BACHELOR OF COMPUTER APPLICATIONS

SEMESTER 4

DCA2203

SYSTEM SOFTWARE

Unit 2

Language Processor

Table of Contents

1. INTRODUCTION

This chapter introduces the concepts underlying the design and implementation of language processors. Mainly, computer programming languages are classified into three categories, they are machine language, assembly language, and high-level language. Machine language is a machine-readable language statements that is a pattern of bits (binary representation) and which can be directly executable by the CPU (Central Processing Unit). Assembly language uses mnemonic codes to represent the low-level machine operations. These instructions should be converted into machine instructions for further processing. For this, one of the language processors called language translator is used. Assembler is the language translator which converts assembly language to machine language.

High-level language (HLL) uses keywords similar to English and is easier to write. Compilers and interpreters are the language translators which convert high-level language into machine language.

In this chapter, we will study different language processing activities in detail. Also, this chapter describes the fundamentals of language processing and language specification. This chapter also explains different language processor development tools like Lex, Yacc, and others.

1.1 Learning Objectives

After studying this unit, learners should be able to:

- *Define Language Processor*
- *Describe different Language Processing Activities*
- *Explain Language specification*
- *Explain different Language Processor Development tools*

2. LANGUAGE PROCESSING ACTIVITIES

The need for language processing arises from the distinction between how software behaves and how it is used in a computer system.The developer expresses the ideas related to the application domain of the software. To implement these ideas, their description should be interpreted as related to the execution domain of the computer system. Semantics represents the rules of the meaning of a domain; the Semantic gap represents the difference between the semantics of two domains. Software implementation using a PL (Programming Language) introduces a new domain, the PL domain. The semantic gap between the application domain and execution domain is bridged by the software engineering steps. The first step bridges the gap between the PL and execution domains, while the second step bridges the gap between the PL and execution domains. We refer to the gap between the application and PL domains as the specification and design gap or simply the *specification gap,* and the gap between the PL and execution domains as the *execution gap.* The specification gap is bridged by the software development team, while the execution gap is bridged by the designer of the programming language processor, using a translator or an interpreter.

Semantic gaps have many consequences like large development time, large development efforts, and poor quality of software. These issues are tackled using programming languages (PL) as problem-oriented languages (are used with specific applications to bridge the gap between the application domain and execution domain) and procedural-oriented languages (used with most of the applications to provide general purpose facilities to bridge the gap between application domain and execution domain).

Steps for using a PL can be grouped into two:

1) Specification, design, and coding steps
2) PL implementation Steps

A ***language processor*** is Software that bridges a specification or execution gap. The activity performed by the language processor is called Language processing. The program that inputs to the language processor are called the source program and its output is the target program. The languages in which these programs are written are called source language

and destination language respectively. A language processor abandons the generation of the target program if it detects errors in the source program.

Some examples of language processors are,

1) A *language translator* bridges the execution gap to the machine language of a computer system. (E.g: Assembler, compiler, interpreter, etc.)
2) The *detranslator* bridges the same execution gap as a language translator but in the reverse direction.
3) *Preprocessor* is another language processor which bridges the execution gap.
4) The *language* migrator bridges the specification gap between two programming languages.

Language processing activities are those that bridge the specification gap and those that bridge the execution gap. These can be divided into two.

1) Program generation activities.
2) Program execution activities.

Program generation activity aims at the automatic generation of a program.

A program execution activity organizes the execution of a program written in a programming language in a computer system.

Program Generation:

A program generator is a software system that accepts the specifications of a program to be generated and executes the program in target programming languages. The program generator is the middleware between the program generator and the application. The specification gap is the gap between the application domain and the program generating domain. Figure 2.1 shows the program generation activities.

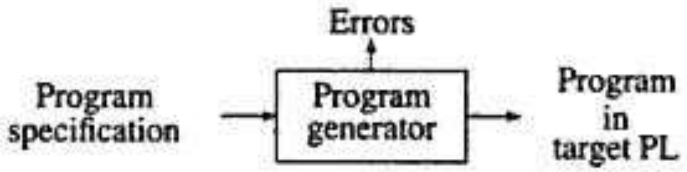


Fig 2.1: Program generation

Reduction in specification gap will increase the reliability of the program. The generator is Bridging the gap to the PL (Programming Language) domain. By testing the correctness of the specification input, we can test an application generated by using a generator. The compiler or interpreter for the PL bridges the execution gap between the target PL domain and the execution domain.

Program Execution:

Translation and interpretation are the two popular models for program execution.

Program Translation:-

This model bridges the execution gap by translating a program written in a PL, which is treated as the source program (SP), into an equivalent program in the machine or assembly language of the computer system which treated as the target program(TP), (see fig 2.2).

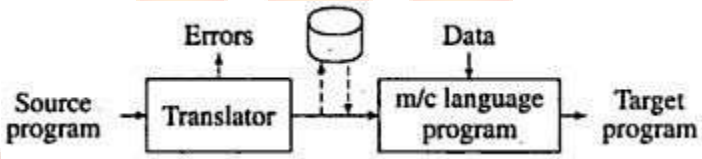


Fig 2.2: Program translation Model

Characteristics of the program translation model are,

1. A program must be translated before it can be executed.
2. The translated program may be saved in a file. The saved program may be executed repeatedly.
3. A program must be retranslated following modifications.

Program interpretation:

The interpreter reads the source program and stores it in memory. During interpretation, it reads a statement, determines its meaning, and performs actions that implement it. This includes computational and input-output actions. Figure 2.3(a) shows the schematics of program interpretation. Fig 2.3(b) shows the schematic of the execution of a machine language program by the CPU of a computer system. CPU uses *Instruction Pointer (IP)* to note the address of the next instruction to be executed. The instruction execution cycle consists of the following steps:

1. Fetch the instruction.
2. Decode the instruction to find out the operation to be performed, and also its operands.

3. Execute the instruction.

At the end of one cycle, the program counter will be updated by the next instruction and the cycle will be repeated.

