Lexical Analysis

Reinhard Wilhelm
Universität des Saarlandes
wilhelm@cs.uni-sb.de
and
Mooly Sagiv
Tel Aviv University
sagiv@math.tau.ac.il

21. Oktober 2011

Subjects

- Role of lexical analysis
- Regular languages, regular expressions
- Finite-state machines
- From regular expressions to finite-state machines
- A language for specifying lexical analysis
- The generation of a scanner
- Flex
“Standard” Structure, interfaces, mechanisms, where treated

1. source (text)
2. lexical analysis (Vol.2, Ch.2) → finite-state machines
3. tokenized-program
4. syntax analysis (Vol.2, Ch.3) → pushdown automata
5. syntax-tree
6. semantic-analysis (Vol.2, Ch.4) → attribute grammar evaluators
7. decorated syntax-tree
8. optimizations (Vol.3) → abstract interpretation + transformations
9. intermediate rep.

“Standard” Structure cont’d

10. intermediate rep.
11. code-generation (Vol.4) → tree automata + dynamic programming + …
12. machine-program
Lexical Analysis (Scanning)

- **Functionality**
  
  **Input:** program as sequence of characters  
  **Output:** program as sequence of symbols (tokens)

- **Produce listing**
- **Report errors, symbols illegal in the programming language**
- **Screening subtask:**
  - Identify language keywords and standard identifiers
  - Eliminate “white-space”, e.g., consecutive blanks and newlines
  - Count line numbers
  - Construct table of all symbols occurring

---

Automatic Generation of Lexical Analyzers

- The symbols of programming languages can be specified by regular expressions.

- **Examples:**
  - program as a sequence of characters.
  - (alpha (alpha | digit)*) for identifiers
  - ‘‘(*)‘‘ until ‘‘(*)‘‘ for comments

- The recognition of input strings can be performed by a finite-state machine.

- A table representation or a program for the automaton is automatically generated from a regular expression.
Automatic Generation of Lexical Analyzers cont’d

Regular-expression(s) -> FLEX -> input-program -> scanner-program -> tokenized-program

Lexical Analysis

Notations

A language, $L$, is a set of words, $x$, over an alphabet, $\Sigma$.  

- $a_1a_2\ldots a_n$, a word over $\Sigma$, $a_i \in \Sigma$  
- $\varepsilon$, The empty word  
- $\Sigma^n$, The words of length $n$ over $\Sigma$  
- $\Sigma^*$, The set of finite words over $\Sigma$  
- $\Sigma^+$, The set of non-empty finite words over $\Sigma$  
- $xy$, The concatenation of $x$ and $y$

Language Operations

- $L_1 \cup L_2$, Union
- $L_1L_2 = \{x.y | x \in L_1, y \in L_2\}$, Concatenation
- $\overline{L}$, Complement
- $L^n = \{x_1 \ldots x_n | x_i \in L, 1 \leq i \leq n\}$
- $L^*$, $= \bigcup_{n \geq 0} L^n$, Closure
- $L^+$, $= \bigcup_{n \geq 1} L^n$
Regular Languages

Defined inductively

- $\emptyset$ is a regular language over $\Sigma$
- $\{\varepsilon\}$ is a regular language over $\Sigma$
- For all $a \in \Sigma$, $\{a\}$ is a regular language over $\Sigma$
- If $R_1$ and $R_2$ are regular languages over $\Sigma$, then so are:
  - $R_1 \cup R_2$,
  - $R_1 R_2$, and
  - $R_1^*$

Regular Expressions and the Denoted Regular Languages

Defined inductively

- $\emptyset$ is a regular expression over $\Sigma$ denoting $\emptyset$,
- $\varepsilon$ is a regular expression over $\Sigma$ denoting $\{\varepsilon\}$,
- For all $a \in \Sigma$, $a$ is a regular expression over $\Sigma$ denoting $\{a\}$,
- If $r_1$ and $r_2$ are regular expressions over $\Sigma$ denoting $R_1$ and $R_2$, resp., then so are:
  - $(r_1 | r_2)$, which denotes $R_1 \cup R_2$,
  - $(r_1 r_2)$, which denotes $R_1 R_2$, and
  - $(r_1)^*$, which denotes $R_1^*$.

