COMPILER CONSTRUCTION

Principles and Practice

Kenneth C. Louden

2. Scanning (Lexical Analysis)

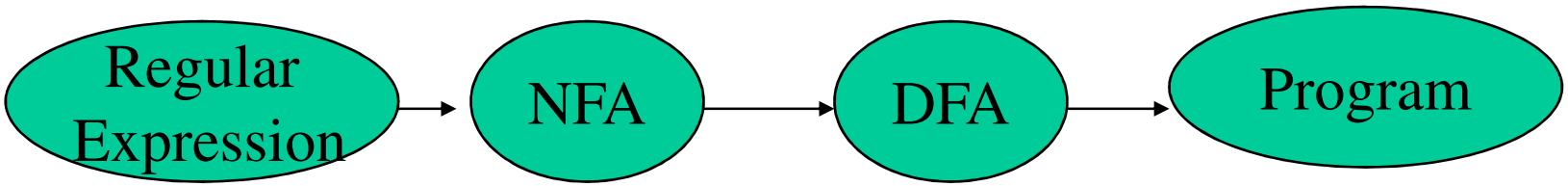
PART TWO

Contents

2.4 From Regular Expression To DFAs

Main Purpose

- Study an algorithm:
  - Translating a regular expression into a DFA via NFA.

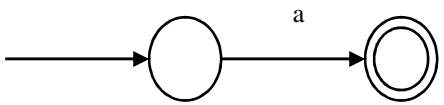


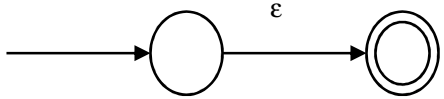
Contents

2.4.1 From a Regular Expression to an NFA

The Idea of Thompson's Construction

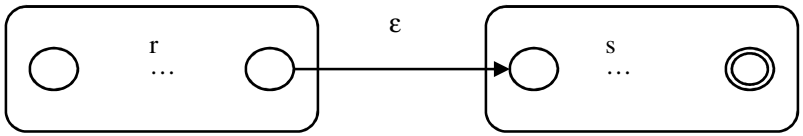
- Use ε-transitions
  - to ***"glue together"*** the machine of each piece of a regular expression
  - to form a machine that corresponds to the whole expression
- Basic regular expression
  - The NFAs for basic regular expression of the form a, ε,or φ





The Idea of Thompson's Construction

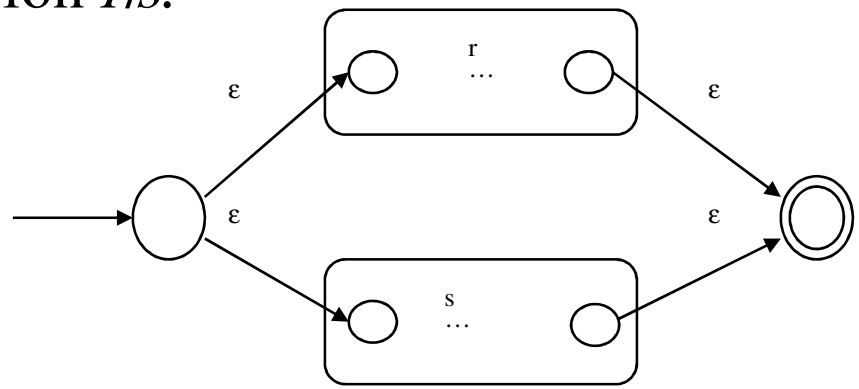
- Concatenation: to construct an NFA equal to ***rs***
  - *To* connect the accepting state of the machine of *r* to the start state of the machine of *s* by an ε-transition.
  - The start state of the machine of *r* as its start state and the accepting state of the machine of *s* as its accepting state.
  - This machine accepts $L(rs) = L(r)\,L(s)$ and so corresponds to the regular expression *rs*.



The Idea of Thompson's Construction

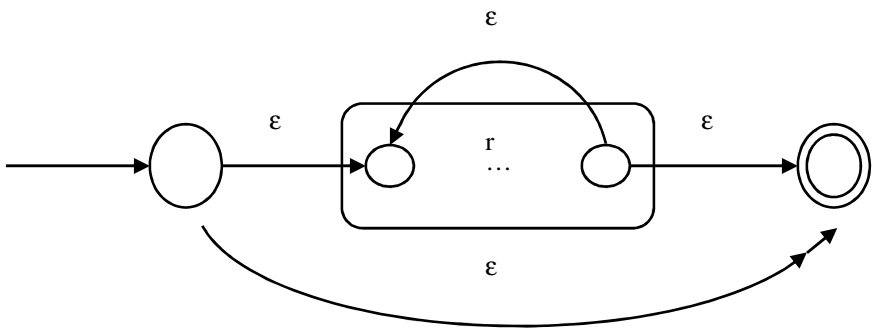
- Choice among alternatives: To construct an NFA equal to ***r / s***
  - To add a new start state and a new accepting state and connected them as shown using ε-transitions.
  - Clearly, this machine accepts the language

    $L(r/s) = L(r) \cup L(s)$, and so corresponds to the regular expression *r/s*.



