1. Language Processor

1.1 INTRODUCTION

Language processing activities arise due to the differences between

- The manner in which a software designer describes the ideas concerning the behavior of a software and
- The manner in which these ideas are implemented in a computer system.

The designer expresses the ideas in terms related to the application domain of the software. To implement these ideas, their description has to be interpreted in terms related to the *execution domain* of the computer system. The term semantics represent the rules of meaning of a domain, and the term semantic gap to represent the difference between the semantics of two domains. Figure 1 depicts the semantic gap between the application and execution domains

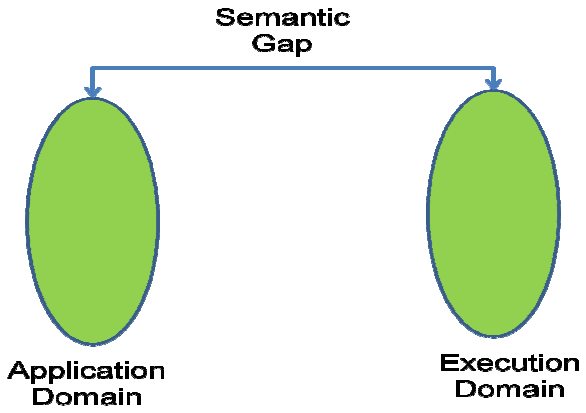


Figure 1: Semantic gap

The semantic gap has many consequences,

- large development times
- large development efforts, and
- poor quality of software.

These issues are tackled by Software engineering through the use of methodologies and programming languages (PLs). The software engineering steps aimed at the use of a PL can be grouped into

1. Specification, design and coding steps
2. PL implementation steps.

Software implementation using a PL introduces a new domain, the PL domain. The semantic gap between the application domain and the execution domain is bridged by the software engineering steps. The first step bridges the gap between the application and PL domains, while the second step bridges the gap between the PL and execution domains. The gap between the application and PL domains is called the specification-and-design gap or simply the specification gap. The gap between the PL and execution domains is 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, viz. a translator or an interpreter

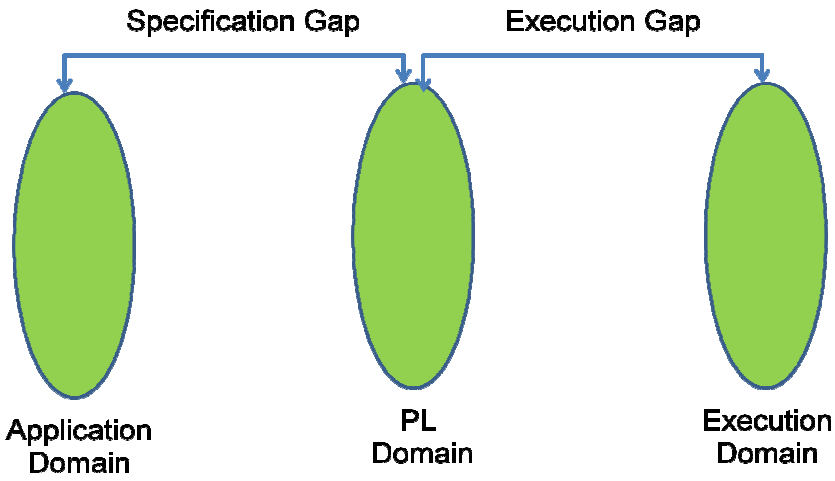


Figure 2: Specification and execution gap

Advantages of introducing the PL domain

- The gap to be bridged by the software designer is now between the application domain and the PL domain rather than between the application domain and the execution domain.

- This reduces the severity of the consequences of semantic gap mentioned earlier.
- Apart from bridging the gap between the PL and execution domains, the language processor provides a diagnostic capability which detects and indicates errors in its input which helps in improving the quality of the software.

Specification gap: Specification gap is the semantic gap between two specifications of the same task.
Execution gap: Execution gap is the gap between the semantics of programs (that perform the same task) written in different programming languages.

We assume that each domain has a specification language (SL). A specification written in an SL is a program in SL. The specification language of the PL domain is the PL itself. The specification language of the execution domain is the machine language of the computer system.

Language processors

A language processor is software which bridges a specification or execution gap.
We use the term language processing to describe the activity performed by a language processor and assume a diagnostic capability as an implicit part of any form of language processing. We refer to the program form input to a language processor as the source program and to its output as the target program. The languages in which these programs are written are called source language and target language, respectively. A language processor typically abandons generation of the target program if it detects errors in the source program.

A spectrum of language processors is defined to meet practical requirements.

1. A **language translator** bridges an execution gap to the machine language (or assembly language) of a computer system. An *assembler* is a language translator whose source language is assembly language. A *compiler* is any language translator which is not an assembler.

2. A **detranslator** bridges the same execution gap as the language translator, but in the reverse direction.
3. A **preprocessor** is a language processor which bridges an execution gap but is not a language translator.
4. A **language migrator** bridges the specification gap between two PLs.

Example 1: Following figure 3 shows two language processors. The language processor of part (a) converts a C++ program into a C program, hence it is a preprocessor. The language processor of part (b) is a language translator for C++ since it produces a machine language program. In both cases the source program is in C++. The target programs are the C program and the machine language program, respectively.

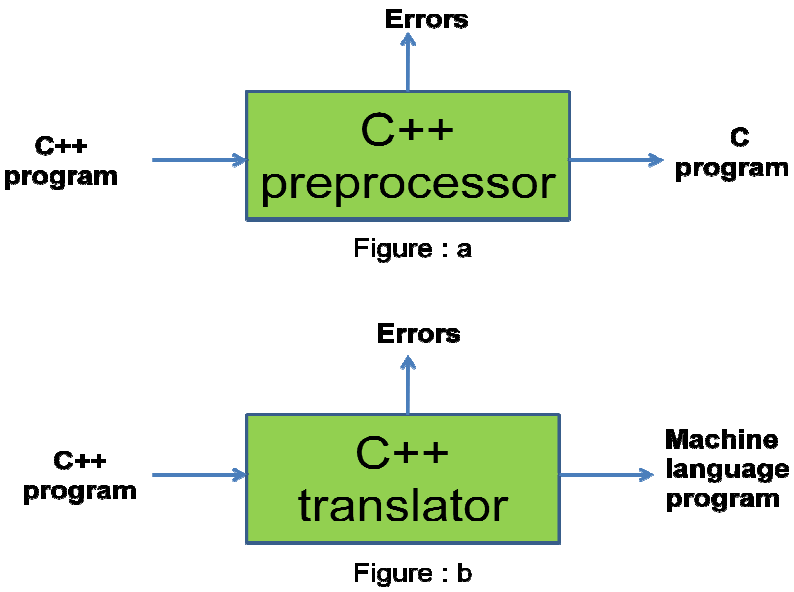


Figure 3: Language processors

Interpreters

An-interpreter is a language processor which bridges an execution gap without generating a machine language program. The interpreter is a language translator.

The absence of a target program implies the absence of an output interface of the

interpreter. Thus the language processing activities of an interpreter cannot be separated from its program execution activities. Hence an interpreter executes a program written in a PL. In essence, the execution gap vanishes totally. Figure 4 is a schematic representation of an interpreter. The interpreter domain encompasses the PL domain as well as the execution domain. Thus, the specification language of the PL domain is identical with the specification language of the interpreter domain.