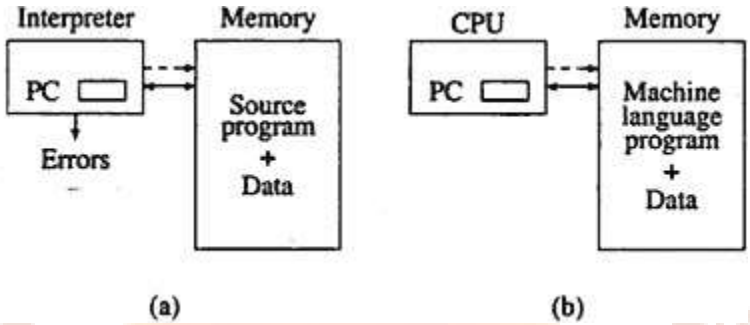


Fig 2.3: Schematics of (a) interpretation, (b) Program execution

The program interpretation cycle consists of the following steps:

1. Fetch the statement.
2. Analyze the statement and find out its meaning, the computation to be performed, and its operands.
3. Execute the meaning of the statement.

In this case, the source program is retained in the source form itself, i.e. no target program form exists, and a statement is analyzed during its interpretation.

Comparison:

In the use of the program translation model, a fixed cost is incurred. If the source program is modified, the translation cost must be incurred again irrespective of the size of the modification. Execution of the target program is efficient since the target program is in the machine language. The use of the interpretation model does not incur translation overheads. This is advantageous if the program is modified between executions, as in program testing and debugging. Interpretation is slower than the execution of a machine language program. Figure 2.4 shows a flow diagram of the Activities of a language processor.

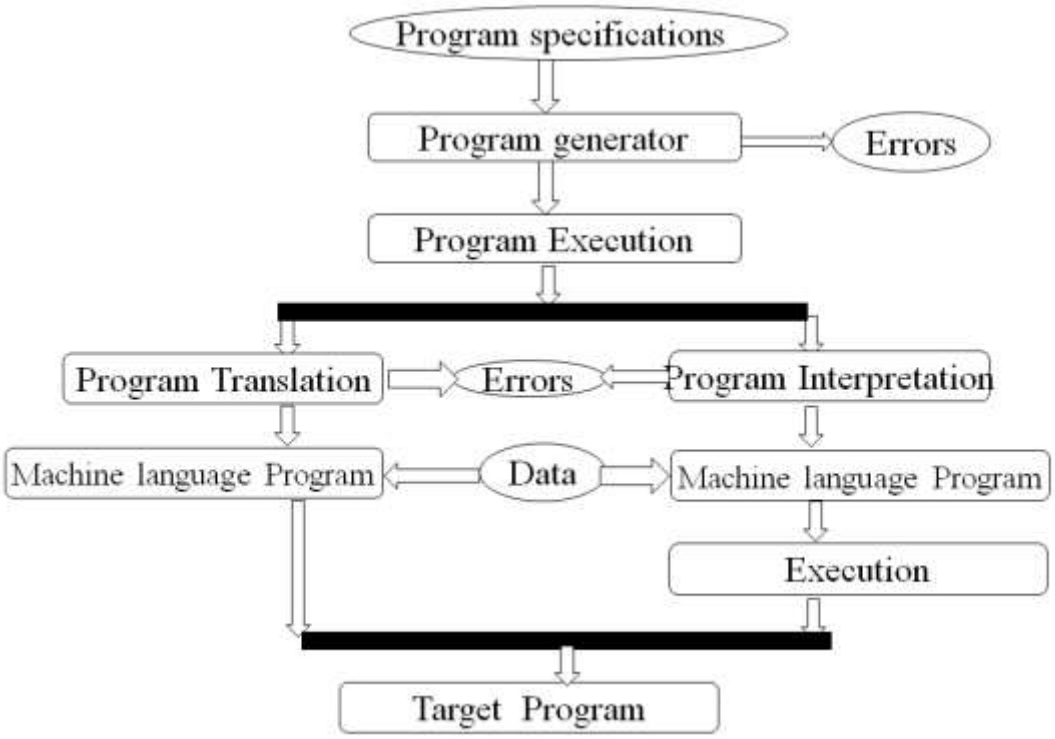


Fig 2.4: Activities of a language processor

SELF ASSESSMENT QUESTIONS – 1

1. The software which bridges a specification or execution gap is called _____.
2. A_____bridges the execution gap to the machine language of a computer system.
3. CPU uses_____to note the address of the next instruction to be executed.
4. _____ and _____ are the two popular models for program execution.

3. FUNDAMENTALS OF LANGUAGE PROCESSING

Analysis of Source Program (SP) + Synthesis of Target Program (TP) is called language processing.

A specification of the source language forms the basis of source program analysis. The specification consists of three components:

1) Lexical rules govern the formation of valid tokens in the source language.
2) Syntax rules govern the formation of valid statements in the source language.
3) Semantic rules associate meaning with valid statements of the language.

To determine relevant information concerning a statement in the source program, the analysis phase uses each component of the source language specification. Thus, analysis of the source statement consists of lexical, syntax, and semantics analysis. The synthesis phase is concerned with the construction of a target language statement that has the same meaning as a source statement. This consists of 2 main activities:

- Creation of data structures in the target program
- Generation of target code

These activities are called memory allocation and code generation respectively.

Phases and passes of a language processor:

A language processor consists of two distinct phases. They are the analysis phase and synthesis phase. Figure 2.5 shows the schematics of the language processor.