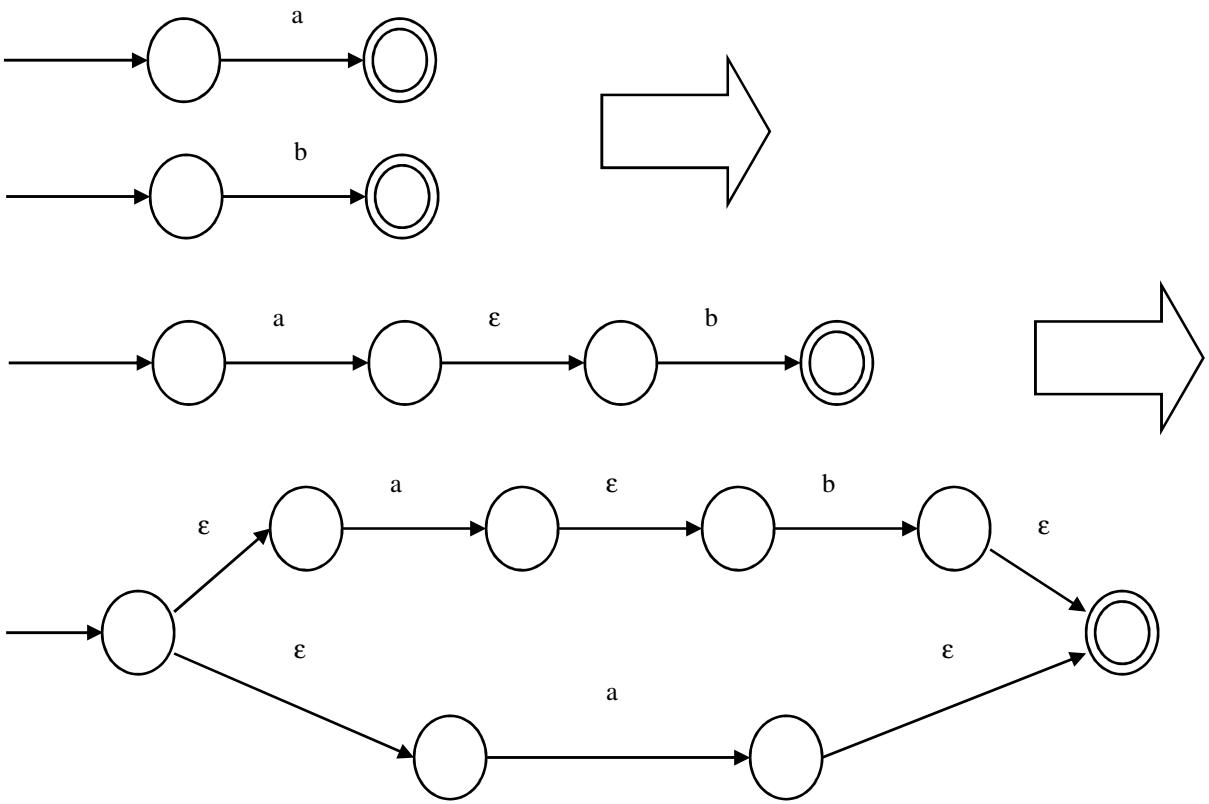
The Idea of Thompson's Construction

- Repetition: Given a machine that corresponds to *r*, Construct a machine that corresponds to r*
  - To add two new states, a start state and an accepting state.
  - The repetition is afforded by the newε-transition from the accepting state of the machine of *r* to its start state.
  - To draw an ε-transition from the new start state to the new accepting state.
  - This construction is not unique, sim-plifications are possible in the many cases.



