Chapter 2

Lexical Analysis

- The basic task of the lexical analyzer is to scan the source-code strings into small, meaningful units, that is, into the tokens we talked about in Chapter 1.

- For example, given the statement
  - if distance >= rate*(end_time - start_time) then distance := maxdist;
- The lexical analyzer must be able to isolate the
  - keywords {if, then}
  - identifiers {distance, rate, …}
  - operators {*, -, :=}
  - relational operator {>=}
  - parenthesis
  - closing semicolon

- The Lexical Analyzer may take care of a few other things as well, unless they are handled by a preprocessor:
  - Removal of Comments
  - Case Conversion
  - Removal of White Space
  - Interpretation of Compiler Directives
  - Communication with the Symbol Table
  - Preparation of Output Listing

1. Tokens and Lexemes

- In our example we had five identifiers. But for parsing purposes all identifiers are the same. On the other hand, it will clearly be important, eventually, to be able to distinguish among them.
- Similarly, for parsing purposes, one relational operator is the same as any other.

- We handle this distinction as follows
  - **Def:** token -- The generic type passed to the parser is the token
  - **Def:** lexeme -- The specific instance of the generic type is the lexeme.
- The Lexical Analyzer must:
  - isolate tokens and take note of particular lexemes.
  - when an ***id*** is found it must confer with the symbol table handler, and its actions depend upon whether we are declaring or using a variable.

2. Buffering

- In principle, the analyzer goes through the source string a character at a time;
- In practice, it must be able to access substrings of the source.
- Hence the source is normally read into a buffer
- The scanner needs two subscripts to note places in the buffer
  - lexeme start & current position

3. Finite State Automata

- The compiler writer defines tokens in the language by means of regular expressions.
- Informally a regular expression is a compact notation that indicates what characters may go together into lexemes belonging to that particular token and how they go together.
- We will see regular expressions in 2.6

- The lexical analyzer is best implemented as a finite state machine or a finite state automaton.
- Informally a Finite-State Automaton is a system that has a finite set of states with rules for making transitions between states.
- The goal now is to explain these two things in detail and bridge the gap from the first to the second.

3.1 State Diagrams and State Tables

- **Def:** State Diagram -- is a directed graph where the vertices in the graph represent states, and the edges indicate transitions between the states.
- Let's consider a vending machine that sells candy bars for 25 cents, and it takes nickels, dimes, and quarters.

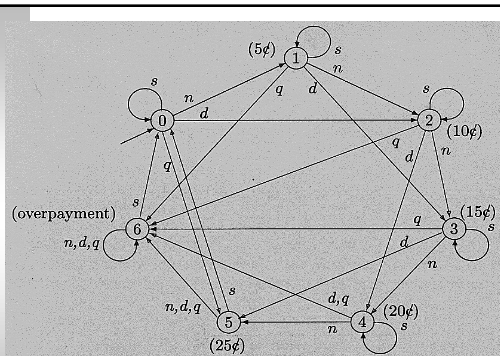

Figure 2.1

- **Def:** State Table -- is a table with states down the left side, inputs across the top, and row/column values indicate the current state/input and what state to go to.
- Table -- pg.. 22

Current state	Inputs: Nickel	Dime	Quarter	Select	
0	1	2	5	0	Next state
1	2	3	6	1	
2	3	4	6	2	
3	4	5	6	3	
4	5	6	6	4	
5	6	6	6	0*	
6	6	6	6	0*	

3.2 Formal Definition

- Def: FSA -- A FSA, *M* consists of
  - a finite set of input symbols Σ (the input alphabet)
  - a finite set of states Q
  - A starting state q_0 (which is an element of Q)
  - A set of accepting states F (a subset of Q) (these are sometimes called final states)
  - A state-transition function $N: (Q \times \Sigma) \rightarrow Q$
- $M = (\Sigma, Q, q_0\ F, N)$

- Example 1:
  - Given Machine Construct Table:

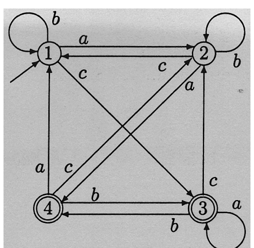

Figure 2.3

Current state	Inputs a	b	c
1	2	1	3
2	4	2	1
<u>3</u>	3	4	2
<u>4</u>	1	3	2

- Example 1 (cont.):
  - State 1 is the Starting State. This is shown by the arrow in the Machine, and the fact that it is the first state in the table.
  - States 3 and 4 are the accepting states. This is shown by the double circles in the machine, and the fact that they are underlined in the table.

