Lexical Analyzer — Scanner

ASU Textbook Chapter 3.1, 3.3, 3.4, 3.6, 3.7, 3.5

Tsan-sheng Hsu

tshsu@iis.sinica.edu.tw

http://www.iis.sinica.edu.tw/~tshsu
Main tasks

- **Read the input characters and produce as output a sequence of tokens** that the parser uses for syntax analysis.

- **Lexeme**: a sequence of characters matched by a given pattern for a **token**.

  - **Example**: 
    - Lexeme: `pi = 3.1416 ;`
    - Token: `ID ASSIGN FLOAT-LIT SEMI-COL`

  - **patterns**:
    - *identifier (variable) starts with a letter and follows by letters, digits or “_”*;
    - *floating point number starts with a string of digits + a dot + another string of digits*;
Strings

- **Definitions and operations.**
  - **alphabet**: a finite set of characters (symbols);
  - **string**: a finite sequence of characters from the alphabet;
  - $|S|$: length of a string $S$;
  - **empty string**: $\epsilon$;
  - $xy$: concatenation of string $x$ and $y$
    - $\epsilon x \equiv x \epsilon \equiv x$;
  - **exponentiation**:
    - $s^0 \equiv \epsilon$;
    - $s^i \equiv s^{i-1}s$, $i > 0$. 
Parts of a string

Parts of a string: example string “necessary”

- prefix: deleting zero or more tailing characters;  
  eg: “nece”

- suffix: deleting zero or more leading characters;  
  eg: “ssary”

- substring: deleting prefix and suffix;  
  eg: “ssa”

- subsequence: deleting zero or more not necessarily contiguous symbols;  
  eg: “ncsay”

- Proper prefix, suffix, substring or subsequence: one that cannot equal to the original string;
Language

- Language: any set of strings over an alphabet.

- Operations on languages:
  - union: \( L \cup M = \{s \mid s \in L \text{ or } s \in M\}; \)
  - concatenation: \( LM = \{st \mid s \in L \text{ and } t \in M\}; \)
  - \( L^0 = \{\epsilon\}; \)
  - **Kleene closure**: \( L^* = \bigcup_{i=0}^{\infty} L^i; \)
  - **Positive closure**: \( L^+ = \bigcup_{i=1}^{\infty} L^i; \)
  - \( L^* = L^+ \cup \{\epsilon\}. \)
Regular expressions

- A regular expression \( r \) denotes a language \( L(r) \), also called a regular set.

- Operations on regular expressions:

<table>
<thead>
<tr>
<th>regular expression</th>
<th>language</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \emptyset )</td>
<td>empty set {}</td>
</tr>
<tr>
<td>( \epsilon )</td>
<td>the set containing the empty string {\epsilon}</td>
</tr>
<tr>
<td>( a )</td>
<td>{a} where ( a ) is a legal symbol</td>
</tr>
<tr>
<td>( r</td>
<td>s )</td>
</tr>
<tr>
<td>( rs )</td>
<td>( L(r)L(s) ) — concatenation</td>
</tr>
<tr>
<td>( r^* )</td>
<td>( L(r)^* ) — Kleene closure</td>
</tr>
</tbody>
</table>

- Example:

  \( a|b \) \( \{a, b\} \)
  \( (a|b)(a|b) \) \( \{aa, ab, ba, bb\} \)
  \( a^* \) \( \{\epsilon, a, aa, aaa, \ldots\} \)
  \( a|a^*b \) \( \{a, b, ab, aab, \ldots\} \)
  C identifier \( (A|\cdots|Z|a|\cdots|z) \ ((A|\cdots|Z|a|\cdots|z|) | (0|1|\cdots|9) | \_ )^* \)
Regular definitions

- For simplicity, give names to regular expressions.
  - format: name → regular expression.
  - example 1: digit → 0|1|2|⋯|9.
  - example 2: letter → a|b|c|⋯|z|A|B|⋯|Z.

- Notational standards:
  - \( r^* \) \( \epsilon \)

- Example: C variable name: \([A − Z a − z_−][A − Z a − z0 − 9_−]^*\)
Non-regular sets

- Balanced or nested construct
  - Example: if \( \cdots \) then \( \cdots \) else
  - Recognized by context free grammar.

- Matching strings:
  - \( \{wcw\} \), where \( w \) is a string of \( a \)'s and \( b \)'s and \( c \) is a legal symbol.
  - Cannot be recognized even using context free grammars.

- Remark: anything that needs to “memorize” something happened in the past.
Finite state automata (FA)

- FA is a mechanism used to recognize tokens specified by a regular expression.

