Translation Software

Lexical Analysis (S1)

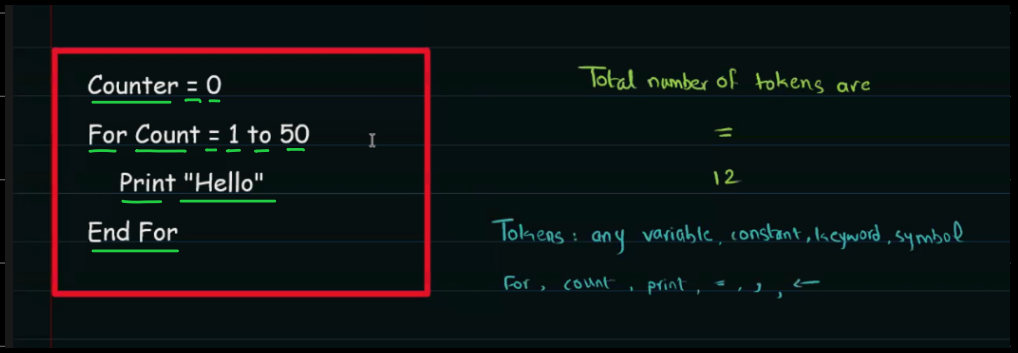
- Unnecessary Characters (white spaces, comments) are removed

- Source program is converted to tokens

Tokenization

Keyword Table

- Reserved words (FOR, TO, OUTPUT)
- Operators
- Their matching tokens



Symbol Table

* See question from papersdock's notes

- Identifiers (variable, constant)
- The data type
- Role (variable, constant, array, procedure)
- Value / location

Q- How the contents of the keyword and symbol tables are used to translate the source code of the program?

- keywords/operators are looked up in the keyword table
- keywords/operators are represented by the tokens
- Identifiers are looked up in the symbol table
- Identifiers are converted to locations/addresses
- Used to create a sequence of tokens for the program.

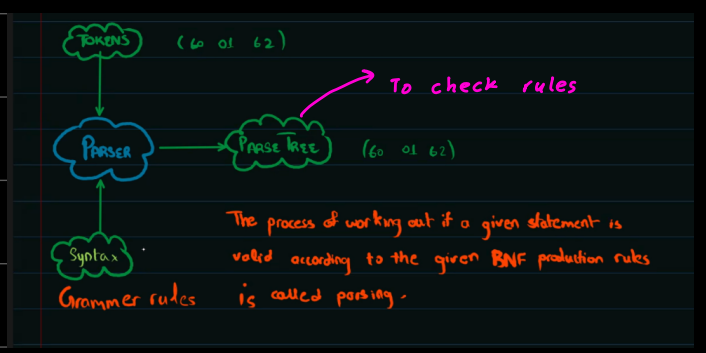
Q- Additional task completed at the lexical analysis stage that does not involve the use of a keyword or a symbol table.

- White spaces are removed
- Redundant characters are removed
- Removal of comments

Syntax Analysis

Tasks:

- Constructing parse tree / parsing
- Checking the table of tokens to ensure that the rules/syntax/ grammar of the language are obeyed.
- Producing error report.

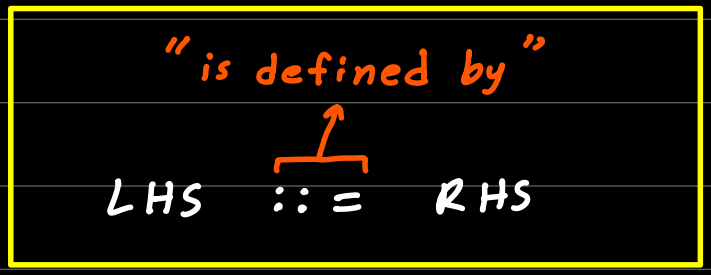


- The grammatical rules or syntax of a programming language need to be set out clearly, so a programmer can write code that obeys the rules and a

compiler can be built to check that a program obeys these rules. The rules can be shown graphically in a syntax diagram or using a meta language such as Backus-Naur Form notation.

