

Programming Languages

Adesh K. Pandey

1. D. 2000

Programming Languages

Principles and Paradigms

Programming Languages

Principles and Paradigms

Adesh K. Pandey



Narosa Publishing House
New Delhi Chennai Mumbai Kolkata

Programming Languages – Principles and Paradigms
356 pgs. | 2 tbs. | 121 figs.

Adesh K. Pandey
Department of Information Technology
Krishna Institute of Engineering and Technology
13km Mile Stone, Ghaziabd-Meerut Road (NH-58)
Muradnagar, Distt. Ghaziabad, India

Copyright © 2008, Narosa Publishing House Pvt. Ltd.

NAROSA PUBLISHING HOUSE PVT. LTD.

22, Delhi Medical Association Road, Daryaganj, New Delhi 110 002
35-36 Greams Road, Thousand Lights, Chennai 600 006
306 Shiv Centre, D.B.C. Sector 17, K.U. Bazar P.O., Navi Mumbai 400 703
2F-2G Shivam Chambers, 53 Syed Amir Ali Avenue, Kolkata 700 019

www.narosa.com

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without prior written permission of the publisher.

All export rights for this book vest exclusively with Narosa Publishing House.
Unauthorised export is a violation of terms of sale and is subject to legal action.

ISBN 978-81-7319-804-5

Published by N.K. Mehra for Narosa Publishing House,
22, Delhi Medical Association Road, Daryaganj, New Delhi 110 002

Printed in India

*Dedicated
To
My Father*

Late Shri Ramesh Chandra Pandey

Preface

This book is an introduction to the wide field of programming languages. It combines principles and paradigms with considerable detail about many modern languages, including some of the newest imperative, functional, logical and object-oriented languages.

This book focus on courses taught at various universities at undergraduate and postgraduate level on programming language design and paradigm. There are many good books that deal with the subject, but there are few that are suitable for a one-semester graduate level course. This book is an attempt to fill that gap. The goal of this course, and hence of this book, is to expose students to a wide range of programming language paradigms and principles, so that they can understand the literature on programming languages. It should improve the students' appreciation of the art of designing programming languages and, to a limited degree, their skill in programming.

This book does not focus on any one particular language; rather examples from the most widely known imperative languages, including Pascal, C, C++, Java, and Ada are included in the book. Also featured are some of the less widely known languages representing other language paradigms, such as Scheme, ML, LISP, Miranda, and Prolog. I do not suggest that the readers of this book are expert of every programming language used in the book but experience with at least one language is necessary. It is also suggested that certain degree of knowledge about compiler design, computational theory and data structures may help to understand the concepts described in the book in better manner.

This book is written in structured and incremental fashion to increase the readability and understanding of concepts. All the chapters help readers to understand the concept gradually.

ORGANIZATION OF BOOK

Chapter 1 is a survey of the basic concepts including characteristics of programming Languages, evolution of programming language, programming methodologies, and desirable features and design issues.

Chapter 2 and 3 provides the established the relation between the programming languages concepts and theory of computation. Chapter 3 focus on various parsing technique in compiler design.

Chapter 4 gives an overview of elementary data types and their implementation. It includes a section on pointers.

Chapters 5 focus on the concepts of structured data type and their implementation. It explains the implementation with the support of various programming language examples. Chapter 6 gives an overview of the concepts like, abstraction, abstract data type, information hiding and programme hierarchy.

Chapter 7 discuss about the sequence control of expression and statements. It also includes the concepts of concurrent programming, semaphore, monitors, threads and exception handling. Chapter 8 explains the sequence control of subprograms and parameter transmission.

Chapter 9 is dedicated to the memory management concepts like: static allocation, dynamic memory management, garbage collection, dangling reference, and memory leaks. Chapter 10 describes the concepts and issues of object oriented languages.

Chapter 11, 12 and 13 delineates the concept of non imperative paradigm like functional programming languages, data flow languages and logical programming languages. Finally chapter 14 provides the comparative study of the languages from all the paradigms.

The book has selected representative programming languages that display particular programming principles, paradigms and language features clearly. These languages are not all generally available or even widely known.

