Lexical Analysis - Part 2

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

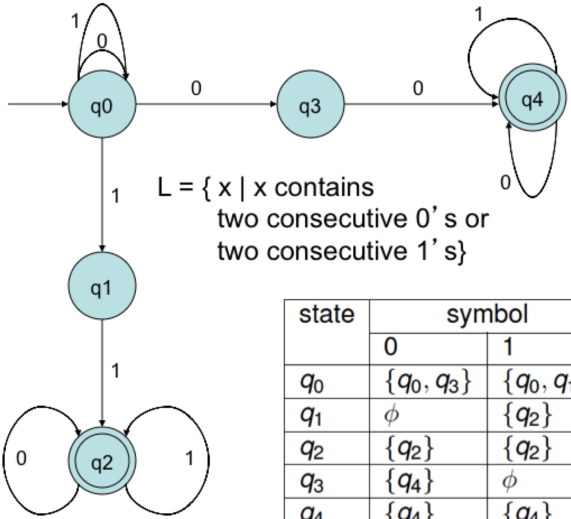
Outline of the Lecture

- What is lexical analysis? (covered in part 1)
- Why should LA be separated from syntax analysis? (covered in part 1)
- Tokens, patterns, and lexemes (covered in part 1)
- Difficulties in lexical analysis (covered in part 1)
- Recognition of tokens - finite automata and transition diagrams
- Specification of tokens - regular expressions and regular definitions
- LEX - A Lexical Analyzer Generator

Nondeterministic FSA

- NFAs are FSA which allow 0, 1, or more transitions from a state on a given input symbol
- An NFA is a 5-tuple as before, but the transition function δ is different
- $\delta(q, a)$ = the set of all states p, such that there is a transition labelled a from q to p
- $\delta : \; Q \times \Sigma \rightarrow 2^Q$
- A string is accepted by an NFA if there *exists* a sequence of transitions corresponding to the string, that leads from the start state to some final state
- Every NFA can be converted to an equivalent deterministic FA (DFA), that accepts the same language as the NFA

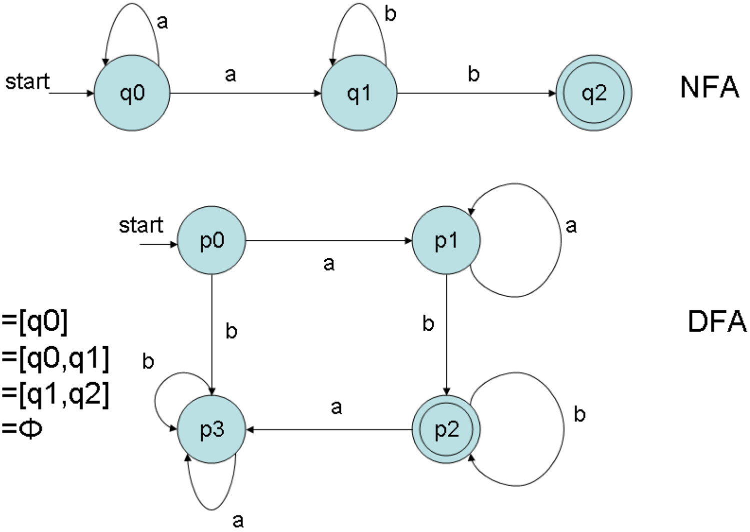
Nondeterministic FSA Example - 1



L = { x | x contains two consecutive 0's or two consecutive 1's}

state	symbol	
	0	1
q_0	$\{q_0, q_3\}$	$\{q_0, q_1\}$
q_1	ϕ	$\{q_2\}$
q_2	$\{q_2\}$	$\{q_2\}$
q_3	$\{q_4\}$	ϕ
q_4	$\{q_4\}$	$\{q_4\}$

