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, \s) and comments
  2. line numbering

Example: Source Code

A Sample Toy Program:

```plaintext
(* 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

<table>
<thead>
<tr>
<th>token</th>
<th>associated value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LET</td>
<td>51</td>
</tr>
<tr>
<td>LPAREN</td>
<td>76</td>
</tr>
<tr>
<td>ID(int)</td>
<td>80</td>
</tr>
<tr>
<td>COLON</td>
<td>86</td>
</tr>
<tr>
<td>EQ</td>
<td>95</td>
</tr>
<tr>
<td>LPAREN</td>
<td>110</td>
</tr>
<tr>
<td>INT</td>
<td>113</td>
</tr>
<tr>
<td>ID(do_nothing1)</td>
<td>150</td>
</tr>
<tr>
<td>INT</td>
<td>162</td>
</tr>
<tr>
<td>RPAREN</td>
<td>170</td>
</tr>
<tr>
<td>ID</td>
<td>189</td>
</tr>
<tr>
<td>RPAREN</td>
<td>198</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>token</th>
<th>associated value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUNCTION</td>
<td>56</td>
</tr>
<tr>
<td>ID</td>
<td>(do_nothing1)</td>
</tr>
<tr>
<td>ID(a)</td>
<td>77</td>
</tr>
<tr>
<td>ID(b)</td>
<td>85</td>
</tr>
<tr>
<td>ID(string)</td>
<td>88</td>
</tr>
<tr>
<td>RPAREN</td>
<td>94</td>
</tr>
<tr>
<td>EQ</td>
<td>95</td>
</tr>
<tr>
<td>ID(do_nothing2)</td>
<td>99</td>
</tr>
<tr>
<td>LPAREN</td>
<td>110</td>
</tr>
<tr>
<td>PLUS</td>
<td>112</td>
</tr>
<tr>
<td>ID(int)</td>
<td>141</td>
</tr>
<tr>
<td>INT</td>
<td>49, 48</td>
</tr>
<tr>
<td>ID</td>
<td>do_nothing1, a,</td>
</tr>
<tr>
<td></td>
<td>int, string</td>
</tr>
<tr>
<td>EOF</td>
<td>203</td>
</tr>
</tbody>
</table>

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>COLON</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STRING</td>
<td>“str”</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:

  - 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 and Regular Languages

- **Given an alphabet** $\Sigma$, the **regular expressions** over $\Sigma$ and their corresponding **regular languages** are

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

  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 $R \cup S$ denotes the language $L_R \cup L_S$, i.e., $\{ x | x \in L_R \text{ or } x \in L_S \}$.

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

  e) if $R$ denotes $L_R$ then $R^*$ denotes the language $L_R^*$ where $L^*$ is the union of all $L^j (j=0, \ldots, \infty)$ and $L^1$ is just $\{ x_1x_2 \ldots x_i | x_1 \in L, \ldots, x_i \in L \}$.

  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)

  $\text{identifier} = \text{letter} \ (\text{letter} \ | \ \text{digit} \ | \ \text{underscore})^*$

  - $\text{or}$

  - $0 \ or \ more$

  - 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.

Example

<table>
<thead>
<tr>
<th>Regular Expression</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a^*$</td>
<td>0 or more $a$</td>
</tr>
<tr>
<td>$a^+$</td>
<td>1 or more $a$</td>
</tr>
<tr>
<td>$(a+b)^*$</td>
<td>all strings of $a$ and $b$ (including $\varepsilon$)</td>
</tr>
<tr>
<td>$(a|b|a|b|)^*$</td>
<td>all strings of $a$ and $b$ of even length</td>
</tr>
<tr>
<td>$[a-zA-Z]$</td>
<td>shorthand for “$a</td>
</tr>
<tr>
<td>$[0-9]$</td>
<td>shorthand for “$0</td>
</tr>
<tr>
<td>$0(0-9)^n0$</td>
<td>numbers that start and end with 0</td>
</tr>
<tr>
<td>$(ab</td>
<td>aab</td>
</tr>
</tbody>
</table>

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

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).

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) $\varepsilon$ can label edges (these edges are called $\varepsilon$-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 $\varepsilon$
  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:

\[ 0 \to 0 \to 1 \to 2 \to 3 \]

Another unsuccessful sequence:

\[ 0 \to 0 \to 0 \to 0 \]

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

Another unsuccessful sequence:

\[ 0 \to 0 \to 0 \to 0 \]

One successful sequence:

\[ 0 \to 1 \to 2 \to 3 \]

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>0</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>

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:

\[ 0 \to 1 \to 2 \to 3 \]

NFA with \(\varepsilon\)-transitions

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

accepts the regular language denoted by \((aa^*|bb^*)\)
Regular Expressions -> NFA

- How to construct NFA (with $\epsilon$-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

- $\epsilon$
- $a \in \Sigma$

2. "Inductive" Construction

- $R_1^*$

Example: RE -> NFA

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

a (in $a|b$) $\implies$ 2 $\rightarrow$ 3
b (in $a|b$) $\implies$ 4 $\rightarrow$ 5

$$a|b \implies 1 \rightarrow 2 \rightarrow 3 \rightarrow 6$$
Example : RE -> NFA (cont’d)

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

\[(a|b)^* \Rightarrow \epsilon \]

\[abb \Rightarrow \text{(several steps are omitted)} \]

Example : RE -> NFA (cont’d)

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

\[(a|b)^* \Rightarrow \epsilon \]

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,

  \[ \epsilon\text{-CLOSURE}(s) = \{ s \} \cup \{ t \mid s \text{ can reach } t \text{ via } \epsilon\text{-transitions} \} \]

- **Definition:** for each set of states \(S\) in NFA,

  \[ \epsilon\text{-CLOSURE}(S) = \bigcup_{s_i \in S} \epsilon\text{-CLOSURE}(s_i) \]

- **Definition:** for each state \(s\) in NFA,

  \[ \epsilon\text{-CLOSURE}(s) = \{ s \} \cup \{ t \mid s \text{ can reach } t \text{ via } \epsilon\text{-transitions} \} \]

- **Definition:** for each set of states \(S\) in NFA,

  \[ \epsilon\text{-CLOSURE}(S) = \bigcup_{s_i \in S} \epsilon\text{-CLOSURE}(s_i) \]

- **Algorithm : converting an NFA \( N \) into a DFA \( D \)

  \[
  D_{states} = \{ \epsilon\text{-CLOSURE}(s_0), s_0 \text{ is } N\text{'s start state} \}
  
  \text{while there is an unmarked D-state } X \text{ do } \{
  \text{mark } X
  
  \text{for each } a \text{ in } S \text{ do } \{
  \text{T = (states reached from any } s_i \text{ in } X \text{ via } a)
  \text{Y = } \epsilon\text{-CLOSURE}(T)
  \text{if } Y \text{ not in } D_{states} \text{ then add } Y \text{ to } D_{states} \text{ “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 \(A = \epsilon\text{-CLOSURE}(0) = \{0, 1, 2, 4, 7\}\); \(Dstates = \{A\}\)

1st iteration: A is unmarked; mark A now;
- \(a\)-transitions: \(T = \{3, 8\}\)
  - a new state \(B = \epsilon\text{-CLOSURE}(3) \cup \epsilon\text{-CLOSURE}(8) = \{3, 6, 1, 2, 4, 7\} \cup \{8\} = \{1, 2, 3, 4, 6, 7, 8\}\)
  - add a transition from A to B labelled with \(a\)
- \(b\)-transitions: \(T = \{5\}\)
  - a new state \(C = \epsilon\text{-CLOSURE}(5) = \{1, 2, 4, 5, 6, 7\}\)
  - 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: \(T = \{3, 8\}\); its \(\epsilon\text{-CLOSURE}\) is B itself.
  - add a transition from B to B labelled with \(a\)
- \(B\)'s \(b\)-transitions: \(T = \{5\}\); its \(\epsilon\text{-CLOSURE}\) is \(\{5\}\)
  - add a transition from B to C labelled with \(b\)

\(Dstates = \{A, B, C, D\}\)

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

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

all states in \(Dstates\) are marked, the DFA is constructed!

Example : NFA -> DFA (cont’d)

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

Other Algorithms

- \textit{How to minimize a DFA} ? (see Dragon Book 3.9, pp141)
- \textit{How to convert RE to DFA directly} ? (see Dragon Book 3.9, pp135)
- \textit{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.

