3.4.3 Translation software

Introduction

Stating the obvious, people and computers do not speak the same language.

People have to write programs in order to instruct a computer what to do. This text is called the **"source code"** and most often it is written in a high level language that people find easy to use and write. On the other hand, a computer runs only binary code '1010101011110101011' within its registers which is almost incomprehensible to people, this is called the **"machine code"** or the **"object code"**

So there needs to be something to convert people-friendly source code into computer-friendly machine code. **This is the role of translators.** In this section, we will discuss translators in depth.

Interpreters and Compilers

When electronic computers were first used, the programs had to be written in machine code. This code was comprised of simple instructions each of which was represented by a binary pattern in the computer. To produce these programs, programmers had to write the instructions in binary. **This not only took a long time, it was also prone to errors.** To improve program writing, assembly languages were developed. Assembly languages allowed the use of mnemonics and names for locations in the memory. Each assembly instruction mapped to a single machine instruction which meant that it was fairly easy to translate a program written in assembly.

To speed up this translation process, **assemblers** were written which could be loaded into the computer and then the computer could translate the assembly language into a machine code; **this process was still tedious and took a long time**.

After assembly languages, came high-level languages which programmers nowadays (and maybe even you) use to write code instructions for the computer to carry out. Several high-level languages were developed. **FORTRAN** (**FOR**mula **TRAN**slation) was developed for science and engineering programs and it used formulae in the same way as would scientists and engineers. Similarly **COBOL** (**C**ommon **B**usiness **O**riented **L**anguage) was developed for business applications. Programs written in these languages needed to be translated to machine code. This led to the birth of compilers.

3.4.3 Translation software

A compiler takes a program written in high-level languages and translates into an equivalent program in machine code. Once this is done, the machine code version can be loaded into the machine and run without any further help as it is complete in itself. The high-level language version of the program is usually called the **source code** and the resulting machine code program is called the **object code**.

Fig. 3.2.a.1

The problem with using a compiler is that it uses a lot of computer resources. It has to be loaded in the computer's memory at the same time as the source code, and there has to be sufficient memory for working storage while the translation is taking place. Another con is that when an error occurs in a program, it is difficult to pin-point its source in the original program.

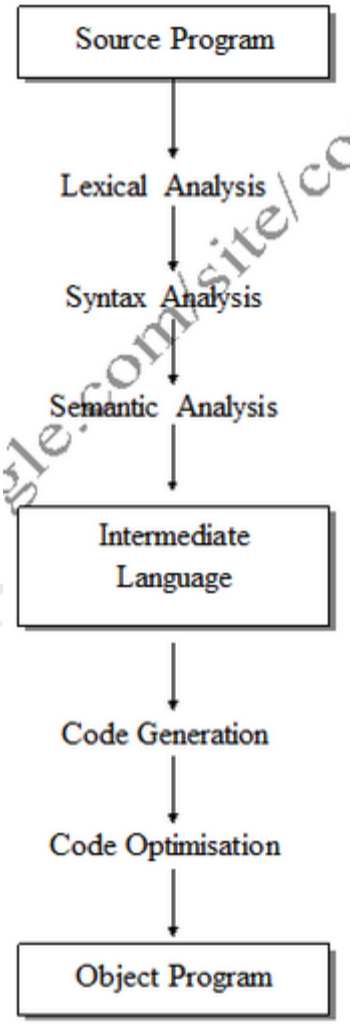
An alternative system is to use interpretation. In this system, each instruction is taken in turn and translated to machine code. The instruction is then executed before the next instruction is translated. This system was developed mainly because early personal computers lacked the power and memory needed for compilation. This method also has the advantage of producing error messages as soon as an error is detected.