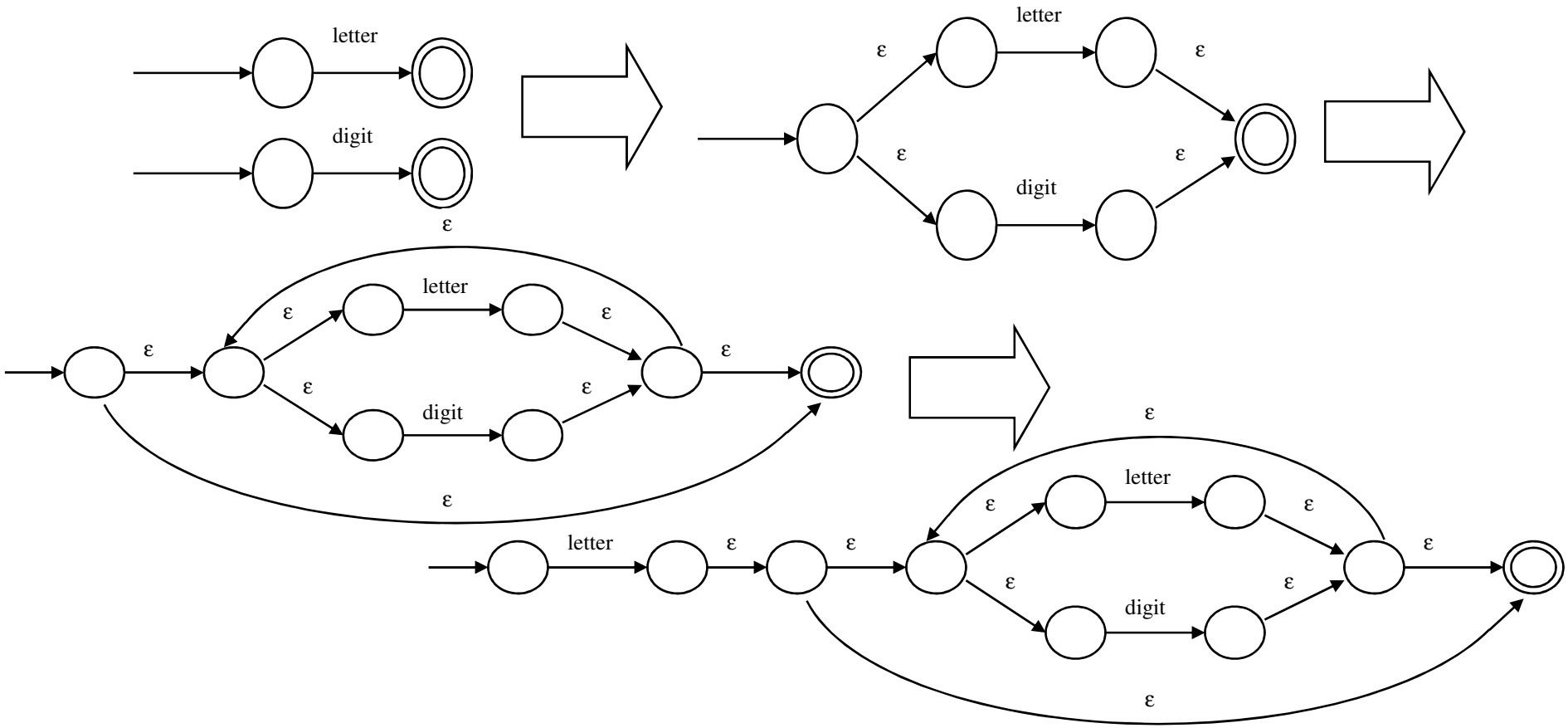
Examples of NFAs Construction

Example 1.12: Translate regular expression **ab|a** into NFA



Examples of NFAs Construction

Example 1.13: Translate regular expression *letter(letter|digit)** into NFA



RET

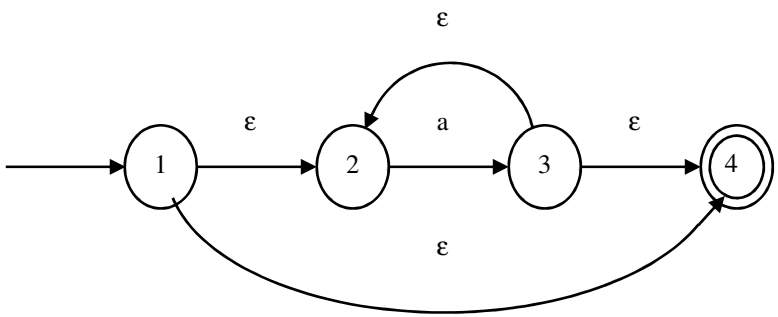
2.4.2 From an NFA to a DFA

Goal and Methods

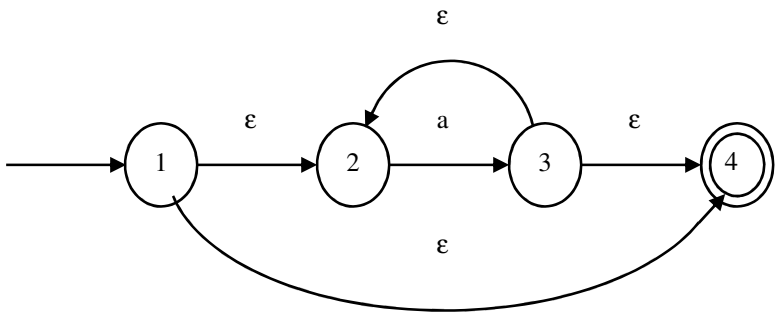
- **Goal**
  - ***Given an arbitrary NFA, construct an equivalent DFA. (i.e., one that accepts precisely the same strings)***
- Some methods
  - (1) Eliminating ε-transitions
    - **ε-closure**: the set of all states reachable by ε-transitions from a state or states
  - (2) Eliminating multiple transitions from a state on a single input character.
    - Keeping track of the set of states that are reachable by matching a single character
  - Both these processes lead us to *consider sets of states instead of single states.* Thus, it is not surprising that the DFA we construct has *sets of states* of the original NFA as its states.

The Algorithm Called **Subset Construction**.

- *The ε-closure of a Set of states:*
  - The ε-closure of a single state *s* is the set of states reachable by a series of zero or more ε-transitions, and we write this set as: $\overline{s}$
- Example 2.14: regular a*



The algorithm called **subset construction**.



$\bar{1} = \{1, 2, 4\}$, $\bar{2} = \{2\}$, $\bar{3} = \{2, 3, 4\}$, and $\bar{4} = \{4\}$.

The ε-closure of a set of states : the union of the ε-closures of each individual state.

$$\bar{S} = \bigcup_{\text{sin } S} \bar{s}$$

$\overline{\{1,3\}} = \bar{1} \cup \bar{3} = \{1, 2, 3\} \cup \{2, 3, 4\} = \{1, 2, 3, 4\}$

The *Subset Construction* Algorithm

(1) Compute the ε-closure of the start state of *M;* to obtain new state $\overline{M}$.

(2) For this set, and for each subsequent set, compute transitions on characters *a* as follows.

Given a set *S* of states and a character *a* in the alphabet,

Compute the set

$S'_a = \{ t \mid$ for some *s* in S there is a transition from *s* to *t* on *a* $\}$.

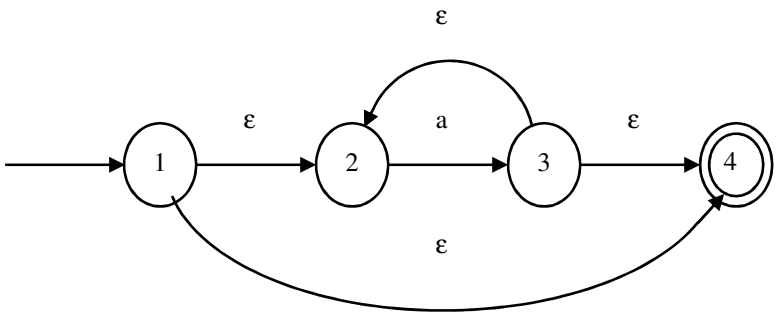
Then, compute $\overline{S_a}'$, the ε-closure of S'_a.

This defines a new state in the subset construction, together with a new transition S$\rightarrow$ $\overline{S_a}'$.

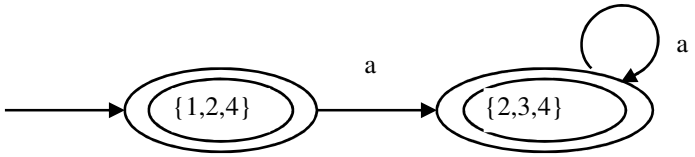
(3) Continue with this process until no new states or transitions are created.

(4) Mark as accepting those states constructed in this manner that contain an accepting state of M.

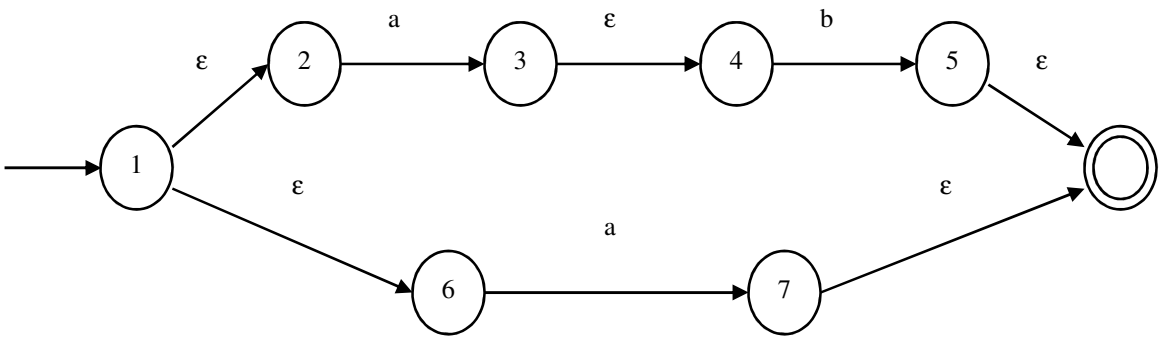
Examples of Subset Construction



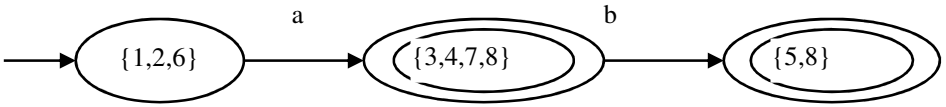
M	ε-closure of M (S)	S'_a
1	1,2,4	3
3	2,3,4	3



