COMPILER CONSTRUCTION

Principles and Practice

Kenneth C. Louden

2. Scanning (Lexical Analysis)

PART ONE

Contents

2.1 The Scanning Process

The Function of a Scanner

- Reading characters from the source code and form them into logical units called tokens
- Tokens are logical entities defined as an enumerated type
  - Typedef enum

    {IF, THEN, ELSE, PLUS, MINUS, NUM, ID,…}

    TokenType;

The Categories of Tokens

- RESERVED WORDS
  - Such as IF and THEN, which represent the strings of characters “if” and “then”
- SPECIAL SYMBOLS
  - Such as PLUS and MINUS, which represent the characters “+” and “-“
- OTHER TOKENS
  - Such as NUM and ID, which represent numbers and identifiers

Relationship between Tokens and its String

- The string is called STRING VALUE or LEXEME of token
- Some tokens have only one lexeme, such as reserved words
- A token may have infinitely many lexemes, such as the token ID

Relationship between Tokens and its String

- Any value associated to a token is called an attributes of a token
  - String value is an example of an attribute.
  - A NUM token may have a string value such as “32767” and actual value 32767
  - A PLUS token has the string value “+” as well as arithmetic operation +
- The token can be viewed as the collection of all of its attributes
  - Only need to compute as many attributes as necessary to allow further processing
  - The numeric value of a NUM token need not compute immediately

Some Practical Issues of the Scanner

- One structured data type to collect all the attributes of a token, called a token record
  - Typedef struct

```
{TokenType tokenval;
        char *stringval;
        int numval;
} TokenRecord
```

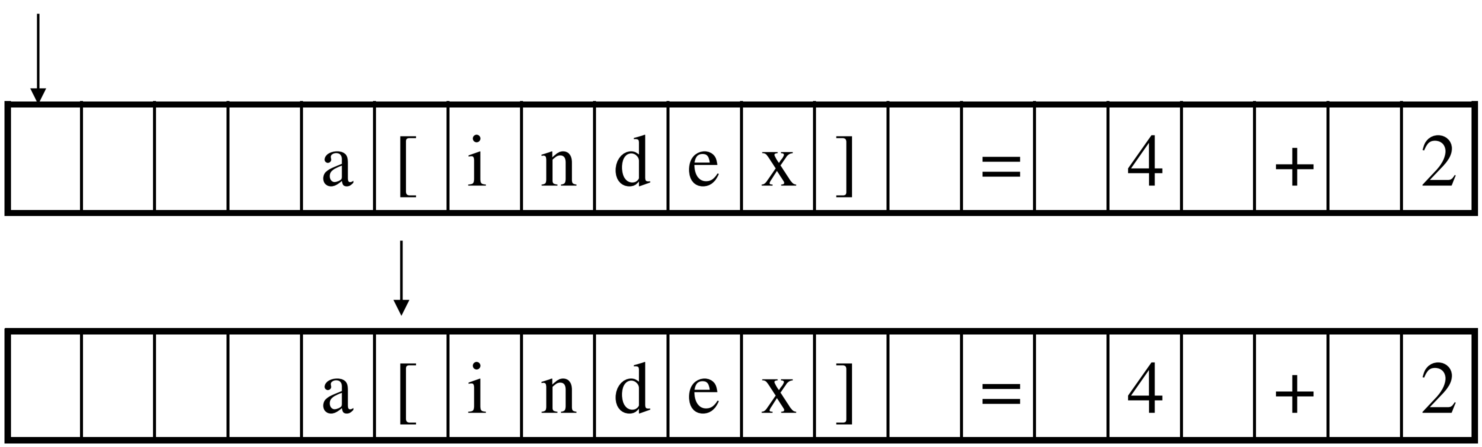
Some Practical Issues of the Scanner

- The scanner returns the token value only and places the other attributes in variables

  TokeType getToken(void)

- As an example of operation of getToken, consider the following line of C code.

  A[index] = 4+2



RET

2.2 Regular Expression

Some Relative Basic Concepts

- Regular expressions
  - represent patterns of strings of characters.
- A regular expression *r*
  - completely defined by the set of strings it matches.
  - The set is called the language of r written as L(r)
- The set elements
  - referred to as symbols
- This set of legal symbols
  - called the alphabet and written as the Greek symbol ∑

Some Relative Basic Concepts

- A regular expression *r*
  - contains characters from the alphabet, indicating patterns, such *a* is the character a used as a pattern
- A regular expression *r*
  - may contain special characters called *meta-characters* or meta-symbols
- An *escape character* can be used to turn off the special meaning of a meta-character.
  - Such as backslash and quotes

More About Regular Expression

2.2.1 Definition of Regular Expression [Open]

2.2.2 Extension to Regular Expression [Open]

2.2.3 Regular Expressions for Programming Language Tokens [Open]

2.2.1 Definition of Regular Expressions

Basic Regular Expressions

- The single characters from alphabet matching themselves
  - a matches the character *a* by writing L(a)={ a }
  - **ε** denotes the empty string, by L(**ε**)={**ε**}

Regular Expression Operations

- Choice among alternatives, indicated by the meta-character |
- Concatenation, indicated by juxtaposition
- Repetition or "closure", indicated by the meta-character *

Choice Among Alternatives

- If r and s are regular expressions, then $r|s$ is a regular expression which matches any string that is matched either by r or by s.
- In terms of languages, the language $r|s$ is the union of language r and s, or $L(r|s) = L(r) \cup L(s)$
- A simple example, $L(a|b) = L(a) \cup (b) = \{a, b\}$
- Choice can be extended to more than one alternative.