Backus-Naur Form

- Is a mathematical way of defining syntax



<Digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 Note: "|" = or

Q- Number contains only 2 digits

<Number> ::= <Digit><Digit>

Q- A Number can only contain three digits.

<Number> ::= <Digit><Digit><Digit>

Q- A Number can be either a single digit or a two digit number

<Number> ::= <Digit> | <Digit><Digit>

<Letter> ::= A | B | C | D | E | F | ....

Q- A word consists of any number of letters

<word> ::= <Letter> | <Letter><word> → Recursion

Q- A Number consists of any number of digits

<Number> ::= <Digit> | <Digit><Number>

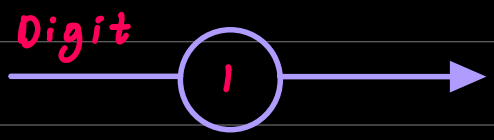
Assignment Statement

- Number = N1 + N2 ;

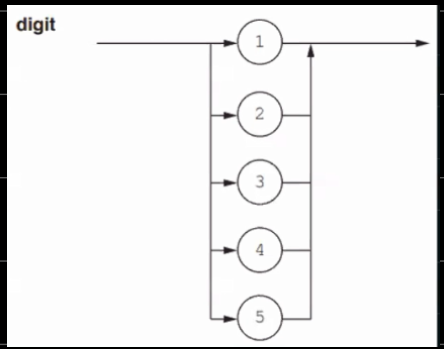
<Assignment Statement> ::= <variable> = <variable> <operator> <variable> ;

Syntax Diagram

Q- <Digit> ::= 1



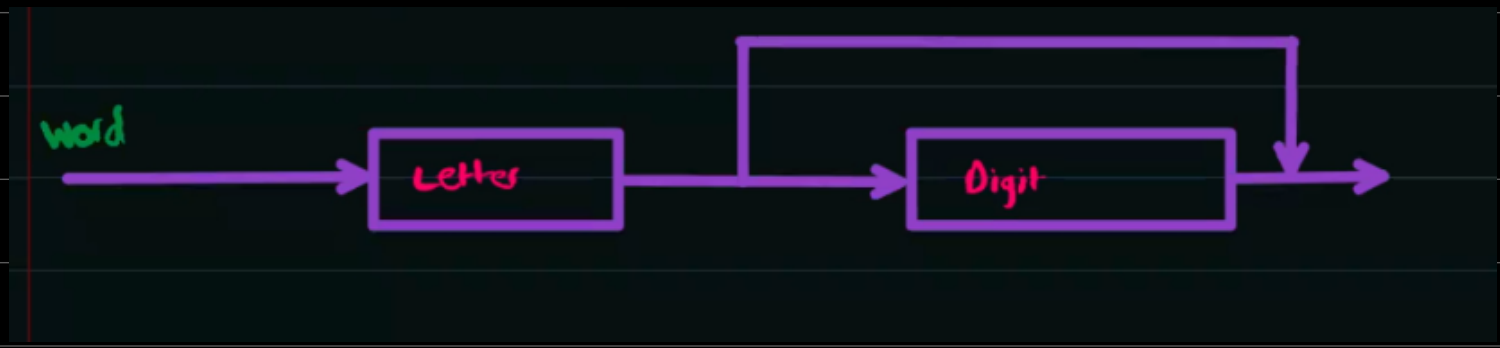
Q- <Digit> ::= 1 | 2 | 3 | 4 | 5



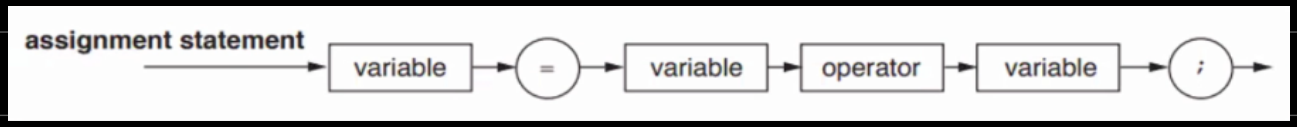
· Constants are represented in circles

Q- < Word > ::= < Letter > | < Letter >< word >

Q- < word > ::= < Letter > | < Letter >< Digit >



< Assignment Statement > ::= < variable > = < variable > < operator > < variable > ;