This means that the instruction causing the problem can easily be identified. Against interpretation, is the fact that **execution of a program is slow** compared to that of a compiled program. This is because; the original program has to be translated every time it is executed. Also, **instructions inside a loop will have to be translated each time the loop is entered.**

3.4.3 Translation software

However, interpretation is very useful during program development as errors can be found and corrected as soon as they are discovered. In fact, many languages, such as Visual Basic, use both an interpreter and a compiler. This enables the programmer to use the interpreter during program development. When the program is fully working, it can be translated by the compiler into machine code. This machine code version can then be distributed to users who do not have access to the original code.

When a compiler or an interpreter is used, the translation from a high-level language to machine code has to go through various stages which are show below.



3.4.3 Translation software

Comparison	
COMPILER	**INTERPRETER**
Fast, creates executable file that runs directly on the CPU	Slower, interprets code one line at a time
Debugging is more difficult. One error can produce many false errors	Debugging is easier. Each line of code is analysed and checked before being executed
More likely to crash the computer. The machine code is running directly on the CPU	Less likely to crash as the instructions are being carried out either on the interpreters' command line or within a virtual machine environment which is protecting the computer from being directly accessed by the code.
Easier to protect Intellectual Property as the machine code is difficult to understand	Weaker Intellectual property as the source code has to be available at run time.
Uses more memory - all the execution code needs to be loaded into memory.	Uses less memory, source code only has to be present one line at a time in memory
Unauthorised modification to the code more difficult. The executable is in the form of machine code. So it is difficult to understand program flow.	Easier to modify as the instructions are at a high level and so the program flow is easier to understand and modify.

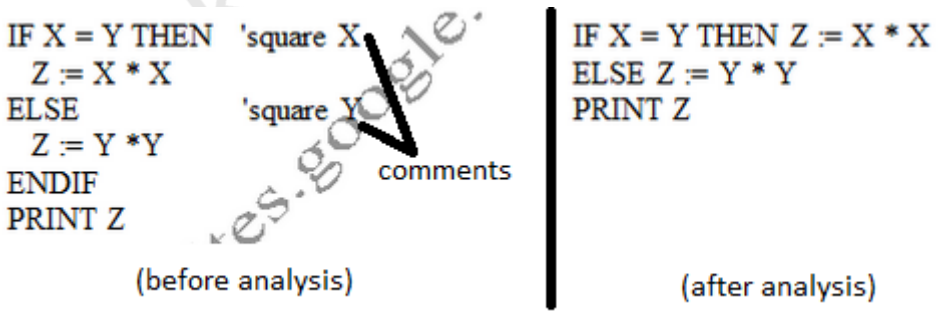
3.4.3 Translation software

Lexical Analysis

The lexical analyzer uses the source program as input and creates a stream of tokens for the syntax analyzer. Tokens are normally 16-bit unsigned integers. Each group of characters is replaced by a token. Single characters, which have a meaning in their own right, are replaced by ASCII values. Multiple character tokens are represented by integers greater than 255 because the ones up to 255 are reserved for the ASCII codes. Variable names will need to have extra information stored about them; this is done by means of a symbol table. This table is used throughout compilation to build up information about names used in the program. During lexical analysis, only the variable's name will be noted. It is during syntax and semantic analysis that details such as the variable's type and scope are added. The lexical analyzer may output some error messages and diagnostics. For example, it will report errors such as an identifier or variable name which breaks the rules of the language.

At various stages during compilation, it will be necessary to look up details about the names in the symbol table. This must be done efficiently! (so a linear search will not be ideal for this task). In fact, it is usual to use a hash table and to calculate the position of a name by hashing the name itself. When two names are hashed to the same address, a linked list can be used to avoid the symbol table filling up.

The lexical analyzer also removes redundant characters such as white spaces (tabs, spaces, etc… which we may find useful to make code more readable, but the computer does not want) and comments. Often the lexical analysis takes longer than the other stages of compilation. This is because it has to handle the original source code, which can have many formats. For example, the following two pieces of code are equivalent although their format is considerably different!



When the lexical analyzer has completed its task, the code will be in a standard format. This means that the syntax analyzer (which is the next stage) can always expect the format of its input to be the same.