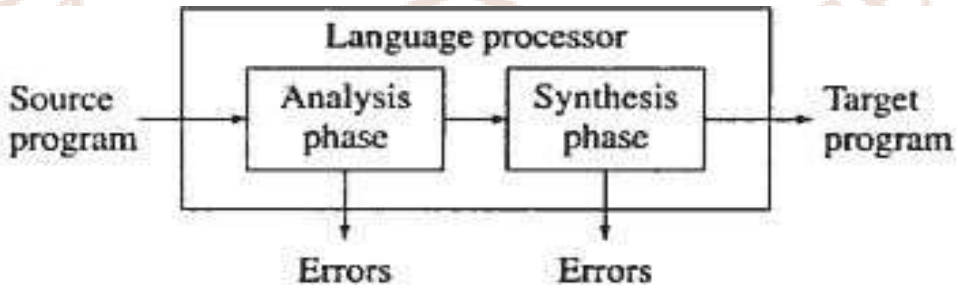


Fig 2.5: Phases of a language processor

This figure shows the impression that: language processing can be performed on a statement by statement basis. That is, analysis of the source statement can be immediately followed by synthesis of equivalent target statements. This may not be feasible due to

- Forward references.
- Issues concerning memory requirements and organization of a language processor.

Forward Reference:

A forward reference of a program entity is a reference to the entity which precedes its definition in the program.

A language processor does not possess all relevant information concerning the referenced entity while processing a statement containing a forward reference. This creates problems during the synthesis phase. This problem can be solved by postponing the generation of the target code until more information concerning the entity becomes available. This also reduces the memory requirements of the language processor and simplifies its organization.

Consider the example:

```
If    (12>10)    then
goto A;
…..
……
A: Display ’12 is greater’
```

Here, the statement ‘goto A’ constitutes a forward reference because the declaration of label ‘A’ occurs later in the program.

Language Processor Pass:

A language processor pass is the processing of every statement in a source program, or its equivalent representation, to perform a language processing function or a set of language processing functions.

Intermediate Representation (IR) of programs:

An intermediate representation (IR) is a representation of a source program that reflects the effect of some, but not all, analysis and synthesis tasks performed during language processing. Figure 2.6 depicts the schematic of a two-pass language processor.

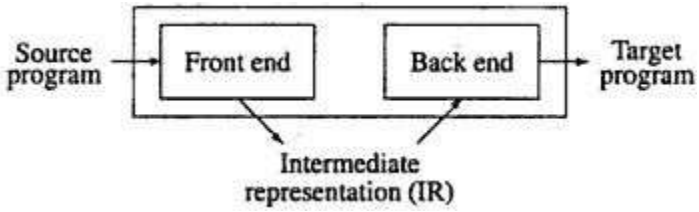


Fig 2.6: Two-pass schematic for language processing

The first pass performs an analysis of the source program and reflects its results in the intermediate representation. To perform synthesis of the target program; the second pass reads and analyses the IR, instead of the source program. This avoids repeated processing of the source program.

The first pass is concerned exclusively with source language issues. Hence it is called the front end of the language processor. The second pass is concerned with program synthesis for a specific target language. Hence it is called the back end of the language processor.

Desirable properties of an IR are:

- Ease of use: IR should be easy to construct and analyze.
- Processing Efficiency: efficient algorithms must exist for constructing and analyzing the IR.
- Memory efficiency: IR must be compact.

The front end of a language processor analyses the source program and constructs an IR. All actions performed by the front end, except lexical and syntax analysis, are called semantic actions. They are the following:

1) Checking semantic validity of constructs in SP
2) Determining the meaning of SP
3) Constructing an IR.

A Toy Compiler:

In the following section, we can see the front end and back end of a toy compiler.

The Front End:

The front end performs lexical, syntax, and semantic analysis of the source program. Each kind of analysis involves the following functions:

1. Determine the validity of a source statement from the viewpoint of analysis.
2. Determine the 'content' of a source statement.
3. Construct a suitable representation of the source statement for use by subsequent analysis functions, or by the synthesis phase of the language processor.

Each analysis represents the 'content' of a source statement in the form of (1) tables of information and (2) description of a source statement. The tables and descriptions at the end of semantic analysis form the IR of the front end.

Output of the front end:

The Intermediate Representation (IR) produced by the front end consists of two components.

1) Tables of information.
2) An intermediate code(IC) is a description of the source program.

Tables:

Tables contain the information obtained during different analyses of SP (Source Program). The most important table which contains information concerning all identifiers used in the SP is the Symbol table. The symbol table is built during lexical analysis. The semantic analysis adds information concerning symbols while processing declaration statements. It may also add new names designating temporary results.

Intermediate Code(IC):

The Intermediate Code(IC) is a sequence of IC units. These IC units represent the meaning of one action in SP. IC units may contain references to the information in various tables.

For example, the following figure 2.7 shows the IR produced by the analysis phase for the program.

```
i: integer; a, b:
real;
a: = b+;
```

In the following example, the symbol table contains information concerning the identifiers and their types. This information is determined during lexical and semantic analysis, respectively. In IC, the specification (Id, #1) refers to the id occupying the first entry in the table. Note that i* and temp are temporary names added during the semantic analysis of the assignment statement.

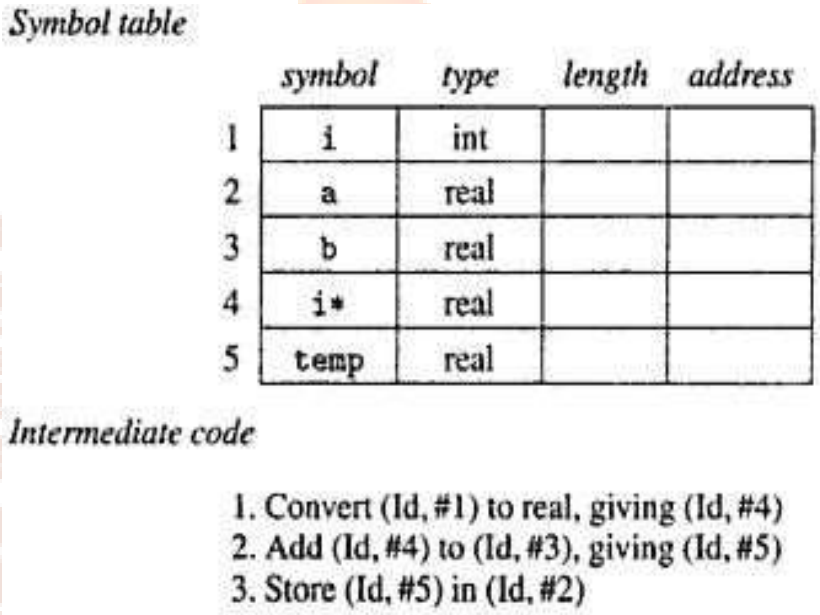


Fig 2.7: IR example

Lexical Analysis (Scanning):

