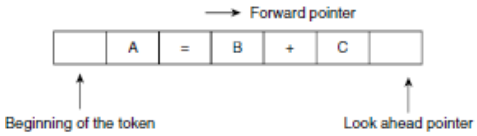
1a)

(i) As characters are read from left to right, each character is stored in the buffer to form a meaningful token as shown below:



We introduce a two-buffer scheme that handles large look aheads safely. We then consider an improvement involving "sentinels" that saves time checking for the ends of buffers.

Buffer Pairs

- A buffer is divided into two N-character halves, as shown below

Each buffer is of the same size N, and N is usually the number of characters on one disk block. E.g., 1024 or 4096 bytes.

- Using one system read command we can read N characters into a buffer.
- If fewer than N characters remain in the input file, then a special character, represented by **eof**, marks the end of the source file.

Two pointers to the input are maintained:

1. Pointer **lexeme beginning** marks the beginning of the current lexeme, whose extent we are attempting to determine.
2. Pointer **forward** scans ahead until a pattern match is found. Once the next lexeme is determined, forward is set to the character at its right end.

The string of characters between the two pointers is the current lexeme. After the lexeme is recorded as an attribute value of a token returned to the parser, lexeme beginning is set to the character immediately after the lexeme just found.

Advancing forward pointer:

Advancing forward pointer requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer. If the end of second buffer is reached, we must again reload the first buffer with input and the pointer wraps to the beginning of the buffer.

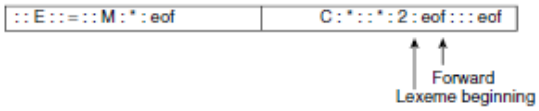
Code to advance forward pointer:

```
if forward at end of first half then begin
reload second half;
forward := forward + 1
end
else if forward at end of second half then begin
reload second half;
move forward to beginning of first half
end
else forward :− forward + 1;
```

Sentinels

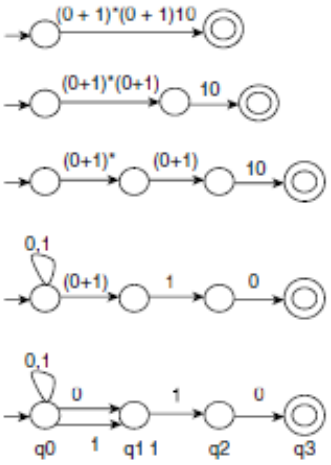
- For each character read, we make two tests: one for the end of the buffer, and one to determine what character is read. We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a sentinel character at the end.
- The sentinel is a special character that cannot be part of the source program, and a natural choice is the character eof.

The sentinel arrangement is as shown below:



1b)

Construct the minimized DFA for the regular expression



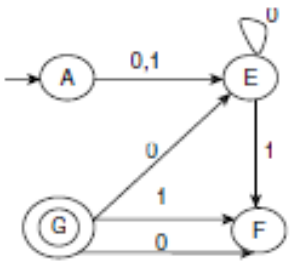
Transition diagram for above diagram

States	0	1
q0	{q0,q1}	{q0,q1}
q1	-	(q2)
q2	{q3}	-
*q3	-	-

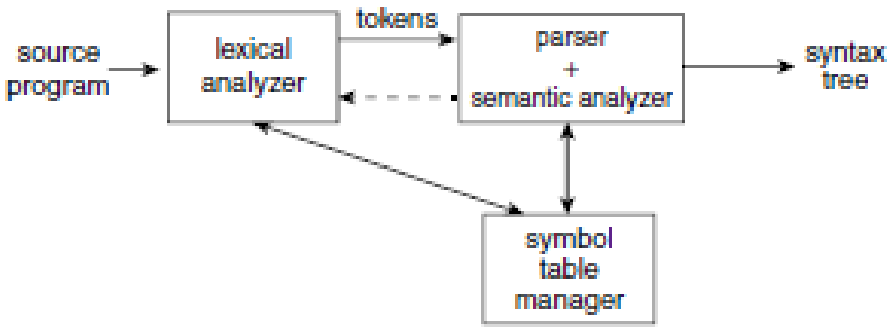
Minimized DFA Table

New State	States	0	1
A	[q0]	[q0,q1]	[q0,q1]
B	[q1]	-	[q2]
C	[q2]	[q3]	-
D	*[q3]	-	-
E	[q0,q1]	[q0,q1]	[q0,q1,q2]
F	[q0,q1,q2]	[q0,q1,q2]	[q0,q1,q2]
G	*[q0,q1,q3]	[q0,q1]	[q0,q1,q2]

Minimized DFA Diagram:



2 a)



Lexical analysis is the process of converting a sequence of characters into a sequence of tokens. A program or function which performs lexical analysis is called a lexical analyzer or scanner. A lexer often exists as a single function which is called by a parser or another function.

The role of the lexical analyzer

- The lexical analyzer is the first phase of a compiler.
- Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis.
- Upon receiving a "get next token" command from the parser, the lexical analyzer reads input characters until it can identify the next token.

Error recovery strategies in lexical analysis:

The following are the error-recovery actions in lexical analysis:

1) Deleting an extraneous character.
2) Inserting a missing character.
3) Replacing an incorrect character by a correct character.
4) Transforming two adjacent characters.
5) Panic mode recovery: Deletion of successive characters from the token until error is resolved.

