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>LET</th>
<th>51</th>
<th>FUNCTION</th>
<th>56</th>
<th>ID(do_nothing1)</th>
<th>65</th>
</tr>
</thead>
<tbody>
<tr>
<td>LPAREN</td>
<td>76</td>
<td>ID(a)</td>
<td>77</td>
<td>COLON</td>
<td>78</td>
</tr>
<tr>
<td>ID(int)</td>
<td>80</td>
<td>COMMA</td>
<td>83</td>
<td>ID(b)</td>
<td>85</td>
</tr>
<tr>
<td>COLON</td>
<td>86</td>
<td>ID(string)</td>
<td>88</td>
<td>RPAREN</td>
<td>94</td>
</tr>
<tr>
<td>EQ</td>
<td>95</td>
<td>ID(do_nothing2)</td>
<td>99</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LPAREN</td>
<td>110</td>
<td>ID(a)</td>
<td>111</td>
<td>PLUS</td>
<td>112</td>
</tr>
<tr>
<td>INT(1)</td>
<td>113</td>
<td>RPAREN</td>
<td>114</td>
<td>FUNCTION</td>
<td>117</td>
</tr>
<tr>
<td>ID</td>
<td>126</td>
<td></td>
<td></td>
<td>RPAREN</td>
<td>137</td>
</tr>
<tr>
<td>ID(d)</td>
<td>138</td>
<td>COLON</td>
<td>139</td>
<td>ID(int)</td>
<td>141</td>
</tr>
<tr>
<td>RPAREN</td>
<td>144</td>
<td>EQ</td>
<td>146</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td>150</td>
<td></td>
<td></td>
<td>LPAREN</td>
<td>161</td>
</tr>
<tr>
<td>ID(d)</td>
<td>162</td>
<td>COMMA</td>
<td>163</td>
<td>STRING(str)</td>
<td>165</td>
</tr>
<tr>
<td>RPAREN</td>
<td>170</td>
<td>IN</td>
<td>173</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td>177</td>
<td></td>
<td></td>
<td>LPAREN</td>
<td>188</td>
</tr>
<tr>
<td>INT(0)</td>
<td>189</td>
<td>COMMA</td>
<td>190</td>
<td>STRING(str2)</td>
<td>192</td>
</tr>
<tr>
<td>RPAREN</td>
<td>198</td>
<td>END</td>
<td>200</td>
<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>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 under-scores</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:
  
  $$\text{identifier} = \text{letter} (\text{letter} \mid \text{digit} \mid \text{underscore})^*$$
  
  $$\text{letter} = \text{a} \mid \ldots \mid \text{z} \mid \text{A} \mid \ldots \mid \text{Z}$$
  
  $$\text{digit} = 0 \mid 1 \mid \ldots \mid 9$$
  
- Second, based on the above **lexical specification**, build the lexical analyzer (to recognize tokens) by hand,

  Regular Expression Spec $\mapsto$ NFA $\mapsto$ DFA $\mapsto$ Transition Table $\mapsto$ Lexical Analyzer
  
- Or just by using **lex** --- the lexical analyzer generator

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

Regular Expressions

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

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

  “or”

- **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.

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 $R \mid S$ 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 $L_RL_S$, 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 $L^n$ ($n=0,\ldots,\infty$) and $L^0$ is just $\{x_1x_2\ldots x_n \mid x_i \in L, \ldots, x_n \in L\}$.
  
  f) if $R$ denotes $L_R$ then $(R)$ denotes the same language $L_R$.

<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's</td>
<td></td>
</tr>
<tr>
<td>$a^+$</td>
<td>1 or more a's</td>
<td></td>
</tr>
<tr>
<td>$(a\mid b)^*$</td>
<td>all strings of a's and b's (including $\varepsilon$)</td>
<td></td>
</tr>
<tr>
<td>$(a\mid ab\mid ba\mid bb)^*$</td>
<td>all strings of a's and b's of even length</td>
<td></td>
</tr>
<tr>
<td>$[a-zA-Z]$</td>
<td>shorthand for $a\mid b\ldots z\mid A\ldots Z$</td>
<td></td>
</tr>
<tr>
<td>$[0-9]$</td>
<td>shorthand for “0\mid 1\mid 2\ldots 9”</td>
<td></td>
</tr>
<tr>
<td>$(0\mid 9)^*0$</td>
<td>numbers that start and end with 0</td>
<td></td>
</tr>
<tr>
<td>$(ab\mid aa\mid bb)^*(a\mid aa\mid \varepsilon)$</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>?</td>
<td>all strings that contain $foo$ as substring</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

  ![Transition Diagram](image)