Lexical analysis identifies the lexical units in a source statement. The units are then entered into various tables after being divided into various lexical classes, such as ids, constants, reserved ids, etc. This classification may be based on the nature of a string or on the specification of the source language. Lexical analysis builds a descriptor called a token, for each lexical unit. A token contains two fields; they are class code and number in class. Class code identifies the class to which a lexical unit belongs. Number in class is the entry number of the lexical unit in the relevant table.

Syntax Analysis (Parsing):

Syntax analysis processes the string of tokens which are built by lexical analysis to determine the statement class, e.g. Assignment statement, if statement, etc. it then builds an IC which

represents the structure of the statement. The IC is passed to semantic analysis to determine the meaning of the statement.

Semantic Analysis:

Semantic analysis of declaration statements differs from the semantic analysis of imperative statements. Semantic analysis of declaration statements results in the addition of information in the symbol table (e.g. type, length, and dimensionality of variables). Semantic analysis of imperative statements identifies the sequence of actions necessary to implement the meaning of a source statement. In both cases, the structure of a source statement guides the application of the semantic rules. When semantic analysis determines the meaning of a subtree in the IC (Intermediate Code), it adds information to a table or adds an action to the sequence of actions. It then modifies the IC to enable further semantic analysis. The analysis ends when the tree has been completely processed. The updated tables and sequence of actions constitute the IR produced by the analysis phase.

The Figure 2.8 front end of the compiler shows the schematic of the front end where arrows indicate the flow of data.

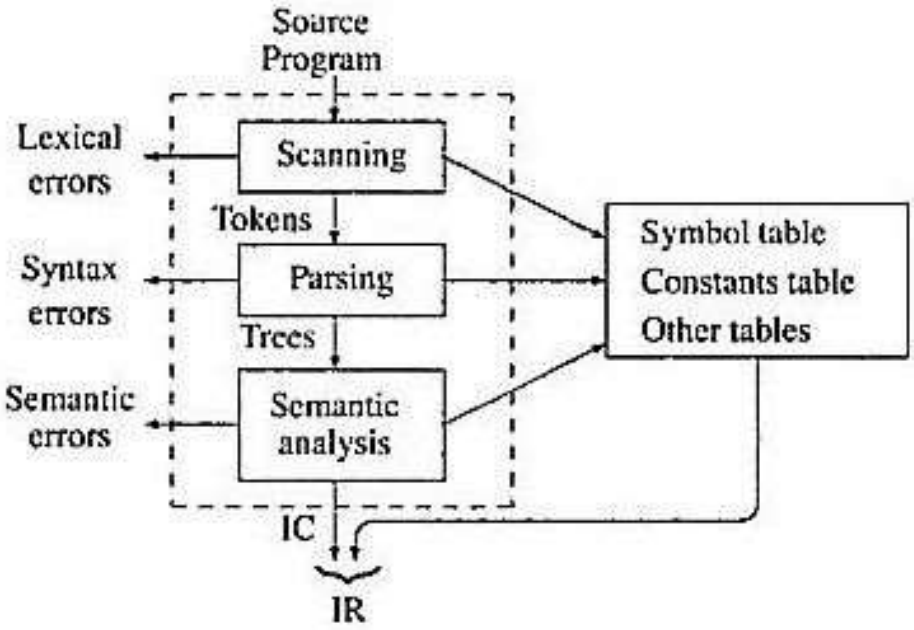


Fig 2.8: Front end of the toy compiler

The Back End:

The back end performs memory allocation and code generation.

Memory allocation:

Memory allocation is a simple task given the presence of the symbol table. The memory requirement of an identifier is computed from its type, length, and dimensionality, and memory is allocated to it. The address of the memory area is entered in the symbol table.

Consider the statement a: =b+i; after memory allocation, the symbol table looks as shown in figure 2.9.

	symbol	*type*	*length*	*address*
1	i	int		2000
2	a	real		2001
3	b	real		2002

Fig 2.9: Symbol table after memory allocation

Code generation:

Code generation uses knowledge of the target architecture, knowledge of instructions, and addressing modes in the target computer, to select appropriate instructions. The important issues in code generation are

1) Determine the places where the intermediate results should be kept, i.e. whether they should be kept in memory locations or held in machine registers. This is a preparatory step for code generation.
2) Determine which instructions should be used for type conversion operations.
3) Determine which addressing modes should be used for accessing variables.

Figure 2.10 back end of the toy compiler shows a schematic of the back end.