This book is the result of my continuous working of two years. I am thankful to Almighty God, without whose grace and blessing nothing is possible. I am thankful to my wife and my family for their continuous support. A special acknowledgement to Dr. Ajay Sharma (Director General, K.I.E.T.) for mentioning me to do the things in different manner. I am also very thankful to the publisher of this book.

Suggestion and critical comments towards the improvement of the book are welcome and would be acknowledged gratefully.

Adesh K. Pandey

Contents

Preface

vii

1. Introduction to Programming Languages Concepts	1
1.1 Introduction to Programming Languages	1
1.2 Programming Paradigms	1
1.3 Classification of Programming Languages	5
1.4 Principles of Programming Languages	8
1.5 Characteristics of Programming Languages	10
1.6 Programming Language Design Issues	12
1.7 Factors Influencing the Evaluation of Programming Languages	13
1.8 Language Processing	15
1.9 The Structure of the Program	16
1.10 Introduction of Translators	18
1.11 The Target Architecture of Compiler	20
1.12 Importance of Imperative Languages	21
1.13 The concept of Binding and Binding Time	23
Exercise	25
2. Syntax, Semantic and Translation	26
2.1 Syntax Vs Semantics of the Languages	26
2.2 Semantics, The Meaning of the Program	26
2.3 Grammars of Programming Languages	28
2.4 The Syntax of a Simple Imperative Language	45
2.5 Introduction to Regular Expression	48
2.6 Deterministic Finite Automata	52
2.7 Why Study Compilers	58
Exercise	70
3. Basic Parsing Techniques	72
3.1 Introduction of Parsers	72
3.2 Top-down-parsing	73

3.3 Recursive-Decent Parsing	82
3.4 Top-down Predictive Parsers	86
3.5 LL(1) Grammars	91
3.6 Introduction to Bottom-up Parsing	95
<i>Exercise</i>	102
4. Elementry Data Types	104
4.1 Introduction To Data	104
4.2 Anonymous Data Items	112
4.3 Renaming (Aliasing)	113
4.4 Concept of Overloading	113
4.5 Type Checking	114
4.6 Primitive Data Types	116
<i>Exercise</i>	124
5. Structured Data Types	126
5.1 Introduction to Structured Data Types	126
5.2 Arrays and Their Declaration	127
5.3 Enumerated Types	140
5.4 Character Strings	142
5.5 Records and Structures	144
<i>Exercise</i>	152
6. Encapsulation	153
6.1 Evolution of Data Type Concept	153
6.2 The Concept of Abstraction	153
6.3 Information Hiding	154
6.4 Programme Hierarchy	155
6.5 Encapsulation by Subprogram	167
<i>Exercise</i>	168
7. Sequence Control	169
7.1 Introduction to Sequence Control	169
7.2 Types of Sequence Control	169
7.3 Expression Sequence Control	170
7.4 Sequence Control between Statement	174
7.5 Iterative Statements	179
7.6 Concurrent Programming	186
7.7 Exception Handling	196
<i>Exercise</i>	203

8. Sequence Control of Subprogram	205
8.1 Introduction	205
8.2 Activation of the Procedure and Activation Record	205
8.3 Stack and Subprogram Calls	206
8.4 Handling Procedure Calls and Retruns	210
8.5 Implementation of Call-Return of Subprograms	213
8.6 Refrencing Environment	214
8.7 Access to Non-local Names	216
8.8 The Block Structure:	217
8.9 Implementation of Dynamic Scoping	219
8.10 Parameter Transmission	222
<i>Exercise</i>	225
9. Memory Management	227
9.1 Introduction To Memory Management	227
9.2 Major Run-Time Elements Requiring Storage	227
9.3 Sub-Division of Run-Time Memory	228
9.4 Memory-Allocation Strategies	230
9.5 Reclamation of Free Storage	234
9.6 Fixed and Variable-Size Element	241
9.7 Recovery in Case of Variable-Size Blocks	242
9.8 Memory Management in Various Language	242
<i>Exercise</i>	246
10. Object Oriented Languages	248
10.1 Introduction	248
10.2 Object-Oriented Languages	251
10.3 Concept of Abstraction	251
10.4 Concept of Object-Oriented Programming	253
10.5 C++	259
10.6 Simula	262
10.7 Small Talk	262
10.8 Module-3	263
10.9 Eiffel	263
10.10 Java	264
<i>Exercise</i>	265
11. Functional Programming Languages	266
11.1 Introduction to Declarative Languages	266
11.2 Functional Programming	266
11.3 Mathmatical Functions	268