  - 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:*

```python
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 \(\varepsilon\)-transitions

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

\(\varepsilon\)-transitions:

- \(0 \rightarrow 1\) with \(\varepsilon\)
- \(3 \rightarrow 4\) with \(b\)
- \(1 \rightarrow 2\) with \(a\)
- \(2 \rightarrow 1\) with \(a\)

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
     ![Diagram 1](image)
   - a ∈ Σ
     ![Diagram 2](image)

2. **Inductive Construction**
   - R₁ | R₂
     ![Diagram 3](image)
   - R₁ R₂
     ![Diagram 4](image)

Example: RE -> NFA

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

- a (in a | b) ===> a
  ![Diagram 5](image)
- b (in a | b) ===> b
  ![Diagram 6](image)
- a | b ===> a
  ![Diagram 7](image)
Example: RE -> NFA (cont’d)

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

\[(a|b)^* \quad \Rightarrow \quad \varepsilon\]

\[\varepsilon \quad \Rightarrow \quad 0\]

\[a \quad \Rightarrow \quad \{1\} \cup \{2\} \cup \{3\} \cup \{4\} \cup \{5\} \cup \{6\} \cup \{7\}\]

\[b \quad \Rightarrow \quad \{1\} \cup \{2\} \cup \{3\} \cup \{4\} \cup \{5\} \cup \{6\} \cup \{7\}\]

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

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,
  \[\varepsilon\text{-CLOSURE}(S) = \cup_{s_i \in S} \varepsilon\text{-CLOSURE}(s) \text{ for all } s_i \in S\]

NFA -> DFA (cont’d)

- each DFA-state is a set of NFA-states
- suppose the start state of the NFA is \(s\), then the start state for its DFA is \(\varepsilon\text{-CLOSURE}(s)\);
  the final states of the DFA are those that include an NFA-final-state
- Algorithm: converting an NFA \(N\) into a DFA \(D\) ----
  \[\text{D} = \{\varepsilon\text{-CLOSURE}(s_0), s_0 \text{ is } N\text{'s start state}\}\]
  Dstates are initially "unmarked"
  while there is an unmarked D-state \(X\) do
    mark \(X\)
    for each \(a \in \Sigma\) do
      \[T = \{\text{states reached from any } s_i \text{ in } X \text{ via } a\}\]
      \[Y = \varepsilon\text{-CLOSURE}(T)\]
      if \(Y \notin \text{Dstates}\) 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 = \varepsilon\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 = \(\varepsilon\text{-CLOSURE}(3) \cup \varepsilon\text{-CLOSURE}(8)\)
  = \(\{3, 6, 1, 2, 4, 7\} \cup \{8\} = \{1, 2, 3, 4, 6, 7, 8\}\)
  add a transition from A to B labelled with \(a\)
  \(b\)-transitions: \(T = \{5\} \)
  a new state C = \(\varepsilon\text{-CLOSURE}(5) = \{1, 2, 4, 5, 6, 7\}\)
  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 \(\varepsilon\text{-CLOSURE} \) is B itself.
  add a transition from B to B labelled with \(a\)
  \(b\)-transitions: \(T = \{5\} \)
  a new state D = \(\varepsilon\text{-CLOSURE}(5) = \{1, 2, 4, 5, 6, 7, 10\}\)
  \(\text{Dstates} = \{A, B, C, D\}\)

Example: NFA -> DFA (cont’d)

  B's \(b\)-transitions: \(T = \{5, 9\} ;\)
  a new state D = \(\varepsilon\text{-CLOSURE}(\{5, 9\}) = \{1, 2, 4, 5, 6, 7, 9\}\)
  add a transition from B to D labelled with \(b\)
  \(\text{Dstates} = \{A, B, C, D\}\)
  then we pick C, and mark C
  C's \(a\)-transitions: \(T = \{3, 8\}; \) T's \(\varepsilon\text{-CLOSURE} \) is C itself.
  add a transition from C to C labelled with \(b\)
  \(b\)-transitions: \(T = \{5\} \)
  a new state D = \(\varepsilon\text{-CLOSURE}(\{5\}) = \{1, 2, 4, 5, 6, 7, 10\}\)
  \(\text{Dstates} = \{A, B, C, D, E\}\); E is a \text{final state} since it has 10;
  next we pick E, and mark E

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

  all states in \(\text{Dstates} \) are marked, the DFA is constructed !
Lex

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

![Lex Diagram]

Lex Specification → NFA → DFA → Transition Tables + Actions → yylex()

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.

![ML-Lex Diagram]

ML-Lex Specification

- **expression** is a regular expression ; **action** is a piece of ML program;
- for details, read the *Lesk & Schmidt* paper.
What does ML-Lex generate?

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

ML-Lex Translation Rules

- Each translation rule (3rd part) are in the form
  
  \langle start-state-list \rangle \ regular expression \rightarrow \ (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
  \( \{ \) and \( \} \) 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) 

  \^x\” any char except newline (equivalent to \[\n\])

  \x match string x exactly even if it contains reserved chars
  \? an optional x
  \* 0 or more x's
  \+ 1 or more x's
  \| 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

  ```ml
  [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

  ```ml
  %%
  1 while => (Tokens.WHILE(...));
  2 [a-zA-Z][a-zA-Z0-9_]* => (Tokens.ID(yytext,...));
  3 "<" => (Tokens.LESS(...));
  4 "<=" => (Tokens.LEE(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`): `%% COMMENT STRING`
- to switch to another start states (in `action`): `YYBEGIN COMMENT`
- *example*: multi-line comments in C

  ```ml
  %%
  %% 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 ------- see Dragon Book page 144-146