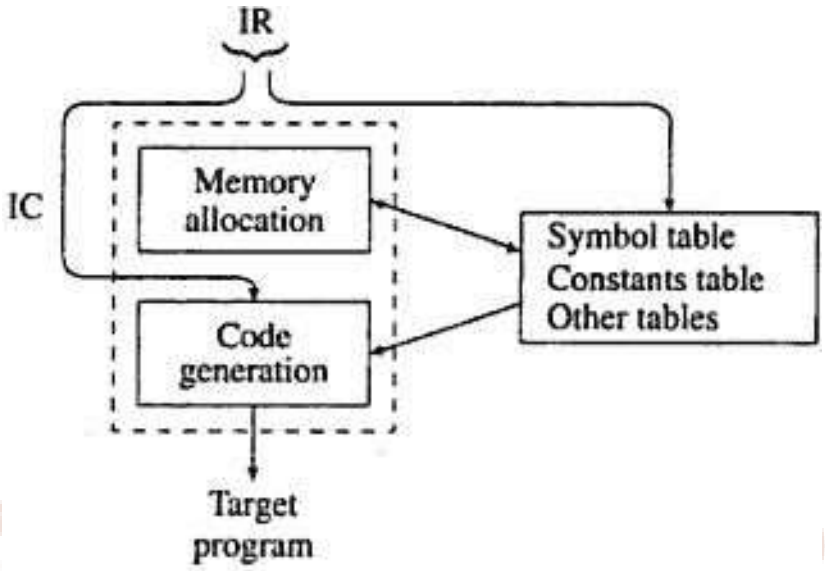


Fig 2.10: Back end of the toy compiler

SELF ASSESSMENT QUESTIONS – 2

5. Analysis of Source Program (SP) + Synthesis of Target Program (TP) is called_____.
6. A_____of a program entity is a reference to the entity which precedes its definition in the program.
7. The most important table which contains information concerning all identifiers used in the Source Program is _____
8. The_____performs memory allocation and code generation

4. FUNDAMENTALS OF LANGUAGE SPECIFICATION

In this section, we will discuss important lexical, syntax, and semantic features of a programming language.

Programming Language Grammars:

The lexical and syntactic features of a programming language are specified by its grammar. A Language L canbe a collection of valid sentences. Each Sentence can be looked upon as a sequence of words and each word as a sequence of letters or graphic symbols acceptable in L. A language specified in this manner is known as a *Formal Language.* A Formal Language grammar is a set of rules which precisely specify the sentences of L. Thenatural languages are not formal languages due to their rich vocabulary. However, PLs are formal languages.

Terminal Symbols, Alphabet, and Strings:

The *alphabet* of L, denoted by the Greek symbol ∑, is the collection of symbols in its character set. We will use lower case letters a, b, c, etc. to denote symbols ∑. A symbol in the alphabet is known as a terminal symbol (T) of L. The alphabet can be represented using the mathematical notation of a set, e.g.

$$\sum \equiv \{a, b \ldots z, 0, 1 \ldots 9\}$$

Here, the symbols {, ',' and} are part of the notation. We call them meta symbols to differentiate them from terminal symbols. We assume that Meta symbols are distinct from terminal symbols. If this is not the case, i.e. if a terminal symbol and Meta symbol are identical, we enclose the terminal symbol in quotes to differentiate it from the meta-symbol. For example, the set of punctuation symbols of English can be defined as

{ :, ; , ',' , ... } , Where ',' denotes the terminal symbol 'comma'.

A string is a finite sequence of symbols. We will represent string by Greek symbols α, þ, ş, etc. Thus α = axy is a string over ∑. The length of the string is number of symbols init. The absence of any symbol is also a string, the null string ∑. The concatenation operation combines two strings into a single string. It is used to build larger strings from existing strings. For example, if α=ab, þ= axy, then the concatenation of α and þ, is represented as α. þ or simply αþ, gives the string abaxy. The null string also participated in the concatenation, thus a. ∑ = ∑.a = a.

Nonterminal Symbols:

A Nonterminal symbol (NT) is the name of a syntax category of a language, e.g. noun, verb, etc. An NT is written as a single capital letter, or as a name enclosed between < … >, e.g. A or <Noun>. During grammatical analysis, a nonterminal symbol represents an instance of the category. Thus, < Noun> represents a noun.

Productions:

A production, also called a rewriting rule, is a rule of grammar. Production has the form,

A nonterminal symbol ::= String of Ts and NTs.

and defines the fact that the NT on the LHS of the production can be rewritten as the string of (Terminal Symbols) Ts and NTs appearing on the RHS. When an NT can be written as one of many different strings, the symbol '|'(standing for 'or') is used to separate the strings on the RHS, e.g.

< Article > := a | an | the

The string on the RHS of a production can be a concatenation of component strings, e.g. the production

< Noun Phrase >:=<Article >< Noun >

expresses the fact that the noun phrase consists of an article followed by a noun.

Each grammar G defines a language LG. G contains an NT called the distinguished symbol or the start NT of G. unless otherwise specified, we use the symbol S as the distinguished symbol of G. A valid string α of LG is obtained by using the following procedure

1. Let α = 'S'.
2. While α is not a string of terminal symbols
   (a) Select an NT appearing in α, say X.
   (b) Replace X with a string appearing on the RHS of production of X.

Derivation, reduction, and parse trees:

A grammar G is used for two purposes, to generate valid strings of LG and to recognize valid strings of LG. The derivation operation helps to generate valid strings while the reduction

operation helps to recognize valid strings. A parse tree is used to depict the syntactic structure of a valid string as it emerges during a sequence of derivations or reductions.

Derivation:

Let production P1 of grammar G be of the form

P1:A ::= α

And let þ be a string such that þ≡şAơ, then the replacement of A by α in string þ constitutes a derivation according to production P1. We use the notation N‹ŋ to denote the direct derivation of ŋ from N and N‹*ŋ to denote the transitive derivation of ŋ from N, respectively. δ is a valid string according to G only if S‹*δ , where S is the distinguished Symbol of G.

Reduction:

Let Production P1 of grammar G be of the form

P1 : A ::= α

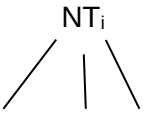
and let σ be a string such that σ = şαδ, then the replacement of α by A in string σ constitutes a reduction according to production P1. We use the notations ŋ‹N and ŋ‹*N to depict direct and transitive reduction, respectively. Thus, α‹ A only if A ::= α is a production of G and α ‹* A if α‹...‹ A. we define the validity of some string δ according to grammar G as follows: δ is a valid string of LG if δ‹* S, where S is a distinguished symbol of G.

