Lexical Analysis

CS2210
Lecture 3

Administrivia

Reading: Aho Ch. 2 & 3 by 9/13

Scanner

source

lexical analyzer

token

get next token

symbol table

parser

CS2210: Compiler Design 2004/05
Lexical Analysis Tasks

- Read input characters produce tokens
- Strip comments and whitespace
- Correlate error message with program source (e.g., line numbers)
- Preprocessor functions (if in language and not separate tool)
- Scanning vs lexical analysis
  - Used interchangeably
  - OR: Lexical Analyzer = 1. Scanner and 2. Lexical Analysis

Why separate lexical analysis phase?

- Keeps parser simpler
  - Don’t have to deal with comments etc
- Efficiency
  - Can read input in chunks and buffer
  - Pass only relevant information on
- Portability
  - Can deal with representation of special symbols directly (e.g., Pascal’s ↑)

Basic Definitions

- Pattern
  - Informal description of strings
- Token
  - Represent some lexical unit (e.g., keyword, identifier etc.)
  - Treated as non-terminals in the parser
- Lexeme
  - Character sequence that matches a token
Example

<table>
<thead>
<tr>
<th>Token</th>
<th>Sample Lexeme</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>const</td>
<td>const</td>
<td>const</td>
</tr>
<tr>
<td>if</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>relop</td>
<td>&lt;=, &gt;=, =, &lt;&gt;</td>
<td>&lt;= or &lt;&gt;</td>
</tr>
<tr>
<td>id</td>
<td>Pi, count, D2</td>
<td>Letter followed by letters and digits</td>
</tr>
<tr>
<td>num</td>
<td>3.14, 1, 07, 02e23</td>
<td>Numeric constant</td>
</tr>
<tr>
<td>literal</td>
<td>&quot;core dumped&quot;</td>
<td>Any character between &quot; and &quot; except &quot;</td>
</tr>
</tbody>
</table>

Token Attributes
- Some tokens represent multiple lexemes
- Specific information kept in attribute
- Attribute values kept in symbol table
  - E.g., line number (for error reporting)
- Tokens used for parsing
- Attributes influence translation

Token Specification
- Alphabet $\Sigma$
- String = finite sequence of characters from $\Sigma$, including the empty string $\varepsilon$
- Language = set of strings over $\Sigma$
- Operations:
  - Concatenation $xy$
  - Union $x | y$
Languages

- \( L \cup M \), union of \( L \) and \( M \)
- \( LM \), concatenation
- Kleene closure \( L^* \) (zero or more concatenations of \( L \))
- Positive closure \( L^+ \) (1 or more concatenations of \( L \))

Regular Expressions

- Formal way to describe patterns for lexemes / tokens
- Regular expressions over \( \Sigma \)
  - \( \varepsilon \) denotes \( \{\varepsilon\} \)
  - For \( a \in \Sigma \): \( a \) is a RE that denotes \( \{a\} \)
  - For \( r,s \) REs:
    - \( (r)\) is a RE denoting \( L(r) \)
    - \( (r)s \) is a RE denoting \( L(r)L(s) \)
    - \( r | (s) \) is a RE denoting \( L(r) \cup L(s) \)
    - \( r^* \) is a RE denoting \( (L(r))^* \)

RE properties / conventions

- Algebraic properties:
  - \( r|s = s|r \) | commutative
  - \( r(s) = (r)s \) | associative
  - \( (rs)t = r(st) \) | concatenation associative
  - \( r(s)t = rs|rt \) | concatenation distributes over |
  - \( \varepsilon r = r, r \varepsilon = r \) | \( \varepsilon \) identity
  - \( r^* = (r)\) | relation * and \( \varepsilon \)
  - \( r^+ = r^* \) | idempotent
Shorthands
- $r^+ = r^*$
- $r? = r | \varepsilon$
- Character classes: $[abc] = a|b|c$, for $a, b, c \in \Sigma$
  similarly $[a-z] = a|...|z$

Token recognition
- REs can be efficiently recognized by finite automata
- Constructed starting from transition diagrams
  - In practice: automatically generated by tools driven by specification language

Transition Diagram Example (1)
- Sample grammar:
  stmt -> if expr then stmt
  | if expr then stmt else stmt
  | \varepsilon
  expr -> term relop term
  | term
  term -> id
  | num
Transition Diagram Example (2)

if -> if
then -> then
else -> else
relop -> < | <= | = | <> | > | >=
id -> letter (letter | digit)*
um -> digit+ (. digit*)? (E(+|-)? digit*)?
letter -> A|B|...|Z|a|...|z
digit -> 0|...|9
delim -> * | \t | \n
Transition Diagram Example (3)

<table>
<thead>
<tr>
<th>RE</th>
<th>Token</th>
<th>Attribute value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ws</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>if</td>
<td>if</td>
<td></td>
</tr>
<tr>
<td>then</td>
<td>then</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td>else</td>
<td></td>
</tr>
<tr>
<td>id</td>
<td>id</td>
<td>pointer to table entry</td>
</tr>
<tr>
<td>num</td>
<td>num</td>
<td>pointer to table entry</td>
</tr>
</tbody>
</table>
<   | relop | LT          |
<=  | relop | LE          |
=   | relop | EQ          |
<   | relop | NE          |
>   | relop | GT          |
>=  | relop | GE          |