Concatenation

- If r and s are regular expression, the rs is their concatenation which matches any string that is the concatenation of two strings, the first of which matches r and the second of which matches s.
- In term of generated languages, the concatenation set of strings S1S2 is the set of strings of S1 appended by all the strings of S2.
- A simple example, (a|b)c matches ac and bc
- Concatenation can also be extended to more than two regular expressions.

Repetition

- The repetition operation of a regular expression, called (Kleene) closure, is written r*, where r is a regular expression. The regular expression r* matches any finite concatenation of strings, each of which matches r.
- A simple example, a* matches the strings epsilon, a, aa, aaa,…
- In term of generated language, given a set of S of string, S* is a infinite set union, but each element in it is a finite concatenation of string from S

Precedence of Operation and Use of Parentheses

- The standard convention

  Repetition * has highest precedence

  Concatenation is given the next highest

  | is given the lowest

- A simple example

  a|bc* is interpreted as a|(b(c*))

- Parentheses is used to indicate a different precedence

Name for regular expression

- Give a name to a long regular expression
  - *digit* = 0|1|2|3|4……|9
  - (0|1|2|3……|9)(0|1|2|3……|9)* *digit* *digit**

Definition of Regular Expression

- A regular expression is one of the following:
  - (1) A basic regular expression, a single legal character a from alphabet $\sum$ or meta-character **ε.**
  - (2) The form r|s, where r and s are regular expressions
  - (3) The form rs, where r and s are regular expressions
  - (4) The form r*, where r is a regular expression
  - (5) The form (r), where r is a regular expression
- Parentheses do not change the language.

Examples of Regular Expressions

Example 1:

- ∑={ a,b,c}
- the set of all strings over this alphabet that contain exactly one b.
- (a|c)*b(a|c)*

Example 2:

- ∑={ a,b,c}
- the set of all strings that contain at most one b.
- (a|c)*|(a|c)*b(a|c)*   (a|c)*(b|ε)(a|c)*
- the same language may be generated by many different regular expressions.

Examples of Regular Expressions

Example 3:

- $\sum = \{ a,b \}$
- the set of strings consists of a single b surrounded by the same number of a's.
- $S = \{b, aba, aabaa, aaabaaa, \ldots\ldots\} = \{ a^n b a^n \mid n \neq 0 \}$
- This set can not be described by a regular expression.
  - "regular expression can't count "
- *not all sets of strings can be generated by regular expressions*.
- **a regular set** : a set of strings that is the language for a regular expression is distinguished from other sets.

Examples of Regular Expressions

Example 4:

- ∑={ a,b,c}
- The strings contain no two consecutive b's
- ((a|c)* | (b(a|c))*)*
- ((a | c) | (b(a | c)))* or (a | c | ba | bc)*
  - Not yet the correct answer

The correct regular expression

- (a | c | ba | bc)* (b |ε)
- ((b |ε) (a | c | ab| cb)*
- (not b |b not b)*(b|ε) not b = a|c

Examples of Regular Expressions

Example 5:

- $\sum$={ a,b,c}
- ((b|c)* a(b|c)*a)* (b|c)*
- Determine a concise English description of the language
- the strings contain an even number of a's

  (nota* a nota* a)* nota*

BACK

2.2.2 Extensions to Regular Expression

List of New Operations

1) one or more repetitions

   r+

2) any character

   period “.   ”

3) a range of characters

   [0-9], [a-zA-Z]

List of New Operations

4) any character not in a given set

~(a|b|c) a character not either a or b or c

[^abc] in Lex

5) optional sub-expressions

– r? the strings matched by r are optional

BACK

2.2.3 Regular Expressions for Programming Language Tokens

Number, Reserved word and Identifiers

Numbers

- *nat* = [0-9]+
- *signedNat* = (+|-)?*nat*
- *number* = *signedNat*(“. ” *nat*)? (E *signedNat*)?

Reserved Words and Identifiers

- *reserved* = if | while | do |………
- *letter* = [a-z A-Z]
- *digit* = [0-9]
- *identifier* = *letter(letter|digit)**

Comments

Several forms:

{ this is a pascal comment } {(~ })*}

; this is a schema comment

-- this is an Ada comment --(~*newline*)* *newline*

/* this is a C comment */

can not written as ba(~(ab))*ab, ~ restricted to single character

one solution for ~(ab) : b*(a*~(a|b)b*)*a*

Because of the complexity of regular expression, the comments will be handled by ad hoc methods in actual scanners.

Ambiguity

Ambiguity: some strings can be matched by several different regular expressions.

- either an identifier or a keyword, **keyword interpretation preferred**.
- a single token or a sequence of several tokens, the single-token preferred.(the principle of **longest sub-string.**)

White Space and Lookahead

White space:

- Delimiters: characters that are unambiguously part of other tokens are delimiters.
- *whitespace* = (n*ewline* | *blank* | *tab* | *comment*)+
- free format or fixed format

Lookahead:

- buffering of input characters , marking places for backtracking

  DO99I=1,10

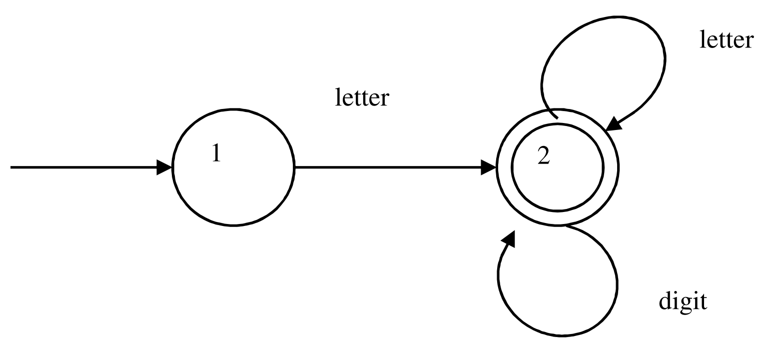
  DO99I=1.10

RET

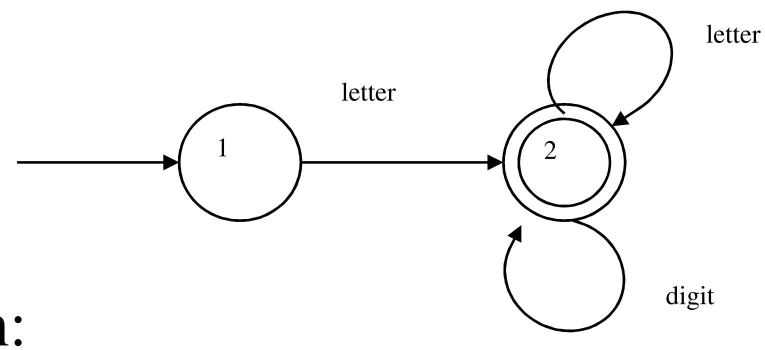
2.3 FINITE AUTOMATA

Introduction to Finite Automata

- Finite automata (finite-state machines) are a mathematical way of describing particular kinds of algorithms.
- A strong relationship between finite automata and regular expression
    - *Identifier = letter (letter | digit)**



Introduction to Finite Automata



- Transition:
  - Record a change from one state to another upon a match of the character or characters by which they are labeled.
- Start state:
  - The recognition process begin
  - Drawing an unlabeled arrowed line to it coming “from nowhere”
- Accepting states:
  - Represent the end of the recognition process.
  - Drawing a double-line border around the state in the diagram

More About Finite Automata

2.3.1 Definition of Deterministic Finite Automata [Open]

2.3.2 Lookahead, Backtracking ,and Nondeterministic Automata [Open]

2.3.3 Implementation of Finite Automata in Code [Open]

2.3.1 Definition of Deterministic Finite Automata

The Concept of DFA

DFA: Automata where the next state is uniquely given by the current state and the current input character.

Definition of a DFA:

A DFA (Deterministic Finite Automation) **M** consist of

(1) an alphabet $\sum$,

(2) A set of states S,

(3) a transition function $T : S \times \sum \rightarrow S$,

(4) a start state $s0 \in S$,

(5)And a set of accepting states $A \subset S$

The Concept of DFA

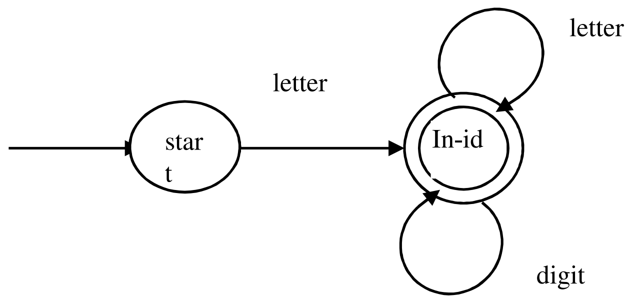
The language accepted by a DFA M, written L(M), is defined to be

the set of strings of characters c1c2c3….cn with each ci ∈∑ such that there exist states s1 = t(s0,c1),s2 = t(s1,c2), sn = T(sn-1,cn) with sn an element of A (i.e. an accepting state).

Accepting state s_n means the same thing as the diagram:

$$\rightarrow s0 \xrightarrow{c1} s1 \xrightarrow{c2} s2 \rightarrow \ldots\ldots\ldots sn\text{-}1 \xrightarrow{cn} sn$$

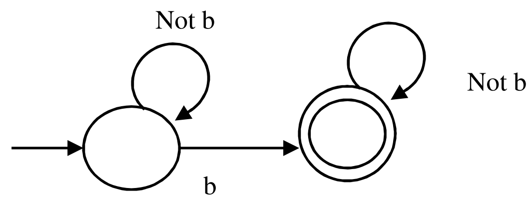
Some differences between definition of DFA and the diagram:



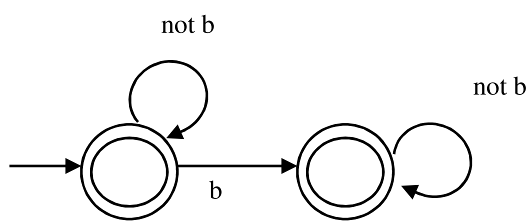
1) The definition does not restrict the set of states to **number**s
2) We have not labeled the transitions with characters but with **names** representing a set of characters
3) definitions T: $S \times \sum \rightarrow S$, T(s, c) must **have a value for every s and c**.
   - In the diagram, T (start, c) defined only if c is a letter, T(in_id, c) is defined only if c is a letter or a digit.
   - Error transitions are not drawn in the diagram but are simply assumed to always exist.