Metacharacters, $\emptyset, \varepsilon, (, ), [, , \ast$ don’t really exist, are replaced by their non-underlined versions.
Clash between characters in $\Sigma$ and metacharacters $\{(, )|[, ]\ast$
Example

<table>
<thead>
<tr>
<th>Expression</th>
<th>Language</th>
<th>Example words</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a</td>
<td>b$</td>
<td>${a, b}$</td>
</tr>
<tr>
<td>$ab^*a$</td>
<td>${a}{b}^*{a}$</td>
<td>$aa, aba, abba, abbbba, \ldots$</td>
</tr>
<tr>
<td>$(ab)^*$</td>
<td>${ab}^*$</td>
<td>$\varepsilon, ab, abab, \ldots$</td>
</tr>
<tr>
<td>abba</td>
<td>${abba}$</td>
<td>$abba$</td>
</tr>
</tbody>
</table>

Regular Expressions for Symbols (Tokens)

Alphabet for the symbol classes listed below:

$\Sigma =$

- integer-constant
- real-constant
- identifier
- string
- comments
- matching-parentheses?
Automata

In the following, we will meet different types of automata.

**Automata**

- process some *input*, e.g. strings or trees,
- make *transitions* from configurations to configurations;
- *configurations* consist of (the rest of) the input and some *memory*;
- the *memory* may be small, just one variable with finitely many values,
- but the memory may also be able to grow without bound, adding and removing values at one of its ends;
- the type of memory an automaton has determines its ability to *recognize* a class of languages,
- in fact, the more powerful an automaton type is, the better it is in *rejecting* input.

---

Finite State Machine

The simplest type of automaton, its memory consists of only one variable, which can store one out of finitely many values, its *states*,
A Non-Deterministic Finite-State Machine (NFSM)

\[ M = (\Sigma, Q, \Delta, q_0, F) \]

- \( \Sigma \) — finite alphabet
- \( Q \) — finite set of states
- \( q_0 \in Q \) — initial state
- \( F \subseteq Q \) — final states
- \( \Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q \) — transition relation

May be represented as a transition diagram

- Nodes — States
- \( q_0 \) has a special “entry” mark
- final states doubly encircled
- An edge from \( p \) into \( q \) labeled by \( a \) if \((p, a, q) \in \Delta\)

---

Example: Integer and Real Constants

<table>
<thead>
<tr>
<th>( \text{Di} )</th>
<th>( {0, 1, \ldots, 9} )</th>
<th>( \varepsilon )</th>
<th>( E )</th>
<th>( \varepsilon )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{1, 2}</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>{1}</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>{2}</td>
<td>{3}</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>{4}</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>{4}</td>
<td>0</td>
<td>{5}</td>
<td>{7}</td>
</tr>
<tr>
<td>5</td>
<td>{6}</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>{7}</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>\emptyset</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\( \text{Di} = \{0, 1, 2, 3, 4, 5, 6, 7\} \)

\( q_0 = 0 \)

\( F = \{1, 7\} \)
Finite-state machines — Scanners

Scanners

- get an input string (a sequence of words),
- start in their initial state,
- attempt to find the end of the next word,
- when found, restart in their initial state with the rest of the input,
- terminate when the end of the input is reached or an error is encountered.

Maximal Munch strategy

Find longest prefix of remaining input that is a legal symbol.

- first input character of the scanner — first “non-consumed” character,
- in final state, and exists transition under the next character: make transition and remember position,
- in final state, and exists no transition under the next character: Symbol found,
- actual state not final and no transition under the next character: backtrack to last passed final state
  - There is none: Illegal string
  - Otherwise: Actual symbol ended there.