- Example 2:
  - Given Table, Construct Machine:

Current state	Inputs a	b	c
1	2	1	3
2	4	2	1
<u>3</u>	3	4	2
<u>4</u>	1	3	2
5	5	4	5

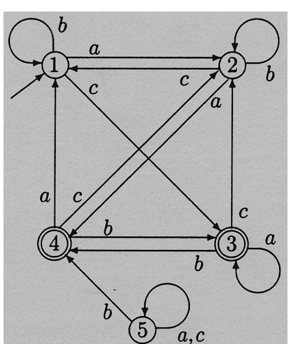

Figure 2.4

- Example 2 (cont.):
  - This machine shows that it is entirely possible to have unreachable states in an automaton.
  - These states can be removed without affecting the automaton.

3.3 Acceptance

- We use FSA's for recognizing tokens
- A character string is recognized (or accepted) by FSA *M* if, when the last character has been read, *M* is in one of the accepting states.
  - If we pass through an accepting state, but end in a non-accepting state, the string is NOT accepted.

- **Def:** <u>language</u> -- A language is any set of strings.
- **Def:** <u>a language over an alphabet Σ</u> is any set of strings made up only of the characters from Σ
- **Def:** <u>L(*M*) -- the language accepted by *M*</u> is the set of all strings over Σ that are accepted by *M*

- if we have an FSA for every token, then the language accepted by that FSA is the set of all lexemes embraced by that token.

- **Def:** <u>equivalent</u> --
  M1 == M2 iff L(M1) = L(M2).

- A FSA can be easily programmed if the state table is stored in memory as a two-dimensional array.

```
table : array[1..nstates,1..ninputs] of byte;
```

- Given an input string w, the code would look something like this:

```
state := 1;
for i:=1 to length(w) do
    begin
    col:= char_to_col(w[i]);
    state:= table[state, col]
    end;
```


4. Nondeterministic Finite-State Automata

- So far, the behavior of our FSAs has always been predictable. But there is another type of FSA in which the state transitions are not predictable.

- In these machines, the state transition function is of the form:
  N: Q x (Σ U {ε}) -> P(Q)
  - Note: some authors use a Greek lambda, λ or Λ

- This means two things:
  - There can be transitions without input. (That is why the ε is in the domain)
  - Input can transition to a number of states. (That is the significance of the power set in the codomain)
- Since this makes the behavior unpredictable, we call it a nondeterministic FSA
  - So now we have DFAs and NFAs (or NDFAs)
- a string is accepted if there is *at least* 1 path from the start state to an accepting state.

- Example: Given Machine
  Trace the input = baabab

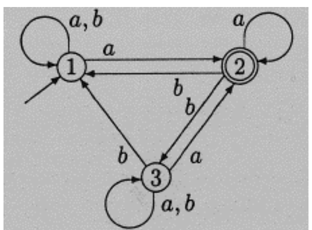

Figure 2.8

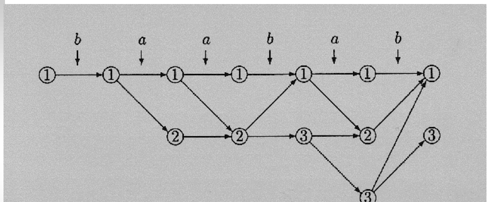

Figure 2.10

4.1 ε-Transitions

- a spontaneous transition without input.
- Example: Trace input aaba

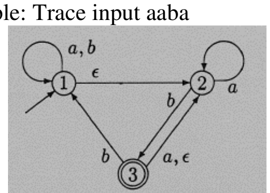

Figure 2.12

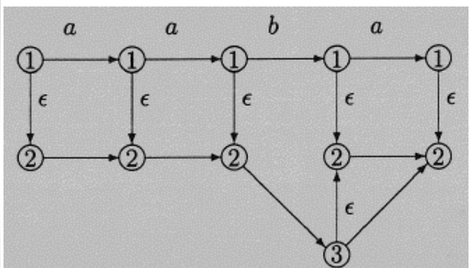

Figure 2.13

4.2 Equivalence

- For every non-deterministic machine *M* we can construct an equivalent deterministic machine *M'*
- Therefore, why study N-FSA?
  - 1.Theory.
  - 2.Tokens -> Reg.Expr. -> N-FSA -> D-FSA

5. The Subset Construction

- Constructing an equivalent DFA from a given NFA hinges on the fact that transitions between *state sets* are deterministic even if the transitions between *states* are not.

- Acceptance states of the subset machine are those subsets that contain at least 1 accepting state.

- Generic brute force construction is impractical.
  - As the number of states in *M* increases, the number of states in *M′* increases drastically (n vs. 2^n). If we have a NFA with 20 states |P(Q)| is something over a million.
  - This also leads to the creation of many unreachable states. (which can be omitted)

- The trick is to only create subset states as you need them.