Examples of DFA

Example 2.6: exactly accept one b



Example 2.7: at most one b



Examples of DFA

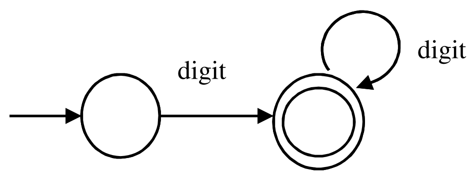
Example 2.8: digit = [0-9]

nat = digit +

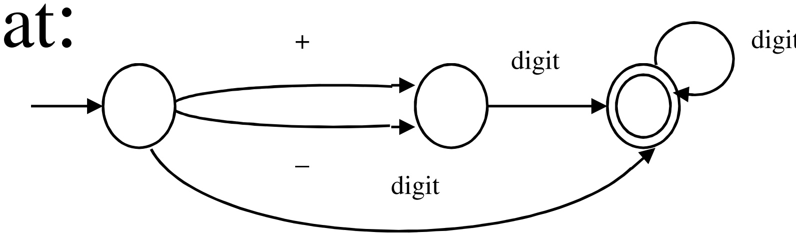
signedNat = (+|-)? nat

Number = singedNat(“.”nat)?(E signedNat)?

A DFA of nat:



A DFA of signedNat:



Examples of DFA

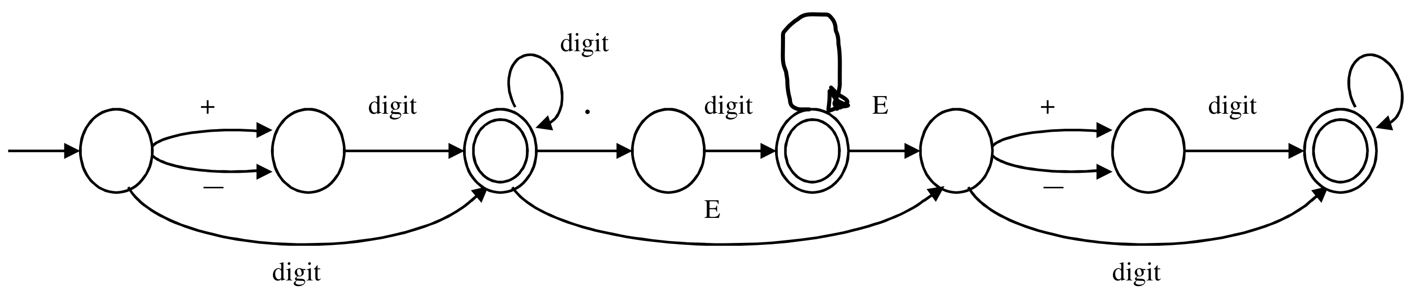
Example 2.8: digit = [0-9]

nat = digit +

signedNat = (+|-)? nat

Number = singedNat(“.”nat)?(E signedNat)?

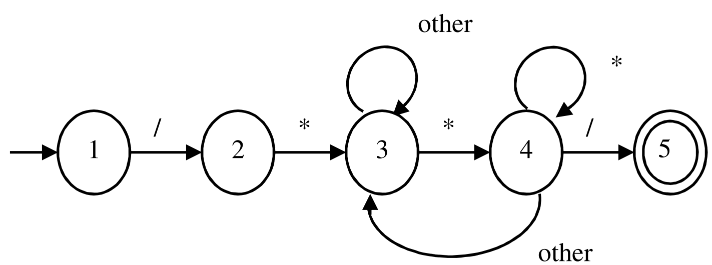
A DFA of Number:



Examples of DFA

Example 2.9 : A DFA of C Comments

(easy than write down a regular expression)

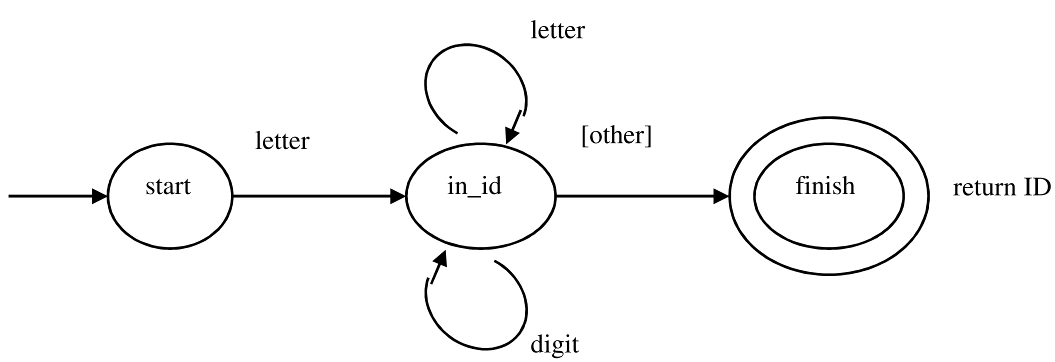


BACK

2.3.2 Lookahead, Backtracking, and Nondeterministic Automata

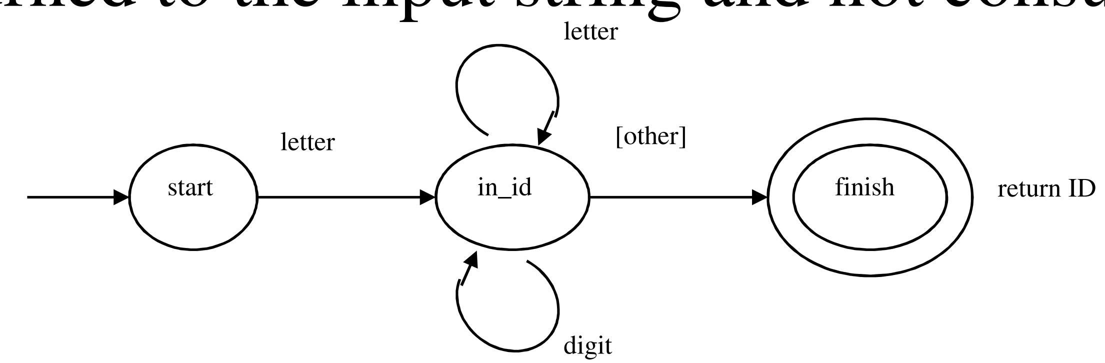
A Typical Action of DFA Algorithm

- Making a transition: move the character from the input string to a string that accumulates the characters belonging to a single token (the token string value or lexeme of the token)
- Reaching an accepting state: return the token just recognized, along with any associated attributes.
- Reaching an error state: either back up in the input (backtracking) or to generate an error token.



