CSE302: Compiler Design

Instructor: Dr. Liang Cheng
Department of Computer Science and Engineering
P.C. Rossin College of Engineering & Applied Science
Lehigh University

February 13, 2007
Outline

- Recap
  - The lexical-analyzer generator Lex
- Implementing lexical-analyzer generators
- Summary and homework
Overview of Flex

- Flex is a scanner generator
  - Input is description of patterns and actions
  - Output is a C program which contains a function yylex() which when called matches patterns and performs actions per input
- Execute the unix command “man flex” for full information
Overview of Flex

- Compile using Flex tool
  - Results in C code
- Compile using C compiler
  - Link to the flex library (-lfl)
- Run the executable and recognize tokens

```
Lex source program: lex.l  →  Lex compiler  →  lex.yy.c

lex.yy.c  →  C compiler  →  a.out

Input stream  →  a.out  →  sequence of tokens
```
% {
declarations
%
} regular definitions
%%
translation rules
%%
auxiliary procedures/functions
Commands

- `flex <prog_name>.l`
  - On CSE Department Suns, flex is in `/usr/sfw/bin/flex`
- `gcc -o sample lex.yy.c -lfl`
- `sample < input.text`
  - flex generates a main routine that is not needed when parsing with Yacc-generated parser
Some Functions and Variables

- **yylex()**
  - The primary function generated
- **input()**
  - Returns the next char from the input
- **unput(int c)**
  - Returns char c to input
- **yyval // Used to pass values to parser**
- **yytext // String with token from input**
- **yylen // Length of string**
- **yyin // File handle**
  - yyin = fopen(args[0], “r”)
Regular Expressions For Tokens

\[\text{ws} \rightarrow (\text{blank|tab|newline})^+\]

\[
\begin{align*}
\text{digit} & \rightarrow [0-9] \\
\text{digits} & \rightarrow \text{digit}^+ \\
\text{number} & \rightarrow \text{digits ( . digits)? ( E [+-]? digits )?} \\
\text{letter} & \rightarrow [A-Za-z] \\
\text{id} & \rightarrow \text{letter ( letter | digit )*} \\
\text{if} & \rightarrow \text{if} \\
\text{then} & \rightarrow \text{then} \\
\text{else} & \rightarrow \text{else} \\
\text{relop} & \rightarrow < | > | <= | >= | = | <> \\
\end{align*}
\]
Example Lex Source Programs

% { /* definitions of constants
    LT, LE, EQ, NE, GT,
    GE, IF, THEN, ELSE,
    ID, NUMBER, RELOP */
%

% { ws }        /* no action */ }
if { return("IF"); }
then { return("THEN"); }
else { return("ELSE"); }
{id} { return("ID"); }
{number} { return("NUMBER"); }
"<" { return("RELOP"); }
"<=" { return("RELOP"); }
"=" { return("RELOP"); }
"<>" { return("RELOP"); }
">" { return("RELOP"); }
">=" { return("RELOP"); }

Instructor: Dr. Liang Cheng  CSE302: Compiler Design  02/13/07
Conflict Resolution in Lex

- When several prefixes of the input match one or more patterns
  - Always prefer a longer prefix to a shorter prefix
  - If the longest possible prefix matches two or more patterns, prefer the pattern listed first in the Lex source program
    - if \( i > 0 \) then \( i = 1 \) else \( i = 0 \)
Outline

- Recap
  - The lexical-analyzer generator Lex
- Implementing lexical-analyzer generators
- Summary and homework
Implementing Lexical-Analyzer Generators

- Regular expressions $\rightarrow$ Nondeterministic finite automata
- Nondeterministic finite automata $\rightarrow$ Deterministic finite automata
- Deterministic finite automata $\rightarrow$ A lexer

- Regular expressions $\rightarrow$ Deterministic finite automata
- Deterministic finite automata $\rightarrow$ A lexer
A transition table based approach

\[ s = 1; \]
\[ \text{while} \ ( s \neq \text{acceptState and } s \neq \text{errorState}) \ { \}
\[ c = \text{next input character}; \]
\[ s = T[ s, c ]; \]
\} 

<table>
<thead>
<tr>
<th>States</th>
<th>States representing transitions ( T( s, c ) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s )</td>
<td>Characters in the alphabet ( c )</td>
</tr>
</tbody>
</table>
Deterministic Finite Automata

- A finite set of states $S$.
- A set of input symbols or characters $\Sigma$ as the input alphabet.
  - The empty string $\varepsilon$ is not a member of $\Sigma$
- A transition function $T: S \times \Sigma \rightarrow S$ that gives a next state for each state and each symbol/character
- A state $s_0$ from $S$ as the initial state.
- A set of states $F$ that is a subset of $S$ as the final/accepting states.
DFA/NFA Accepting Regular Expressions