**Definition:**
- A finite set of states.
- A set of transitions, labeled by characters.
- A starting state.
- A set of final (accepting) states.

**Example:** transition graph for the regular expression \((abc^+)^+\)
Transition graph and table for FA

- **Transition graph:**

  - Transition graph:

  ![Transition Graph Diagram]

  - Transition table:

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

  - Rows are input symbols.
  - Columns are current states.
  - Entries are resulting states.
  - Along with the table, a start state and a set of accepting states are also given.

  This is also called a GOTO table.
Types of FA’s

- **Deterministic FA (DFA):**
  - must “has a unique next state for a transition”
  - and “does not contain $\epsilon$-transitions,” that is, a transition takes $\epsilon$ as the input symbol.

- **Nondeterministic FA (NFA):**
  - either “could have more than one next state for a transition;”
  - or “contains $\epsilon$-transitions.”
  - Example: $aa^*|bb^*$.

![Diagram of NFA](attachment:image.png)
How to execute a DFA

Algorithm:

\[
\begin{align*}
  &s \leftarrow \text{starting state}; \\
  &\text{while there are inputs do} \\
  &\quad s \leftarrow \text{Table}[s, \text{input}] \\
  &\text{end while} \\
  &\text{if } s \in \text{accepting states then ACCEPT else REJECT}
\end{align*}
\]

Example: input “abccabc”. The accepting path:

\[
\begin{align*}
  &0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 3 \rightarrow 1 \rightarrow 2 \rightarrow 3
\end{align*}
\]
How to execute an NFA (informally)

- An NFA accepts an input string $x$ if and only if there is some path in the transition graph initiating from the starting state to some accepting state such that the edge labels along the path spell out $x$.
- Could have more than one path. (Note DFA has at most one.)
- Example: regular expression: $(a|b)^*abb$; input $aabb$

![NFA diagram]

<table>
<thead>
<tr>
<th></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>
</tbody>
</table>

$0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$ Accept!

$0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$ Reject!
From regular expressions to NFA’s

- **Structural decomposition:**
  - atomic items: $\emptyset$, $\epsilon$ and a legal symbol.

![Diagram of NFA's](image)

- $r | s$
- $\epsilon$
- start state for $r$
- $\epsilon$
- NFA for $r$
- $\epsilon$
- start state for $s$
- $\epsilon$
- NFA for $s$
- $\epsilon$
- NFA for $r$
- $\epsilon$
- start state for $r$
- $\epsilon$
- accepting states for $r$

- $r^*$
- $\epsilon$
- start state for $r$
- $\epsilon$
- NFA for $r$
- $\epsilon$
- accepting states for $r$

- $r s$
- $\epsilon$
- start state for $r$
- $\epsilon$
- NFA for $r$
- $\epsilon$
- start state for $s$
- $\epsilon$
- NFA for $s$
- $\epsilon$

- $r s$
- $\epsilon$
- convert all accepting states in $r$ into non accepting states and add $\epsilon$ transitions
This construction produces only $\epsilon$-transitions, never multiple transitions for an input symbol.

It is possible to remove all $\epsilon$-transitions from an NFA and replace them with multiple transitions for an input symbol, and vice versa.
Construction theorems

- **Theorem #1:**
  - Any regular expression can be expressed by an NFA.
  - Any NFA can be converted into a DFA.

- That is, any regular expression can be expressed by a DFA.

- **How to convert an NFA to a DFA:**
  - Find out what is the set of possible states that can be reached from an NFA state using $\epsilon$-transitions.
  - Find out what is the set of possible states that can be reached from an NFA state on an input symbol.

- **Theorem #2:**
  - Every DFA can be expressed as a regular expression.
  - Every regular expression can be expressed as a DFA.
  - DFA and regular expressions have the same expressive power.

- **How about the power of DFA and NFA?**
Definitions: let $T$ be a set of states and $a$ be an input symbol.

- $\epsilon$-closure($T$): the set of NFA states reachable from some state $s \in T$ using $\epsilon$-transitions.
- $\text{move}(T, a)$: the set of NFA states to which there is a transition on the input symbol $a$ from state $s \in T$.
- Both can be computed using standard graph algorithms.
- $\epsilon$-closure($\text{move}(T, a)$): the set of states reachable from a state in $T$ for the input $a$.