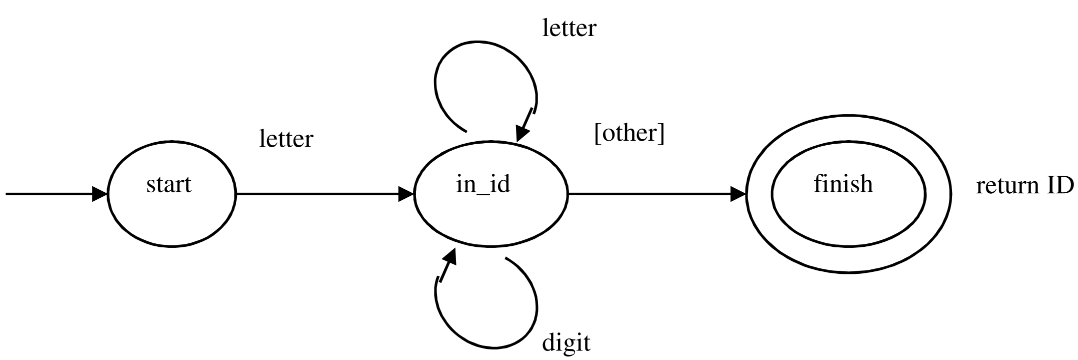
Finite automation for an identifier with delimiter and return value

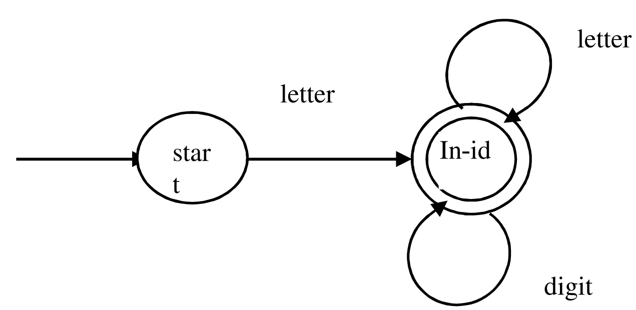
- **The error state** represents the fact that either an identifier is not to be recognized (if came from the start state) or a delimiter has been seen and we should now accept and generate an identifier-token.
- **[other]: indicate that the delimiting character** should be considered look-ahead, it should be returned to the input string and not consumed.



Finite automation for an identifier with delimiter and return value

- This diagram also **expresses the principle of longest sub-string** described in Section 2.2.4: the DFA continues to match letters and digits (in state in_id) until a delimiter is found.
- **By contrast the old diagram allowed the DFA to accept at any point while reading an identifier string**.



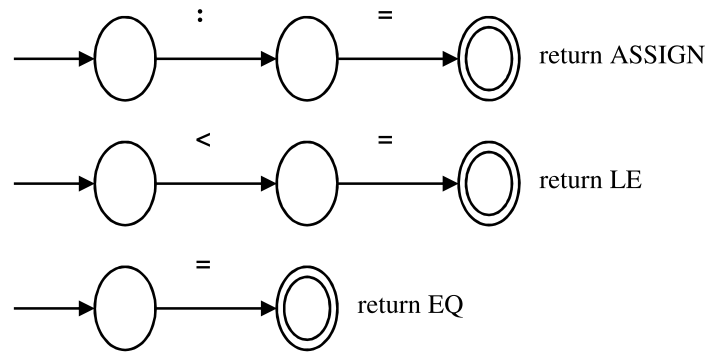


How to arrive at the start state in the first place

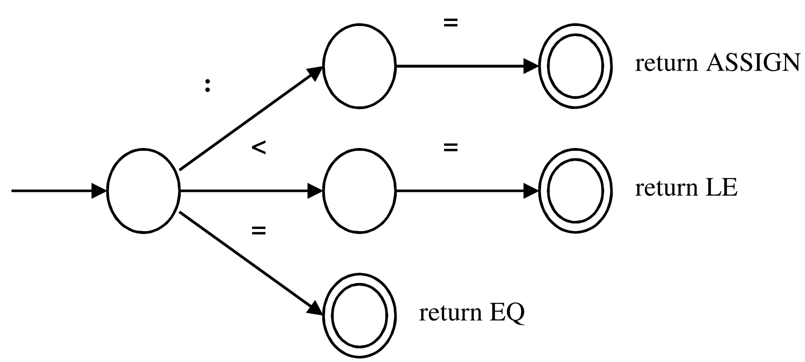
(combine all the tokens into one DFA)

Each of these tokens begins with a different character

- Consider the tokens given by the strings : =, <=, and =
- Each of these is a fixed string, and DFAs for them can be written as right