3.4.3 Translation software

Syntax Analysis

During this stage of compilation the code generated by the lexical analyzer is parsed (broken into small units) to check if it is grammatically correct. All languages have rules of grammar and computer languages are no exception. The grammar of programming languages is defined by means of BNF notation. It is against these rules that the code has to be checked.

For example, taking a very elementary language, an assignment statement may be defined to be of the form:

<variable> <assignment_operator> <expression>

And expression is:

<variable> <arithmetic_operator> <variable>

And the parser must take the output from the lexical analyzer and check that it is of this form.

If the statement is:

Sum := sum + number

Then the parser will receive:

<variable> <assignment_operator> <variable> <arithmetic_operator> <variable>

This then becomes:

<variable> <assignment_operator> <expression>

And then:

<Assignment statement>

(which is valid!)

If the original statement was:

Sum := sum ++ number

This will be input as:

<variable> <assignment_operator> <variable> <arithmetic_operator> <arithmetic_operator> <variable>

3.4.3 Translation software

And this does not represent a valid statement, hence an error message will be returned.

It is at this stage that invalid names can be found such as **“PRIT”** instead of **“PRINT”** as “PRIT” will be read as a variable name instead of a reserved word. This will mean that the statement containing “PRIT” will not parse to be a valid statement. Note that in languages that require variables to be declared before being used, the lexical analyzer may pick up this error because **“PRIT”** has not been declared as a variable and so it won’t be in the symbol table.

Most compilers will report errors found during syntax analysis as soon as they are found and will attempt to show where the error has occurred. However, **they may not be very accurate** in their conclusions **nor may the error message be very clear.**

During syntax analysis, certain semantic checks are carried out. These include label checks, flow of control checks and declaration checks.

Some languages allow **GOTO** statements (not recommended by the authors) which allow control to be passed, unconditionally, to another statement which has a label. The **GOTO** statement specifies the label to which the control must pass. The compiler must check that such a label exists.

Certain control constructs can only be places in certain parts of a program. For example in C (and in C++) the **CONTINUE** statement can only be placed inside a loop and the **BREAK** statement can only be placed inside a loop or a SWITCH statement. The compiler must ensure that statements like these are used in the correct place.

Many languages insist on the programmer declaring variables and their types. It is at this stage that the compiler verifies that all variables have been properly declared and that they are used correctly.

3.4.3 Translation software

Code Generation:

It is at this stage, when all the errors due to incorrect use of the language have been removed, that the program is translated into code suitable for use by the computer's processor.

During lexical and syntax analysis a table of variables has been built up which includes details of the variable name, its type and the block in which it is valid. The address of the variable is now calculated and stored in the symbol table. This is done as soon as the variable is encountered during code generation.

Before the final code can be generated, an intermediate code is produced. This intermediate code can then be interpreted or translated into machine code. In the latter case, the code can be saved and distributed to computer systems as an executable program. Two methods can be used to represent the high-level language in machine code. This syllabus does not require knowledge of either of these methods, but a brief outline is given for those who may be interested.

One uses a tree structure and the other a three address code (TAC). TAC allows no more than three operands. Instructions take the form:

$Operand_1$:= $Operand_2$ **Operator** $Operand_3$

For example, using three address code, the assignment statement

A:= (B + C) * (D – E) / F

Can be evaluated in the following order, where “Ri” represents an intermediate result.