11.4 Referential Transparent	268
11.5 Type System of Function Programming Languages	269
11.6 Names, Binding, Environments and Scope of FPLs	271
11.7 Some FPLs	272
11.8 Concepts of FPLs	272
11.9 Application of Functional Language	276
11.10 The Lambda Calculus	276
<i>Exercise</i>	285
12. Logic Programming Languages	287
12.1 Introduction to Logic Programming	287
12.2 Introduction to Predicate Logic	288
12.3 Rules for Constructing Well-formed Formula	289
12.4 Transcribing English to Predicate Logic WFPS	290
12.5 Definite Clauses	291
12.6 SLD-Resolution	295
12.7 Negative Information	299
<i>Exercise</i>	301
13. Data-flow Languages	302
13.1 Introduction to Data-flow Languages	302
13.2 Data-flow Computers	302
13.3 Features of Data-flow Languages	303
13.4 TDEL (Textual Data-flow Languages)	303
13.5 LAU	304
13.6 LAPSE	305
13.7 Real-Time System	305
<i>Exercise</i>	310
14. Programming Languages from Different Paradigms	311
14.1 Java	311
14.2 ML	318
14.3 LISP	320
14.4 Ada	327
14.5 Prolog	333
<i>Index</i>	339

CHAPTER 1

Introduction to Programming Languages Concepts

1.1 INTRODUCTION TO PROGRAMMING LANGUAGES

The concept of programming languages has been taken from the natural languages.

Like natural languages (Hindi; English, ...) programming languages are also notations, with which common man can communicate to the computers.

"According to Nico Habermann "An ideal language allows us to express what is useful for the programming task and the same time makes it difficult to write what leads to incorrect programs"

"According to Robert Harper "Good language makes it easier to establish, verify and maintain the relationship between code and its properties".

We can study programming language in two domains

- (i) Paradigm of programming languages
- (ii) Principles of programming languages

1.2 PROGRAMMING PARADIGMS

A programming paradigm is a general approach to programming or to the solution of problems using a programming language. Thus programming languages that share similar characteristics are clustered together in the same paradigm.

A programming paradigm is both a method of problem solving and an approach to programming language design. A distinction can also be made between a programming language, and a programming environment. Often programming in language X may be considered much easier than in language Y simply because X has a much richer programming environment. This environment has two important aspects First, the user interface editor, compiler, debuggers and other tools provided to help the user to develop the programs. Second, the libraries of procedures available. There is also an issue of reusability and ease of library functions.

A distinction can be made between symbolic and non-symbolic (or numerical) programming. In the early history of computers, it was thought that all one could do was

2 Programming Languages-Principles and Paradigms

various kinds of calculation, and indeed this was the main use of computers. Thus language such as FORTRAN were largely used for performing complex calculations. In time, characters and strings became data types that could be manipulated within conventional programming languages. On the other hand symbolic programming languages (such as LISP and PROLOG) were expressly developed to manipulate abstract symbols, and in particular to do list processing. Any programs which require some form of "understanding" reasoning or complex interpretation need to do symbolic processing and are therefore normally best done using a language such as LISP or PROLOG. This is why LISP and PROLOG are main languages used in AI.

The major paradigms are".

- (i) Imperative Languages/Procedural Languages
- (ii) Object - Oriented Languages
- (iii) Functional Programming Languages
- (iv) Logical Programming Languages
- (v) Rule - based Programming Languages
- (vi) Data base query languages
- (vii) Visual programming languages
- (viii) Scripts programming languages
- (ix) Programming by Demonstration.

