Lexical Analyzer — Scanner

ALSU Textbook Chapter 3.1–3.4, 3.6, 3.7, 3.5, 3.8

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 to be used by the parser for syntax analysis.
  - tokens: terminal symbols in grammar.

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

- **Examples:**
  - lexemes: \( \pi = 3.1416 \);
  - tokens: ID ASSIGN FLOAT-LIT SEMI-COL
  - patterns:
    - identifier (variable name) starts with a letter or “_”, and follows by letters, digits or “_”;
    - floating point number starts with a string of digits, follows by a dot, and terminates with another string of digits;
Strings

- **Definitions.**
  - alphabet: a finite set of symbols or characters;
  - string: a finite sequence of symbols chosen from the alphabet;
  - $|S|$: length of a string $S$;
  - empty string: $\epsilon$;

- **Operations.**
  - **concatenation** of strings $x$ and $y$: $xy$
    - $\epsilon x \equiv x \epsilon \equiv x$;
  - **exponentiation**:
    - $s^0 \equiv \epsilon$;
    - $s^i \equiv s^{i-1}s$, $i > 0$. 
Parts of a string: 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: a set of strings over an alphabet.

Operations on languages:

- **union**: $L \cup M = \{ s \mid s \in L \text{ or } s \in M \}$;
- **concatenation**: $LM = \{ st \mid s \in L \text{ and } t \in M \}$;
- $L^0 = \{ \epsilon \}$;
- $L^1 = L$;
- $L^i = LL^{i-1}$ if $i > 1$;
- **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 $L(r)$ which is also called a regular set [Kleene 1956].

- Atomic items of regular expressions and operations on them:

<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>${\epsilon}$ where $\epsilon$ is the empty string</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:

- $a|b$                | $\{a, b\}$ |
- $(a|b)(a|b)$          | $\{aa, ab, ba, bb\}$ |
- $a^*$               | $\{\epsilon, a, aa, aaa, \ldots\}$ |
- $a|a^*b$            | $\{a, b, ab, aab, \ldots\}$ |
Algebraic laws of R.E.

- Assume \( r, s \) and \( t \) are arbitrary regular expressions.

<table>
<thead>
<tr>
<th>Law</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r \mid s = s \mid r )</td>
<td>(union) is commutative</td>
</tr>
<tr>
<td>( r \mid (s \mid t) = (r \mid s) \mid t )</td>
<td>is associative</td>
</tr>
<tr>
<td>( r(st) = (rs)t )</td>
<td>Concatenation is associative</td>
</tr>
<tr>
<td>( r(s \mid t) = rs \mid rt )</td>
<td>Concatenation distributes over union</td>
</tr>
<tr>
<td>( (s \mid t)r = sr \mid tr )</td>
<td></td>
</tr>
<tr>
<td>( \epsilon \mid r = r \mid \epsilon = r )</td>
<td>( \epsilon ) is the identity for union</td>
</tr>
<tr>
<td>( \epsilon r = r \epsilon = r )</td>
<td>( \epsilon ) is the identity for concatenation</td>
</tr>
<tr>
<td>( r^* = (r \mid \epsilon)^* )</td>
<td>( \epsilon ) is guaranteed in a closure</td>
</tr>
<tr>
<td>( r^{**} = r^* )</td>
<td>* is idempotent</td>
</tr>
</tbody>
</table>

- Algebraic structure:
  - Without the Kleene closure operation, it is a semi-ring, i.e., a ring without an inverse for union.
  - With the Kleene closure operation, it is a Kleene algebra.
Regular definitions

- For simplicity, give names to regular expressions and use names later in defining other regular expressions.
  - similar to the idea of macros or subroutine calls without parameters
  - format:
    - name → regular expression

- examples:
  - digit → 0 | 1 | 2 | ... | 9
  - letter → a | b | c | ... | z | A | B | ... | Z

- Notational standards:
  - \{ r \} → r is a regular definition
  - r* | r+ | \epsilon
  - r+ | rr*
  - r? | r | \epsilon
  - [abc] → a | b | c
  - [a-zA-Z] → a | b | c | ... | z

- Example: C variable name
  - [A-Za-z][A-Za-z0-9_]*
  - {letter}{digit}{letter}{digit}{letter}{digit}{
Non-regular sets

- Balanced or nested construct
  - Example:
    
    \[
    \text{if } \text{cond}_1 \text{ then if } \text{cond}_2 \text{ then } \cdots \text{ else } \cdots \text{ else } \cdots
    \]
  - Can be recognized by **context free grammars**.