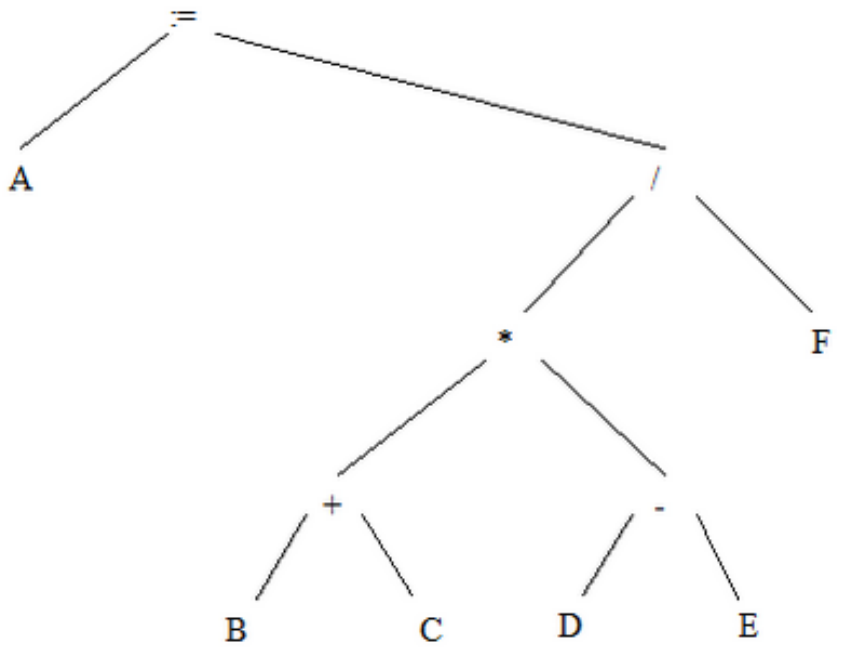
R_1 := B + C

R_2 := D – E

R_3 := R1 * R2

A := R3 / F

3.4.3 Translation software



This can also be represented in a tree format shown below.

Other statements can be represented in similar ways and the final stage of compilation can then take place.

The compiler has to consider, at this point the type of code that is required. Code can be improved in order to increase the speed of execution or in order to make the size of the program as small as possible. Often compilers try to compromise between the two. This process of improving the code is known as “Code Optimization”.

An example of code optimization is shown in the table below. Where the code on the left has been changed to that on the right so that “r1 – b” is only evaluated once. And at a later stage, the whole expression of “r4” can be removed. r5’s definition will be changed and would refer to r2 directly.

```
a := 5 + 3
b := 5 * 3
r1 := a + b
r2 := r1 - b
r3 := r2 / a
r4 := r1 - b
r5 := r4 + 6
c := r3 – r5
```

```
a := 5 + 3
b := 5 * 3
r1 := a + b
r2 := r1 - b
r3 := r2 / a
r4 := r2
r5 := r4 + 6
c := r3 – r5
```

```
a := 5 + 3
b := 5 * 3
r1 := a + b
r2 := r1 - b
r3 := r2 / a
r5 := r2 + 6
c := r3 – r5
```

3.4.3 Translation software

There are several other ways of optimizing code, but hopefully the above demonstration has made the concept clear. Candidates will not be expected to optimize code, but should be aware of what it is.

Backus-Naur Form and Syntax Diagrams

Since all programming languages have to be translated to machine code by means of a computer, they must be clearly defined. Each statement must be of a prescribed form.

An example of the start of a **FOR** loop in Visual Basic is: **For count = 1 To 10**

Whereas C++ expects: **for (count = 1, count <= 10, count++)**

A visual basic compiler would not understand the C++ syntax and vice versa. We therefore need, for each language, a set of rules that specify precisely every part of the language. These rules are specified using the Backus Naur Form (BNF) or syntax diagrams.

All languages use integers, so we shall start with the definition of an integer. **An integer is a sequence of digits 0,1,2,3,….,9.** Now the number of digits in an integer is arbitrary. That is, it can be any number. A particular compiler will restrict the number of digits only because of the storage space set aside for an integer has been exceeded. But a computer language does not restrict the number of digits.

Thus the following are valid integers:

- 0
- 2
- 415
- 3040513002976
- 0000000123

Thus, an integer can be a single digit. We can write this as : <integer>:= <digit>

The symbol (::=) is read as 'is defined to be'

But we now must define a digit. A digit is 0 or 1 or 2 or…….. Or 9 and we write this as:

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

The vertical line is read as OR. Notice that all the digits have to be specified and that they are not inside angle brackets. (< and >) like **<integer>** and **<digit>** This is because integer and digit have definitions elsewhere.