Other programming paradigm include simulation (that is SIMULA) and spread sheets. This Book will concentrate on just three paradigm:

- Imperative programming languages
- Functional programming languages
- Logic programming languages

1.2.1 Imperative/Procedural Programming

Procedural programming is by far the most common form of programming. A program is a series of instructions which operate on variables. It is also known as imperative programming. Procedural programming bears a close relation to the Von Neumann form of computer architecture, and early porcedural languages were little more complex than assemblers.

Examples of Procedural programming languages include FORTRAN, ALGOL, pascal, C, MODULA 2, Ada, BASIC. Despite their differences they all share the common characteristics of procedural programming.

As a method of design, procedural programming attempts to encapsulate the human problem solving method of carrying out a sequence of operations. First, one carries out step 1, then step 2, etc. In addition there are control structures such as IF-THEN-ELSE in all modern procedural languages. Variables play a central role in programs, and their scope is an important notion in block structured languages.

Advantages of procedural programming include its relative simplicity, and ease of implementation of Compilers and iterpreters. Procedural languages were important in the history of computers due to their low memory utilisation. Dijkstra believed that the introduction of micro computers set back the development of computer science. This was in part due to the fact that the small memory microcomputers could only run languages like BASIC with 8K byte

interpreters, and languages that were run on main frame and micro computers (such as LISP) that needed larger memory could not be implemented on the micros. All this has now changed with larger memory micros, but history has never the less affected the market place and people's perception of programming languages.

Disadvantages of procedural programming include the difficulties of reasoning about programs and to some degree difficulty of parallelisation. Procedural programming tends to be relatively low level compared to some other paradigms, and as a result can be very much less productive.

1.2.2 Object Oriented Programming

Object oriented porgramming was introduced by Xerox with the language small talk. This was intended to be a salution to the software crisis by providing an end-user programming language. However, small talk has a complex syntax and has not found to be easy to use, and certainty is not realistic for end-user.

At the same time as small talk was developed another part of Xerox developed an object system within LISP, known as LOOPS. This was the basis of the more recent CLOS (Common LISP object system).

The most widely used object oriented language is C++ which provides object extension to C, but this is rapidly being overtaken by Java.

Object oriented programming is characterised by the defining of classes of object, and their properties. Inheritance of properties is one way of reducing the amount of programming, and provision of class libraries in the programming environment can also reduce the effort required.

1.2.3 Functional Programming

Functional programming is based upon the notion of a program as a function in similar sense to its usage in mathematics. Programs are designed by the composition of functions.

The earliest functional language is LISP which was developed by John Mc Arthy at MIT in the late 1950s. This was based on Lambda calculus. However, early in the development of LISP non-functional elements were entroduced and it is possible to design programs in LISP in a procedural style. LISP stands for LISP processing, and it is the main language used for list processing applications.

Among other interesting qualities of LISP is the ability to compile functions at run time. LISP programs are themselves data structures which can be manipulated by LISP, and therefore it is straight forward to write programs which changes them selves as they run. It is also possible to design and compile programs to down without any need for lower level functions to be designed earlier. Since the modification of the design of a single function has no side effects of other functions, this makes modular programming straight forward. LISP is also widely used to design new programming languages. Due to LISP's pre-eminence in the development of AI programs, it has always had sophisticated programming environment. Symbolics Inc. developed a range of computers whose CPUs were specifically designed to execute LISP efficiently, and their system had a button on the keyboard marked DWIM (Do what I Mean). Thus a user can expect the LISP system to in some sense understand his intentions, and this shows something of the possible sophistication. LISP system were also one of the first to develop interface editors that allowed a drawing type of interface, and automatic construction of code. Other functional languages includes SCHEME, HOPE and ML.

The major advantages of functional programming are the programs can be easy to understand and to formally reason about, and that functions are very reusable. However, as with other language it is also possible to design functional programs that are incomprehensible. It is possible to develop and maintain very large programs consisting of thousand of functions, because functions have no side effect and it is therefore straight forward to fully test functions and the resulting system.