An NFA and an Equivalent DFA



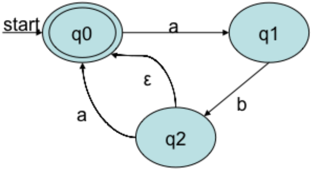
p0=[q0]
p1=[q0,q1]
p2=[q1,q2]
p3=Φ

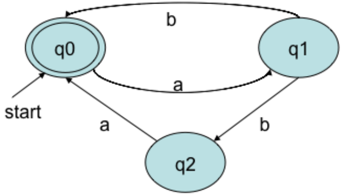
Example of NFA to DFA conversion

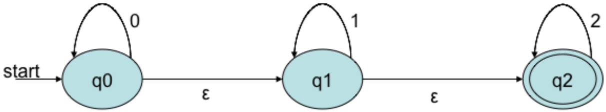
- The start state of the DFA would correspond to the set $\{q_0\}$ and will be represented by $[q_0]$
- Starting from $\delta([q_0], a)$, the new states of the DFA are constructed on *demand*
- Each subset of NFA states is a *possible* DFA state
- All the states of the DFA containing some final state as a member would be final states of the DFA
- For the NFA presented before (whose equivalent DFA was also presented)
  - $\delta[q_0], a) = [q_0, q_1], \;\; \delta([q_0], b) = \phi$
  - $\delta([q_0, q_1], a) = [q_0, q_1], \;\; \delta([q_0, q_1], b) = [q_1, q_2]$
  - $\delta(\phi, a) = \phi, \;\; \delta(\phi, b) = \phi$
  - $\delta([q_1, q_2], a) = \phi, \;\; \delta([q_1, q_2], b) = [q_1, q_2]$
  - $[q_1, q_2]$ is the final state
- In the worst case, the converted DFA may have 2^n states, where n is the no. of states of the NFA

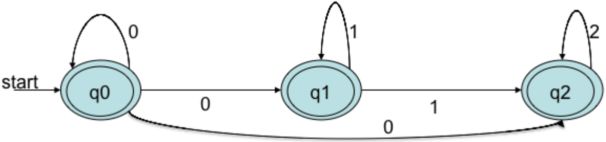
NFA with ϵ-Moves

ϵ-NFA is equivalent to NFA in power









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, $L(r^*) = R^* = \bigcup_{i=0}^{\infty} R^i$

    (L^* is called the *Kleene closure* or *closure* of L)

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
   $r = (0+1)^*00(0+1)^*$
3. $L = \{w \in \{0,1\}^* \mid w$ has two or three occurrences of 1, the first and second of which are not consecutive$\}$
   $r = 0^*10^*010^*(10^* + \epsilon)$
4. $r = (1+10)^*$
   L = set of all strings of 0's and 1's, beginning with 1 and not having two consecutive 0's
5. $r = (0+1)^*011$
   L = set of all strings of 0's and 1's ending in 011

Examples of Regular Expressions

6. $r = c^*(a + bc^*)^*$
   L = set of all strings over {a,b,c} that do not have the substring *ac*
7. $L = \{w \mid w \in \{a, b\}^* \wedge w$ ends with $a\}$
   $r = (a + b)^* a$
8. L = {if, then, else, while, do, begin, end}
   $r = if + then + else + while + do + begin + end$

Examples of Regular Definitions

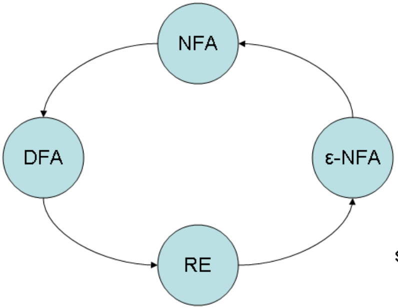
A *regular definition* is a sequence of "equations" of the form $d_1 = r_1;\ d_2 = r_2;\ ...\ ; d_n = r_n$, where each d_i is a distinct name, and each r_i is a regular expression over the symbols $\Sigma \cup \{d_1, d_2, ..., d_{i-1}\}$

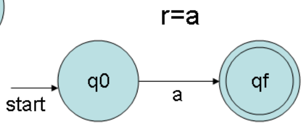
1. identifiers and integers
   $letter = a + b + c + d + e;\ \ digit = 0 + 1 + 2 + 3 + 4;$
   $identifier = letter(letter + digit)^*;\ \ number = digit\ digit^*$
2. unsigned numbers
   $digit = 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9;$
   $digits = digit\ digit^*;$
   $optional\_fraction = .digits + \epsilon;$
   $optional\_exponent = (E(+|-|\epsilon)digits) + \epsilon$
   $unsigned\_number =$
   $digits\ optional\_fraction\ optional\_exponent$

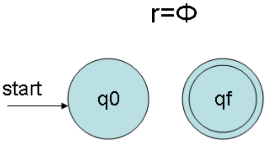
Equivalence of REs and FSA

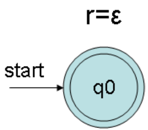
- Let r be an RE. Then there exists an NFA with ϵ-transitions that accepts $L(r)$. The proof is by construction.
- If L is accepted by a DFA, then L is generated by an RE. The proof is tedious.