- 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” “non-constant” amount of information happened in the past cannot be recognized by regular expressions.
Finite state automata (FA)

- FA is a mechanism used to recognize tokens specified by a regular expression.
- **Definition:**
  - A finite set of states, i.e., vertices.
  - A set of transitions, labeled by characters, i.e., labeled directed edges.
  - A starting state, i.e., a vertex with an incoming edge marked with “start”.
  - A set of final (accepting) states, i.e., vertices of concentric circles.
- **Example:** transition graph for the regular expression $(abc^+)^+$
Transition graph and table for FA

- **Transition graph:**

![Transition graph diagram](image)

- **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>\Ø</td>
<td>\Ø</td>
</tr>
<tr>
<td>1</td>
<td>\Ø</td>
<td>{2}</td>
<td>\Ø</td>
</tr>
<tr>
<td>2</td>
<td>\Ø</td>
<td>\Ø</td>
<td>{3}</td>
</tr>
<tr>
<td>3</td>
<td>{1}</td>
<td>\Ø</td>
<td>{3}</td>
</tr>
</tbody>
</table>

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

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

- **Deterministic FA (DFA):**
  - 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.”
  - Note: can have both of the above two.
  - Example: regular expression: $aa^* | bb^*$.

![Diagram of FA's](image-url)
How to execute a DFA

**Algorithm:**

\[
s \leftarrow \text{starting state};
\]

while there are inputs and \( s \) is a legal state do

\[
s \leftarrow \text{Table}[s, \text{input}]
\]

end while

if \( s \in \text{accepting states} \) then ACCEPT else REJECT

**Example:** input: `abccabc`. The accepting path:

\[
0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{c} 3 \xrightarrow{c} 3 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{c} 3
\]
How to execute an NFA (informally) (1/2)

- 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: $aabb$. 

![Diagram of NFA accepting input string aabb](image-url)
How to execute an NFA (informally) (2/2)

- **Goto table:**

<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>$\emptyset$</td>
<td>{2}</td>
</tr>
<tr>
<td>2</td>
<td>$\emptyset$</td>
<td>{3}</td>
</tr>
</tbody>
</table>

- **Two possible traces.**

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!**
Structural decomposition:
- atomic items:
  - $\emptyset$
  - $\epsilon$
  - a legal symbol

Diagram:
- Start state:
- Transition: $a$
From regular expressions to NFA’s (2/3)

- **union**

- **concentration**

  convert all accepting states in $r$ into non accepting states and add $\varepsilon$-transitions
From regular expressions to NFA’s (3/3)

- Kleene closure

![Diagram of NFA for r*]

Compiler notes #2, 20130314, Tsan-sheng Hsu ©
Example: \((a|b)^*((ab)b)\)

- This construction produces only \(\epsilon\)-transitions, and never produce 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.
- Theorem [Thompson 1969]:
  - Any regular expression can be expressed by an NFA.
Converting an NFA to a DFA

- **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.
  - $move(T, a)$: 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($move(T, a)$): the set of states reachable from a state in $T$ for the input $a$.

- **Example:** NFA for $(a|b)^*(((ab)b)$

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

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

- **Subset construction algorithm**: [Rabin & Scott 1959]
  Initially, we have an unmarked state labeled with $\epsilon$-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
    ▶ $U \leftarrow \epsilon$-closure(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:

- \(\epsilon\text{-closure}(\{0\}) = \{0,1,2,4,6,7\}\)
- \(\text{move}(\{0, 1, 2, 4, 6, 7\}, a) = \{3,8\}\)
- \(\epsilon\text{-closure}(\{3,8\}) = \{0,1,2,3,4,6,7,8,9\}\)
- \(\text{move}(\{0, 1, 2, 4, 6, 7\}, b) = \{5\}\)
- \(\epsilon\text{-closure}(\{5\}) = \{0,1,2,4,5,6,7\}\)
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:

<table>
<thead>
<tr>
<th></th>
<th>$a$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>$B$</td>
<td>$D$</td>
</tr>
<tr>
<td>$B$</td>
<td>$B$</td>
<td>$C$</td>
</tr>
<tr>
<td>$C$</td>
<td>$B$</td>
<td>$E$</td>
</tr>
<tr>
<td>$D$</td>
<td>$B$</td>
<td>$D$</td>
</tr>
<tr>
<td>$E$</td>
<td>$B$</td>
<td>$D$</td>
</tr>
</tbody>
</table>
Construction theorems (I)

- **Facts:**
  - Lemma [Thompson 1968]: Any regular expression can be expressed by an NFA.
  - Lemma [Rabin & Scott 1959]
    - Any NFA can be converted into a DFA.
    - By using the Subset Construction Algorithm.

- **Conclusion:**
  - Theorem: Any regular expression can be expressed by a DFA.
  - Note: It is possible to convert a regular expression directly into a DFA [McNaughton & Yamada 1960].
Construction theorems (II)

Facts:
- Theorem [previous slide]: Any regular expression can be expressed by a DFA.
- Lemma [Brzozowski & McCluskey 1963]: Every DFA can be expressed as a regular expression.
  - Define extended FA that has labels of regular expressions on the edges.
  - Repeatly merge states.