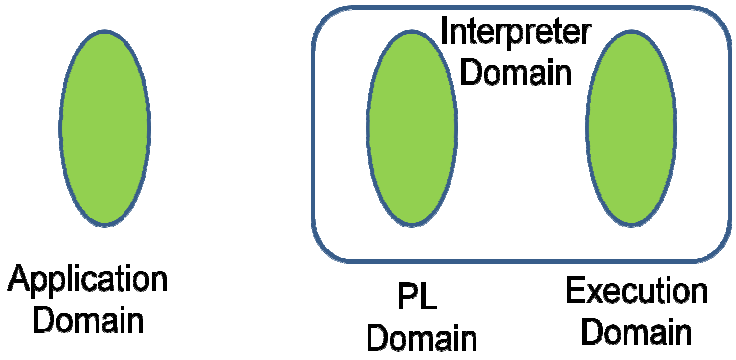


Figure 4: Interpreter

Problem oriented and procedure oriented languages

The three consequences of the semantic gap are in fact the consequences of a specification gap which are as follows:

- Large development times
- Large development efforts, and
- Poor quality of software.

Software systems are poor in quality and require large amounts of time and effort to develop due to difficulties in bridging the specification gap. A classical solution is to develop a PL such that the PL domain is very close or identical to the application domain. PL features now directly model aspects of the application domain, which leads to a very small specification gap. Such PLs can only be used for specific applications; hence they are called problem oriented languages. They have large execution gaps, however this is acceptable because the gap is bridged by the translator or interpreter and does not concern the software designer.

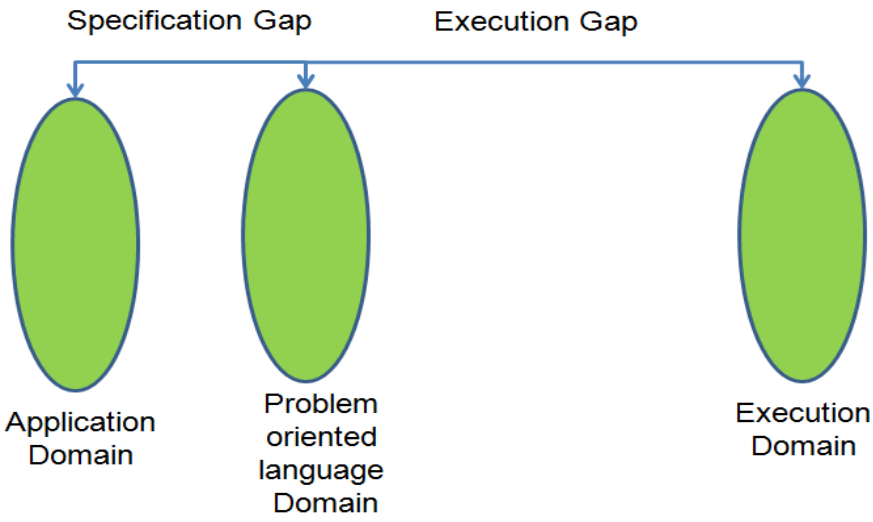


Figure 5: Problem oriented language domain

A procedure oriented language provides general purpose facilities required in most application domains. Such a language is independent of specific application domains and results in a large specification gap which has to be bridged by an application designer.

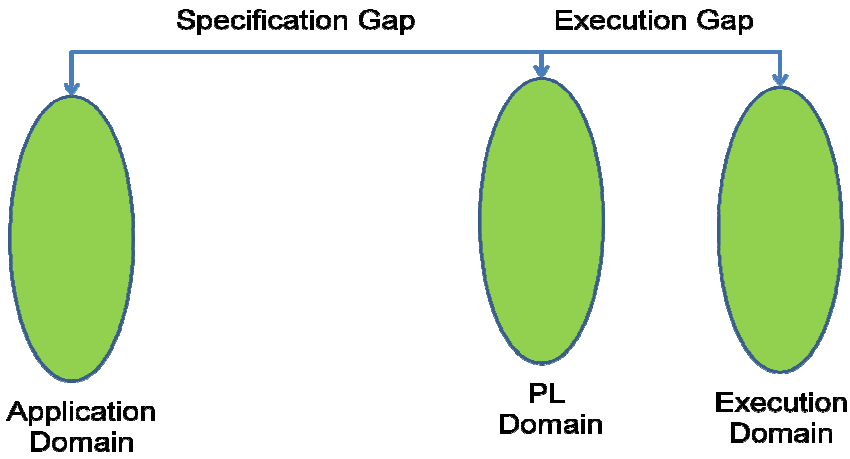


Figure 6: Procedure oriented language

1.2 LANGUAGE PROCESSING ACTIVITIES

The fundamental language processing activities can be divided into those that bridge the specification gap and those that bridge the execution gap. We name these activities as

1. **Program generation activities**: A program generation activity aims at automatic generation of a program. The source language is a specification language of an application domain and the target language is typically a procedure oriented PL.
2. **Program execution activities**: A program execution activity organizes the execution of a program written in a PL on a computer system. Its source language could be a procedure oriented language or a problem oriented language

1.2.1 Program Generation

Figure 7 depicts the program generation activity. The program generator is a software system which accepts the specification of a program to be generated, and generates a program in the target PL. So the program generator introduces a new domain called the program generator domain between the application and PL domains which is shown in figure 8.. The specification gap is now the gap between the application domain and the program generator domain. This gap is smaller than the gap between the application domain and the target PL domain.

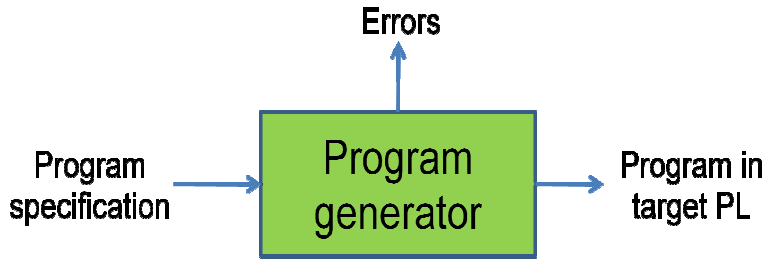


Figure 7: Program generation

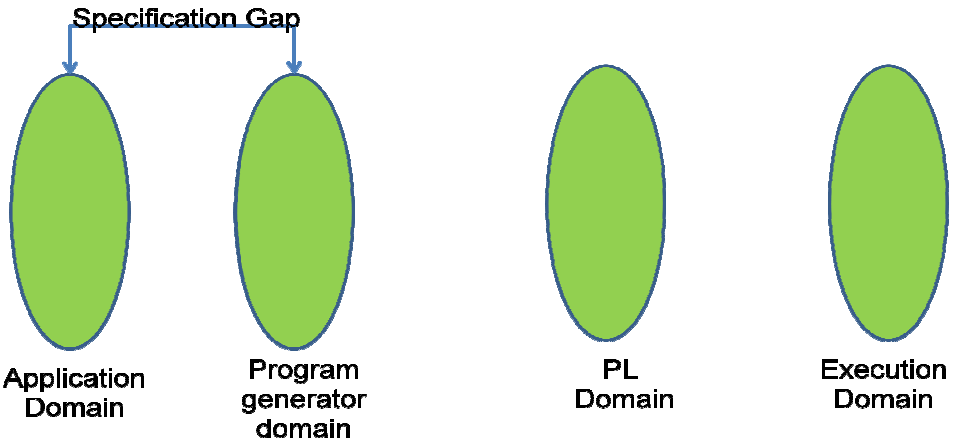


Figure 8: Program generator domain

Reduction in the specification gap increases the reliability of the generated program. Since the generator domain is close to the application domain, it is easy for the designer or programmer to write the specification of the program to be generated.