2 b) Via wikipedia, we can categorize the single noman numerals into 4 groups:
I, II, III | I V | V, V I, V II, V III | I X
then get the production:
digit -> smallDigit | I V | V smallDigit | I X
smallDigit -> I | II | III | ε
and we can find a simple way to map roman to arabic numerals. For example:
XII => X, II => 10 + 2 => 12
CXCIX => C, XC, IX => 100 + 90 + 9 => 199
MDCCCLXXX => M, DCCC, LXXX => 1000 + 800 + 80 => 1880
via the upper two rules, we can derive the production:
romanNum -> thousand hundred ten digit
thousand -> M | MM | MMM | ε
hundred -> smallHundred | C D | D smallHundred | C M
smallHundred -> C | CC | CCC | ε
ten -> smallTen | X L | L smallTen | X C
smallTen -> X | XX | XXX | ε
digit -> smallDigit | I V | V smallDigit | I X
smallDigit -> I | II | III | ε

3 a)

Intermediate code generation:

Intermediate code generation gets input from the semantic analysis and converts the input into output as intermediate code such as three-address code.

This code is in variety of forms as quadruple, triple and indirect triple .The three-address code consists of a sequence of instructions, each of which has almost three operands.

Example

t1:=int to float(60)
t2:= rate *t1
t3:=initial+t2
position:=t3

Code Optimization:

Code Optimization gets the intermediate code as input and produces optimized intermediate code as output. This phase reduces the redundant code and attempts to improve the intermediate code so that faster-running machine code will result.

During the code optimization, the result of the program is not affected.

To improve the code generation, the optimization involves

– deduction and removal of dead code (unreachable code).
– calculation of constants in expressions and terms.
– collapsing of repeated expression into temporary string.
– loop unrolling.
– moving code outside the loop.
– removal of unwanted temporary variables.

Eample:

t1:= rate *60
position:=initial+t1

Code Generaton:

Code Generation gets input from code optimization phase and produces the target code or object code as result.

Intermediate instructions are translated into a sequence of machine instructions that

Perform the same task.

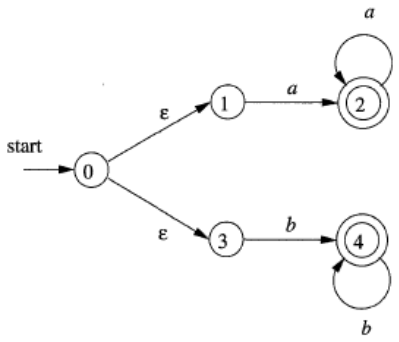
The code generation involves

– allocation of register and memory
– generation of correct references
– generation of correct data types
– generation of missing code

Machine instructions:

MOV rate, R1
MUL #60, R1
MOV initial, R2
ADD R2, R1
MOV R1, position

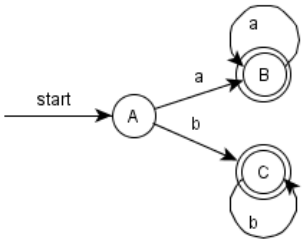
3 b)



Transition table

NFA State	DFA State	a	b
{0,1,3}	A	B	C
{2}	B	B	ø
{4}	C	ø	C

DFA



4 a)

4.1.1 The Role of the Parser

In our compiler model, the parser obtains a string of tokens from the lexical analyzer, as shown in Fig. 4.1, and verifies that the string of token names can be generated by the grammar for the source language. We expect the parser to report any syntax errors in an intelligible fashion and to recover from commonly occurring errors to continue processing the remainder of the program. Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing. In fact, the parse tree need not be constructed explicitly, since checking and translation actions can be interspersed with parsing, as we shall see. Thus, the parser and the rest of the front end could well be implemented by a single module.

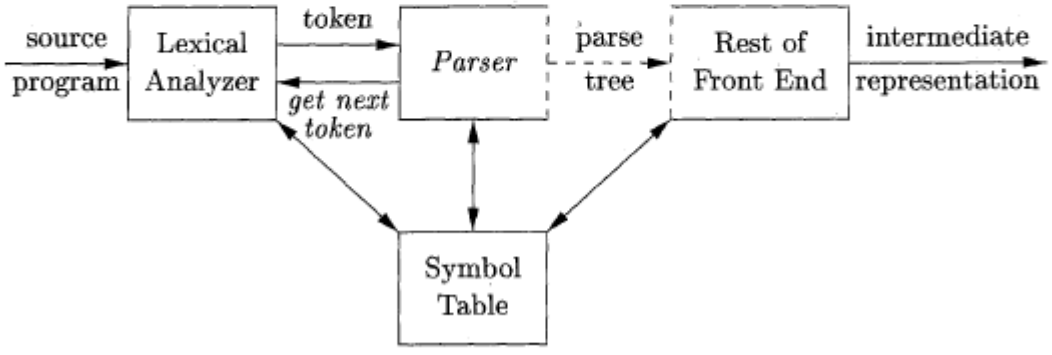


Figure 4.1: Position of parser in compiler model

Common programming errors can occur at many different levels.
Lexical errors include misspellings of identifiers, keywords, or operators -e.g., the use of an identifier ellipse size instead of ellipse size – and missing quotes around text intended as a string.

Syntactic errors include misplaced semicolons or extra or missing braces; that is, '((" or ")." As another example, in C or Java, the appearance of a case statement without an enclosing switch is a syntactic error (however, this situation is usually allowed by the parser and caught later in the processing, as the compiler attempts to generate code).
Semantic errors include type mismatches between operators and operands. An example is a return statement in a Java method with result type void.
Logical errors can be anything from incorrect reasoning on the part of the programmer to the use in a C program of the assignment operator = instead of the comparison operator ==. The program containing = maybe well formed; however, it may not reflect the programmer's intent.
4 b)

```
E -> E E op | num
```

```
2. list -> list , id | id
```