Conclusion:
- Theorem: DFA and regular expression have the same expressive power.
- Q: How about the power of DFA and NFA?
Algorithm for executing an NFA

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

  $S \leftarrow \epsilon$-closure($\{s_0\}$)

  while next input $a$ is not EOF do
    ▶ $S \leftarrow \epsilon$-closure($\text{move}(S, a)$)
  end while

  if $S \cap F \neq \emptyset$ 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 $\epsilon$-closure($T$) assuming using an adjacency matrix representation and a constant-time hashing routine with linear-time preprocessing 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 alphabet.

- Have better algorithms by using compact data structures and techniques.
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 $O(2^r)$ DFA states by converting an $r$-state NFA.
  - The converting algorithm may also take $O(2^r \cdot c)$ time in the worst case.

  ▶ For typical cases, the execution time is $O(r^3)$.

- 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 [Lesk 1975].
  - It has been ported to lots of OS’s and platforms.
    - Flex (GNU version), and JFlex and JLex (Java versions).
- An easy way to use regular expressions to specify “patterns”.
- Convert your LEX program into an equivalent C program.
- Depending on implementation, may use NFA or DFA algorithms.

```
file.l → lex file.l → lex.yy.c
lex.yy.c → cc -ll lex.yy.c → a.out
```

- May produce .o file if there is no main().
- input → a.out → output a sequence of tokens
- May have slightly different implementations and libraries.
LEX formats (1/2)

- **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

- **Built-in global variables:**
  - `yytext`: current matched string
  - `yyleng`: length of the current matched string
  - ...

- **Built-in service routines:**
  - `yylex()`: the scanner routine
    - returns the value 0 when EOF is encountered
  - `yywrap()`: called when EOF is encountered
  - `yyerror()`: called when there is an error
  - ...

Compiler notes #2, 20130314, Tsan-sheng Hsu ©
LEX formats (2/2)

- **Declarations:**
  - C language code between `%{` and `%}`.
    - variables;
    - manifest constants, i.e., 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
  - regular expressions $\rightarrow$ DFA
test.l — Declarations

{%
    /* some initial C programs */
#define START_OF_SYMBOLS 1
// 0 is reserved for EOF
#define BEGINSYM 1
#define INTEGER 2
#define IDNAME 3
#define REAL 4
#define SEMICOLONSYM 6
#define ASSIGNSYM 7
#define END_OF_SYMBOLS 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);}
%%
/* some final C programs */
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();
    i = yylex();
    while(i>=START_OF_SYMBOLS && i <= END_OF_SYMBOLS){
        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
<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):

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

- watch out for precedence and associative rules of these operators.
LEX and regular expressions

- **LEX assumes input is a stream of strings, not just one string.**
  - How to know it is the end of a lexeme?

- **LEX allows the specification of multiple regular expressions.**
  - Assume you have regular expressions $R_1$ and $R_2$.
  - Assume $L(R_i)$ is the language, i.e., set of strings, defined by $R_i$.
  - Potential problems or ambiguities:
    - $L(R_1) \cap L(R_2) \neq \emptyset$.
    - $\exists s_1 \in L(R_1)$ such that $s_1$ is a proper prefix of a string $s_2$ and $s_2 \in L(R_2)$.

- **LEX allows “conditional matches”.**
  - Lookahead symbols.
  - Accept a string only if it is followed by another string.
LEX internals

- **LEX code:**
  - regular expression #1 \{action #1\}
  - regular expression #2 \{action #2\}
  - ...
Ambiguity in matching (1/2)

- **Definitions:**
  - for a given prefix of the input output “accept” for more than one pattern;
    - *that is, the languages defined by two patterns have some intersection.*
  - output “accept” for two different prefixes.
    - *An element in a language is a proper prefix of another element in a different language.*

- **When there is any ambiguity in matching, prefer**
  - longest possible match;
  - earlier expression if more than one longest match.

- **White space is needed only when there is a chance of ambiguity.**
Ambiguity in matching (2/2)

- **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 there are more than one longest match?**
  - Assign numbers 1, 2, ... to the accepting states using the order they appear (from top to bottom) in the expressions.
  - If you are in multiple accepting states, execute the action associated with the least indexed accepting state.

- **What does `yylex()` do?**
  - Find the longest possible prefix from the current input stream that can be accepted by “the regular expression” defined.
  - Extract this matched prefix from the input stream and assign its token meaning according to rules discussed.
Lookahead symbols

- **Multi-character lookahead**: how many more characters ahead do you have to look in order to decide which pattern to match?
  - Extensions to regular expression when there are ambiguity in matching.
- **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 it implemented in LEX?
**key word** 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 the scanner larger*
  - *solution: treat them as identifiers, and use a table to check whether it is a reserved word.*