- Example:
  - Given: NFA

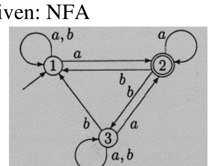

Figure 2.8

  - Build DFA out of NFA (Table & Graph pg. 34)

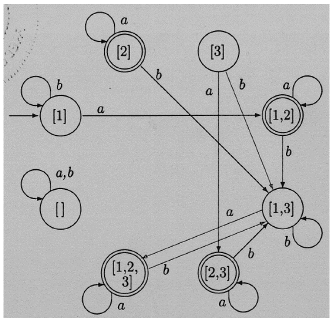

Figure 2.15

6. Regular Expressions

- Lexical Analysis and Syntactic Analysis are typically run off of tables. These tables are large and laborious to build. Therefore, we use a program to build the tables.
- But there are two major problems:
  - How do we represent a token for the table generating program?
  - How does the program convert this into the corresponding FSA?

- Tokens are described using regular expressions.
- Informally a regular expression of an alphabet Σ is a combination of characters from Σ and certain operators indicating concatenation, selection, or repetition.
  - b^* -- 0 or more b's (Kleene Star)
  - b^+ -- 1 or more b's
  - | -- a|b -- choice

- **Def:** Regular Expression:
  - any character in Σ is an RE
  - ε is an RE
  - if R and S are RE's so are
    RS, R|S, R^*, R^+, S^*, S^+.
- Only expressions formed by these rules are regular.
- L(RS) = {vw| v in L(R), w in L(S)}.
  -- concatenation.

- REs can be used to describe only a limited variety of languages, but they are powerful enough to be used to define tokens.
- One limitation -- many languages put length limitations on their tokens, RE's have no means of enforcing such limitations.

7. Regular Expressions and Finite-State Machines

- This machine recognizes ε

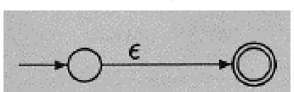

Figure 2.17(a)

- This machine will recognize a character *a* in Σ

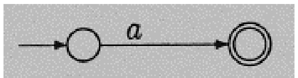

Figure 2.17(b)

- To recognize RS connect the machines as shown

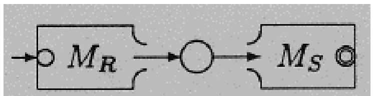

Figure 2.17(c)

- To recognize R|S, connect the machines this way.

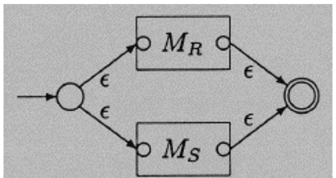

Figure 2.17(d)

- R^*
  - Begin with R^+

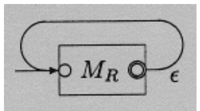

Figure 2.17(e)

  - Now add the zero or more to go from R^+ to R^*

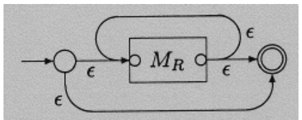

Figure 2.17(f)

8. The Pumping Lemma

- Given a machine with n states
- and a string w in $L(M)$ has length n
- w must go through $n+1$ states, therefore something is repeated (call it y)
- therefore $w = xyz$ and y can be looped.
- so xy^*z is also part of the language.

- The goal of the pumping lemma is to show that there are some languages that are not regular.
- For Example:
  - $L_R = \{wcw^R \mid w \text{ in } (0,1)^*\}$
  - L_P -- matching parens
    - this is handled in syntax analysis.

9. Application to Lexical Analysis

- Now you are ready to put it all together:
  - Given 2 tokens' regular expression
    - X = aa*(b|c)
    - Y = (b|c)c*
  - Construct the NDFA
  - Construct the DFA

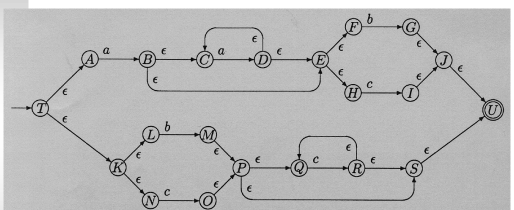

Figure 2.20

9.1 Recognizing Tokens

- The scanner must ignore white space (except to note the end of a token)
  - Add white space transition from Start state to Start state.
- When you enter an accept state, announce it
  - (therefore you cannot pass through accept states)
  - The string may be the entire program.

- One accept state for each token, so we know what we found.
- Identifier/Keyword differences
  - Accept everything as an identifier, and then look up keywords in table. Or pre-load the Symbol Table with Keywords.
- When you read an identifier, you read the next character in order to tell it was the end. You need to back up (put it back on the input stream).

- Comments
  - Recognize the beginning of comment, and then ignore everything until the end of comment.
  - What if there are multiple types of comments?
- Character Strings
  - single or double quotes?

10. Summary

- Lesk & Schmidt -- "LEX - a lexical analyzer generator" in *Unix Programmers Manual* Vol. 2
- Mason & Brown -- *Lex & Yacc* -- O'Reilly & Associates.