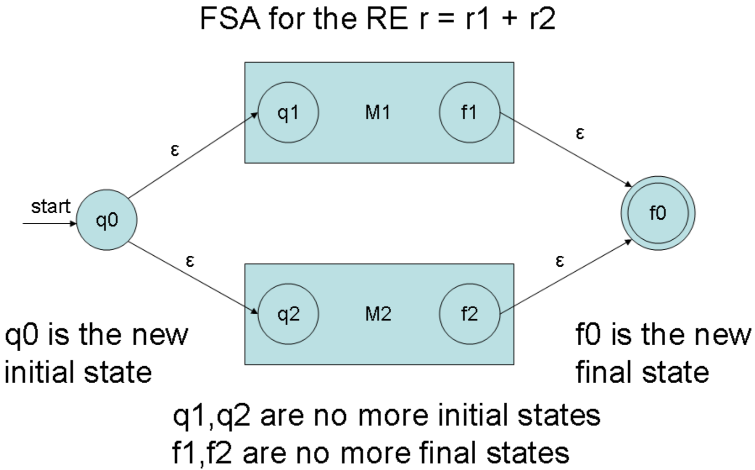
Construction of FSA from RE - r = ϕ, ϵ, or a



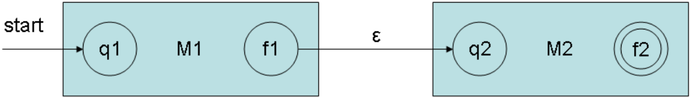







FSA for the RE r = r1 + r2
start
q0
ε
ε
q1
M1
f1
q2
M2
f2
ε
ε
f0
q0 is the new
initial state
f0 is the new
final state
q1,q2 are no more initial states
f1,f2 are no more final states