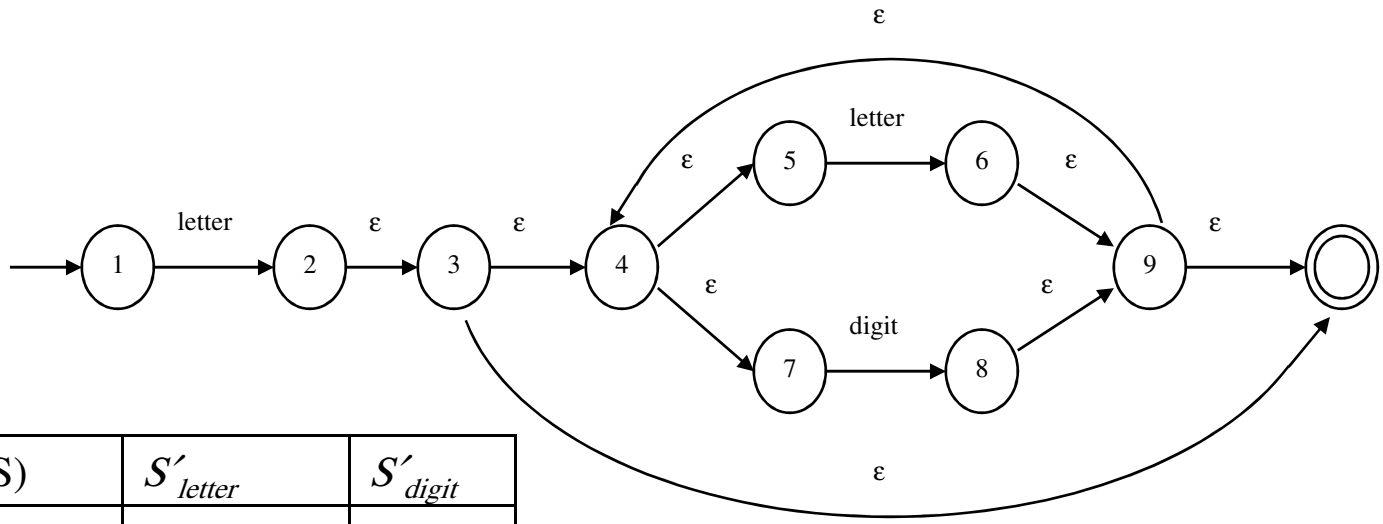
Examples of Subset Construction



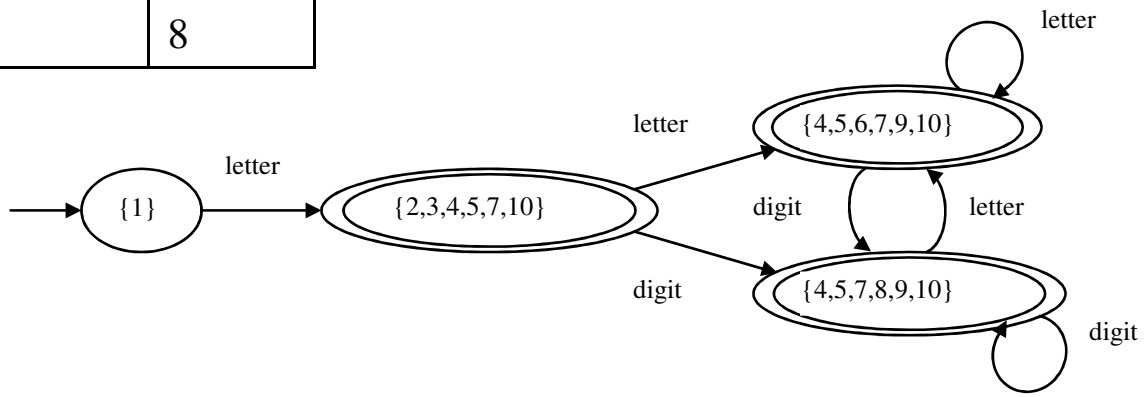
M	ε-closure of M (S)	S'_a	S'_b
1	1,2,6	3,7	
3,7	3,4,7,8		5
5	5,8		



Examples of Subset Construction

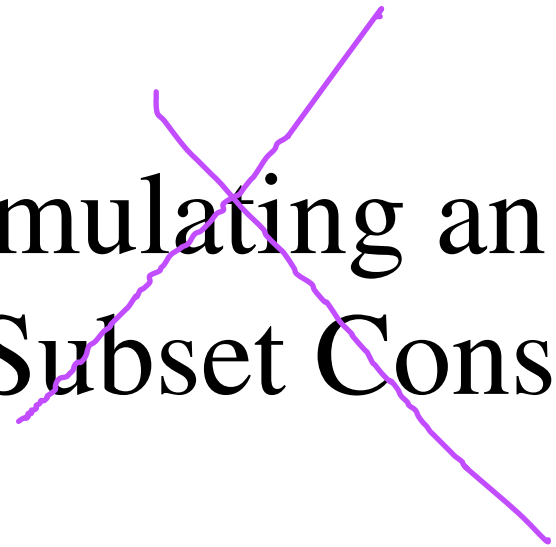


M	ε-closure of M (S)	S'_{letter}	S'_{digit}
1	1	2	
2	2,3,4,5,7,10	6	8
6	4,5,6,7,9,10	6	8
8	4,5,7,8,9,10	6	8



RET

2.4.3 Simulating an NFA using the Subset Construction

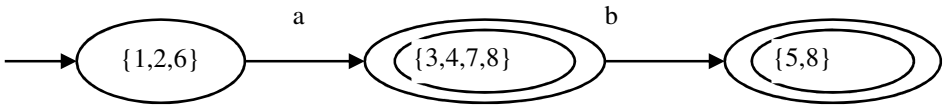


One Way of Simulating an NFA

- NFAs can be implemented in similar ways to DFAs, except that NFAs are nondeterministic
  - Many different sequences of transitions that must be tried.
  - Store up transitions that have not yet been tried and backtrack to them on failure.

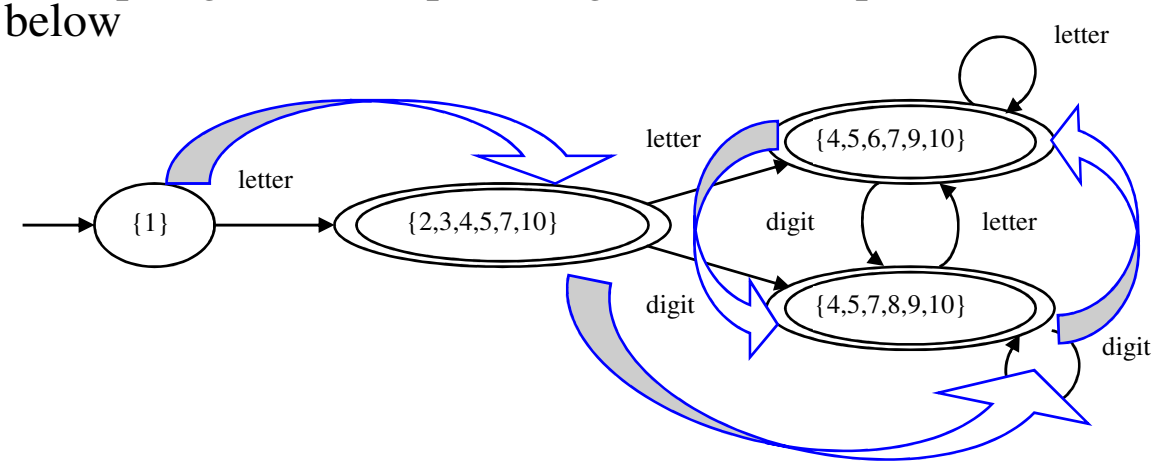
An Other Way of Simulating an NFA

- Use the subset construction
  - Instead of constructing all the states of the associated DFA
  - Construct only the state at each point that is indicated by the next input character
- The advantage: Not need to construct the entire DFA
  - Example: input single character *a*, construct the start state {1,2,6}and then the second state {3,4,7,8} to move and match the *a*.
  - Since no following *b*, accept without generating the state {5,8}