Our full definition of a single digit integer is:

3.4.3 Translation software

<integer> ::= <digit>

<digit> ::= 0|1|2|3|4|5|6|7|8|9

This is called Backus Naur Form (BNF).

But how are we going to specify integers of any length?

Consider the integer **"147"**

This is a single digit integer (1) followed by the integer **47**. But **47** is a single digit integer (4) followed by a single digit integer (7). Thus, all integers of more than one digit start with a single digit and are followed by an integer. Eventually the final integer is a single digit integer.

Thus, an indefinitely long integer is defined as: **<integer> ::= <digit><integer>**

This is a recursive definition as integer is defined in terms of itself. Applying this definition several times produces the sequence:

<Integer>::= <digit><integer>

=<digit><digit><integer>

=<digit><digit><digit><integer>

To stop this ongoing process, we use the fact that eventually, <integer> is a single digit.

So we can write: **<Integer> ::= <digit>|<digit><integer>**

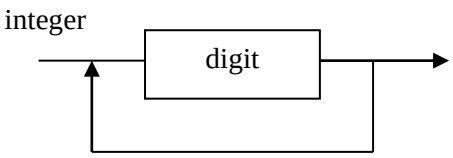
3.4.3 Translation software

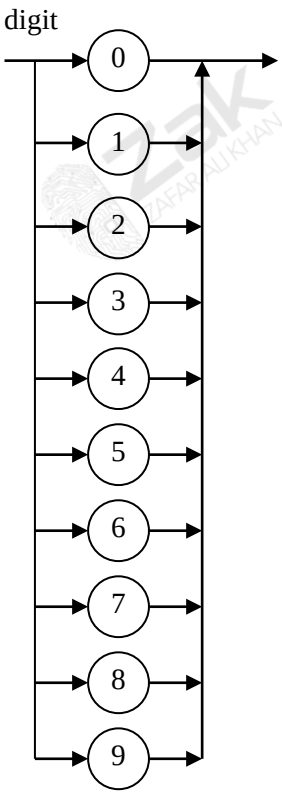
That is, **<integer>** is a **<digit>** or a **<digit>** followed by an **<integer>**. This means that at any time **<integer>** can be replaced by **<digit>** and the recursion stops. Strictly speaking we have defined an unsigned integer as we have not allowed a leading plus sign **(+)** or minus sign **(-)**.

This will be dealt with later but we now have the full definition of an unsigned integer which, in BNF, is:

<unsigned integer> ::= <digit>|<digit><unsigned integer>
<digit> ::= 0|1|2|3|4|5|6|7|8|9

The definition of an unsigned integer can also be described by means of syntax diagrams.





3.4.3 Translation software

Now we shall define a signed integer such as:

- +27
- -3415

This is simply an unsigned integer preceded by a "+" or a "-" sign.

Thus **<unsigned integer> ::= +<unsigned integer> | -<unsigned integer>**

And we can use the earlier definition of <unsigned integer>. It is usual to say that an integer is **signed** or **unsigned**. If we do this, we get the following definition, in BNF , of an integer.

<integer> ::= <unsigned integer>|<signed integer>

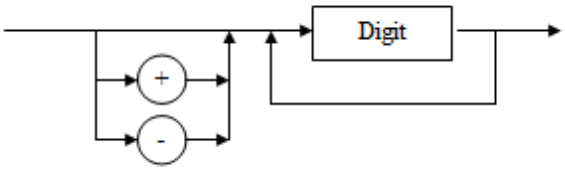
<signed integer> ::= + <unsigned integer>| - <unsigned integer>