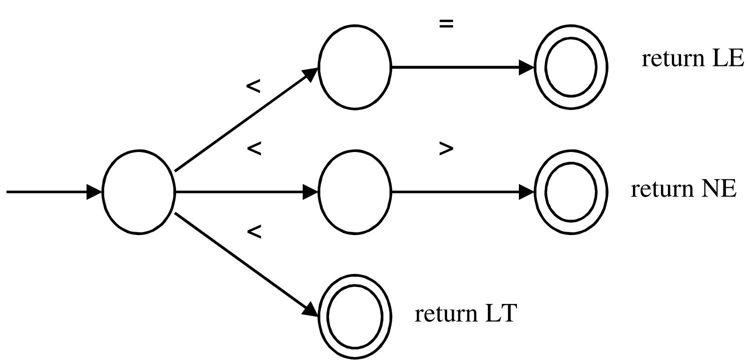


- Uniting all of their start states into a single start state to get the DFA

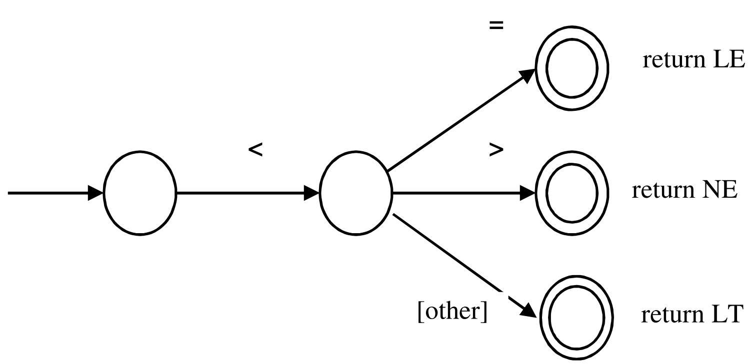


Several tokens beginning with the same character

- They cannot be simply written as the right diagram, since it is not a DFA



- The diagram can be rearranged into a DFA



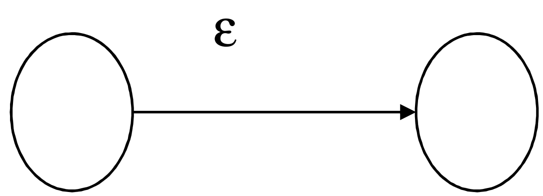
Expand the Definition of a Finite Automaton

- One solution for the problem is to **expand the definition of a finite automaton**
- **More than one transition from a state** may exist for a particular character

  (NFA: non-deterministic finite automaton,)
- Developing an algorithm for systematically turning these NFA into DFAs

ε-transition

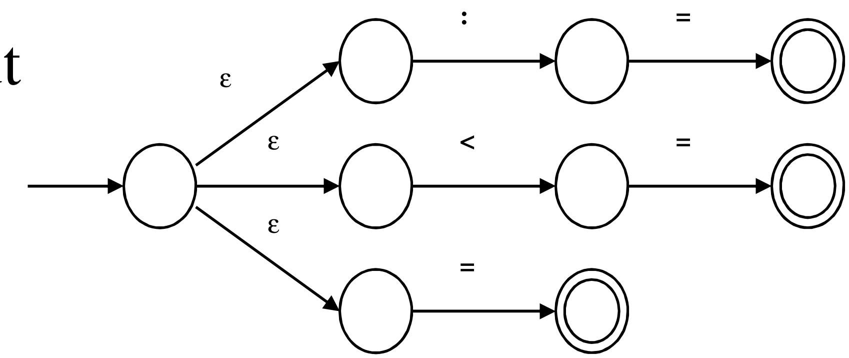
- A transition that may occur without consulting the input string (and without consuming any characters)



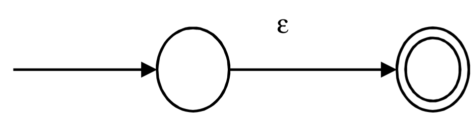
- It may be viewed as a "match" of the empty string.
- (This should not be confused with a match of the characterεin the input)

ε-Transitions Used in Two Ways.

- First: to **express a choice of alternatives** in a way without combining states
  - Advantage: keeping the original automata intact and only adding a new start state to connect them



- Second: to explicitly **describe a match of the empty string**.



Definition of NFA

- An **NFA** (non-deterministic finite automaton) *M* consists of
  - an alphabet Σ, a set of states *S*,
  - a transition function $\boldsymbol{T: S \times (\Sigma \cup \{\varepsilon\}) \rightarrow \wp(S)}$,
  - a start state *s0* from *S*, and a set of accepting states *A* from *S*

- The language accepted by *M*, written L(*M*),
  - is defined to be the set of strings of characters *c1c2*…. *cn* with
  - each *ci* from $\Sigma \cup \{\varepsilon\}$ such that
  - there exist states *s1* in *T(s0 ,c1*), *s2* in (s*1,* c2),..., *sn* in *T(sn-1 , cn)* with *sn* an element of A.

Some Notes

- Any of the c_i in $c_1c_2\ldots\ldots c_n$ may be ε, and
  - the string that is actually accepted is the string $c_1c_2\ldots c_n$ with the ε's removed (since the concatenation of *s* with ε is *s* itself).
  - Thus, **the string** $c_1c_2\ldots c_n$ **may actually have fewer than *n* characters in it**

- The sequence of states $s_1, \ldots, s_n$ are chosen from the *sets* of states $T(s_0, c_1), \ldots, T(s_{n-1}, c_n)$, and this choice will not always be uniquely determined.
  - **The sequence of transitions that accepts a particular string is not determined at each step by the state and the next input character.**
  - Indeed, arbitrary numbers of ε's can be introduced into the string at any point, corresponding to any number of ε-transitions in the NFA.

Some Notes

- An NFA does not represent an algorithm.

  However, it can be simulated by an algorithm that backtracks through every non-deterministic choice.