Code Generation (S3)

- The code generation stage produces an object code
- The program should have correct syntax
- The object code is in machine-readable form.

Code Optimization (S4)

Q. Why Code Optimization is necessary?

- Optimization means the code will have fewer instructions
- Optimized code occupies less space in memory
- Fewer instructions reduces the execution time of the program.

E.g

Original code	w = x + y v = x + y + z	Object code	LDD x ADD y STO w LDD x ADD y ADD z STO v
Optimised code	w = x + y v = w + z	Object code	LDD x ADD y STO w ADD z STO v

Benefits for optimizing code:

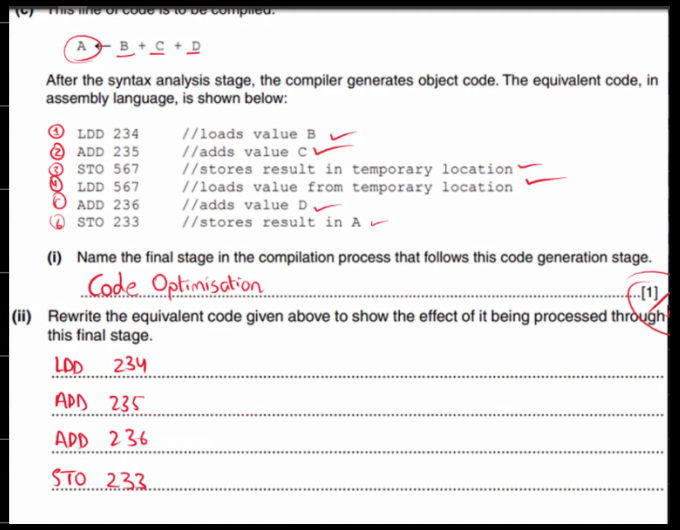
- Code has fewer instructions/occupies less space in memory
- time taken to execute the whole program decreases.

Reverse Polish Notation

- It is a method of expressing an arithmetic expression

Infix Form: A - B // Human understandable form of arithmetic expression

Reverse Polish Notation: A B -



(c) This line of code is to be compiled.

A ← B + C + D

After the syntax analysis stage, the compiler generates object code. The equivalent code, in assembly language, is shown below:

```
1  LDD 234     //loads value B
2  ADD 235     //adds value C
3  STO 567     //stores result in temporary location
4  LDD 567     //loads value from temporary location
5  ADD 236     //adds value D
6  STO 233     //stores result in A
```

(i) Name the final stage in the compilation process that follows this code generation stage.

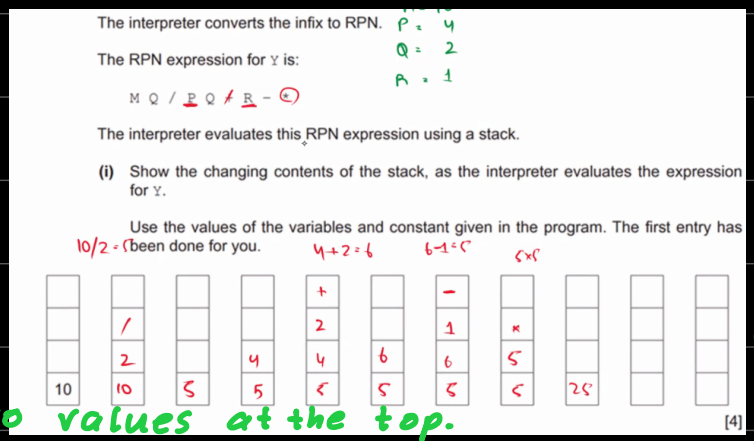
Code Optimisation [1]

(ii) Rewrite the equivalent code given above to show the effect of it being processed through this final stage.