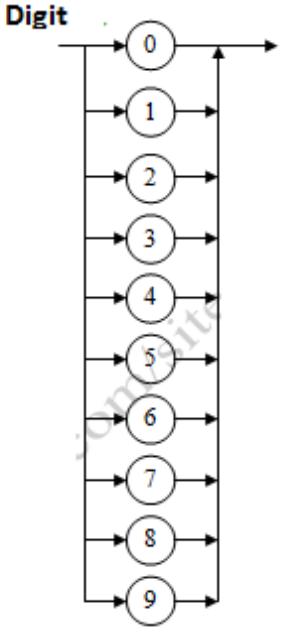
<unsigned integer> ::= <digit>|<digit><unsigned integer>

<digit> ::= 0|1|2|3|4|5|6|7|8|9

There are other ways of writing these definitions. However, it is better to use several definitions than try to put all the possibilities into a single definition. In other words, try to start at the top with a general definition and then try to break the definitions down into simpler and simpler ones. That is, we have used top-down design when creating these definitions. We have broken the definitions down until we have terms whose values can be easily determined.

Here is a corresponding syntax diagram.





3.4.3 Translation software

Care must be taken when positioning the recursion in the definitions using BNF. Suppose we define a variable as a sequence of one or more characters starting with a letter. The characters can be any letter, digit or the underscore sign.

Valid examples are:

- A
- x
- sum
- total24
- mass_of_product
- MyAge

Let us see what happens if we use a similar definition to that for an unsigned integer.

<variable> ::= <letter>|<character><variable>

<character> ::= <letter>|<digit>|<under-score>

So the phrase **"2Sum"** is valid. As we use: **<character><variable>**

With <character> = 2 and <variable> = Sum. Continuing in this way, we use **2**, **S** and **u** for <character> and then **m** for **<letter>**. This means that our definition simply means that we must end with a letter and not start with one. We must rewrite our definition in such a way as to ensure that the first character is a letter. Moving the recursive call to the front of **<character>** can do this for us. This means that the last time it is called, it will be a letter and this will be at the head of the variable.

The correct definition is:

<variable> ::= <letter>|<variable><character>

<character> ::= <letter>|<digit>|<under-score>

<letter> ::= <uppercase>|<lowercase>

<uppercase>::=A|B|C|D|E|F|G|H|I|J|K|ZL|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

<lowercase> ::=a|b|c|d|e|f|g|h|i|j|k|zl|m|n|o|p|q|r|s|t|u|v|w|x|y|z

<digit> ::= 0|1|2|3|4|5|6|7|8|9

<under-score> ::= _

3.4.3 Translation software

A syntax diagram can also do this. (which is left as an exercise). You should also note that, in the definition of a integer, we used tail recursion, but here we have used head recursion.

Let us now use our definition of integer to define a real numbers such as:

- 0.347
- -2.862
- +14.34
- 00235.006

The result is very simple, it is: **<real number>::= <integer> . [dot] <unsigned integer>**

Finally, suppose we don't want to allow leading zeros in our integers.

That is:

00135 is not allowed

0 is allowed

This means that an integer can be a:

- Zero digit
- Non-zero digit
- Non-zero digit followed by any digit.

This means that an integer is: **Zero** or **Digits**

Where digits must start with a non-zero digit. In BNF, this is **<integer> ::= <zero>|<digits>**

<digits> must be a single non-zero digit or a non-zero digit followed by any digits.

This gives us: **<digits> ::= <non-zero digit>|<digits><digit>**

Where:

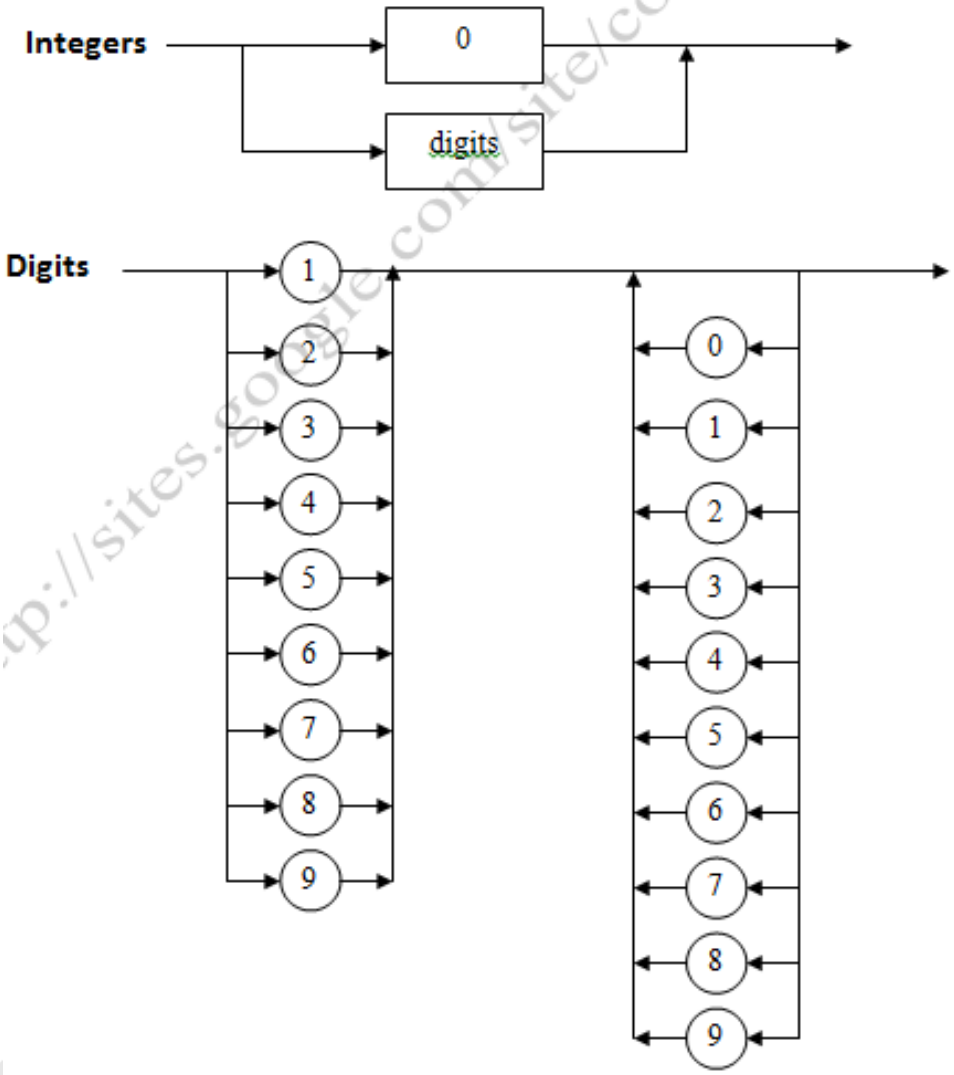
<zero> ::= 0

<non-zero integer> ::= 1|2|3|4|5|6|7|8|9

3.4.3 Translation software

<digit> ::= <zero>|<non-zero digit>

Here is the corresponding syntax diagram



3.4.3 Translation software

Reverse Polish notation

Reverse Polish notation (RPN) is a mathematical notation wherein every operator follows all of its operands. It is also known as the **Prefix notation** and is parenthesis-free. **In Reverse Polish notation, the operators follow their operands.** For instance, to add 3 and 4, one would write **"3+4"** rather than **"3 4 +".** If there are multiple operations, and the operator is given immediately after its second operand; so the expression written "3 – 4 + 5" in conventional infix notation would be written as "3 4 – 5 +" in RPN.

First subtract 4 from 3, and then add 5 to that. An advantage of RPN is that it obviates the need for parentheses that are needed for infix or general notations.

Interpreters of Reverse Polish notation are often stack-based; that is, operands are pushed onto a stack, and when an operation is performed, its operands are popped from a stack and its result pushed back in. So the value of postfix expression is on the top of the stack. Stacks, and therefore RPN, have an advantage of being easy to implement with quick access time.