Examples of NFAs

Example 2.10

- The string **abb** can be accepted by either of the following sequences of transitions:

  a b ε b
  →1→2→4→2→4

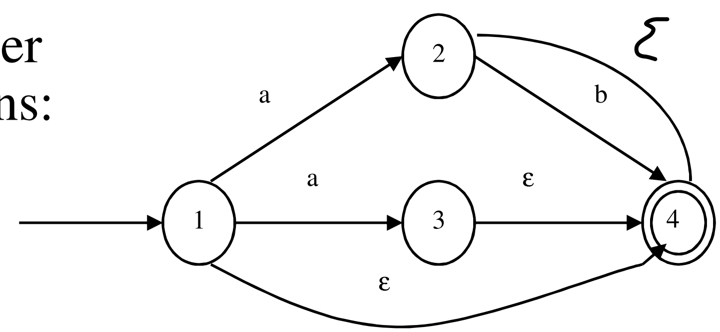
  a ε ε b ε b
  →1→3→4→2→4→2→4

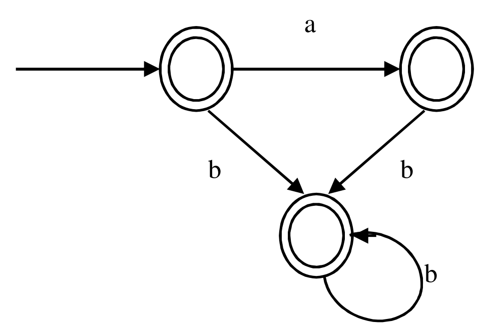
- This NFA accepts the languages as follows:

  regular expression: (a|ε)b*

  ab+|ab*|b*

- Left DFA accepts the same language.

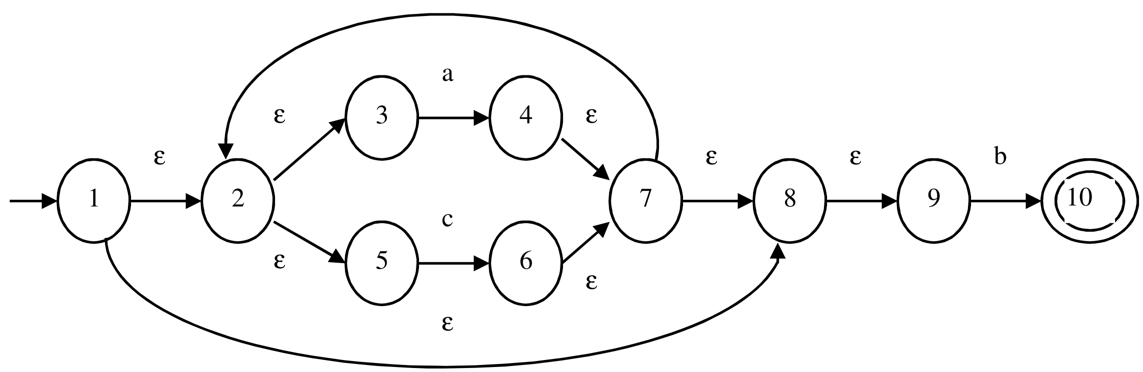




Examples of NFAs

Example 2.11

- It accepts the string acab by making the following transitions:
  - <u>(1)(2)(3)a(4)(7)(2)(5)(6)c(7)(2)(3)a(4)(7)(8)(9)b(10)</u>
- It accepts the same language as that generated by the regular expression : (a | c) *b



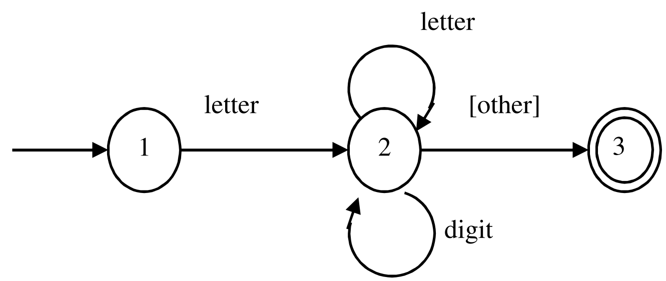
BACK

2.3.3 Implementation of Finite Automata in Code

Ways to Translate a DFA or NFA into Code

The code for the DFA accepting identifiers:

- { starting in state 1 }
- **if** *the next character is a letter* **then**
- *advance the input;*
- { now in state 2 }
- **while** the next character is a letter or a digit **do** advance the input; { stay in state 2 }
- **end while;**
- { go to state 3 without advancing the input}
- *accept;*
- **else**
- { error or other cases }
- **end if;**



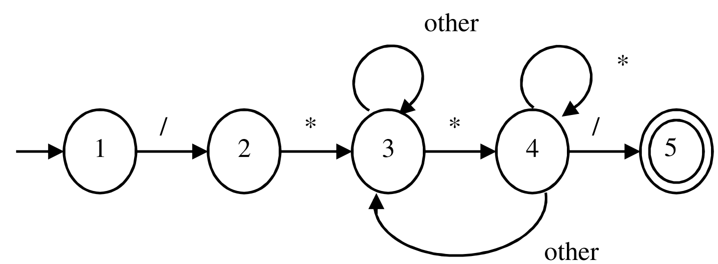
Two drawbacks:

- **It is ad hoc—that is, each DFA has to be treated slightly differently, and it is difficult to state an algorithm that will translate every DFA to code in this way.**
- **The complexity of the code increases dramatically as the number of states rises or, more specifically, as the number of different states along arbi-trary paths rises.**

Ways to Translate a DFA or NFA into Code

The Code of the DFA that accepts the C comments:

- { state 1 }
- **if** *the next character is "/"* **then** *advance the input; (* state *2 }*
- **if** *the next character is* " * " then
- *advance the input; {* state 3 }
- *done* := **false;**
- **while not** *done* **do**
- **while** *the next input character is not* "*" do
- *advance the input;* **end while;**
- *advance the input; (* state 4 }
- **while** *the next input character is* "*" do
- *advance the input;*
- **end while;**
- **if** *the next input character is "/"* **then**
- *done* := **true; end if;**
- *advance the input;* **end while;**
- *accept; {* state 5 }
- **else** { other processing }
- **end if;**
- **else** { other processing } **end if;**

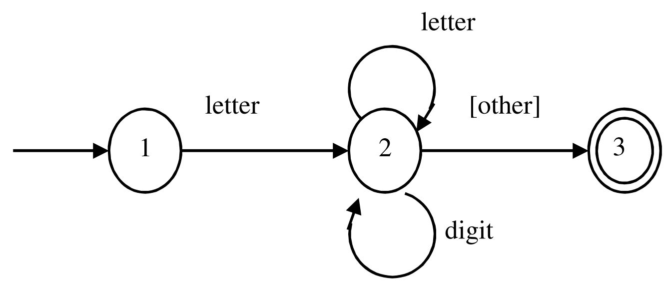


Ways to Translate a DFA or NFA into Code

A better method:

- Using a **variable to maintain the current state** and
- **writing the transitions as a doubly nested case statement inside a loop**,
- where the first case statement tests the current state and the nested sec-ond level tests the input character.

The code of the DFA for identifier:



- state := 1; { start }
- while state = 1 or 2 do
- case state of
- 1: case input character of
- letter: advance the input :
- state := 2;
- else state := ….{ error or other };
- end case;
- 2: case input character of
- letter , digit: advance the input;
- state := 2; { actually unnecessary }

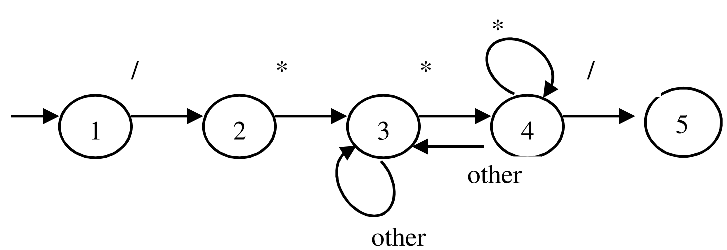
else state := 3;

- end case;
- end case;
- end while;
- if state = 3 then accept else error;

Ways to Translate a DFA or NFA into Code

The code of the DFA for C comments

- *state* := 1; { start }
- **while** *state* = 1, 2, 3 *or* 4 **do**
- ***case*** *state* ***of***
- **1: case** *input character* **of**
- "/" : *advance the input;*
- *state* := 2;
- **else** state :=...{ error or other};
- **end case;**
- **2: case** *input character* **of**
- "*": *advance the input;*
- *state* ::= 3;
- **else** *state* :=...{ error or other };
- **end case;**
- **3: case** *input character* **of**
- "*": *advance the input;*
- *state* := 4;
- **else** *advance the input* { and stay in state 3 };
- **end case;**
-
- **4: case** *input character* of
- "/" *advance the input;*
- *state* := 5;
- "*": *advance the input; {* and stay in state 4 }
- **else** *advance the input;*
- *state* := 3;
- **end case;**
- **end case;**
- **end while;**
- *if state* = 5 **then** *accept* **else** *error*;



Ways to Translate a DFA or NFA into Code

Generic code:

Express the DFA as a data structure and then write "generic" code;

A **transition table,** or two-dimensional array, indexed by state and input character that expresses the values of the transition function T

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

Ways to Translate a DFA or NFA into Code

The transition table of the DFA for identifier:

Input char / state	letter	digit	other	*Accepting*
1	2			No
2	2	2	[3]	no
3				yes

Brackets indicate "noninput-consuming" transitions

This column indicates accepting states

Assume :the first state listed is the start state

Ways to translate a DFA or NFA into Code

The transition table of the DFA for C comments:

Input char / state	/	*	Other	*Accepting*
1	2			no
2		3		no
3	3	4	3	no
4	5	4	3	no
5				yes

The code scheme:

- *state* := 1;
- *ch* := *next input character;*
- **while not** *Accept[state]* and not *error(state)* do
- *newstate* := *T[state,ch];*
- if *Advance[state,ch]* **then** *ch* := *next input char;*
- *state* := *newstate;*
- **end while;**
- **if** *Accept[state]* **then** *accept;*

Assumes :

- The transi-tions are kept in a transition array T indexed by states and input characters;
- The transi-tions that advance the input (i.e., those not marked with brackets in the table) are given by the Boolean array *Advance,* indexed also by states and input characters;
- Accepting states are given by the Boolean array *Accept,* indexed by states.

Features of Table-Driven Method

Table driven: use tables to direct the progress of the algorithm.

The advantage:

- The size of the code is reduced, the same code will work for many different problems, and the code is easier to change (maintain).

The disadvantage:

- The tables can become very large, causing a significant increase in the space used by the program. Indeed, much of the space in the arrays we have just described is wasted.
- Table-driven methods often rely on table-compression methods such as sparse-array representations, although there is usually a time penalty to be paid for such compression, since table lookup becomes slower. Since scanners must be efficient, these methods are rarely used for them.

NFAs can be implemented in similar ways to DFAs, except NFAs are nondeterministic,

- there are potentially many different sequences of transitions that must be tried.
- A program that simulates an NFA must store up transitions that have not yet been tried and backtrack to them on failure.

RET

End of Part One

THANKS