An Other Way of Simulating an NFA

- The disadvantage: A state may be constructed many times, if the path contains loops
  - Example: given the input string r2d3, the sequence of states as showing below



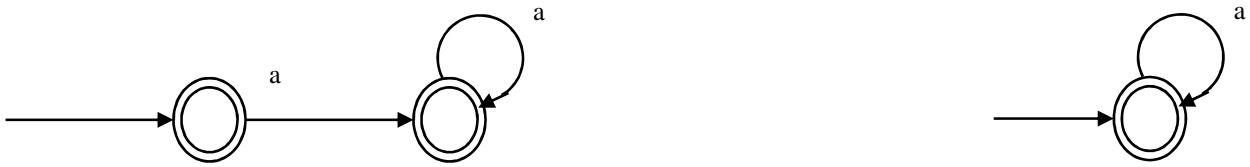
- If these states are constructed as the transitions occur, then the states of the DFA have been constructed and the state {4,5,7,8,9,10}has even been constructed twice
  - Less efficient than constructing the entire DFA

RET

2.4.4 Minimizing the Number of States in a DFA

Why need Minimizing ?

- The process of deriving a DFA algorithmically from a regular expression has the unfortunate property that
  - the resulting DFA may be more complex than necessary.
- The derived the DFA for the regular expression a* and an equivalent DFA



An Important Result from Automata Theory for Minimizing

- Given any DFA, there is an equivalent DFA containing a minimum number of states, and, that this minimum-state DFA is unique (except for renaming of states)

- It is also possible to directly obtain this minimum-state DFA from any given DFA.

Algorithm obtaining Mini-States DFA

1. It begins with the most optimistic assumption possible. Creates two sets: one consisting of all the accepting states and the other consisting of all the non-accepting states.

2. Given this partition of the states of the original DFA, consider the transitions on each character *a* of the alphabet.

   (1) If all accepting states have transitions on *a* to accepting states, then this defines an a-transition from the new accepting state (the set of all the old accept-ing states) to itself.

   (2) If all accepting states have transitions on *a* to non-accepting states, then this defines an a-transition from the new accepting state to the new non-accepting state (the set of all the old non-accepting states).

Algorithm obtaining Mini-States DFA

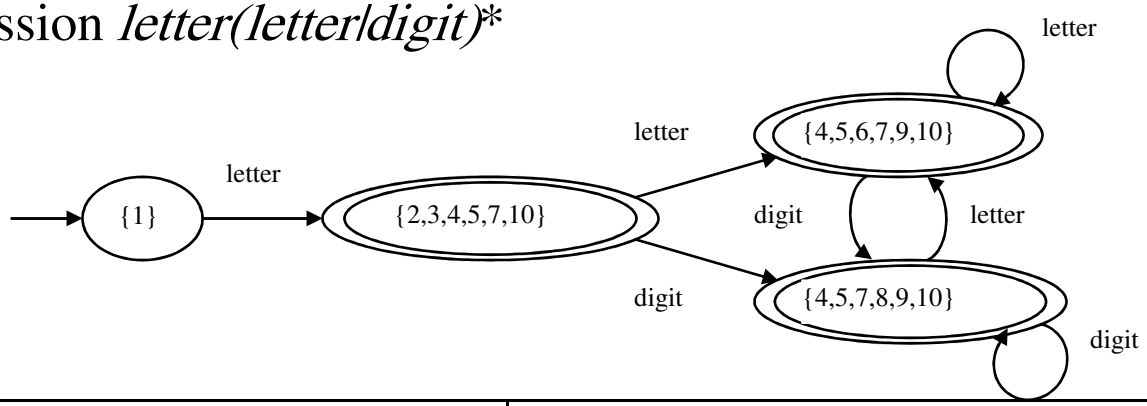
(3) On the other hand, if there are two accepting states *s* and *t* that have transitions on *a* that land in different sets, then no a-transition can be defined for this grouping of the states. We say that *a* distinguishes the states *s* and *t*

(4) We must also consider error transitions to an error state that is non-accepting. If there are accepting states *s* and *t* such that *s* has an a-transition to another accepting state, while *t* has no a-transition at all (i.e., an error transition), then a distinguishes *s* and *t.*

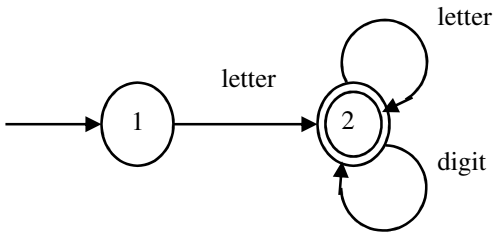
3. If any further sets are split, we must return and repeat the process from the beginning. This process continues until either all sets contain only one element (in which case, we have shown the original DFA to be minimal) or until no further splitting of sets occurs.

Examples of Minimizing DFA

Example 2.18: The regular expression *letter(letter|digit)**

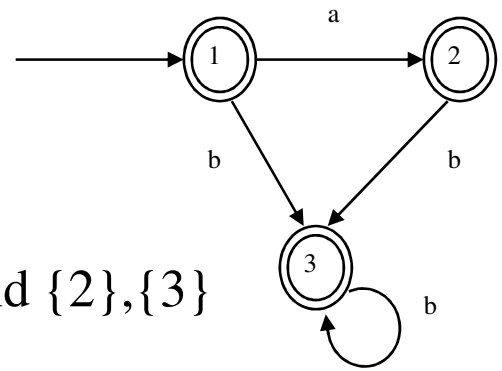


The accepting sets	{2,3,4,5,7,10},{4,5,6,7,9,10},{4,5,7,8,9,10}
The nonaccepting sets	{1}



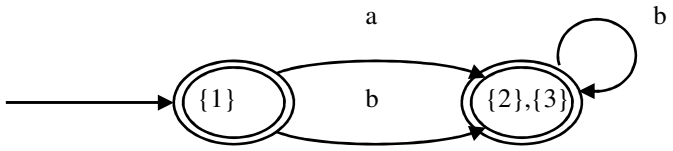
Examples of Minimizing DFA

Example 2.18: the regular expression (al ε)b*



a distinguishes state 1 from states 2 and 3,
and we must repartition the states into the sets {1} and {2},{3}

The accepting sets	{1},{2},{3}
The non-accepting sets	



RET

2.5 Implementation of a Tiny Scanner

The Tiny language

- The features of a program in TINY:
  - a sequence of statements separated by semicolons
  - no procedure, no declarations
  - all variables are integer,
  - two control statement : if-else and repeat
  - read and write statements
  - comments with curly brackets; but can not be nested
  - expressions are Boolean and integer arithmetic expressions (using < ,=), (+,-,* /, parentheses, constants, variables), Boolean expressions are only as tests in control statements.