LDD 234
ADD 235
ADD 236
STO 233

CASE 1: Using RPN to evaluate expressions using stacks

- Always solve left to right
- Whenever operators enter the stack, use that operator on two values at the top.
- In division, numerator is the bigger number and in subtraction smallest number is subtracted from bigger number.



The interpreter converts the infix to RPN.

The RPN expression for Y is:

M Q / P Q ↑ R - ×

P = 4
Q = 2
R = 1

The interpreter evaluates this RPN expression using a stack.

(i) Show the changing contents of the stack, as the interpreter evaluates the expression for Y.

Use the values of the variables and constant given in the program. The first entry has been done for you.

10/2 = 5 4+2 = 6 6-1 = 5 5×5

				+		-					
	/			2		1	×				
	2		4	4	6	6	5				
10	10	5	5	5	5	5	5	25			

[4]

CASE 2 : Infinix to RPN Conversion.

Q1- A + B

= AB+

2- A - B * C

Note: Use BODMAS rule always.

= A - BC *

= ABC * -

3- (P + Q) + (R - P)

= (PQ+) + (RP-)

= PQ+RP-+

4- (a - b) * (a + c) / 7

Note: · Power of divide and multiply is same
· Power of add and subtract is same
· Always choose from left to right in case if *, / or +, - is encountered at the same time.
· If not, then apply BODMAS rule.

= ab- * ac+ / 7

= ab-ac+* / 7

= ab-ac+*7/

5- P + Q - R / S

= PQ+ - RS/

= PQ+RS/-

6- - A / B * 4 / (C - D)

= -A/B*4/CD-

= A- / B * 4 / C D-

= A-B/ * 4 / CD-

= A-B/4* / CD-

= A-B/4*CD-/

CASE 3: RPN to Infinix Conversion

Note: Use stack concept to solve such questions

Q1- AB-

= A-B

2- PQ+ M * RP-- Always start from left to right

= (P+Q) M * RP--

= (P+Q) * M RP--

= (P+Q) * M (R-P)-

= (P+Q) * M - (R-P)

3- x wz + y - *

= x (w+z) y - *

= x (w+z) - y *

= x * [(w+z) - y] * // x * (w+z-y)

4- ba * cda++ -

= b * a cda++-

= (b * a) c (d+a)+-

= (b * a) (c+d+a)-

= (b * a) - (c+d+a)

CASE 4: Theoratical Questions

Q1- Explain how RPN is used by an interpreter to evaluate expressions

- Expressions are always evaluated left to right
- Each operator uses the two previous values on the stack (except unary minus)
- If element is a number, push that number onto the stack.
- If element is an operator, then pop the first two numbers from stack
- Perform that operation on those numbers
- Push the result back onto stack
- End, once the last item in the expression has been dealt with.

Advantages for using RPN for the evaluation for an expression.

- No need for rules of precedence (BODMAS)
- No need for brackets
- In RPN, evaluation of operators is always left to right

Q- What does the statement "No need for rules of precedence" means?

- Rules of precedence means different operators have different priorities (Multiply is done before addition)
- In RPN, evaluation of operators is left to right
- No need for brackets

Q- Identify with reasons, a data structure that could be used to evaluate an expression in RPN.

- Structure: STACK
- Reason: Two operands are popped from the stack in the reverse order to how they were pushed.

Q- Explain how an interpreter executes a program without producing a complete translated version of it.

- An interpreter examines source code's one statement at a time.
- Checks each statement for errors
- If no error is found, the statement is executed
- If an error is found, it is reported and the interpreter halts
- Interpretation is repeated for every iteration in repeated sections of code
- Interpretation has to be repeated every time the program is run.