Example: NFA for $(\mathit{a} | \mathit{b})^* \mathit{abb}$

- $\epsilon$-closure($\{0\}$) = $\{0, 1, 2, 4, 6, 7\}$, that is the set of all possible start states
- $\text{move}(\{2, 7\}, a) = \{3, 8\}$
Subset construction algorithm

- In the converted DFA, each state represents a subset of NFA states.
  \[ T \xrightarrow{a} \epsilon\text{-closure}(\text{move}(T, a)) \]

- **Subset construction algorithm:**
  Initially, we have an unmarked state labeled with $\epsilon\text{-closure}(\{s_0\})$, where $s_0$ is the starting state.

  ```plaintext
  while there is an unmarked state with the label T do
    ▶ mark the state with the label T
    ▶ for each input symbol a do
      ▶ $U \leftarrow \epsilon\text{-closure}(\text{move}(T, a))$
      ▶ if $U$ is a subset of states that is never seen before
      ▶ then add an unmarked state with the label $U$
    ▶ end for
  end while
  ```

- New accepting states: those contain an original accepting state.
First step:

- \( \epsilon\text{-closure}\{0\} = \{0,1,2,4,6,7\} \)
- \( move(\{0,1,2,4,6,7\}, a) = \{3,8\} \)
- \( \epsilon\text{-closure}(\{3,8\}) = \{0,1,2,3,4,6,7,8\} \)
- \( move(\{0,1,2,4,6,7\}, b) = \{5\} \)
- \( \epsilon\text{-closure}(\{5\}) = \{0,1,2,4,5,6,7\} \)
Example — cont.

states:
- \(A = \{0, 1, 2, 4, 6, 7\}\)
- \(B = \{0, 1, 2, 3, 4, 6, 7, 8, 9\}\)
- \(C = \{0, 1, 2, 4, 5, 6, 7, 10, 11\}\)
- \(D = \{0, 1, 2, 4, 5, 6, 7\}\)
- \(E = \{0, 1, 2, 4, 5, 6, 7, 12\}\)

transition table:

\[
\begin{array}{c|cc}
 & a & b \\
\hline
A & B & D \\
B & B & C \\
C & B & E \\
D & B & D \\
E & B & D \\
\end{array}
\]
Algorithm for executing an NFA

- **Algorithm:** $s_0$ is the starting state, $F$ is the set of accepting states.

  \[
  S \leftarrow \epsilon\text{-closure}(\{s_0\})
  
  \text{while next input } a \text{ is not EOF do}
  
  \quad \triangleright \ S \leftarrow \epsilon\text{-closure}(move(S, a))
  
  \text{end while}
  
  \text{if } S \cap F \neq \emptyset \text{ then ACCEPT else REJECT}
  
- **Execution time is** $O(r^2 \cdot s)$, where
  - $r$ is the number of NFA states, and $s$ is the length of the input.
  - Need $O(r^2)$ time in running $\epsilon\text{-closure}(T)$ assuming using an adjacency matrix representation and a linear-time hashing routine to remove duplicated states.

- **Space complexity is** $O(r^2 \cdot c)$ using a standard adjacency matrix representation for graphs, where $c$ is the cardinality of the alphabets.

- May have slightly better algorithms.
Trade-off in executing NFA’s

- Can also convert an NFA to a DFA and then execute the equivalent DFA.
  - Running time: linear in the input size.
  - Space requirement: linear in the size of the DFA.

- Catch:
  - May get $O(2^r \cdot c)$ DFA states by converting an $r$-state NFA.
  - The converting algorithm may also takes $O(2^r)$ time.

- Time-space tradeoff:

<table>
<thead>
<tr>
<th></th>
<th>space</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFA</td>
<td>$O(r^2 \cdot c)$</td>
<td>$O(r^2 \cdot s)$</td>
</tr>
<tr>
<td>DFA</td>
<td>$O(2^r \cdot c)$</td>
<td>$O(s)$</td>
</tr>
</tbody>
</table>

- If memory is cheap or programs will be used many times, then use the DFA approach;
- otherwise, use the NFA approach.
LEX

- An UNIX utility.
- An easy way to use regular expressions to do lexical analysis.
- Convert your LEX program into an equivalent C program.
- Depending on implementation, may use NFA or DFA algorithms.

\[ \text{file.l} \rightarrow \text{lex file.l} \rightarrow \text{lex.yy.c} \]

\[ \text{lex.yy.c} \rightarrow \text{cc -ll lex.yy.c} \rightarrow \text{a.out} \]

- May produce .o file if there is no main().

\[ \text{input} \rightarrow \text{a.out} \rightarrow \text{output sequence of tokens} \]
LEX formats

- **Source format:**
  - Declarations — a set of regular definitions, i.e., names and their regular expressions.
  - `%%`
  - Translation rules — actions to be taken when patterns are encountered.
  - `%%`
  - Auxiliary procedures

- **Global variables:**
  - `yyleng`: length of current string
  - `yytext`: current string
  - `yylex()`: the scanner routine
LEX formats – cont.