Parse Trees:

A sequence of derivations or reductions reveals the syntactic structure of a string with respect to G. We represent the syntactic structure in the form of a parse tree. Derivation according to the production A::= α gives rise to the following elemental parse tree:

(Sequence of Ts and NTs constituting α)

A subsequent step in the derivation replaces an NT in α, say NTi, with a string. We can build another elemental parse tree to depict this derivation,



We can combine the two trees by replacing the node of NTi in the first tree with this tree. In essence, the parse tree has grown in the downward direction due to a derivation. We can obtain a parse tree from a sequence of reductions by performing the converse actions. Such a tree would grow in the upward direction.

Recursive Specification:

Consider a Grammar say, Grammar 2.1,

Grammar 2.1

<exp> ::= <exp>+<term> | <term>

<term> ::= <term>*<factor>|<factor>

<factor> ::= <factor><primary>| <primary>

<primary> ::= <id>|<constant>|(<exp>)

<id> ::= <letter>|<id>|<letter>|<digit>

<const> ::= [+|-]<digit>|<const><digit>

<letter> ::= a|b|c|...|z

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

This is a complete grammar for an arithmetic expression containing the operators ‡ (exponentiation), * and +. This grammar uses the notation known as the Backus NaurForm (BNF). Apart from the familiar elements ::=, | and <...>, a new element here is [. . .],which is used to enclose an optional specification. Thus, the rules for <id> and <const> in the above grammar are equivalent to the rules

<id> ::= <letter>|<id><letter>|<id><digit>

<const> ::= <digit>+<digit>|-<digit>|<const><digit>

This grammar uses recursive specification, whereby the NT being defined in a production itself occurs in a RHS string of the production, e.g. X ::= ...X....

The RHS alternative employing recursion is called a recursive rule. Recursive rules simplify the specification of recurring constructs.

Classification of Grammars:

Grammars are classified based on the nature of the productions used in them. Each grammar class has its own characteristics and limitations.

Type-0 Grammars:

These grammars are known as phrase structure grammars, contain productions of the form

$$\alpha ::= þ$$

where both α and þ can be strings of Ts and NTs. Such productions permit arbitrary substitution of strings during derivation or reduction; hence they are not relevant to the specification of programming languages.

Type - 1 Grammars:

These grammars are known as context-sensitive grammars because their productions specify that derivation or reduction of strings can take place only in specific contexts. A Type-1 production has the form

$$\alpha A þ ::= \alpha \pi þ$$

Thus, a string π in a sentential form can be replaced by 'A' only when it is enclosed by the strings α and þ. These grammars are also not particularly relevant for PL specification since recognition of PL constructs is not context-sensitive in nature.

Type - 2 Grammars:

These grammars impose no context requirements on derivations or reductions. A typical Type-2 production is of the form

$$A ::= \pi$$

which can be applied independently of its context. These grammars are therefore known as context-free grammars (CFG). CFGs are ideally suited for programming language specification.

Type - 3 Grammars:

Type-3 grammars are characterized by productions of the form

A::= tB | t or

A ::= Bt | t

Note that these productions also satisfy the requirements of Type-2 grammars. The specific form of the RHS alternatives- namely a single T or a string containing a single T and a single NT- gives some practical advantages in scanning. However, the nature of the productions restricts the expressive power of these grammars, e.g. nesting of constructs or matching of parentheses cannot be specified using such productions. Hence, the use of Type-3 productions is restricted to the specification of lexical units, e.g. identifiers, constants, labels, etc. The Productions for <constant> and <identifier> in the grammar (2.1) are in fact Type-3 in nature. This can be seen clearly when we rewrite the production for < id >in the form B t | t, viz.

<id> ::= l | <id> l | <id> d

where l and d stand for a letter and digit respectively.

Type-3 grammars are also known as linear grammars or regular grammars. These are further categorized into left-linear and right-linear grammars depending on whether the NT in the RHS alternative appears at the extreme left or extreme right.

Operator grammars:

An operator grammar is a grammar none of whose productions contain two or more consecutive NTs in any RHS alternative.

Thus, nonterminals occurring in an RHS string are separated by one or more terminal symbols. All terminal symbols occurring in the RHS strings are called Operators of the grammar.

Ambiguity in Grammatic specification:

Ambiguity implies the possibility of different interpretations of a source string. In natural languages, ambiguity may concern the meaning or syntax category of a word, or the syntactic structure of a construct. For example, a word can have multiple meanings or can be both noun and verb (e.g. the word 'base') and a sentence can have multiple syntactic structures (e.g. 'police ordered to stop speeding on roads').

Formal language grammars avoid ambiguity at the level of a lexical unit or a syntax category. This is achieved by the simple rule that identical strings cannot appear on the RHS of more than one product in the grammar. The existence of ambiguity at the level of the syntactic structure of a string would mean that more than one parse tree can be built for the string. In turn, this would mean that the string can have more than one meaning associated with it.

Eliminating ambiguity:

An ambiguous grammar should be rewritten to eliminate ambiguity. In figure

2.10 the first tree does not reflect the conventional meaning associated with a+b*c, while the second tree does. Hence the grammar must be rewritten such that the reduction of '*' precedes the reduction of '+' in a+b*c. the normal method of achieving this is to use a hierarchy of NTs in the grammar and to associate the reduction or derivation of an operator with an appropriate NT.