- The language or regular expression accepted by a DFA or NFA $D$, written as $L(D)$

- The set of strings of symbols $c_1c_2...c_n$ with each $c_i$ such as there exist states $s_1 = \pi(s_0, c_1), ..., s_n = \pi(s_{n-1}, c_n)$, with $s_n$ an accepting/final state.
Nondeterministic Finite Automata

- A finite set of states $S$.
- A set of input symbols or characters $\Sigma$ as the input alphabet.
  - The empty string $\varepsilon$ is not a member of $\Sigma$
- A transition function $T: S \times \Sigma \rightarrow S$ that gives a set of next states for each state and each symbol/character in $\Sigma \cup \{\varepsilon\}$
- A state $s_0$ from $S$ as the initial state.
- A set of states $F$ that is a subset of $S$ as the final/accepting states.
Implementing Lexical-Analyzer Generators

- Regular expressions → Nondeterministic finite automata
- Nondeterministic finite automata → Deterministic finite automata
- Deterministic finite automata → A lexer

- Regular expressions → Deterministic finite automata
- Deterministic finite automata → A lexer
MYT Algorithm

- Constructing an NFA from a regular expression \( r \) by McNaughton-Yamada-Thompson algorithm
  - Organizing \( r \) into its constituent sub-expressions
    - Sub-expressions with no operators
    - Operators
  - Using basic rules to construct NFA for Sub-expressions with no operators
  - Using inductive rules to construct larger NFA based on the constructed NFA for operations of sub-expressions
Basic Rules to Construct NFA

- For expression $\varepsilon$
  
  ![Diagram](image1)

- For any subexpression $a$, i.e. $\{a\}$
  
  ![Diagram](image2)
Inductive Rules to Construct Larger NFA For Operations

- Assume $M(s)$ and $M(t)$ are NFA for regular expressions $s$ and $t$, respectively
  - Parenthesis operation $r=(s)$
    - Use the NFA $M(s)$ as $M(r)$
  - Union operation $r=s|t$
  - Concatenation operation $r=st$
  - Repetition operation $r=s^*$
Implementing Lexical-Analyzer Generators

- Regular expressions $\rightarrow$ Nondeterministic finite automata
- Nondeterministic finite automata $\rightarrow$ Deterministic finite automata
- Deterministic finite automata $\rightarrow$ A lexer
Conversion of NFA to DFA

Subset construction algorithm

- Input: An NFA $N$
- Output: A DFA $D$ accepting the same language as $N$
- Algorithm: construct a transition table $D_{tran}$ corresponding to $D$

Initially, $\varepsilon$-closure($s_0$) is the only state in $D_{states}$, and it is unmarked;
while ( there is an unmarked state $T$ in $D_{states}$ ) {
    mark $T$;
    for ( each input symbol $a$ ) {
        $U = \varepsilon$-closure(move($T, a$));
        if ( $U$ is not in $D_{states}$ ) add $U$ as an unmarked state to $D_{states}$;
        $D_{tran}[T, a] = U$;
    }
}
\( \varepsilon\text{-closure}(s) \) and \( \varepsilon\text{-closure}(T) \)

- \( \varepsilon\text{-closure}(s) \): a set of NFA states reachable from NFA state \( s \) on \( \varepsilon \)-transitions alone
- \( \varepsilon\text{-closure}(T) \): a set of NFA states reachable from some NFA state \( s \) in the set \( T \) on \( \varepsilon \)-transitions alone
  
  \[ \bigcup_{s \in T} \varepsilon\text{-closure}(s) \]

push all states of \( T \) onto stack;
initialize \( \varepsilon\text{-closure}(T) \) to \( T \);

\textbf{while} ( stack is not empty ) {
  pop \( t \), the top element, off stack;
  \textbf{for} ( each state \( u \) with an edge from \( t \) to \( u \) labeled \( \varepsilon \) )
    \textbf{if} ( \( u \) is not in \( \varepsilon\text{-closure}(T) \) ) {
      add \( u \) to \( \varepsilon\text{-closure}(T) \); push \( u \) onto stack;
    }
}
move(T, a)

- A set of NFA states to which there is a transition on input symbol a from some state $s$ in T
Examples
Outline

- Recap
  - The lexical-analyzer generator Lex
- Implementing lexical-analyzer generators
- Summary and homework
Homework (Due on 02/19 at 11:55 PM)

5.1. (10 points). Using flex and based on the Example 3.8 (pages 128-129 in the textbook), generate a lexer that scans the following input stream and outputs the following output stream.

- Input stream: if i>0 then i=1 else i=0

Please provide a readme file explaining how you generate and test your lexer.

5.2. Conversion of a NFA to a DFA will be posted at the Blackboard.