Lecture 3

Lexical Analysis

Course Leader:

K S Suvidha

Objectives

At the end of this lecture, the student will be able to answer

- Specification of tokens - regular expressions and regular definitions
- LEX - A Lexical Analyzer Generator

Regular Expressions

Let Σ be an alphabet. The REs over Σ and the languages they denote (or generate) are defined as below

1. ϕ is an RE. $L(\phi) = \phi$
2. ϵ is an RE. $L(\epsilon) = \{\epsilon\}$
3. For each $a \in \Sigma$, a is an RE. $L(a) = \{a\}$
4. If r and s are REs denoting the languages R and S, respectively
   - (rs) is an RE, $L(rs) = R.S = \{xy \mid x \in R \wedge y \in S\}$
   - $(r + s)$ is an RE, $L(r + s) = R \cup S$
   - (r^*) is an RE, $L(r^*) = R^* = \bigcup_{i=0}^{\infty} R^i$

Examples of Regular Expressions

1. L = set of all strings of 0's and 1's

   $r = (0 + 1)^*$

   - How to generate the string 101 ?
   - $(0 + 1)^* \Rightarrow^4 (0 + 1)(0 + 1)(0 + 1)\epsilon \Rightarrow^4 101$

2. L = set of all strings of 0's and 1's, with at least two consecutive 0's

   $r = (0 + 1)^*00(0 + 1)^*$

1. identifiers and integers

   $letter = a + b + c + d + e;\ digit = 0 + 1 + 2 + 3 + 4;$

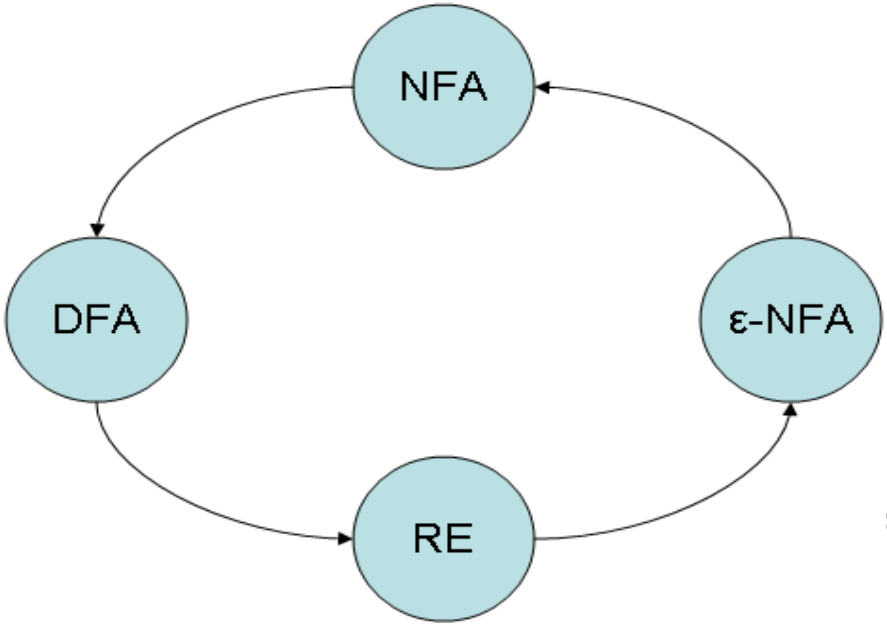
   $identifier = letter(letter + digit)^*;\ number = digit\ digit^*$

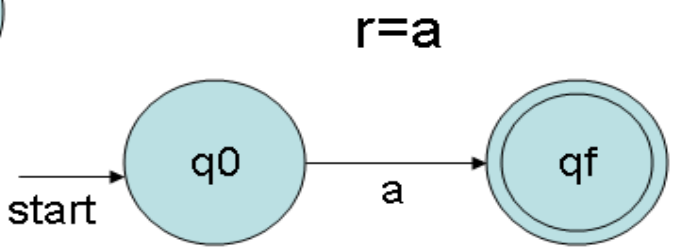
Equivalence of REs and FSA

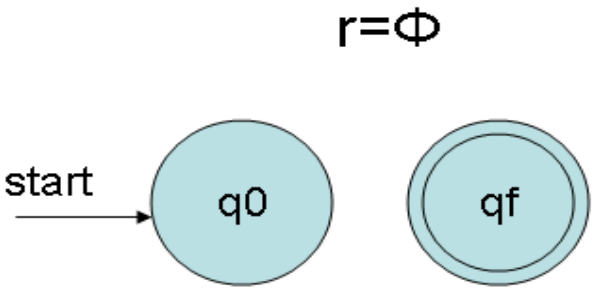
- Let r be an RE. Then there exists an NFA with –transitions that accepts L(r). The proof is by construction

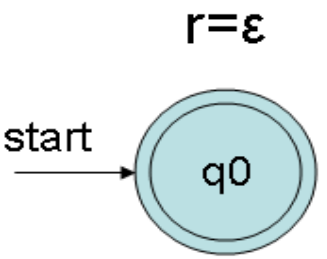
- If L is accepted by a DFA, then L is generated by an RE. The proof is tedious