Transition diagram

- >=
  - Relop see board
- id: letter or digit

return(gettoken(),install_id())
Practical Considerations

- Distinguishing keywords from identifiers
  - Seed symbol table with keywords
  - Saves many states
  - General idea: use symbol table to reduce states
    (used in parser tool -> simplifies grammar)
- unget() operation to return characters to the input buffer
- Look for frequent tokens first
  - Compose transition diagrams intelligently
  - Saves recognition time

From RE to Recognizer

- Finite automata (NFA or DFA) recognize precisely regular languages
- NFAs and DFAs recognize the same language class
  - NFAs easy to construct directly from RE
  - DFAs faster but can be (much) larger
  - Algorithmic conversion from NFA to DFA possible

Nondeterministic Finite Automata

- 5-tuple (S, Σ, M, S₀, F)
  - S = set of states
  - Σ alphabet
  - M transition function: S × Σ ∪ {ε} -> S
  - S₀ ∈ S a start state
  - F ⊆ a set of final states
Nondeterministic Finite Automata

- 5-tuple \((S, \Sigma, M, S_0, F)\)
  - \(S\) = set of states
  - \(\Sigma\) alphabet
  - \(M\) transition relation: \(S \times \Sigma \cup \{\varepsilon\} \rightarrow 2^S\)
  - \(S_0 \subseteq S\) a start state
  - \(F \subseteq\) a set of final states

NFA Example

```
\[
\begin{array}{c}
\text{start} \\
0 \\
\text{1} \\
\text{2} \\
\text{3} \\
\text{4}
\end{array}
\]
```

- \((a, b)^*\)

Thompson’s Algorithm

- Input: RE \(r\) over \(\Sigma\)
- Output: NFA accepting \(L(r)\) input: RE \(r\) over \(\Sigma\)
- See board:
  - NFA for \(s\)
  - NFA for \(a \in \Sigma\)
  - NFA for \(s|t\)
  - NFA for \(st\)
  - NFA for \(s^*\)
  - NFA for \((s)\)
From NFA to DFA

- DFA = special NFA
  - w/o ε transitions
  - for each state s and input symbol a there is at most 1 edge labeled a leaving s

Büchi’s Algorithm (aka subset construction)

- Special operations: (T set of NFA states)
  - ε-closure(s) = set of NFA states reachable from NFA state s on ε-transitions alone
  - ε-closure(T) = set of NFA states reachable from some NFA state s in T on ε-transitions alone
  - move(T, a) = set of NFA states to which there is a transition on input symbol a from some NFA state s in T

Initially, ε-closure(s₀) only state in Dstate, unmarked

while unmarked state T in Dstates do
  mark T;
  for each input symbol a do
    U := ε-closure(move(T, a))
    if U is not in Dstates then
      add U as an unmarked state to Dstates
    Dtran[T, a] := U
  end
end
Büchi’s Algorithm - Example

NFA for \((a|b)^*abb\) transition table

<table>
<thead>
<tr>
<th>State</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Lexical Analysis in Practice

Implementation alternatives:
- Generate NFA, convert to DFA and implement with transition tables
  - Lazy DFA construction used in pattern matching tools (e.g., egrep)
- Simulate NFA directly
  - On the fly subset generation (algorithm 3.4 in Dragon book)
- Generate DFA directly
  - Cf. Dragon book

Tradeoffs

- \(r\) regular expression, \(x\) input string
- NFA
  - \(O(|r|)\) space
  - \(O(|r| \times |x|)\) time
- DFA
  - \(O(2^{|r|})\) space, usually much less in practice
  - \(O(|x|)\) time
What actual tools do
- GNU flex
  - Builds NFA
  - Converts to DFA
  - Generates transition tables and driver (similar to figure 3.16 in Dragon Book)
- Important optimizations
  - State minimization ("equivalence classes")
  - Table compression

DFA State Minimization
- Idea
  - Merge indistinguishable states
  - s distinguished from t by string w \iff
    starting from s w leads to accepting state
    but t w to non-accepting state (or vice versa)

State Minimization Algorithm
Input DFA M with states S, alphabet \( \Sigma \),
transitions for all inputs \& states, F final states, s, start state
Construct initial partition \( \Pi_{\text{in}} : F \) and S-F
\begin{algorithm}
\begin{algorithmic}
\STATE \( \Pi := \Pi_{\text{in}} \)
\STATE \( \Pi_{\text{new}} := \text{partition}(\Pi) \)
\WHILE {\( \Pi = \Pi_{\text{new}} \)}
\STATE \( \Pi_{\text{new}} := \Pi \)
\ENDWHILE
\STATE Choose one state from each group as representative, s, for the start state for \( M' \)
\STATE Remove all dead states d from \( M' \) (i.e., d not accepting, transitions from d to d on all input symbols). Remove unreachable states.
\end{algorithm}
\end{algorithm}
Partition Algorithm

\begin{enumerate}
\item for each group $G$ of $\Pi$ do
\item partition $G$ into subgroups such that $s$ and $t$ are in the same subgroup $\iff$ for all input symbols $a$ stats $s$ and $t$ have transitions on $a$ to states in the same group of $\Pi$
\item replace $G$ in $\Pi_{raw}$ by the set of all formed subgroups
\end{enumerate}

Example: homework

Table Compression (1)

- Want to implement table as 2-dim array ($t[s,a] = s'$)
- 1-dim array of linked list with outgoing transitions smaller but slow
- Fast combined implementation with 4 special arrays

Table Compression (2)

\begin{verbatim}
nextstate(s,a) = next[base[s]+a] if check(base[s]+a) = s
otherwise s = default[s] and repeat
\end{verbatim}