The harder task of bridging the gap to the PL domain is performed by the generator. This arrangement also reduces the testing effort. To test an application generated by using the generator, it is necessary to only verify the correctness of the specification input to the program generator. This is a much simpler task than verifying correctness of the generated program. This task can be further simplified by providing a good diagnostic (i.e. error indication) capability in the program generator which would detect inconsistencies in the specification.

It is more economical to develop a program generator than to develop a problem oriented language. This is because a problem oriented language suffers a very large execution gap between the PL domain and the execution domain, whereas the program generator has a smaller semantic gap to the target PL domain, which is the domain of a standard procedure oriented language. The execution gap between the target PL domain and the execution domain is bridged by the compiler or interpreter for the PL.

Example 2: A screen handling program handles screen IO in a data entry environment. It displays the field headings and default values for various fields in the screen and accepts data values for the fields. Figure 9 shows a screen for data entry of employee

information. A data entry operator can move the cursor to a field and key in its value. The screen handling program accepts the value and stores it in a data base.

A screen generator generates screen handling programs. It accepts a specification of the screen to be generated (we will call it the screen spec) and generates a program that performs the desired screen handling. The specification for some fields in Figure 9 could be as follows:

```
Employee name:    char :  start(line=2,position=25)
                          end(line=2,position=80)
Married:          char :  start(line=10,position=25)
                          end(line=10,position=27)
                          default('Yes')
```

Errors in the specification, e.g. invalid start or end positions or conflicting specifications for a field, are detected by the generator. The generated screen handling program validates the data during data entry, e.g. the *age* field must only contain digits, the *sex* field must only contain M or F, etc

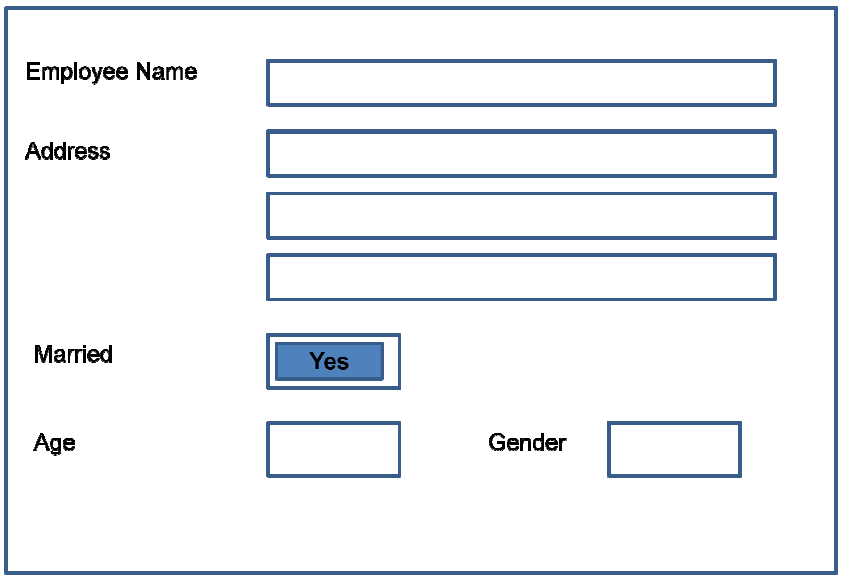


Figure 9: Screen displayed by a screen handling program

1.2.2 Program Execution

Two popular models for program execution are

- Translation and
- Interpretation

Program translation

The program translation model bridges the execution gap by translating a program written in a PL, called the source program (SP), into an equivalent program in the machine or assembly language of the computer system, called the target program. (Figure 10)

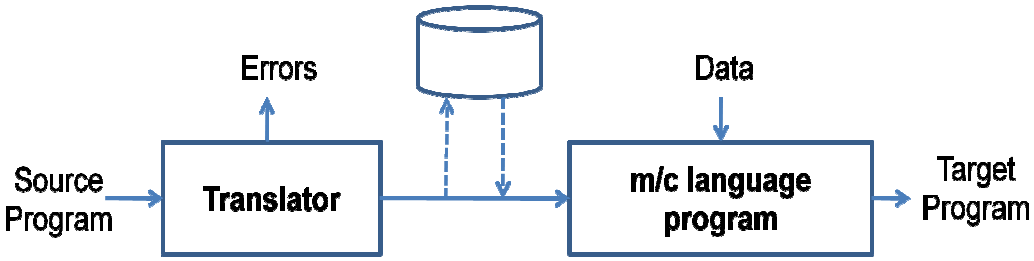


Figure 10: Program translation model

Characteristics of the program translation model are:

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

Program interpretation

Figure 11a and 11b shows a schematic of program interpretation and program execution respectively. The interpreter reads the source program and stores it in its memory. During interpretation it takes a source statement, determines its meaning and performs actions which implement it. This includes computational and input-output

actions.

The CPU uses a program counter (PC) to note the address of the next instruction to be executed. This instruction is subjected to the instruction execution cycle consisting of the following steps:

1. Fetch the instruction.
2. Decode the instruction to determine the operation to be performed, and also its operands.
3. Execute the instruction.

At the end of the cycle, the instruction address in PC is updated and the cycle is repeated for the next instruction.

Program interpretation can proceed in an analogous manner. Thus, the PC can indicate which statement of the source program is to be interpreted next. This statement would be subjected to the interpretation cycle, which could consist of the following steps:

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

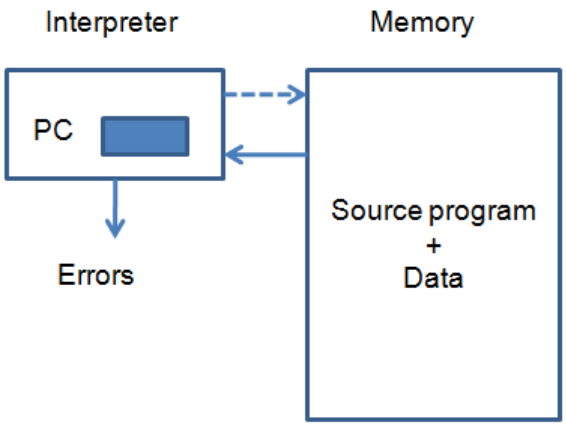


Figure 11a: Interpretation

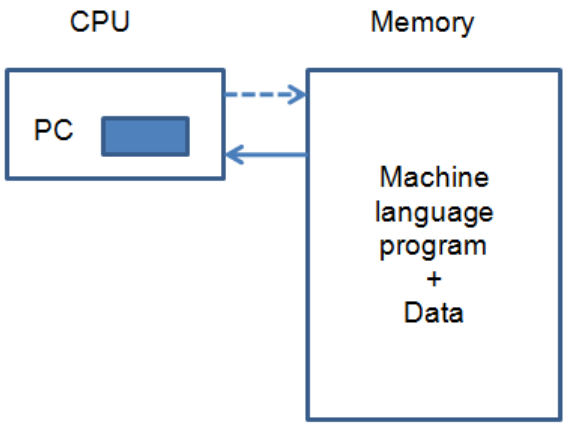


Figure 11b: Program execution

From this analogy, we can identify the following characteristics of interpretation:

- The source program is retained in the source form itself, i.e. no target program form exists,
- A statement is analyzed during its interpretation.