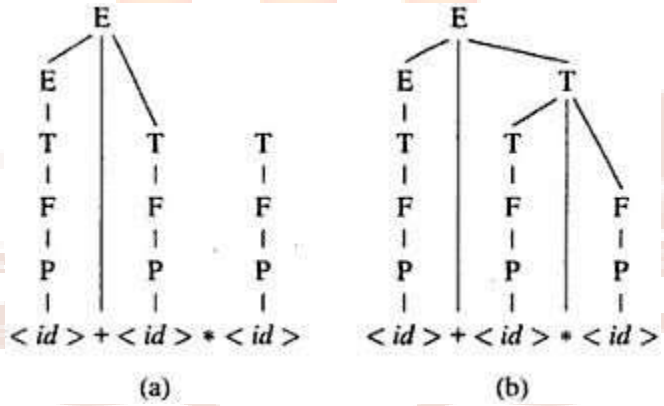


Fig 2.11: Ensuring a unique parse tree for an expression

Figure 2.11 illustrates the reduction of a+b*c according to grammar 2.1. Part (a) depicts an attempt to reduce a+b to <exp>. This attempt fails because the resulting string <exp>*<id> cannot be reduced to<exp>.part(b) depicts the correct reduction of a+b*c in which b*c is first reduced to<exp>. this sequence of reductions can be explained as follows:

Grammar 2.1 associates the recognition of '*' with the reduction of a string to a <term>, which alone can take part in a reduction involving '+'. Consequently, in a+b+c,'*' must be necessarily reduced before'+'. This yields the conventional meaning of the string. Other NTs, viz.<factor> and <primary>, similarly take care of the operator ' ‡ ' and the parentheses '(...)'. Hence there is no ambiguity in grammar.

Binding and Binding Times:

Each program entity pei in program P has a set of attributes Ai ≡ {aj} associated with it. If pei is an identifier, it has an attribute kind whose value indicates whether it is a variable, a procedure, or a reserved identifier (i.e. a Keyword). A variable has attributes like type, dimensionality, scope, memory address, etc. Note that the attribute of one program entity may be another program entity. For example, type is an attribute of a variable. It is also a program entity with its own attributes, e.g. size (i.e. number of memory bytes). The values of the attributes of the type typ should be determined sometime before a language processor processes a declaration statement using that type.

Var: type;

Binding (Definition): A binding is the association of an attribute of a program entity with a value.

Binding time is the time at which binding is performed. Thus the type attribute of variable var is bound to *typ* when its declaration is processed. The size attribute of *typ* is bound to a value sometime prior to this binding. We are interested in the following binding times:

1. Language Definition time of L
2. Language implementation time of L
3. Compilation time of P
4. Execution init time of proc
5. Execution time of proc.

Where L is a programming language, P is a program written in L and proc is a procedure in P. Note that language implementation time is the time when a language translator is designed. The preceding list of binding times is not exhaustive; other binding times can be defined, viz. binding at the linking time of P. The language definition of L specifies binding times for the attributes of various entities of a program written in L.

Importance of binding times:

The way a language processor can handle using an entity depends on the binding time of an attribute of a programme entity. A compiler can generate code specifically tailored to a binding performed during or before compilation time. However, a compiler cannot generate

such code for bindings performed during or before compilation time. However, a compiler cannot generate such code for bindings performed later than compilation time. This affects the execution efficiency of the target program.

Static and Dynamic bindings:

Static binding is a binding *performed before* the execution of a program begins.

Dynamic binding is a binding *performed after* the execution of a program has begun.

Static binding led to a more efficient execution of a program than dynamic bindings.

SELF ASSESSMENT QUESTIONS – 3

9. The lexical and syntactic features of a programming language are specified by its_____.
10. A rule of the grammar also called a rewriting rule is_____.
11. A grammar in which no productions contain two or more consecutive NTs in any RHS alternativeis called _____
12. Association of an attribute of a program entity with a value is called _____.
13. A binding *performed before* the execution of a program begins is called _____.

5. LANGUAGE PROCESSOR DEVELOPMENT TOOLS

The analysis phase of a language processor has a standard form irrespective of its purpose; the source text is subjected to lexical, syntax, and semantic analysis, and the results of the analysis are represented in an IR. Thus writing of language processors is a well-understood and repetitive process that ideally suits the program generation approach to software development. This has led to the development of a set of language processor development tools (LPDTs) focusing on the generation of the analysis phase of language processors.

Figure 2.12 shows the schematic of an LPDT which generates the analysis phase of a language processor whose source language is L.

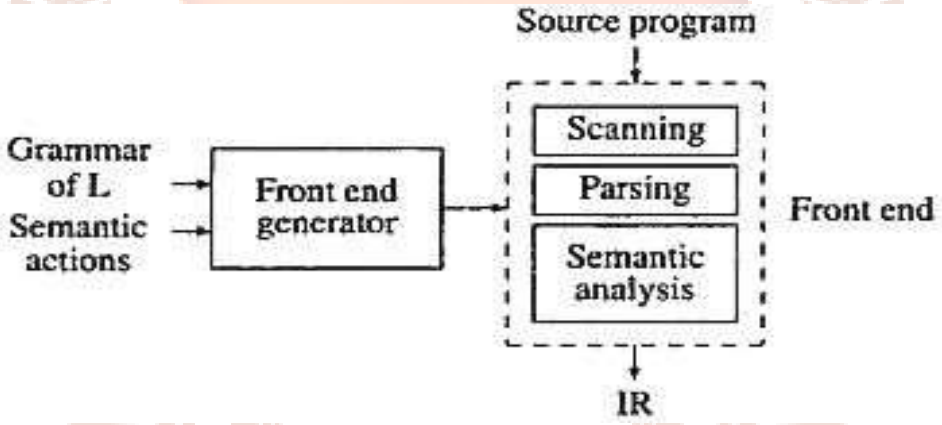


Fig 2.12: A language processor development tool (LPDT)

The LPDT requires the following two inputs:

1. Specification of a grammar of language L
2. Specification of semantic actions to be performed in the analysis phase

It generates programs that perform lexical, syntax, and semantic analysis of the source program and constructs the IR. These programs collectively form the analysis phase of the language processor.