```
Lex Specification

Lex
```

```
lex.yy.c
```

```
a.out
```

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

Lex Specification

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

ML-Lex

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

```
Lex Specification

ML-Lex
```

```
foo.lex.sml
```

```
module Mlex
```

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

ML-Lex Specification

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

```
ML-Lex Specification

ML-Lex
```

```
foo.lex.sml
```

```
module Mlex
```

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

```
Lex Specification

ML-Lex
```

```
lex.yy.c
```

```
a.out
```

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

```
Lex Specification

Lex
```

```
lex.yy.c
```

```
a.out
```

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

Lex Specification

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

```
Lex Specification

Lex
```

```
lex.yy.c
```

```
a.out
```

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

Lex Specification

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

```
Lex Specification

Lex
```

```
lex.yy.c
```

```
a.out
```

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

Lex Specification

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

```
Lex Specification

Lex
```

```
lex.yy.c
```

```
a.out
```

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

Lex Specification

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

```
Lex Specification

Lex
```

```
lex.yy.c
```

```
a.out
```

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

Lex Specification

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

```
Lex Specification

Lex
```

```
lex.yy.c
```

```
a.out
```

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

Lex Specification

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

```
Lex Specification

Lex
```

```
lex.yy.c
```

```
a.out
```

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

Lex Specification

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

```
Lex Specification

Lex
```

```
lex.yy.c
```

```
a.out
```

**Implementation of Lex:**
Lex Spec -> NFA -> DFA -> Transition Tables + Actions -> yylex()
What does ML-Lex generate?

ML-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();
```

ML-Lex Definitions

- Things you can write inside the "ml-lex definitions" section (2nd part):
  ```ml
  \%
  define new start states
  \%
  to reject a match
  \%
  count the line number
  \%
  the resulting structure name
  \%
  (the default is Mlex)
  \%
  \%
  \%
  (hint: you probably don't need use \%reject, \%count, or \%structure
  for assignment 2.)
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  \%```
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

ML-Lex Translation Rules (cont’d)

• 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

```%
A

while (Tokens.WHILE(...));
B

[a-zA-Z] [a-zA-Z0-9_]* (Tokens.ID(yytext,...));
C

<= (Tokens.LESS(...));
D

< (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
  #s COMMENT
  %%
  <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