Warning: Certain overlapping symbol definitions will result in quadratic runtime: Example: (a|a*; )
Lexical Analysis

Other Example Automata

- integer-constant
- real-constant
- identifier
- string
- comments

Lexical Analysis

The Language Accepted by a Finite-State Machine

- $M = \langle \Sigma, Q, \Delta, q_0, F \rangle$
- For $q \in Q$, $w \in \Sigma^*$, $(q, w)$ is a configuration
- The binary relation step on configurations is defined by:

  $(q, aw) \vdash_M (p, w)$

  if $(q, a, p) \in \Delta$

- The reflexive transitive closure of $\vdash_M$ is denoted by $\vdash_M^*$
- The language accepted by $M$

  $L(M) = \{ w \mid w \in \Sigma^* \mid \exists q_f \in F : (q_0, w) \vdash_M^* (q_f, \varepsilon) \}$
From Regular Expressions to Finite Automata

Theorem

(i) For every regular language $R$, there exists an NFSM $M$, such that $L(M) = R$.
(ii) For every regular expression $r$, there exists an NFSM that accepts the regular language defined by $r$.

A Constructive Proof for (ii) (Algorithm)

- A regular language is defined by a regular expression $r$
- Construct an “NFSM” with one final state, $q_f$, and the transition
  \[
  q_f \xrightarrow{r} q_f
  \]
- Decompose $r$ and develop the NFSM according to the following rules
  \[
  q \xrightarrow{r_1 r_2} p \quad \Rightarrow \quad q \xrightarrow{r_1} q \xrightarrow{r_2} p
  \]
  \[
  q \xrightarrow{r_1} p \quad \Rightarrow \quad q \xrightarrow{r_1} q \xrightarrow{r_2} p
  \]
  \[
  q \xrightarrow{r} p \quad \Rightarrow \quad q \xrightarrow{\varepsilon} q_{1} \xrightarrow{r} q_{2} \xrightarrow{\varepsilon} p
  \]
- until only transitions under single characters and $\varepsilon$ remain.
Examples

- $a(a|0)^*$ over $\Sigma = \{a, 0\}$
- Identifier
- String

Nondeterminism

- Several transitions may be possible under the same character in a given state
- $\varepsilon$-moves (next character is not read) may “compete” with non-$\varepsilon$-moves.
- Deterministic simulation requires “backtracking”
Deterministic Finite-State Machine (DFSM)

- No $\varepsilon$-transitions
- At most one transition from every state under a given character, i.e. for every $q \in Q$, $a \in \Sigma$,

$$|\{q' \mid (q, a, q') \in \Delta\}| \leq 1$$

From Non-Deterministic to Deterministic Automata

**Theorem**

For every **NFSM**, $M = \langle \Sigma, Q, \Delta, q_0, F \rangle$ there exists a **DFSM**, $M' = \langle \Sigma, Q', \delta, q'_0, F' \rangle$ such that $L(M) = L(M')$.

A **Scheme of a Constructive Proof (Subset Construction)**

Construct a DFSM whose states are **sets of states** of the NFSM. The DFSM simulates all possible transition paths under an input word in parallel.

Set of new states

$$\{\{q_1, \ldots, q_n\} \mid n \geq 1 \land \exists w \in \Sigma^* : (q_0, w) \vdash^*_M (q_i, \varepsilon)\}$$

\[\vdash^*_M (q_i, \varepsilon)\]
The Construction Algorithm

Used in the construction: the set of ε-Successors,
\[ \varepsilon-SS(q) = \{ p \mid (q, \varepsilon) \vdash^*_M (p, \varepsilon) \} \]
- Starts with \( q_0 = \varepsilon-SS(q_0) \) as the initial DFSM state.
- Iteratively creates more states and more transitions.
- For each DFSM state \( S \subseteq Q \) already constructed and character \( a \in \Sigma \),
  \[ \delta(S, a) = \bigcup_{q \in S} \bigcup_{(q, a, p) \in \Delta} \varepsilon-SS(p) \]
  if non-empty
  add new state \( \delta(S, a) \) if not previously constructed;
  add transition from \( S \) to \( \delta(S, a) \).
- A DFSM state \( S \) is accepting (in \( F' \)) if there exists \( q \in S \) such that \( q \in F \)

