tr>
<td>1</td>
<td>{-}</td>
<td>{2}</td>
</tr>
<tr>
<td>2</td>
<td>{-}</td>
<td>{3}</td>
</tr>
<tr>
<td>3</td>
<td>{-}</td>
<td>{-}</td>
</tr>
</tbody>
</table>

For DFA, each entry is a unique state:

<table>
<thead>
<tr>
<th>STATE</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

NFA with \(\epsilon\)-transitions

1. NFA can have \(\epsilon\)-transitions --- edges labelled with \(\epsilon\)

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
   - \( a \in \Sigma \)

2. "Inductive" Construction
   - \( R_1 | R_2 \)
   - \( R_1^* \)
   - \( R_1 R_2 \)

---

**Example: RE -> NFA**

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

\( a \) (in \( a|b \)) \[ \Rightarrow \]
\( b \) (in \( a|b \)) \[ \Rightarrow \]
\( a|b \) \[ \Rightarrow \]
Example: RE -> NFA (cont’d)

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

\((a|b)^*====>\)

\[0\rightarrow 1\rightarrow 2\rightarrow 3\rightarrow 6\rightarrow 7\]

\[\varepsilon\rightarrow a\rightarrow \varepsilon\rightarrow b\rightarrow \varepsilon\rightarrow \varepsilon\]

\[abb====>(several\ steps\ are\ omitted)\]

NFA -> DFA

- NFA are non-deterministic; need DFA in order to write a deterministic program.
- 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,
  \[\varepsilon\text{-CLOSURE}(s) = \{ s \} \cup \{ t \mid s \text{ can reach } t \text{ via } \varepsilon\text{-transitions} \}\]
- Definition: for each set of states \(S\) in NFA,
  \[\varepsilon\text{-CLOSURE}(S) = \bigcup \varepsilon\text{-CLOSURE}(s) \text{ for all } s_i \text{ in } S\]

Example: RE -> NFA (cont’d)

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

\((a|b)^*abb====>\)

\[0\rightarrow 1\rightarrow 2\rightarrow 3\rightarrow 6\rightarrow 7\]

\[\varepsilon\rightarrow a\rightarrow \varepsilon\rightarrow b\rightarrow \varepsilon\rightarrow \varepsilon\rightarrow \varepsilon\]

\[abb====>(several\ steps\ are\ omitted)\]

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 \(\varepsilon\text{-CLOSURE}(s)\); the final states of the DFA are those that include a NFA-final-state
- Algorithm: converting an NFA \(N\) into a DFA \(D\)

\[\text{Dstates} = \{ \varepsilon\text{-CLOSURE}(s_0), s_0 \text{ is } N\text{'s start state} \}\]
\[\text{Dstates are initially “unmarked”}\]
\[\text{while there is an unmarked D-state } X \text{ do } \{
\text{mark } X
\text{ for each } a \text{ in } S \text{ do } \{
T = (\text{states reached from any } s_i \text{ in } X \text{ via } a)
Y = \varepsilon\text{-CLOSURE}(T)
\text{if } Y \text{ not in Dstates then add } Y \text{ to Dstates “unmarked”}
\text{add transition from } X \text{ to } Y, \text{ labeled with } a
\}\}\]
Example : NFA -> DFA

• converting NFA for \((a|b)*abb\) to a DFA

The start state \(A = e\text{-closure}(0) = \{0, 1, 2, 4, 7\}; \text{Dstates}=\{A\}\)

1st iteration: A is unmarked; mark A now;

a-transitions: \(T = \{3, 8\}\)

a new state \(B = e\text{-closure}(3) \cup e\text{-closure}(8)\)

\(= \{3, 6, 1, 2, 4, 7\} \cup \{8\} = \{1, 2, 3, 4, 6, 7, 8\}\)

add a transition from A to B labelled with a.

b-transitions: \(T = \{5\}\)

a new state \(C = e\text{-closure}(5) = \{1, 2, 4, 5, 6, 7\}\)

add a transition from A to C labelled with b.

\text{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 e-closure is B itself.

add a transition from B to B labelled with a.

B’s b-transitions: \(T = \{5\}\)

add a transition from B to C labelled with b.

\text{Dstates} = \{A, B, C, D\}

Example : NFA -> DFA (cont’d)

next we pick D, and mark D

D’s a-transitions: \(T = \{3, 8\}\); its e-closure is B.

add a transition from D to B labelled with a.

D’s b-transitions: \(T = \{5, 10\}\)

a new state \(E = e\text{-closure}(\{5, 10\}) = \{1, 2, 4, 5, 6, 7, 10\}\)

\text{Dstates} = \{A, B, C, D, E\}; E is a final state since it has 10;

next we pick E, and mark E

all states in \text{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 a **lexical specification** as input, and produces a **lexical processor** written in C.

```
Lex Specification foo.l  ->  lex.yy.c
lex.yy.c    ->  C Compiler  ->  a.out
input text  ->  a.out       ->  sequence of tokens
```

- **Implementation of Lex:**
  Lex Spec -> NFA -> DFA -> Transition Tables + Actions -> yyex()

ML-Lex

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

```
Lex Specification foo.lex  ->  foo.lex.sml
foo.lex.sml  ->  ML Compiler  ->  module Mlex
input text  ->  M lex       ->  sequence of tokens
```

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

Lex Specification

```
expression  => (action);
DIGITS [0-9]
integer    => (print("INT"));
SPACE=[ \t\n\012];
DIGITS=[0-9];
%%
%%
%%
char getc() { ...... }
```

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

ML-Lex Specification

```
type pos = int
val lineNum = ...
val lexresult = ....
...
%%
%%
%%
%%
%%
%%
%%
expression => (action);
integer    => (print("INT"));
SPACE=[ \t\n\012];
DIGITS=[0-9];
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
%%
##
What does ML-Lex generate?

```
foo.lex   ML-Lex   foo.lex.sml

sample foo.lex.sml:
structure Mlex =
  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()
```

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 $lex$)

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

  Definition of named regular expressions :

  identifier = regular expression

  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| 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)

  !b 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{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 start-state` ---- 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. longest match is preferred
  B. among rules which matched the same number of characters, the rule given first 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)

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

Implementation of Lex

• construct NFA for sum of Lex translation rules (regexp/action);
• convert NFA to DFA, then minimize the DFA
• to recognize the input, simulate DFA to termination; find the last DFA state that includes NFA final state, execute associated action (this picks `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