Comparison

A fixed cost (the translation overhead) is incurred in the use of the program translation model. If the source program is modified, the translation cost must be incurred again irrespective of the size of the modification. However, execution of the* target program is efficient since the target program is in the machine language. Use of the interpretation model does not incur the translation overheads. This is advantageous if a program is modified between executions, as in program testing and debugging. Interpretation is however slower than execution of a machine language program because of Step 2 in the interpretation cycle.

1.3 FUNDAMENTALS OF LANGUAGE PROCESSING

Language Processing: Language Processing = Analysis of SP + Synthesis of TP.

The collection of language processor components engaged in analyzing a source program is referred as the analysis phase of the language processor. Components engaged in synthesizing a target program constitute the synthesis phase.

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

1. **Lexical rules** which govern the formation of valid lexical units in the source language.
2. **Syntax rules** which govern the formation of valid statements in the source language.
3. **Semantic rules** which associate meaning with valid statements of the language.

The analysis phase uses each component of the source language specification to determine relevant information concerning a statement in the source program. Thus, analysis of a source statement consists of lexical, syntax and semantic analysis.

Example 3: Consider the statement

percent_profit := (profit * 100) / cost_price;

in some programming language.

- Lexical analysis identifies : =, * and / as operators, 100 as a constant and the remaining strings as identifiers.
- Syntax analysis identifies the statement as an assignment statement with percent_profit as the left hand side and (profit * 100) / cost-price as the expression on the right hand side.
- Semantic analysis determines the meaning of the statement to be the assignment of (profit x 100)/cost_price to percent_profit

The synthesis phase is concerned with the construction of target language statements) which have the same meaning as a source statement. Typically, this consists of two main activities:

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

These activities are referred as memory allocation and code generation, respectively.

Example 4: A language processor generates the following assembly language statements for the source statement of above example

```
MOVER           AREG, PROFIT
MULT            AREG, 100
DIV             AREG, COST_PRICE
MOVEM           AREG, PERCENT_PROFIT
----------
PERCENT_PROFIT  DW   1
PROFIT          DW   1
COST_PRICE      DW   1
```

where MOVER and MOVEM move a value from a memory location to a CPU register and vice versa, respectively, and DW reserves one or more words in memory.

Phases and passes of a language processor

A language processor consists of two distinct phases—the analysis phase and the synthesis phase. Figure 12 shows a schematic of a language processor. This schematic may give the impression that language processing can be performed on a statement-by-statement basis that is, analysis of a source statement can be immediately followed by synthesis of equivalent target statements. This may not be feasible due to:

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

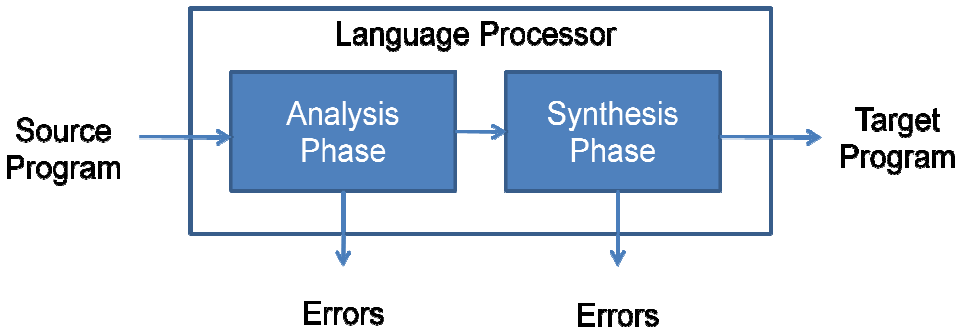


Figure 12: Phases of a language processor

We discuss these issues in the following.

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

While processing a statement containing a forward reference, a language processor does not possess all relevant information concerning the referenced entity. This creates difficulties in synthesizing the equivalent target statements. This problem can be solved by postponing the generation of target code until more information concerning the entity becomes available. Postponing the generation of target code may also reduce memory requirements of the language processor and simplify its organization.

Example 5: Consider the statement percent_profit := (profit * 100) / cost_price to be a part of the following program in some programming language:

```
percent_profit := (profit * 100) / cost_price;
--------
long profit;
```

The statement long profit; declares profit to have a double precision value. The reference to profit in the assignment statement constitutes a forward reference because the declaration of profit occurs later in the program. Since the type of profit is not known while processing the assignment statement, correct code cannot be generated for it ill a statement-by-statement manner.

Departure from the statement-by-statement leads to the multipass model of language processing.

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

Here 'pass' is an abstract noun describing the processing performed by the language processor.

Example 6: It is possible to process the program fragment of

```
percent_profit := (profit * 100) / cost_price;
--------
long profit;
```

in two passes as follows:

Pass I: Perform analysis of the source program and note relevant information

Pass II: Perform synthesis of target program

Information concerning the type of profit is noted in pass I. This information is used during pass II to perform code generation.

Intermediate representation of programs

The language processor performs certain processing more than once. In pass I, it

analyses the source program to note the type information. In pass II, it once again analyses the source program to generate target code using the type information noted in pass I. This can be avoided using an intermediate representation of the source program.

Intermediate Representation (IR): An intermediate representation (IR) is a representation *of* a source program which reflects the effect *of* some, but not all, analysis and synthesis tasks performed during language processing.

Figure 13 depicts the schematic of a two pass language processor. The first pass performs analysis of the source program and reflects its results in the intermediate representation. The second pass reads and analyses the IR, instead of the source program, to perform synthesis of the target 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. The front and back ends of a language processor need not coexist in memory. This reduces the memory requirements

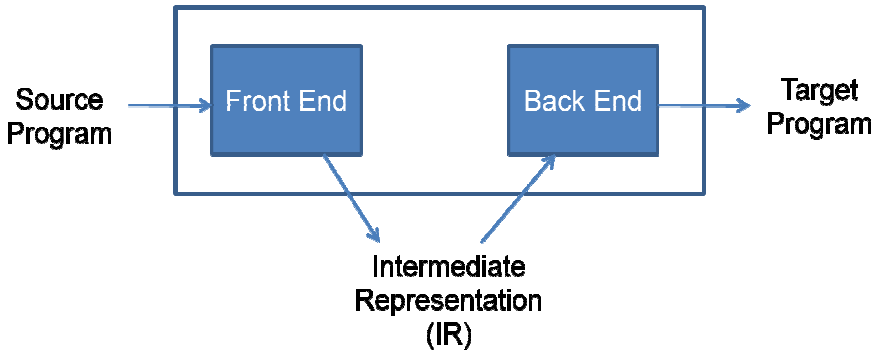


Figure 13: Two pass schematic for language processing

Desirable properties of an IR are:

- **Ease of use***:* IR should be easy to construct and analyse.
- **Processing efficiency***:* efficient algorithms must exist for constructing and

analyzing the IR.

- **Memory efficiency***:* IR must be compact.

Semantic actions

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. These include actions for the following:

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

1.3.1 A Toy Compiler

The front end and back end of a toy compiler for a Pascal-like language is described here.

1.3.1.1 The Front End

The front end performs

- Lexical analysis,
- Syntax analysis and
- Semantic analysis of the source program.

Each kind of analysis involves the following functions:

1. Determine validity of a source statement from the viewpoint of the 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.

The word 'content' has different connotations in lexical, syntax and semantic analysis.

- In lexical analysis, the content is the lexical class to which each lexical unit belongs,
- In syntax analysis it is the syntactic structure of a source statement.

- In semantic analysis the content is the meaning of a statement—for a declaration statement, it is the sef of attributes of a declared variable (e.g. type, length and dimensionality), while for an imperative statement, it is the sequence of actions implied by the statement.