Practical implications:

- Reverse Polish expression can be processed directly from left to right
- RPN is free from ambiguities.
- Does not require brackets: the user simply performs calculations in the order that is required, letting the automatic stack store intermediate results on the fly for later use.
- There is no requirement for the precedent rules required in infix notation.
- Calculations occur as soon as an operator is specified. Thus, expressions are not entered wholesale from left to right but calculated one piece at a time.
- The automatic stack permits the automatic storage of intermediate results for later use. **This key feature is what permits RPN calculators to easily evaluate expression of arbitrary complexity also; they do not have limits on the complexity of expressions they can evaluate.**
- In RPN calculators, no equals key is required to force computation to occur.
- Users must know the size of the stack, because practical implementations of RPN use different sizes for the stack.

- **When writing RPN on paper, adjacent numbers need a separator between them. Using a space requires clear handwriting to prevent confusion. For example, "12 34+" could look like "123 4+", while something like 12,34 is straightforward.**

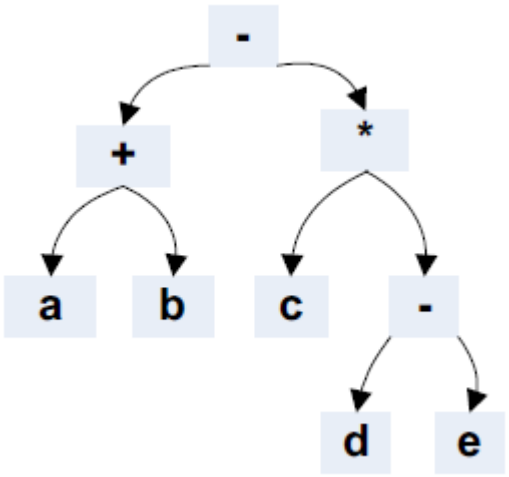
Reverse Polish Notation using Binary Trees:

RPN can be easily made using binary trees. Using BODMAS, an infix (general) notation can easily be turned into a binary tree; representing the original infix notation. This tree can be read in post traversal (left-right-root) to make an equivalent RPN.

In simple words, a binary tree can be used to convert an infix notation to RPN.

See how the following infix expression can be represented as a binary tree below.

(a+b) – c * (d-e)



If we read the above tree in post order traversal (left-right- root) then we achieve the following RPN as a result:

a b + c d e - * -

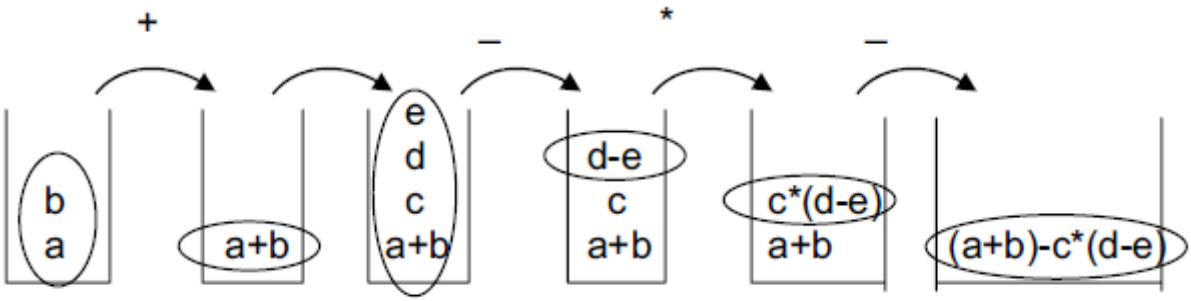
3.4.3 Translation software

Converting RPN back to Infix notation using stacks:

Reading an RPN left to right, pushing every operand to the stack, as soon as an operator is read, you pop two values from the stack top and apply that operator to them, and push the final expression back into the stack. This is the way to convert RPN back to Infix notation.

Example: See with the help of this example, how a stack is used to turn a RPN into Infix notation.

ab+cde-*-



And there you have it!

The expression is now in Infix (standard) form.