- **Declarations:**
  - variables: using C format
  - manifest constants: using C format; identifiers declared to represent constants
  - regular expressions.

- **Translation rules:**

  $P_1 \{\text{action}_1\}$

  if regular expression $P_1$ is encountered, then action$_1$ is performed.

- **LEX internals:** regular expressions $\rightarrow$ NFA $\rightarrow$ DFA
test.l — Declarations

%
    /* some initial C programs */
#define BEGINSYM 1
#define INTEGER 2
#define IDNAME 3
#define REAL 4
#define STRING 5
#define SEMICOLONSYM 6
#define ASSIGNSYM 7
%
Digit        [0-9]
Letter       [a-zA-Z]
IntLit       {Digit}+
Id           {Letter}({Letter}|{Digit}|_)*
%%
[ \t\n] { /* skip white spaces */}
[Bb][Ee][Gg][Ii][Nn] {return(BEGINSYM);}
{IntLit} {return(INTEGER);}
{Id}
{
    printf("var has %d characters, ", yyleng);
    return(IDNAME);
}
({IntLit}[.]{IntLit})([Ee][+-]?{IntLit})? {return(REAL);}
"[^"\n]*" {stripquotes(); return(STRING);}
";" {return(SEMICOLONSYM);}
";=" {return(ASSIGNSYM);}
. {printf("error --- %s\n", yytext);}
stripquotes()
{
    /* handling string within a quoted string */
    int frompos, topos=0, numquotes = 2;
    for(frompos=1; frompos<yyleng; frompos++)
    {
        yytext[topos++] = yytext[frompos];
    }
    yyleng -= numquotes;
    yytext[yyleng] = '\0';
}

void main()
{
    int i;
    i = yylex();
    while(i>0 && i < 8)
    {
        printf("<%s> is %d\n", yytext, i);
        i = yylex();
    }
}
Sample run

austin% lex test.l
austin% cc lex.yy.c -ll
austin% cat data

Begin
123.3 321.4E21
x := 365;
"this is a string"
austin% a.out < data

<Begin> is 1
<123.3> is 4
<321.4E21> is 4
var has 1 characters, <x> is 3
<=> is 7
<365> is 2
<;> is 6
<br this is a string> is 5
%austin
More LEX formats

- **Special format requirement:**

  \[
  P_1
  \{
  \text{action}_1
  \ldots
  
  \}
  \]

  Note: \{ and \} must indent.

- **LEX special characters (operators):**

  `' ' \[ \] ^ - ? . * + | ( ) $ { } % < >`

- **When there is any ambiguity in matching, prefer**
  - longest possible match;
  - earlier expression if all matches are of equal length.
LEX internals

LEX code:
- regular expression #1 {action #1}
- regular expression #2 {action #2}
- ...

![Diagram of LEX code](image)
LEX internals – cont.

- **How to find a longest possible match if there are many legal matches?**
  - If an accepting state is encountered, do not immediately accept.
  - Push this accepting state and the current input position into a stack and keep on going until no more matches is possible.
  - Pop from the stack and execute the actions for the popped accepting state.
  - Resume the scanning from the popped current input position.

- **How to find the earliest match if all matches are of equal length?**
  - Number the accepting states according to the order in the expressions.
  - If you are in multiple accepting states, execute the action associated with the least indexed accepting state.
Practical considerations

- **key words** v.s. **Reserved word**

  - **key word:**
    - def: word has a well-defined meaning in a certain context.
    - example: FORTRAN, PL/1, ... 
      
      ```
      if if then else = then ;
      id id id
      ```
    - Makes compiler to work harder!

  - **reserved word:**
    - def: regardless of context, word cannot be used for other purposes.
    - example: COBOL, ALGOL, PASCAL, C, ADA, ...
    - task of compiler is simpler
    - reserved words cannot be used as identifiers
    - listing of reserved words is tedious for the scanner, also makes scanner large
    - solutions: treat them as identifiers, and use a table to check whether it is a reserved word.
Multi-character lookahead: how many more characters ahead do you have to look in order to decide which pattern to match?

FORTRAN: lookahead until difference is seen without counting blanks.
- DO 10 I = 1, 15 ≡ a loop statement.
- DO 10 I = 1.15 ≡ an assignment statement for the variable DO10I.

PASCAL: lookahead 2 characters with 2 or more blanks treating as one blank.
- 10..100: needs to look 2 characters ahead to decide this is not part of a real number.

LEX lookahead operator “/”: \( r_1/r_2 \): match \( r_1 \) only if it is followed by \( r_2 \); note that \( r_2 \) is not part of the match.
- This operator can be used to cope with multi-character lookahead.
- How is this implemented in LEX?