Each analysis represents the 'content' of a source statement in the form of (1) tables of information, and (2) description of the source statement. Subsequent analysis uses this information for its own purposes and either adds information to these tables and descriptions, or constructs its own tables and descriptions.

For example, syntax analysis uses information concerning the lexical class of lexical units and constructs a representation for the syntactic structure of the source statement. Semantic analysis uses information concerning the syntactic structure and constructs a representation for the meaning of the statement. The tables and descriptions at the end of semantic analysis form the IR of the front end.

Output of the front end

The IR produced by the front end consists of two components:

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

Tables

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

***Intermediate code* (I C)**

The IC is a sequence of IC units; each IC unit representing the meaning of one action in SP. IC units may contain references to the information in various tables.

Example 7: Figure 14 shows the IR produced by the analysis phase for the program

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

Symbol Table:

No.	Symbol	Type	Length	Address
1	i	int		
2	a	real		
3	b	real		
4	i*	real		
5	temp	real		

Intermediate code

1. Convert (id, #1) to real, giving (id, #4)
2. Add (id, #4) to (id, #3) giving (id, #5)
3. Store (id, #5) in (Id, #2)

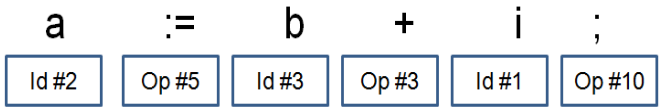
The symbol table contains information concerning the identifiers and their types. This information is determined during lexical and semantic analysis, respectively. In IC, the specification (Id, #1) refers to the id occupying the first entry in the table. i* and temp are temporary names added during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

Lexical analysis (Scanning)

Lexical analysis identifies the lexical units in a source statement. It then classifies the units into different lexical classes, e.g. id's, constants, reserved id's, etc, and enters them into different tables. Lexical analysis builds a descriptor, called a *token*, for each lexical unit. A token contains two fields—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.

Example 8: The statement a := b+i; is represented as the string of tokens



Syntax analysis (Parsing)

Syntax analysis processes the string of tokens 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

Example 9: Figure 14 shows IC for the statements a, b : real; and a := b + i;. A tree form is chosen for IC because a tree can represent the hierarchical structure of a PL statement appropriately. Each node in a tree is labeled by an entity.

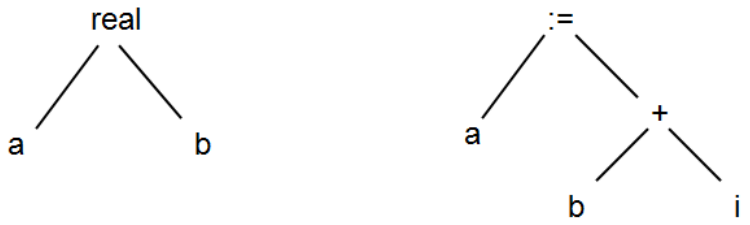


Figure 14: IC for the statement: a, b :real; a := b + i;

Semantic analysis

Semantic analysis of declaration statements differs from the semantic analysis of imperative statements. The former results in addition of information to the symbol table, e.g. type, length and dimensionality of variables. The latter 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, it adds

Example 10: Semantic analysis of the statement a := b + i; proceeds as follows:

1. Information concerning the type of the operands is added to the IC tree. The

IC tree now looks as in Figure 15(a).

2. Rules of meaning governing an assignment statement indicate that the expression on the right hand side should be evaluated first. Hence focus shifts to the right subtree rooted at '+'.
3. Rules of addition indicate that type conversion of i should be performed to ensure type compatibility of the operands of '+'. This leads to the action

   (i) Convert i to real, giving i*.

   which is added to the sequence of actions. The IC tree under consideration is modified to represent the effect of this action (see Figure 15(b)). The symbol i* is now added to the symbol table.

4. Rules of addition indicate that the addition is now feasible. This leads to the action

   (ii) Add i* to b, giving temp.

   The IC tree is transformed as shown in Figure 15(c), and temp is added to the symbol table.
5. The assignment can be performed now. This leads to the action

   (iii) Store temp in a.

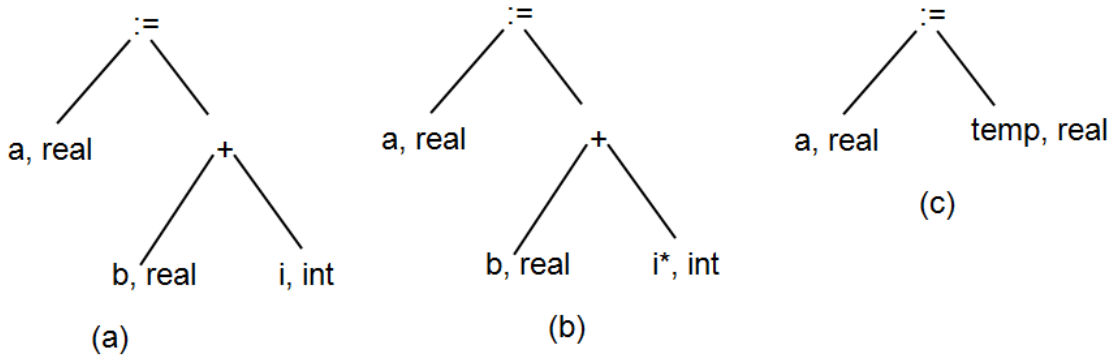


Figure 15: Steps in semantic analysis of an assignment statement

Figure 16 shows the schematic of the front end where arrows indicate flow of data.

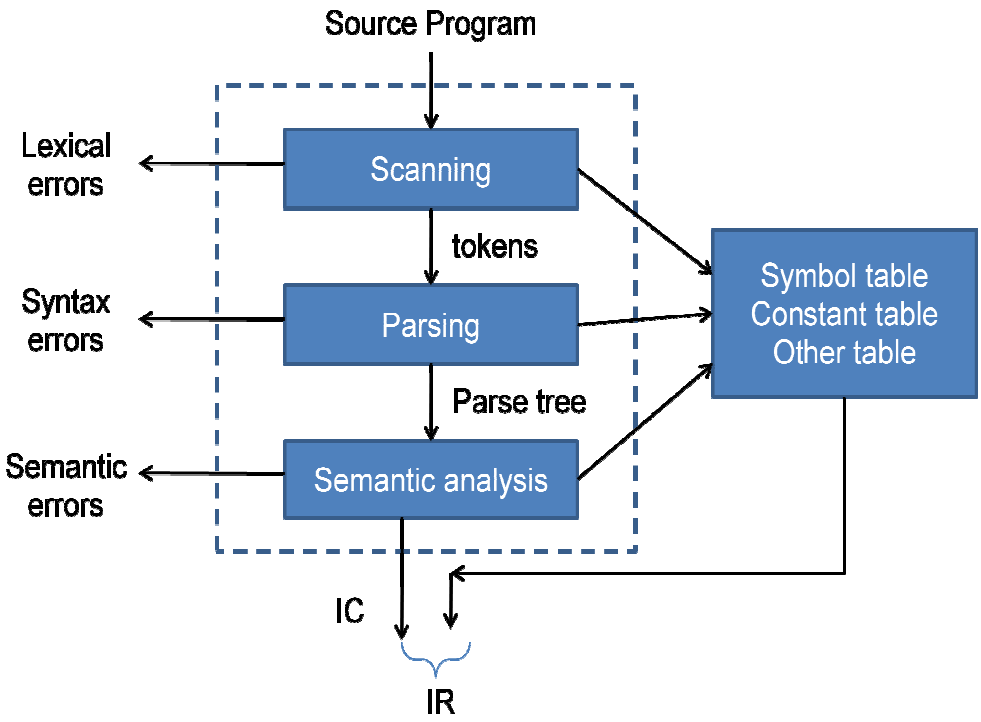


Figure 16: Front end of the toy compiler

1.3.1.2 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.

