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)
  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>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</td>
<td>138</td>
</tr>
<tr>
<td>RPAREN</td>
<td>144</td>
</tr>
<tr>
<td>ID(do_nothing1)</td>
<td>150</td>
</tr>
<tr>
<td>ID(d)</td>
<td>162</td>
</tr>
<tr>
<td>ID(do_nothing2)</td>
<td>176</td>
</tr>
<tr>
<td>ID(do_nothing1)</td>
<td>177</td>
</tr>
<tr>
<td>INT</td>
<td>189</td>
</tr>
<tr>
<td>COMMA</td>
<td>190</td>
</tr>
<tr>
<td>RPAREN</td>
<td>198</td>
</tr>
</tbody>
</table>

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>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LET</td>
<td>51</td>
</tr>
<tr>
<td>FUNCTION</td>
<td>56</td>
</tr>
<tr>
<td>ID(do_nothing1)</td>
<td>65</td>
</tr>
<tr>
<td>LPAREN</td>
<td>76</td>
</tr>
<tr>
<td>ID(a)</td>
<td>77</td>
</tr>
<tr>
<td>COLON</td>
<td>86</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>ID(a)</td>
<td>111</td>
</tr>
<tr>
<td>PLUS</td>
<td>112</td>
</tr>
<tr>
<td>INT</td>
<td>113</td>
</tr>
<tr>
<td>RPAREN</td>
<td>114</td>
</tr>
<tr>
<td>FUNCTION</td>
<td>117</td>
</tr>
<tr>
<td>ID</td>
<td>126</td>
</tr>
<tr>
<td>COLON</td>
<td>139</td>
</tr>
<tr>
<td>ID(int)</td>
<td>141</td>
</tr>
<tr>
<td>EQ</td>
<td>146</td>
</tr>
<tr>
<td>ID(do_nothing2)</td>
<td>150</td>
</tr>
<tr>
<td>COMMA</td>
<td>163</td>
</tr>
<tr>
<td>STRING</td>
<td>165</td>
</tr>
<tr>
<td>LPAREN</td>
<td>170</td>
</tr>
<tr>
<td>IN</td>
<td>173</td>
</tr>
<tr>
<td>ID</td>
<td>177</td>
</tr>
<tr>
<td>COMMA</td>
<td>190</td>
</tr>
<tr>
<td>STRING</td>
<td>192</td>
</tr>
<tr>
<td>RPAREN</td>
<td>198</td>
</tr>
<tr>
<td>END</td>
<td>200</td>
</tr>
</tbody>
</table>

<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:
  
  
  ```
  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)*
  ```

- 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><code>a*</code></td>
<td>0 or more a's</td>
</tr>
<tr>
<td><code>a+</code></td>
<td>1 or more a's</td>
</tr>
<tr>
<td>`(a</td>
<td>b)*`</td>
</tr>
<tr>
<td>`(aa</td>
<td>ab</td>
</tr>
<tr>
<td><code>[a-zA-Z]</code></td>
<td>shorthand for &quot;a</td>
</tr>
<tr>
<td><code>[0-9]</code></td>
<td>shorthand for &quot;0</td>
</tr>
<tr>
<td><code>0{0-9}</code></td>
<td>numbers that start and end with 0</td>
</tr>
<tr>
<td>`(ab</td>
<td>aab</td>
</tr>
<tr>
<td><code>?</code></td>
<td></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

- 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:

```c
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) $\epsilon$ can label edges (these edges are called $\epsilon$-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 $\epsilon$
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:

Another unsuccessful sequence:

input string: aabb

The successful sequence:

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 \(\varepsilon\)-transitions

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

\(\varepsilon\)-transitions allows 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

2. **Inductive Construction**
   - \(R_1 \mid R_2\)
   - \(R_1 \cdot R_2\)

Example: RE -> NFA

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

- \(a\) (in \(a\mid b\)) \(\Rightarrow\) \(2 \rightarrow 3\)
- \(b\) (in \(a\mid b\)) \(\Rightarrow\) \(4 \rightarrow 5\)

\(a\mid b\) \(\Rightarrow\) \(1 \rightarrow 4 \rightarrow 5\)
Example : RE -> NFA (cont’d)

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

\((a|b)^* \implies \epsilon\)

\(abb \implies \epsilon\) (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,
  \[\epsilon\text{-CLOSURE}(s) = \{ s \} \cup \{ t \mid s \text{ can reach } t \text{ via } \epsilon\text{-transitions}\}\]
- **Definition:** for each set of states \(S\) in NFA,
  \[\epsilon\text{-CLOSURE}(S) = \bigcup_i \epsilon\text{-CLOSURE}(s_i) \text{ for all } s_i \in S\]

NFA -> DFA (cont’d)

- Each DFA-state is a set of NFA-states
- Suppose the **start state** of the NFA is \(s_0\), then the **start state** for its DFA is \(\epsilon\text{-CLOSURE}(s_0)\)
  - The **final states** of the DFA are those that include a NFA-final-state
- **Algorithm:** converting an NFA \(N\) into a DFA \(D\)
  
  \[\text{Dstates} = \{ \epsilon\text{-CLOSURE}(s_0), s_0 \text{ is } N\text{'s start state} \}\]
  
  Dstates are initially “unmarked”
  
  \[\text{while there is an unmarked D-state } X \text{ do }\{
  \text{mark } X
  \text{ for each } a \text{ in } S \text{ do }\{
  \text{T} = \{ \text{states reached from any } s_i \text{ in } X \text{ via } a\}
  \text{Y} = \epsilon\text{-CLOSURE}(T)
  \text{if } Y \text{ not in Dstates then add } Y \text{ to Dstates “unmarked”}
  \text{add transition from } X \text{ to } Y \text{, labeled with } a
  \}\}
  \}\]
Example : NFA -> DFA

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

  The start state \(A = \varepsilon\)-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 \(B = \varepsilon\)-CLOSURE(3) \(\cup\) \(\varepsilon\)-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 \(C = \varepsilon\)-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 \(\varepsilon\)-CLOSURE is B itself.
  add a transition from B to B labelled with \(a\)

  \(b\)-transitions: \(T = \{5\}\)
  add a transition from B to C labelled with \(b\)

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

  then we pick C, and mark C
  C’s \(a\)-transitions: \(T = \{3, 8\}\); its \(\varepsilon\)-CLOSURE is B.
  add a transition from C to B labelled with \(a\)
  C’s \(b\)-transitions: \(T = \{5\}\); its \(\varepsilon\)-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 \(\varepsilon\)-CLOSURE is B.
  add a transition from D to B labelled with \(a\)
  D’s \(b\)-transitions: \(T = \{5, 10\}\);
  a new state \(E = \varepsilon\)-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

---

Example : NFA -> DFA (cont’d)

- 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 *lexical specification* as input, and produces a *lexical processor* written in C.

![Lex Diagram](image)

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

Lex Specification

- **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.

![ML-Lex Diagram](image)

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

ML-Lex Specification

- **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?

ML-Lex

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()

input filename

each call returns one token !

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.)

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, \\

  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
  
  \[ [a-z]+ \Rightarrow (\text{print } \text{yytext}); \]
  
  \[ [0-9]{3} \Rightarrow (\text{print } \text{Char.ord(sub(yytext,0))}); \]
• 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

```plaintext
1 2 3 4

while \Rightarrow (\text{Tokens.WHILE(...)});
[a-zA-Z][a-zA-Z0-9_]* \Rightarrow (\text{Tokens.ID(yytext,...)});
<"\Rightarrow (\text{Tokens.LESS(...)});
<=" \Rightarrow (\text{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

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