We briefly discuss two LPDTs widely used in practice. These are the lexical analyzer generator LEX, and the parser generator YACC. The input to these tools is a specification of the lexical and syntactic constructs of L, and the semantic actions to be performed on recognizing the constructs. The specification consists of a set of translation rules of the form

<String specification>{ < semantic action>}

Where <semantic action> consists of C code. This code is executed when a string matching <string specification> is encountered in the input. LEX and YACC generate C programs that contain the code for scanning and parsing, respectively, and the semantic actions contained in the specification. A YACC generated parser can use a LEX generated scanner as a routine if the scanner and parser use the same conventions concerning the representation of tokens. Fig 2.12 shows a schematic for developing the analysis phase of a compiler for language L using LEX and YACC.

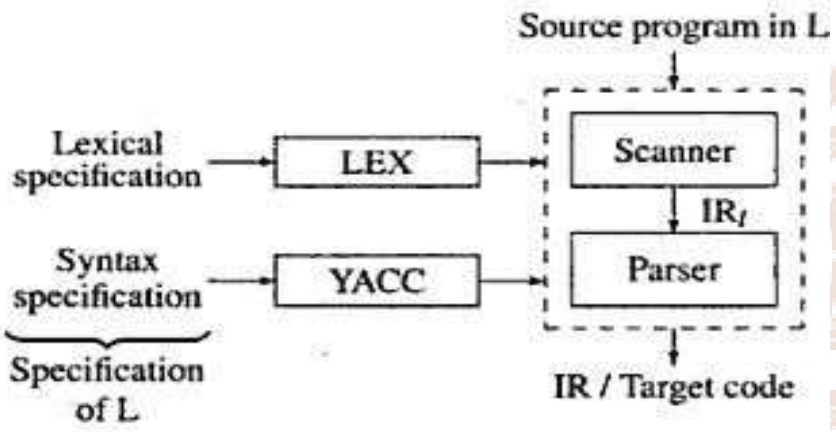


Fig 2.13: Using LEX and YACC

The analysis phase processes the source program to build an intermediate representation (IR). A single pass compiler can be built using LEX and YACC if the semantic actions are aimed at generating target code instead of IR. Note that the scanner also generates an intermediate representation of a source program for use by the parser. We call it IRl in figure 2.13 to differentiate it from the IR of the analysis phase.

LEX:

LEX accepts an input specification that consists of two components. The first component is a specification of strings representing the lexical units in L, e.g. id's and constants. This specification is in the form of regular expressions. The second component is a specification of semantic actions aimed at building an IR. The IR consists of a set of tables of lexical units and a sequence of tokens for the lexical units occurring in a source statement. Accordingly, the semantic actions make new entries in the tables and build tokens for the lexical units.

YACC:

Each string specification in the input to YACC resembles a grammar production. The parser generated by YACC performs reductions according to this grammar. The actions associated with a string specification are executed when a reduction is made according to the specification. An attribute is associated with every nonterminal symbol. The value of this attribute can be manipulated during parsing. The attribute can be given any user-designed structure. A symbol '$n' in the action part of a translation rule refers to the attribute of the n^{th} symbol in the Right Hand Side (RHS) of the string specification.'$$' represents the attribute of the Left Hand Side (LHS) symbol of the string specification.

SELF ASSESSMENT QUESTIONS – 4

14. Language processor development tools (LPDTs) focuses on generation of the _____phase of language processors.
15. Two LPDTs widely used in practice are_____and_____.

6. SUMMARY

Let us recapitulate the important concepts discussed in this unit:

- Computer programming languages are classified into three categories, they are, machine language, assembly language, and high-level language.
- A language processor is Software that bridges a specification or execution gap.
- A language translator is a program that takes input written in a language and produces output written in other programming languages.
- Assembler, compiler, interpreter are examples of language processor.
- Language processing contains different phases, lexical analysis, syntax analysis, and semantic analysis.
- Lexical analysis performs scanning and syntax analysis performs parsing of a given sentence.
- A forward reference of a program entity is a reference to the entity which precedes its definition in the program.
- Syntax analysis processes the string of tokens built by lexical analysis to determine the statement class.
- A binding is the association of an attribute of a program entity with a value.
- The popular language processor development tools are Lex and Yacc.

7. GLOSSARY

Assembly language: A low-level programming language, that corresponds closely to the instruction set of a given computer, allows symbolic naming of operations and addresses and usually results in a one-to-one translation of program instructions [mnemonics] into machine instructions.

Compilation: Translating a program expressed in a problem-oriented language or a procedure-oriented language into object code. In contrast with assembling, interpret

Error: A fault in a program that causes the program to perform in an unintended or unanticipated manner

Syntax: The structural or grammatical rules that define how symbols in a language are to be combined to form words, phrases, expressions, and other allowable constructs.

Translation: Converting from one language form to another. See: assembling, compilation, interpret.

8. TERMINAL QUESTIONS

Short Answer Questions

1. What is language processing? Explain its activities in detail.
2. Define Grammar. Explain different types of Grammars.
3. Write a short note on Lex and Yacc.
4. What is parsing? Why is it required?
5. What is binding? When does it take place?

8.1 Answers

A. Self-Assessment Questions

1. Language Processor
2. Language Translator
3. Instruction Pointer
4. Translation and Interpretation
5. Language Processing
6. Forward reference

7. Symbol table
8. Back end
9. Grammar
10. Production
11. Operator grammar
12. Binding
13. Static binding
14. Analysis
15. LEX and YACC

Short Answer Questions

1. Language processing activities are those that bridge the specification gap and those that bridge the execution gap. (Refer section 2 for detail)
2. The lexical and syntactic features of a programming language are specified by its grammar. (Refer section 4 for detail)
3. LEX and YACC are two language processor development tools. (Refer section 5 for detail)
4. Refer section 3, Syntax analysis (parsing) for detail.
5. A Binding is the association of an attribute of a program entity with a value. Refer section 4, Binding and Binding times for detail.

9. SUGGESTED BOOKS

- Dhamdhere (2002). Systems programming and operating systems Tata McGraw-Hill.
- M. Joseph (2007). System software, Firewall Media.