FSA for RE r = r1 r2

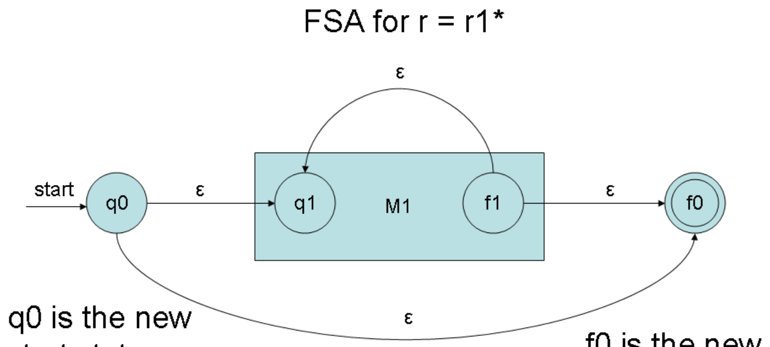


q1 is the new start state

f2 is the new final state

f1 is no more a final state
q2 is no more a start state

FSA for r = r1*

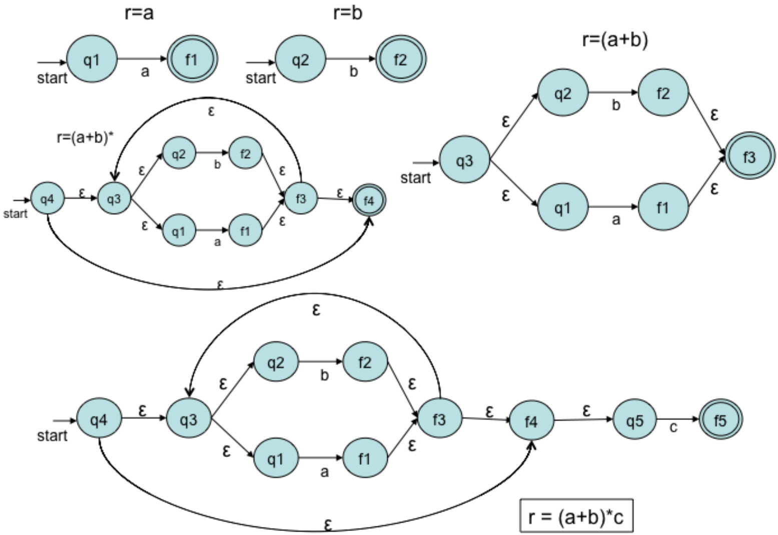


q0 is the new start state

f0 is the new final state

q1 is no more a start state
f1 is no more a final state

NFA Construction for r = (a+b)*c

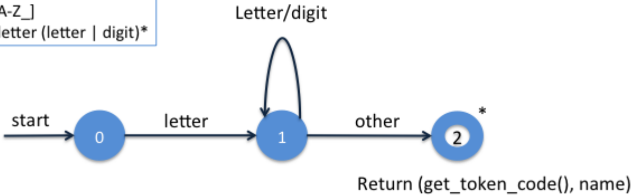


Transition Diagrams

- 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

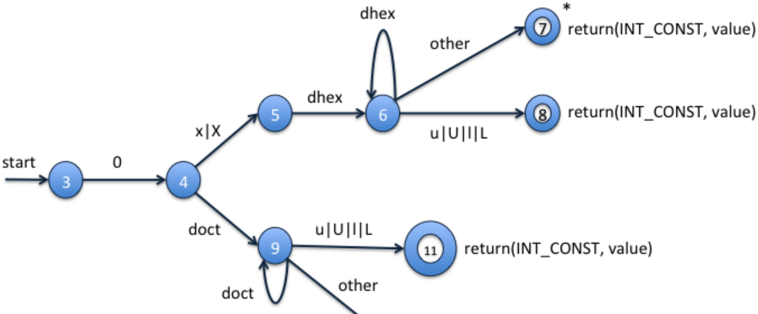
Transition Diagram for Identifiers and Reserved Words

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)

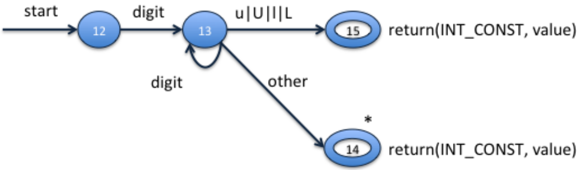
Transition Diagrams for Hex and Oct Constants



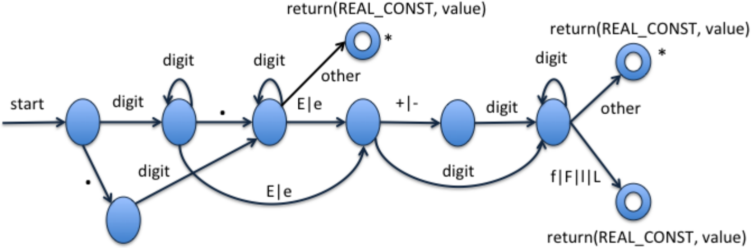
hex_const = 0 (x | X) dhex $^+$ (qualifier | ε)
oct_const = 0 doct $^+$ (qualifier | ε)
qualifier = u | U | l | L
dhex = [0-9A-F]
doct = [0-7]

Transition Diagrams for Integer Constants

int_const = digit $^+$ (qualifier | ε)
qualifier = u | U | l | L
digit = [0-9]

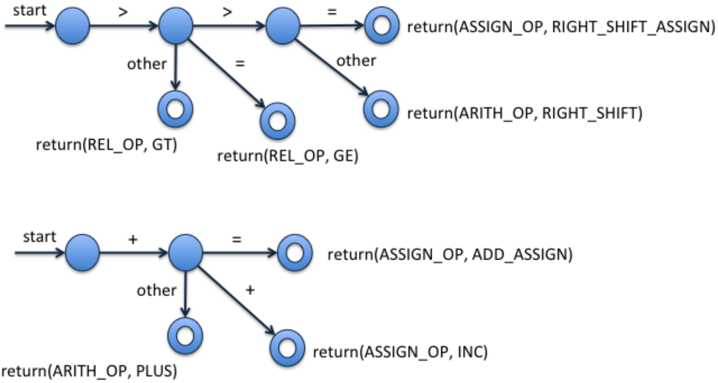


Transition Diagrams for Real Constants



real_const = (digit $^+$ exponent (qualifier | ε)) |
(digit* "." digit $^+$ (exponent | ε) (qualifier | ε)) |
(digit $^+$ "." digit* (exponent | ε) (qualifier | ε))
exponent = (E|e)(+|-|ε) digit $^+$
qualifier = f | F | l | L
digit = [0-9]