Example: \( a(a|0)^* \)
DFSM minimization

DFSM need not have minimal size, i.e. minimal number of states and transitions.

$q$ and $p$ are undistinguishable (have the same acceptance behavior) iff

for all words $w$ $(q, w) \vdash_M^*$ and $(p, w) \vdash_M^*$ lead into either $F'$ or $Q' = F'$.

Undistinguishability is an equivalence relation.
Goal: merge undistinguishable states $\equiv$ consider equivalence classes as new states.

DFSM minimization algorithm

- Input a DFSM $M = \langle \Sigma, Q, \delta, q_0, F \rangle$
- Iteratively refine a partition of the set of states, where each set in the partition consists of states so far undistinguishable.
- Start with the partition $\Pi = \{F, Q - F\}$
- Refine the current $\Pi$ by splitting sets $S \in \Pi$ if there exist $q_1, q_2 \in S$ and $a \in \Sigma$ such that
  - $\delta(q_1, a) \in S_1$ and $\delta(q_2, a) \in S_2$ and $S_1 \neq S_2$
- Merge sets of undistinguishable states into a single state.
Example: $a(a|0)^*$

A Language for specifying lexical analyzers

$$(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$$

$$(\varepsilon.(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*)$$

$$(\varepsilon|E(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)))$$
Descriptional Comfort

Character Classes:
Identical meaning for the DFSM (exceptions!), e.g.,
\( le = a - z \ A - Z \)
\( di = 0 - 9 \)
Efficient implementation: Addressing the transitions indirectly through an array indexed by the character codes.

Symbol Classes:
Identical meaning for the parser, e.g.,
Identifiers
Comparison operators
Strings

Descriptional Comfort cont’d

Sequences of regular definitions:

\[
\begin{align*}
A_1 &= R_1 \\
A_2 &= R_2 \leftarrow A_1 \\
\vdots \\
A_n &= R_n \leftarrow A_{1.1} \cdots A_{n.1}
\end{align*}
\]
Sequences of Regular Definitions

Goal: Separate final states for each definition

1. Substitute right sides for left sides
2. Create an NFSM for every regular expression separately;
3. Merge all the NFSMs using ε transitions from the start state;
4. Construct a DFSM;
5. Minimize starting with partition

\[ \{F_1, F_2, \ldots, F_n, Q - \bigcup_{i=1}^{n} F_i\} \]
%{
    extern int line_number;
    extern float atof(char *);
}%

DIG    [0-9]
LET    [a-zA-Z]

[=#+<>+-*]    { return(*yytext); }

({DIG}+)    { yylval.intc = atoi(yytext); return(301); }

({DIG}\.{DIG}+(E(\+|\-)?{DIG}+)?)
    { yylval.realc = atof(yytext); return(302); }

"\"(\"\.|[^"\\])\""    { strcpy(yylval.strc, yytext);
    return(303); }

"<="    { return(304); }

:    { return(305); }
\
\n    { return(306); }

ARRAY    { return(307); }
BOOLEAN   { return(308); }
DECLARE   { return(309); }

\{LET\}\{LET\}\{DIG\}*
    { yylval.symb = look_up(yytext);
    return(310); }

\t    { /* White space */ } 
\n    { line_number++; }
.    { fprintf(stderr,
            "WARNING: Symbol '%c' is illegal, ignored!\n", yytext); }

%}