One Sample Program in TINY: Factorial Function

```
Read x; {input an integer}
If x>0 then {don't compute if x <=0}
    Fact:=1;
    Repeat
        Fact :=fact *x;
        X:=x-1;
    Until x=0;
    Write fact {output factorial of x}
End
```

2.5.1 Implementing a Scanner for the Sample Language TINY

Defining the tokens and their attributes.

The tokens and token classes of TINY are summarized as follows:

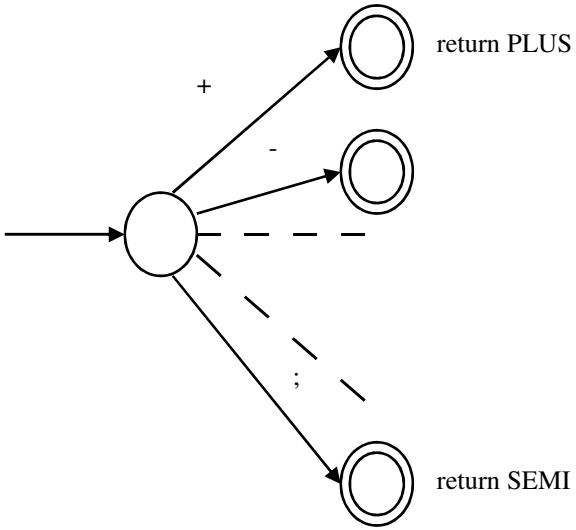
Reserved Words	Special Symbols	Other
if	**+**	**number**
then	**-**	**(1 or more digits)**
else	*****	
end	**/**	
repeat	**=**	
until	**<**	**identifier**
read	(	**(1 or more letters)**
write	)	
	;	
	:=	

TINY has the following **lexical conventions**.

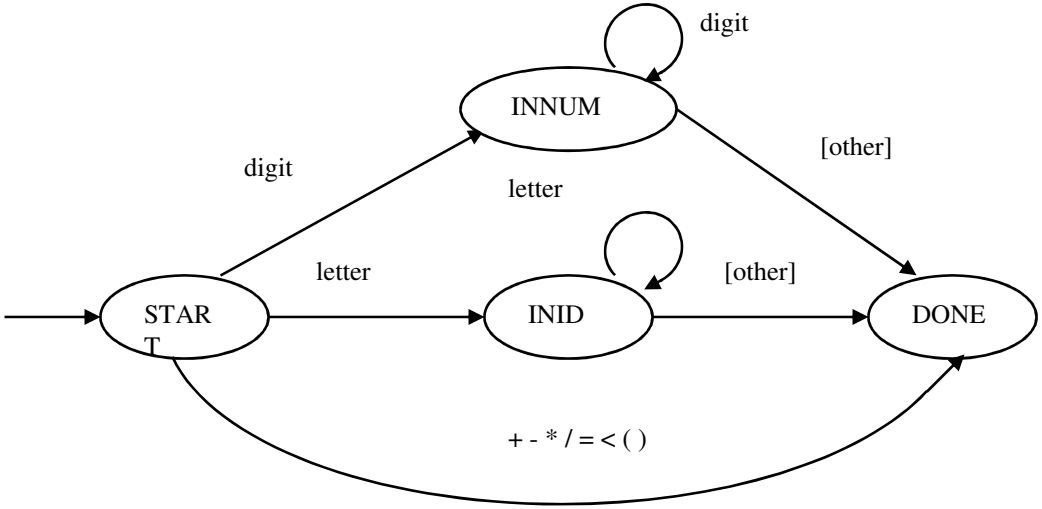
1. Comments are enclosed in curly brackets {…} and cannot be nested;
2. The code is free format; white space consists of blanks, tabs, and newlines;
3. The principle of longest substring is followed in recognizing tokens.

The DFAs for the Tokens of TINY

The DFA for the special symbols except assignment:

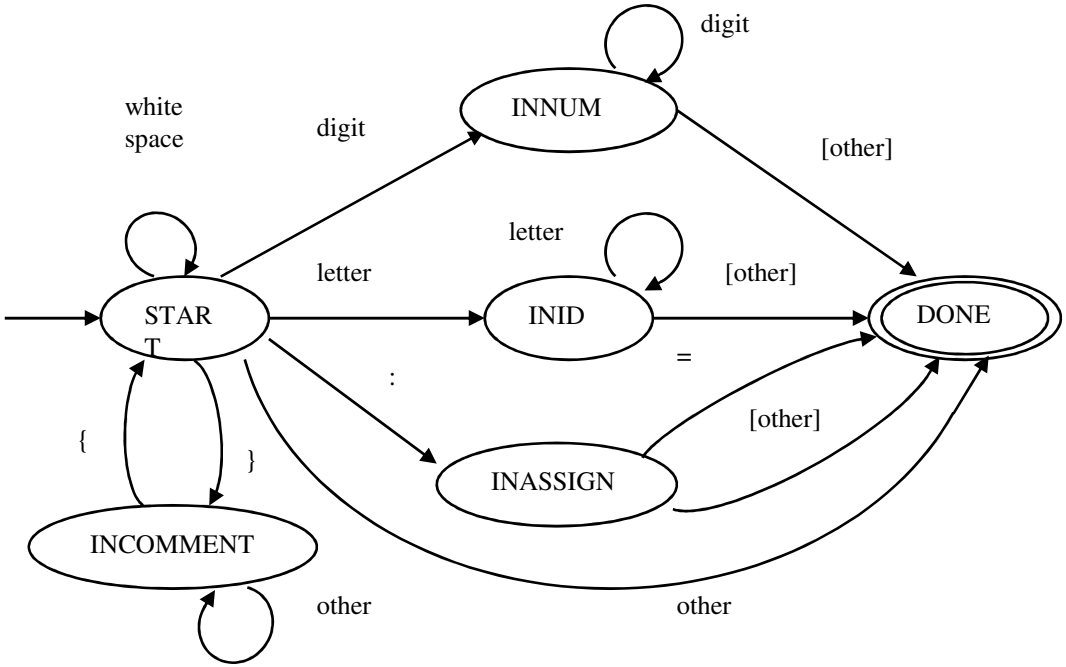


The DFA combined with DFAs that accept numbers and identifiers:



The DFAs for the Tokens of TINY

- The DFA extended by adding comments, white space, and assignment to this DFA
- The DFA considers reserved words to be the same as identifiers, and then to look up the identifiers in a table of reserved words



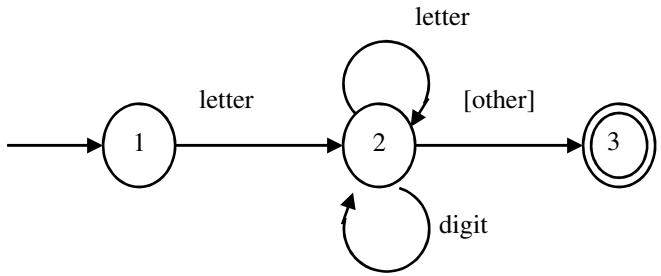
Ways to Translate a DFA or NFA into Code

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 second 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;

The Code to Implement This DFA

Appendix B :(p511-516) Scan.h and Scan.c

The principal procedure : **getToken** (lines 674-793)

- consumes input characters and returns the next token recognized according to the DFA
- uses the **doubly nested case analysis** described in Section 2.3.3,
- a large case list based on the state, within which are individual case lists based on the current input character.

The code to implement this DFA

Appendix B :(p511-516) Scan.h and Scan.c

The **tokens are defined as an enumerated type** in globals.h (lines 174-186)

- which include all the tokens listed above together with the bookkeeping tokens endfile (when the end of the file is reached) and ERROR (when an erroneous character is encountered)
- The states of the scanner are defined as an enumerated type, but within the scanner itself (lines 612-614).

The code to implement this DFA

Appendix B :(p511-516) Scan.h and Scan.c

The only attribute computed is the lexeme, or string value of the token recognized

- placed in the variable **tokenString**.

This variable and **getToken** are the only services offered to other parts of the compiler,

- Their definitions collected in the header file scan.h (lines 550-571).
- **tokenString** is declared with a fixed length of 41, so that identifiers cannot be more than 40 characters (plus the ending null character).

The code to implement this DFA

Appendix B :(p511-516) Scan.h and Scan.c

The scanner makes use of three global variables:

- the file variables **source** and **listing**,
- the integer variable **lineno** declared in **globals.h,** allocated and initialized in **main. c**.

The code to implement this DFA

Appendix B :(p511-516) Scan.h and Scan.c

The table **reservedWords** (lines 649-656} and the procedure **reservedLookup** (lines 658-666}

- perform a lookup of reserved words after an identifier is recognized by **getToken**
- the value of **current Token** is changed accordingly
- A flag variable **save** is used to indicate whether a character is to be added to **tokenString.**

The code to implement this DFA

Appendix B :(p511-516) Scan.h and Scan.c

Character input to the scanner is provided by the **getNextChar** function (lines 627-642),

- fetches characters from **lineBuf,** a 256-character buffer internal to the scanner.
- If the buffer is exhausted, **getNextChar** refreshes the buffer from the **source** file using the standard C procedure **fgets**,
- assuming each time that a new **source** code **line is** being fetched (and incrementing **lineno**).
- While this assumption allows for simpler code, a TINY program with lines greater than 255 characters will not be handled quite correctly

The code to implement this DFA

Appendix B :(p511-516) Scan.h and Scan.c

ungetNextChar procedure (lines 644-647) backs up one character in the input buffer.

Sample program in the TINY language

```
{       sample program
In TINY language  -
Computes  factorial
}
 read x; { input on integer }
 if  0 < x then  { don't compute if x <= 0 }
fact := 1 ;
repeat
fact := fact * x;
x := x - 1
until x = 0;
write fact { output factorial of x }
end
```

Output of scanner given the TINY program

```
TINY COMPILATION: sample.tny
 1:  { Sample program
 2:    in TINY language –
 3:    computes factorial
 4:  }
 5:  read x; { input an integer }
5: reserved word: read
5: id, name= x
5: ;
 6: if 0 < x then { don't compute if x <= 0 }
6: reserved word: if
6: mum, val= 0
6: <
6: id, name= x
6: reserved word: then
7: fact := 1;
7: id, name= fact
7: :=
7: num, val= 1
7: ;
8: repeat
8: reserved word: repeat
9: fact := fact * x;
9: id, name= fact
9: :=
9: id, name= fact
9: *
9: id, name= x
9: ;
10:x := x - 1
10: id, name= x
10: :=
10: id, name=x
10: -
10: mum, val = 1
11:until x = 0;
11: reserved word: until
11: id, name= x
11: =
11: mum, val= 0
11: ;
12: write fact { output factorial of x }
12: reserved words: write
12: id, name= fact
13:end
13: reserved word: end
14: EOF
```

2.5.2 Reserved Words Versus Identifiers

Recognizing Reserved Words

- **First considering them as identifiers and then looking them up in a table of reserved words**
  - Efficiency depends on the lookup process in the reserved word table
    - linear search
    - binary search
    - hash table (**Minimal perfect hash functions**)
- Reserved words use the same table that stores identifiers

2.5.3 Allocating space for identifiers

Some Issues for Allocation

- In the TINY scanner: token strings can only be a maximum of 40 characters, but identifiers may be arbitrarily long
- Allocate a 40-character array for each identifier, then much of the space is wasted
  - In TINY compiler, the utility function **copyString** allocates only the necessary space
  - A solution to the size limitation of **tokenString is to** only allocate space on an as needed basis, using the realloc standard C function.
- An alternative is to allocate an initial large array for all identifiers and then to perform do-it-yourself memory alloca-tion within this array

2.6 Use of Lex to Generate a Scanner Automatically

Introduction to Lex

- Use the Lex scanner generator to generate a scanner from a description of the tokens of TINY as regular expressions
- A number of different versions of Lex exist, the most popular version of Lex is called flex {for Fast Lex)

Introduction to Lex

- Lex is a program
  - Input : a text file containing regular expressions, together with the actions to be taken when each expression is matched
  - Output : Contains C source code defining a procedure yylex that is *a* table-driven implementation of a DFA corresponding to the regular expressions of the input file, and that operates like a **getToken** procedure
- The Lex output file, usually called lex.yy.c or lexyy.c
  - compiled and linked to a main program to get a running program.

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.

2.6.1 Lex conventions for regular expression

Conventions

- Matching of single characters, or strings of characters, by writing the characters in sequence.
- Metacharacters matched as actual characters by surrounding the characters in quotes; Quotes written around characters that are not metacharacters, where they have no effect.
  - match a left parenthesis, we must write " (“
  - an alternative is to use the backslash metacharacter \
  - match the character sequence (* , have to write \(* or " (* "

Conventions

- Metacharacters : *, +, (,) , |, ?
  - The set of strings of **a**'s and **b**'s that begin with either **aa** or **bb** and have an optical **c** at the end.
  - **(aa|bb)(a|b)*c? ("aa"|"bb")("a"|"b")*"c“**
- The Lex convention for character classes (sets of characters) is to write them between square brackets.
  - The above example cab be writen as:
  - (aa|bb) [ab]*c?

