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

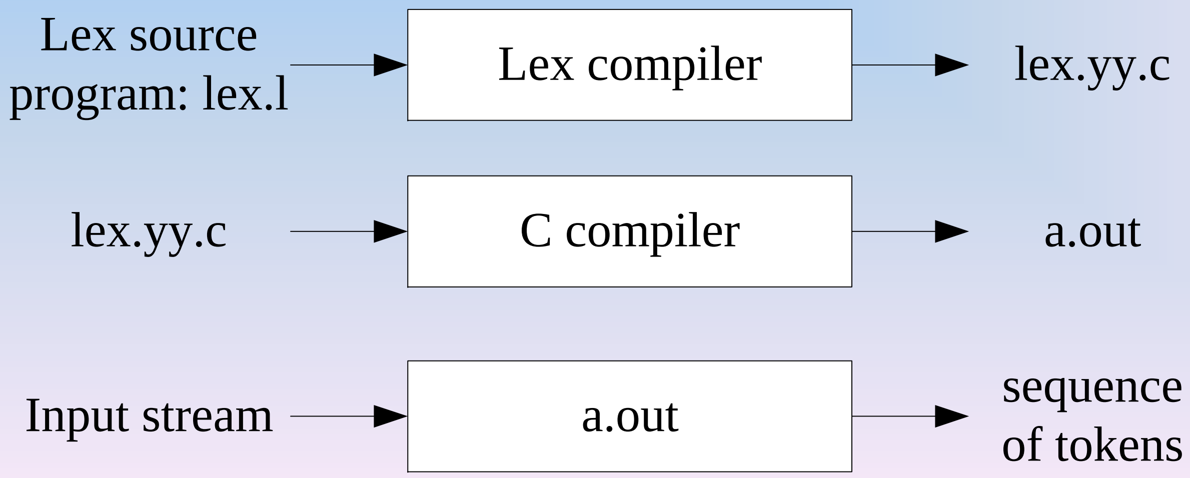
- Recap
  - The lexical-analyzer generator Lex
- Implementing lexical-analyzer generators
- Summary and homework

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



Flex Source Program Format

```
%{
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
- yylval // Used to pass values to parser
- yytext // String with token from input
- yyleng // Length of string
- yyin // File handle
  - yyin = fopen(args[0], “r”)

Regular Expressions For Tokens

ws → (**blank**|**tab**|**newline**)+

digit →	[0-9]
digits →	$digit^+$
number →	*digits* (. *digits*)? (E [+-]? *digits*)?
letter →	[A-Za-z]
id →	*letter* (*letter* | *digit*)*
if →	if
then →	then
else →	else
relop →	< | > | <= | >= | = | <>

Example Lex Source Programs

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

delim     [ \t\n]
ws        {delim}+
letter    [A-Za-z]
digit     [0-9]
id        {letter}({letter}|{digit})*
number
{digit}+(\.{digit}+)?(E[+-]?{digit}+)?
```

```
%%
{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");}
%%
```

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

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

Deterministic Finite Automata → A Lexer

- A transition table based approach
  - *s* = 1;

    while(*s*!=acceptState and *s*!=errorState) {

    c = next input character;

    s = T[*s*,*c*];

    }

	Characters in the alphabet *c*
States *s*	States representing transitions T(*s*,*c*)

Deterministic Finite Automata

- A finite set of states *S*.
- A set of input symbols or characters Σ as the input alphabet.
  - The empty string ε is not a member of Σ
- 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 $s_1 = T(s_0, c_1)$, ..., $s_n = T(s_{n-1}, c_n)$, with s_n an accepting/final state

Nondeterministic Finite Automata

- A finite set of states S.
- A set of input symbols or characters Σ as the input alphabet.
  - The empty string ε is not a member of Σ
- 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

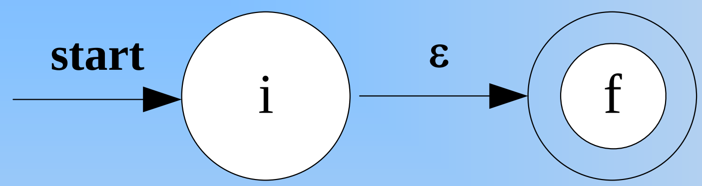
- Regular expressions → Deterministic finite automata
- Deterministic finite automata → A lexer

MYT Algorithm

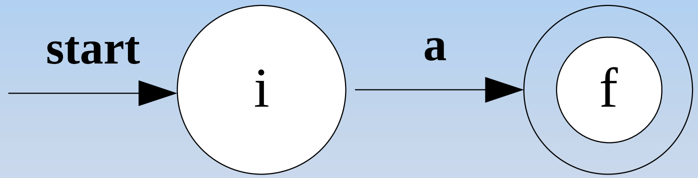
- Constructing an NFA from a regular expression ***r*** by McNaughton-Yamada-Thompson algorithm
  - Organizing ***r*** into its constituent sub-expressions
    - Sub-expressions with no operators
    - Operators
  - Using basic rules to construct NFA for Sub-expressions with no operators
  - Using inductive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

Basic Rules to Construct NFA

- For expression ε



- For any subexpression **a**, i.e. {a}



Inductive Rules to Construct Larger NFA For Operations

- Assume *N*(***s***) and *N*(***t***) are NFA for regular expressions ***s*** and ***t***, respectively
  - Parenthesis operation ***r***=(***s***)
    - Use the NFA *N*(***s***) as *N*(***r***)
  - Union operation ***r***=***s***|***t***
  - Concatenation operation ***r***=***st***
  - Repetition operation ***r***=***s****

Examples

Implementing Lexical-Analyzer Generators

- Regular expressions → Nondeterministic finite automata
- Nondeterministic finite automata → Deterministic finite automata
- Deterministic finite automata → 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 Dtran corresponding to D

```
Initially, ε-closure(s0) is the only state in Dstates, and it is unmarked;
while ( there is an unmarked state T in Dstates ) {
    mark T;
    for ( each input symbol a) {
      U = ε-closure(move(T,a));
      if ( U is not in Dstates ) add U as an unmarked state to Dstates;
      Dtran[T,a] = U;
    }
}
```

ε-closure(*s*) and ε-closure(T)

- ε-closure(*s*): a set of NFA states reachable from NFA state *s* on ε-transitions alone
- ε-closure(T): a set of NFA states reachable from some NFA state s in the set T on ε-transitions alone
  - $\cup_{s \text{ in } T}$ ε-closure(*s*)

```
push all states of T onto stack;
initialize ε-closure(T) to T;
while ( stack is not empty ) {
   pop t, the top element, off stack;
   for ( each state u with an edge from t to u labeled ε)
     if ( u is not in ε-closure(T) ) {
          add u to ε-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
  - Output stream: IF ID:i RELOP:GT NUMBER:0 THEN ID:i RELOP:EQ NUMBER:1 ELSE ID:i RELOP:EQ NUMBER: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.