Example 11: After memory allocation, the symbol table looks as shown in Figure 17. The entries for i* and temp are not shown because memory allocation is not needed for these id's.

No.	Symbol	Type	Length	Address
1	i	int		2000
2	a	real		2001
3	b	real		2002

Figure 17: Symbol table after memory allocation

Certain decisions have to precede memory allocation, for example, whether i* and temp should be allocated memory. These decisions are taken in the preparatory steps of code generation.

Code generation

Code generation uses knowledge of the target architecture, viz. knowledge of instructions and addressing modes in the target computer, to select the 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.

Example 12: For the sequence of actions for the assignment statement a :* b+i;

(i) Convert i to real, giving i*.

(ii) Add i* to b, giving temp,

(iii) Store temp in a.

the synthesis phase may decide to hold the values of i* and temp in machine registers and may generate the assembly code

```
CONV_R    AREG, I
ADD_R     AREG, B
MOVEM     AREG, A
```

where CONV_R converts the value of I into the real representation and leaves the result in AREG. ADD_R performs the addition in real mode and MOVEM puts the result into

the memory area allocated to A.

Figure 18 shows a schematic of the back end.

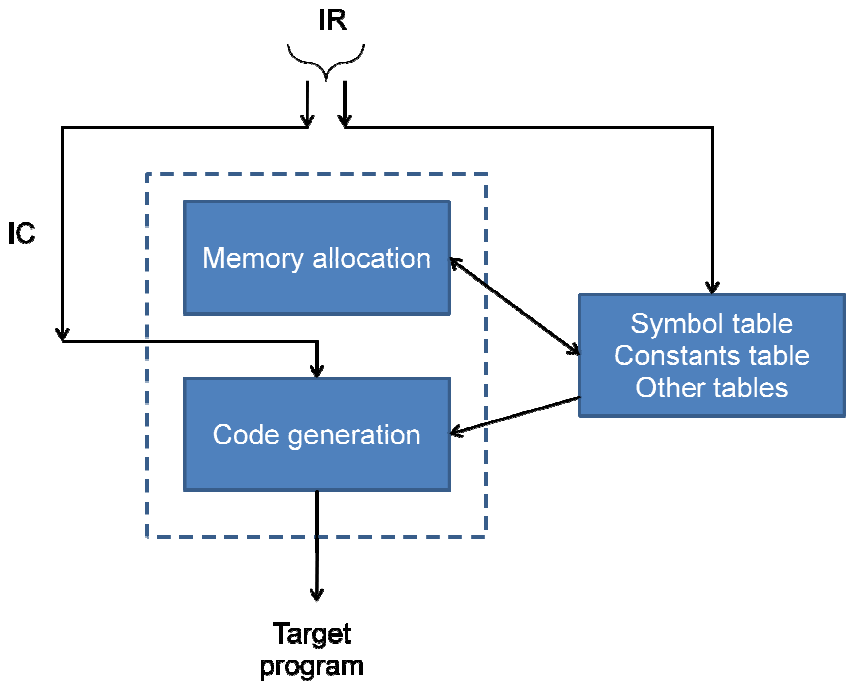


Figure 18: Back end of the toy compiler

1.4 FUNDAMENTALS OF LANGUAGE SPECIFICATION

A specification of the source language forms the basis of source program analysis. In this section, important lexical, syntactic and semantic features of a programming language shall be discussed.

1.4.1 Programming Language Grammars

The lexical and syntactic features of a programming language are specified by its grammar. A language L can be considered to be 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. It is clear that natural 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 Σ, is the collection of symbols in its character set. Lower case letters a, b, c, etc. are used to denote symbols in Σ. 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.

$$\Sigma = \{a, b \ldots z, 0, 1 \ldots 9\}$$

Here the symbols {, ',' and } are part of the notation. These symbols are called metasymbols to differentiate them from terminal symbols. If this is not the case, i.e. if a terminal symbol and a metasymbol are identical, we enclose the terminal symbol in quotes to differentiate it from the metasymbol. 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. Strings are represented by Greek symbols α, β, γ, etc. Thus α = axy is a string over Σ. The length of a string is the number of symbols in it. The absence of any symbol is also a string, the null string Ɛ. The concatenation operation combines two strings into a single string. It is used to build larger strings from existing strings. The null string can also participate in a concatenation, thus a.Ɛ = Ɛ.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 the grammar. A 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 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 expresses the fact that the noun phrase consists of an article followed by a noun.

Each grammar G defines a language L_G. 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 a of L_G is obtained by using the following procedure

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

Example 13: Following grammar defines a language consisting of noun phrases in English

<Noun Phrase> ::= <Article> <Noun>
<Article> ::= a| an | the
<Noun> ::= boy | apple

Grammar: A *grammar G of* a language L_G is a quadruple (L, SNT, S, P) where

Σ is the alphabet *of* L_G, i.e. the set *of T*s,
SNT is the set *of NTs,*
S is the distinguished symbol, and
P is the set *of* productions.

Derivation, reduction and parse trees

A grammar G is used for two purposes, to generate valid strings of L_G and to 'recognize' valid strings of L_G. 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 β be a string such that β = γAθ, then replacement of A by α in string β constitutes a *derivation* according to production P1. We use the notation N => η to denote direct derivation of η from N and N=>* η to denote transitive derivation of η (i.e. derivation in zero or more steps) from N, respectively. Thus, A => α only if A :: = α is a production of G and A =>* δ if A =>...=> δ. δ is a valid string according to G only if S =>* δ, where S is the distinguished symbol of G.

Example 14: Consider the grammar G

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

<Article> ::= a| an | the

<Noun> ::= boy | apple

Derivation of the string the boy according to grammar can be depicted as

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

=> the < *Noun* >

=> the boy

Example 15: Consider the grammar G

<Sentence> :: = <Noun Phrase> <Verb Phrase>

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

<Verb Phrase> ::= <Verb> <Noun Phrase>

<Article> ::= a| an| the

<Noun> ::= boy | apple

<Verb> ::= ate

The following strings are sentential form of L_G

<Sentence>

<Noun Phrase> <Verb Phrase>

<Article> <Noun> <Verb Phrase>

<Article> <Noun> <Verb> <Noun Phrase>

the <Noun> <Verb> <Article> <Noun>

the boy <Verb> <Article> <Noun>

the boy ate <Article> <Noun>

the boy ate an <Noun>

the boy ate an apple

Reduction

Let production P1 of grammar G be of the form

P1 : A :: = α

and let σ be a string such that σ = γαθ, then replacement of α by A in string σ constitutes a *derivation* according to production P1. We use the notation η→ N and η →* 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. δ is a valid string according to G only if δ →* S, where S is the distinguished symbol of G.