Equivalence of REs and FSA







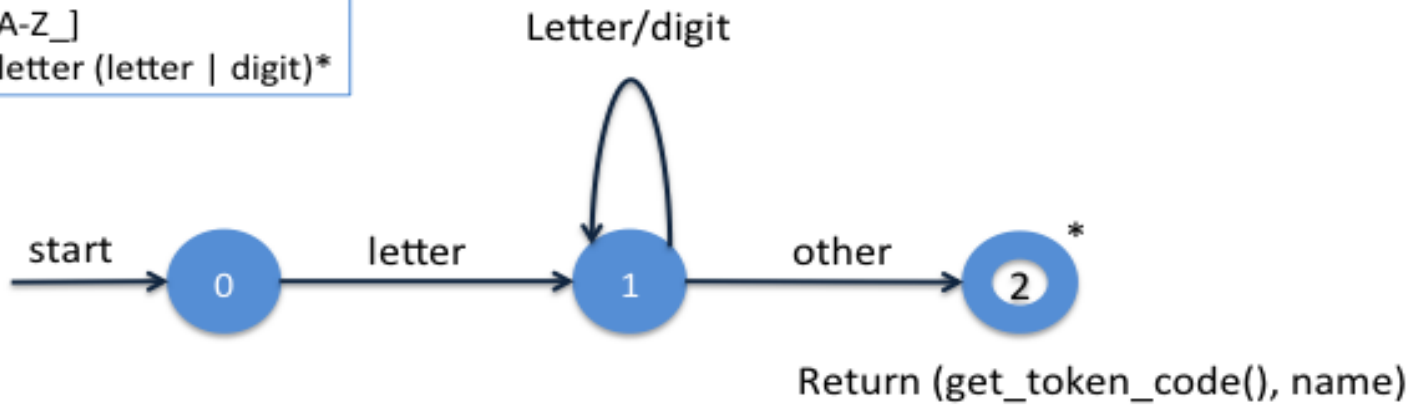


Transition Diagrams

- Transition diagrams are generalized DFAs with the following differences
  - Edges may be labelled by a symbol, a set of symbols, or a regular definition
  - Some accepting states may be indicated as retracting states, indicating that the lexeme does not include the symbol that brought us to the accepting state
  - Each accepting state has an action attached to it, which is executed when that state is reached. Typically, such an action returns a token and its attribute value
- Transition diagrams are not meant for machine translation but only for manual translation

Transition Diagram for Identifiers and Reserved Words

letter = [a-zA-Z_]
Identifier = letter (letter | digit)*



- '*' indicates retraction state
- get_token_code() searches a table to check if the name is a reserved word and returns its integer code, if so
- Otherwise, it returns the integer code of IDENTIFIER token, with name containing the string of characters forming the token (name is not relevant for reserved words)

Lexical Analyzer Implementation from Trans. Diagrams

```
TOKEN gettoken() {
   TOKEN mytoken; char c;
   while(1) { switch (state) {
     /* recognize reserved words and identifiers */
       case 0: c = nextchar(); if (letter(c))
               state = 1; else state = failure();
               break;
       case 1: c = nextchar();
               if (letter(c) || digit(c))
               state = 1; else state = 2; break;
       case 2: retract(1);
               mytoken.token = search_token();
               if (mytoken.token == IDENTIFIER)
               mytoken.value = get_id_string();
               return(mytoken);
```

LEX - A Lexical Analyzer Generator

- LEX has a language for describing regular expressions
- It generates a pattern matcher for the regular expression specifications provided to it as input
- General structure of a LEX program
  {definitions} – Optional
  %%
  {rules} – Essential
  %%
  {user subroutines} – Essential
- Commands to create an LA
  - lex ex.l – creates a C-program *lex.yy.c*
  - gcc -o ex.o lex.yy.c – produces ex.o
  - ex.o is a *lexical analyzer*, that carves tokens from its input

Definition Section

- Definitions Section contains definitions and included code

  Definitions are like macros and have the following form:
  name translation

  digit [0-9]

  number {digit} {digit}*

- Included code is all code included between **%{** and **%}**

  %{

  float number; int count=0;

  %}

Rules Section

- Contains patterns and C-code
- A line starting with white space or material enclosed in %{ and %} is C-code
- A line starting with anything else is a pattern line
- Pattern lines contain a pattern followed by some white space and C-code
  $\{pattern\} \quad \{action\ (C-code)\}$
- C-code lines are copied verbatim to the the generated C-file
- Patterns are translated into NFA which are then converted into DFA, optimized, and stored in the form of a table and a driver routine
- The action associated with a pattern is executed when the DFA recognizes a string corresponding to that pattern and reaches a final state

Strings and Operators

- **Examples of strings**: integer a57d hello
- **Operators**:

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

  \ can be used as an escape character as in C
- **Character classes**: enclosed in [and]
  Only \, -, and ^ are special inside [ ]. All other operators are irrelevant inside [ ]
  Examples:

```
[-+][0-9]+    ---> (-|+)(0|1|2|3|4|5|6|7|8|9)+
[a-d][0-4][A-C]    ---> a|b|c|d|0|1|2|3|4|A|B|C
[^abc]  ---> all char except a,b, or c,
            including special and control char
[+\-][0-5]+  ---> (+|-)(0|1|2|3|4|5)+
[^a-zA-Z]  ---> all char which are not letters
```

Lex Actions

- Default action is to copy input to output, those characters which are unmatched
- We need to provide patterns to **catch** characters
- **yytext**: contains the text matched against a pattern copying **yytext** can be done by the action **ECHO**
- **yyleng**: provides the number of characters matched
- LEX always tries the rules in the order written down and the *longest match* is preferred

```
integer    action1;
[a-z]+     action2;
```

The input *integers* will match the second pattern

A sample code

```
%{

#include<stdio.h>
int Upper=0;
int Lower=0;
%}

%%
[A-Z] {printf("Uppercase\t");Upper++;}
[a-z] {printf("Lowercase\t");Lower++;}
%%

int yywrap()
{
return 1;
}

main()
{
printf("Enter a string\n");
yylex();

printf("Uppercase=%d and Lowercase=%d",Upper,Lower);
}
```

Summary

- Regular Expression are used to recognize the tokens
- Transition diagrams are not meant for machine translation but only for manual translation
- Lexical Analyzer Generator – Lex is a tool used to identify tokens.