Conventions

- Ranges of characters written using a hyphen
  - The expression [0-9] means in Lex any of the digits zero through nine.
- A period is a metacharacter represents a set of characters:
  - It represents any character except a new-line.
- Complementary sets written in this notation, using the carat ^ as the first character inside the brackets
  - [^0-9abc] means any character that is not a digit and is not one of the letters *a, b,* or *c.*

Conventions

- One curious feature is that ***inside square brackets (representing a character class), most of the metacharacters lose their special status and do not need to be quoted.***
  - written [**-+**] instead of (**"+" | "-"**) . (but not **[+-]** because of the metacharacter use of - to express a range of characters).
  - **[."?]** means any of the three characters period, quotation mark, or question mark
- ***Some characters, however, are still metacharacters even inside the square brackets,*** and to get the actual character, we must precede the character by a backslash .
  - [\^ \ \] means either of the actual characters ^ or \.

Conventions

- A further important metacharacter convention in Lex is the use of curly brackets to denote names of regular expressions.
  - that a regular expression can be given a name, and that these names can be used in other regular expressions as long as there are no recursive references.
    - **nat [0-9]+**
    - **signedNat (+|-)?{nat}**

The table of Conventions

Pattern	Meaning
a	the character *a*
“a”	the character a, even if a is a metacharacter
\a	the character *a* when a is a metacharacter
a*	zero or more repetitions of a
a+	one or more repetitions of a
a?	an optional *a*
a|b	*a or b*
(a)	*a* itself
[abc]	any of the characters *a, b,* or c .
[a-d]	any of the characters *a. b, c.* or *d*
[^ab]	any character except *a* or *b*
.	any character except a newline
{xxx}	the regular expression that the name xxx represents

2.6.2 The format of a Lex input file

The format

{ definitions }
%%
{ rules }
%%
{ auxiliary routines

- The definition section occurs before the first %%.
  - any C code that must be inserted external to any function should appear in this section between the delimiters %{and %}
- Names for regular expressions must also be defined in this section.
  - A name is defined by writing it on a separate line starting in the first column and following it (after one or more blanks) by the regular expression it represents.

The format

{ definitions }

%%

{ rules }

%%

{ auxiliary routines}

- The second section: rules
- These consist of a sequence of regular expressions
  - followed by the C code that is to be executed when the corresponding reg-ular expression is matched.

The format

{ definitions }

%%

{ rules }

%%

{ auxiliary routines}

- The third section: auxiliary routines
- Routines are called in the second section and not defined elsewhere.
  - This section may also contain a main program, if we want to compile the Lex output as a standalone program.
  - This section can also be missing. (the second %% need not be written. The first %% is always necessary.)

Examples

(1) The following Lex input specifies a scanner that adds line numbers to text, sending its output to the screen.

```
%{
/*  a Lex program that adds line numbers
    to lines  of  text,  printing  the
    new  text to the standard  output
 */
#include <stdio.h>
int  lineno = l;
%}
line  .*\n
%%
{line}  { printf ("%5d  %s",lineno++,yytext) ; }
%%
main( )
{ yylex( );  return 0;  }
```

Examples

(1) Running the program obtained from Lex on this input file itself gives the following output:

```
 1  %{
 2  /*  a Lex program that adds line numbers
 3     to lines  of  text,  printing  the
 4     new  text to the standard
 5   */
 6   #include <stdio.h>
 7   int  lineno = l;
 8   %}
 9   line  .*\n
10  %%
11  {line}  { printf ("%5d  %s",lineno++, yytext) ; }
12  %%
13  main( )
14  { yylex( );  return 0;  }
```

Examples

Example 2.21 the Lex input file:

```
%{
/*  a Lex program that changes all numbers from decimal to hexadecimal notation,  printing a summary statiatic to stdeer
*/
#include  <stdlib.h>
#include  <stdio.h>
int  count=0;
%}
digit   [0-9]
number   {digit}+
%%
{number} { int  n = atoi(yytext);
          printf("%x", n);
          if (n > 9) count++;}
%%
main( )
{ yylex();
 fprintf(stderr, "number of replacements = %d", count);
 return 0;
 }
```

Examples

Example 2.22 the following Lex input file:

```
%{
/*  Selects only lines that end or
       begin with the letter  'a'.
   Deletes everything else.
*/
#include <stdio.h>
%}
ends_with_a   .*a\n
begins_with_a  a.*\n
%%
{ends_with_a}  ECHO;
{begins_with_a} ECHO;
.*\n  ;
%%
main( )
{ yylex( );  return 0; }
```

Additional feature of Lex input

- **Lex has a priority system for resolving such ambiguities**.
  - First, Lex always matches the longest possible substring {so Lex always generates a scanner that follows the longest substring principle).
  - Then, if the longest substring still matches two or more rules, Lex picks the first rule in the order they are listed in the action section.
- If the rules and actions as follows:
  - .*\n ;
  - {ends_with_a} ECHO;
  - {begins_with_a} ECHO;
- The program produced by Lex would generate no output at all for any file, since every line of input will be matched by the first rule.

Summary

- **Ambiguity resolution**
  - Lex's output will always first match the longest possible substring to a rule.
  - If two or more rules cause substrings of equal length to be matched, then Lex's output will pick the rule listed first in the action section.
  - If no rule matches any nonempty sub-string, then the default action copies the next character to the output and continues.

Summary

- **Insertion of c code**
  - Any text written between %{ and %} in the definition section will be copied directly to the output program external to any procedure.
  - Any **text** in the auxiliary procedures section will be copied directly to the output program at the end of the Lex code.
  - Any code that follows a regular expression (by at least one space) in the action section (after the first %%) will be inserted at the appropriate place in the recognition procedure **yylex** and will be executed when a match of the corresponding regular expression occurs.
  - The C code representing an action may be either a single C statement or a compound C statement consisting of any declara-tions and statements surrounded by curly brackets.

Lex internal names

Lex Internal Name	Meaning/Use
lex.yy.c or lexyy.c	Lex output file name
yylex	Lex scanning routine
yytext	string matched on current action
yyin	Lex input file (default: stdin)
yyout	Lex output file (default: stdout)
input	Lex buffered input routine
ECHO	Lex default action (print yytext to yyout)

2.6.3 A tiny scanner using Lex

Appendix B gives a listing of a Lex input file tiny.l

End of Chapter Two

THANKS