Example 16: To determine the validity of the string

the boy ate an apple

according to the following grammar

<Sentence> :: = <Noun Phrase> <Verb Phrase>

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

<Verb Phrase> ::= <Verb> <Noun Phrase>

<Article> ::= a| an| the

<Noun> ::= boy | apple

<Verb> ::= ate

We perform the following reductions:

Step	String
1	the boy ate an apple
2	<Article> boy ate an apple
3	<Article> <Noun> ate an apple
4	<Article> <Noun> <Verb> an apple
5	< Article> <Noun> <Verb> <Article> apple
6	< Article> <Noun> <Verb> <Article> <Noun>
7	<Noun Phrase> <Verb> <Article> <Noun>
8	<Noun Phrase> <Verb> <Noun Phrase>
9	<Noun Phrase> <Verb Phrase>
10	<Sentence>

Parse Tree

A sequence of derivations or reductions reveals the syntactic structure of a string with respect to G. The syntactic structure can be depicted in the form of a parse 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.

Example 17: Figure 19 shows the parse tree of the string '**the boy ate an apple'.** The superscript associated with a node in the tree indicates the step in the reduction sequence which led to the subtree rooted at that node. Reduction steps 1 and 2 lead to reduction of the and boy to <Article> and <Noun>, respectively. Step 3 combines the parse trees of < Article > and < noun > to give the subtree rooted at < Noun Phrase >.

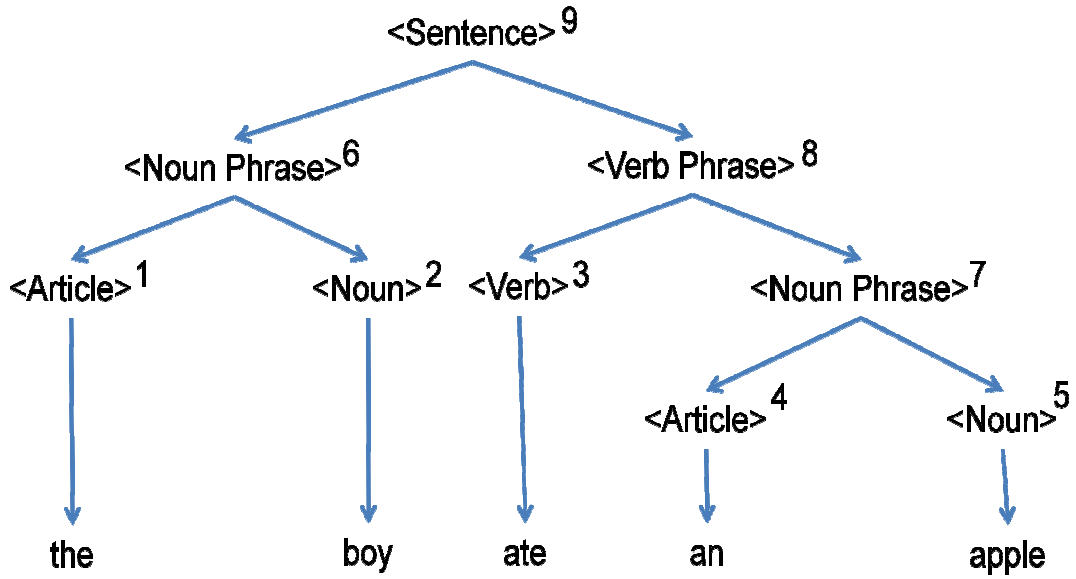


Figure 19: Parse Tree

An identical tree would have been obtained if the boy ate an apple was derived from S.

Recursive specification

Following grammar is a complete grammar for an arithmetic expression containing the operators f (exponentiation), * and +.

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

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

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

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

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

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

<letter> ::= a | b | …. | z

<digit> ::= 0 | 1|….| 9 ----------Grammar (a)

This grammar uses the notation known as the Backus Naur Form (BNF). A new element here is [...], which is used to enclose an optional specification. Thus, the rules for <id > and < const > in grammar (a) are equivalent to the rules

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

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

Grammar (a) 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.*

Example 18: A non-recursive specification for expressions containing the '+' operator would have to be written as

<exp> ::= < term >|< term > + < term > / <term > + < term > - < term >/ ...

Using recursion, <exp> can be specified simply as

<exp> ::= < exp > + < term > | < term > ……..(b)

The first alternative on the RHS of grammar (b) is recursive. It permits an unbounded number of '+' operators in an expression. The second alternative is non- recursive. It provides an 'escape' from recursion while deriving or recognizing expressions according to the grammar.

Recursive rules are classified into *left-recursive rules* and *right-recursive rules* depending on whether the NT being defined appears on the extreme left or extreme right in the recursive rule. For example, all recursive rules of grammar (a) are left-recursive rules.

Indirect recursion occurs when two or more NTs are defined in terms of one another. Such recursion is useful for specifying nested constructs in a language. In grammar (a), the alternative <primary>*::= (*<exp>*)* gives rise to indirect recursion because <exp> =>* <primary>*.* This

Direct recursion is not useful in situations where a limited number of occurrences is required. For example, the recursive specification

< id > *::* = < letter > | < id > [< letter > / < digit >]

permits an identifier string to contain an unbounded number of characters, which is not correct. In such cases, controlled recurrence may be specified as

<id> ::= <letter> {<letter> | <digit>}$_0^{15}$

1.4.1.1 Classification of Grammars

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

Type-0 grammars

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

$$\alpha ::= \beta$$

where both a and P can be strings of Ts and NTs. Such productions permit arbitrary substitution of strings during derivation or reduction, hence they are not relevant to 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

$$\alpha A \beta \ : : = \ \alpha \pi \beta$$

Thus, a string *n* in a sentential form can be replaced by 'A' (or vice versa) only when it is enclosed by the strings a and p. 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 independent 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

$$A ::= tB \mid t \text{ or}$$

A ::= Bt | t

These productions also satisfy the requirements of Type-2 grammars. 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

Example 19: E ::= E + E | E * E | (E) | id

1.4.1.2 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. 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 production in the grammar. 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.

Example 20: Consider the expression grammar

E ::= E + E | E * E | (E) | id

Id ::= a| b| c

Two parse trees exist for the source string a+b*c according to this grammar —one in which a+b is first reduced to *<exp>* and another in which b*c is first reduced to *<exp>.* Since semantic analysis derives the meaning of a string on the basis of its parse tree, clearly two different meanings can be associated with the string.

Eliminating ambiguity

An ambiguous grammar should be rewritten to eliminate ambiguity. 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.

Example 21: E ::= E + E | E * E | (E) | id

After eliminating ambiguity from grammar

E ::= E + T | T
T ::= T * F | F
F ::= (E) | id

1.4.2 Binding and Binding Times

Each program entity pe_i in program P has a set of attributes $A_i = \{a_j\}$ associated with it. If pe_i 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. 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).

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

Binding time is the time at which a 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.

Language implementation time is the time when a language translator is designed. The

language definition of L specifies binding times for the attributes of various entities of a program written in L.

Example 22: Consider the Pascal program

```
program bindings (input, output);
    var
    i : integer; a,b : real;
    procedure proc (x : real; j : integer);
     var
        info : array [1.. 10, 1..5] of integer;
        p : ↑integer;
     begin
        new (p);
    end;
begin
    proc(a, i)
end.
```

- Binding of the keywords of Pascal to their meanings is performed at language definition time. This is how keywords like **program, procedure, begin** and **end** get their meanings. These bindings apply to all programs written in Pascal.
- At language implementation time, the compiler designer performs certain bindings. For example, the size of type 'integer' is bound to *n* bytes where *n* is a number determined by the architecture of the target machine.
- Binding of type attributes of variables is performed at compilation time of program **bindings.**
- The memory addresses of local variables **info** and p of procedure proc are bound at every execution init time of procedure proc.
- The value attributes of variables are bound (possibly more than once) during an execution of proc. The memory address of p↑ is bound when the procedure call **new** (p) is executed.