Transition Diagrams for a few Operators

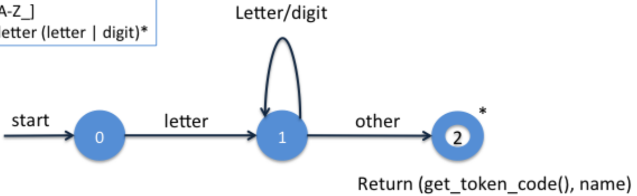


Lexical Analyzer Implementation from Trans. Diagrams

```
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);
```

Transition Diagram for Identifiers and Reserved Words

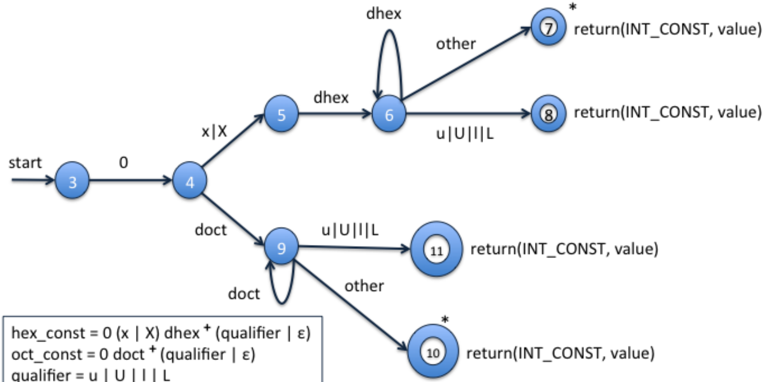
letter = [a-zA-Z_]
Identifier = letter (letter | digit)*



- '*' 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)

```
/* recognize hexa and octal constants */
  case 3: c = nextchar();
          if (c == '0') state = 4; break;
          else state = failure();
  case 4: c = nextchar();
          if ((c == 'x') || (c == 'X'))
          state = 5; else if (digitoct(c))
          state = 9; else state = failure();
          break;
  case 5: c = nextchar(); if (digithex(c))
          state = 6; else state = failure();
          break;
```

Transition Diagrams for Hex and Oct Constants



hex_const = 0 (x | X) dhex$^+$ (qualifier | ε)
oct_const = 0 doct$^+$ (qualifier | ε)
qualifier = u | U | l | L
dhex = [0-9A-F]
doct = [0-7]

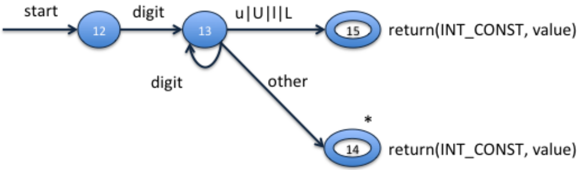
```
    case 6: c = nextchar(); if (digithex(c))
            state = 6; else if ((c == 'u')||
            (c == 'U')||(c == 'l')||
            (c == 'L')) state = 8;
            else state = 7; break;
    case 7: retract(1);
/* fall through to case 8, to save coding */
    case 8: mytoken.token = INT_CONST;
            mytoken.value = eval_hex_num();
            return(mytoken);
    case 9: c = nextchar(); if (digitoct(c))
            state = 9; else if ((c == 'u')||
            (c == 'U')||(c == 'l')||(c == 'L'))
            state = 11; else state = 10; break;
```

Lexical Analyzer Implementation from Trans. Diagrams

```
    case 10: retract(1);
/* fall through to case 11, to save coding */
    case 11: mytoken.token = INT_CONST;
             mytoken.value = eval_oct_num();
             return(mytoken);
```

Transition Diagrams for Integer Constants

int_const = digit $^+$ (qualifier | ε)
qualifier = u | U | l | L
digit = [0-9]



```
    /* recognize integer constants */
        case 12: c = nextchar(); if (digit(c))
                 state = 13; else state = failure();
        case 13: c = nextchar(); if (digit(c))
                 state = 13;else if ((c == 'u')||
                 (c == 'U')||(c == 'l')||(c == 'L'))
                 state = 15; else state = 14; break;
        case 14: retract(1);
    /* fall through to case 15, to save coding */
        case 15: mytoken.token = INT_CONST;
                 mytoken.value = eval_int_num();
                 return(mytoken);
        default: recover();
        }
    }
}
```