Importance of binding times

The binding time of an attribute of a program entity determines the manner in which a language processor can handle the use of the 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 later than compilation time. This affects execution efficiency of the target program.

Example 23: Consider the PL/1 program segment

```
procedure pl1_proc (x, j, info_size, columns)
        declare x float;
        declare (j, info_size, columns) fixed;
        declare pl1_info (1: info_size, 1: columns) fixed;
        ….
end pl1_proc
```

Here the size of array pll-info is determined by the values of parameters info-size and columns in a specific call of pl1_proc. This is an instance of execution time binding. The compiler docs not know the size of array pllJ.nf o. Hence it may not be able to generate efficient code for accessing its elements.

The dimension bounds of array info in program bindings are constants. Thus, binding of the dimension bound attributes can be performed at compilation time. This enables the Pascal compiler to generate efficient code to access elements of info. Thus the PL/1 program ‘pl1_info’ may execute slower than the Pascal program ‘binding; for more details). However, the PL/1 program provides greater flexibility to the programmer since the dimension bounds can be specified during program execution.

An early binding provides greater execution efficiency whereas a late binding provides greater flexibility in the writing of a program.

Static and dynamic bindings

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

Dynamic binding*:* A dynamic binding is a binding performed after the execution *of* a program

has begun.

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

1.5 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 analysis are represented in an IR. Thus writing of language processors is a well understood and repetitive process which 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 generation of the analysis phase of language processors.

Figure 20 shows a schematic of an LPDT which generates the analysis phase of a language processor whose source language is L. 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.

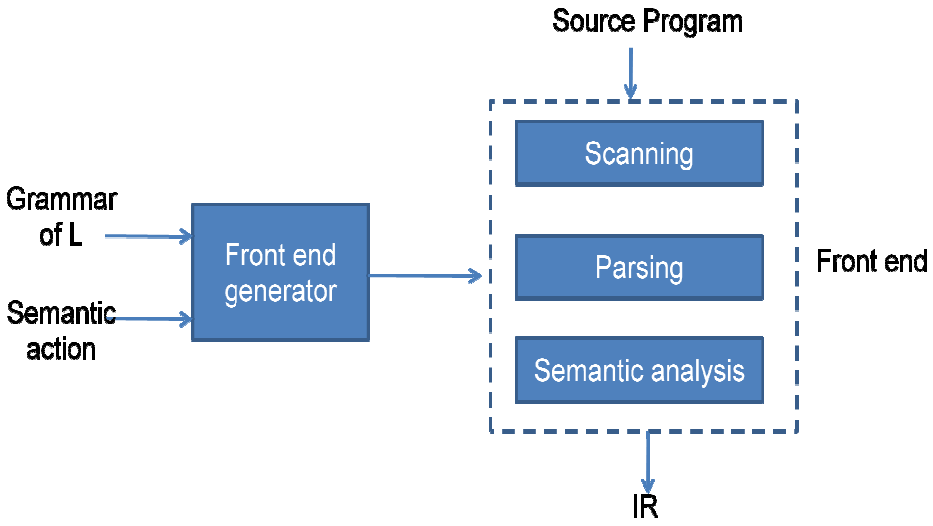


Figure 20: A Language Processor Development Tool (LPDT)

It generates programs that perform lexical, syntax and semantic analysis of the source

program and construct the IR. These programs collectively form the analysis phase of the language processor.

Two LPDTs are 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 which 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 same conventions concerning the representation of tokens. Figure 21 shows a schematic for developing the analysis phase of a compiler for language L using LEX and YACC. The analysis phase processes the source program to build an intermediate representation. A single pass compiler can be built using LEX and YACC if the semantic actions are aimed at generating target code instead of IR. The scanner also generates an intermediate representation of a source program for use by the parser.

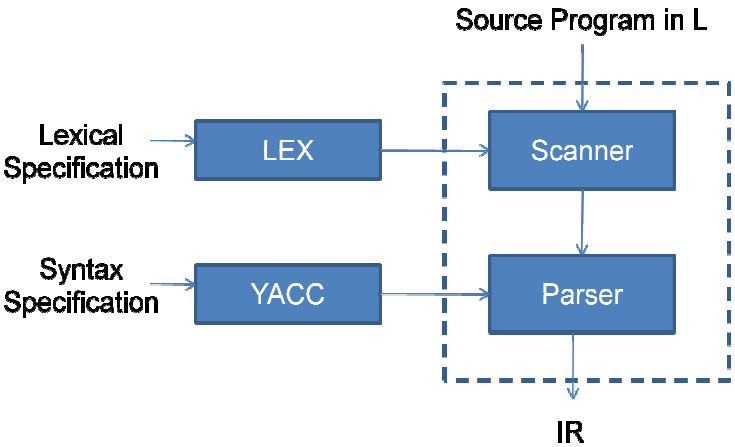


Figure 21: using LEX and YACC

1.5.1. LEX

LEX accepts an input specification which 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.

Example 24: Figure 22 shows a sample input to LEX. The input consists of four components, three of which are shown here. The first component (enclosed by %{ and %}) defines the symbols used in specifying the strings of L. It defines the symbol letter to stand for any upper or lower case letter, and digit to stand for any digit. The second component enclosed between %% and %% contains the translation rules. The third component contains auxiliary routines which can be used in the semantic actions.

```
%{
        letter              [A-Za-z]
        digit               [0-9]
 }%
%%
   begin                    {return(BEGIN);}
   end                      {return(END);}
   {letter} ( {letter}|{digit})*    {yylval = enter_id(); return(ID);}
   {digit}+                 {yylval=enter_num(); return(NUM);}
%%
enter_id() { /*enters id in symbol table*/ }
enter_num() {/* enters number in constabts table */}
```

Figure 22: A sample LEX specification

The sample input in Figure 22 defines the strings begin, end, := (the assignment operator), and identifier and constant strings of L. When an identifier is found, it is entered in the symbol table (if not already present) using the routine enter.id. The pair

(ID, entry *#)* forms the token for the identifier string. By convention *entry #* is put in the global variable yylval, and the class code ID is returned as the value of the call on scanner. Similar actions are taken on finding a constant, the keywords begin and end and the assignment operator.

1.5.2 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 RHS of the string specification. '$$' represents the attribute of the LHS symbol of the string specification.

Example 25: Figure 23 shows sample input to YACC. The input consists of four components, of which only two are shown. It is assumed that the attribute of a symbol resembles the attributes used in Fig. 1.15. The routine gendesc builds a descriptor containing the name and type of an id or constant. The routine gencode takes an operator and the attributes of two operands, generates code and returns with the attribute

```
%%
  E   :     E+T          {$$ = gencode('+', $1, $3);}
      |     T            {$$ = $1;}
  T   :     T*V          {$$ = gencode('*', $1, $3);}
      |     V            {$$ = $1;}
  V   :     id           {$$ = gendesc($1);}
%%
gencode (operator, operand_1, operand_2){ }
gendesc(symbol) {  }
```

Figure 23: A sample YACC specification

Parsing of the* string b+c*d where b, c and d are of type real, using the parser gener-

ated by YACC from the input of Figure 23 leads to following-calls on the C routines:

```
Gendesc (id#1);
Gendesc (id#2);
Gendesc (id#3);
Gencode (*, [c,real], [d, real]);
Gencode (+, [b, real], [t, real]);
```

where an attribute has the f*orm* <name>, <type> and t is the name of a location used to store the result of c*d in the code generated by the first call on gencode.