The major disadvantage of functional programming is the difficulty of doing input-output since this is inherently non-functional. There are also other aspects of problem solving that cannot easily or sensibly be performed in a functional manner. Nevertheless large programs can be developed with about 80% of the code being designed purely functionally.

1.2.4 Rule - Based Programming

Rule based programming is also known as IF-THEN programming and as expert system design. It shares many similarities with logic based programming, and in principle is simply a variant of it. Nevertheless, practical rule-based systems are very different in their design and usage to PROLOG.

Rule-based programming has its origin in the very first electronic computer; Rule based system are also known sometimes as post production system and post was probably the first developer of the idea.

Like logic programming the designer need not worry about the execution details of the program. These are left to the inference engine of the expert system to sort out. Thus this raises the level of abstraction of the programming.

Part of the development of rule-based programming methods has come from attempting to model the problem solving behaviour of human experts. The idea has been that if the expert's thinking can be suitably captured and conceptualised in the expert system, then it will be both easy to design such systems, and to use them to perform the task correctly. Expertise is primarily encapsulated in the notion of IF-THEN rules.

Rule-based systems are often used in the development of systems at the outset of the work there is no clear idea about exactly how the task is performed. But by detailed analysis of the expert's behaviour in solving problems and much testing it is possible to reveal the underlying logic in the expert's behaviour. It is partly because rule-based systems are excellent at rapid prototyping that they are used in this way. Arguments that the development of rule-based systems is very slow should consider the fact that this is due to the initial ignorance of the task domain, and that if the development was done in say a procedural language the development time would be even longer.

In recent years Schank and others have argued that experts do not solve problems using rules but using cases. Schank makes a convincing argument, for example in relation to medical and legal reasoning. This has led to the development of Case Based Reasoning systems.

A major advantage of rule-based programming is that prototypes can often be developed very quickly. Some rule-based languages are also easy to learn.

The main disadvantages of rule-based systems include slowness of execution and the difficulties of debugging. More advanced rule-based systems include the ability to automatically generate procedural code (typically in C) which performs the same function as the expert system. This usually solves the execution speed problem.

1.2.5 Data Base Query Languages

Data base query languages allow the development of application on data bases. Language include DBASE, ORACLE, ACCESS, PARADOX .

The major advantages are the ease of use.

Disadvantages are execution speed and in some cases limited processing capabilities. Naturally they are limited to applications which involve the querying data bases.

1.2.6 Visual Programming

There are two different kinds of visual programming. The first and more common is a visual programming environment that is an adjunct to an existing programming paradigm.

Examples include Visual BASIC and Visual C++. The second kind of visual programming sees program construction as a visual task similar to the solving of jig saw puzzles. The user manipulates program shapes in a graphical interface.

1.2.7 Script Based Programming

Script based programming uses a direct representation of user's actions as a script. Thus the programming language is designed in a more expressive manner to other porgramming languages. Furthermore, a program is constructed to correspond directly to the operations that a user would execute. Thus programs can often be constructed by means of recording the sequence of actions done by the user. However, such recording are literal and invalve no attempt at generalisation (unlike programming by demonstration).

Example include Hyper Talk, the scripting language of Hypercard, and its derivaties in other system.

Other example would be apple script which works at the level of the operating system.

1.2.8 Programming by Demonstration

Programming by Demonstration (PBD) is the most recent approach to programming and is largely only used as present within research laboratories and in limited application such as drawing. The essential notion is that the user demonstrates a program to the system by working through one or more examples.

The system then generalises the user's sequence of actions to porduce a general program. One of the best known such systems is cyphir's EAGFR system which can automatically detect repetitions in user's action and build a pogram to perform them in future.

1.3 CLASSIFICATION OF PROGRAMMING LANGUAGES

To facilitate discussion of any subject it is desirable and convenient to group together similar facts of the subject according to some grouping notation. Computer programming languages are not an exception to this, however, there are many categorizations that we can adopt. The most common are as follows:

6 Programming Languages-Principles and Paradigms

1.3.1 Machine, Assembly and High Level Languages

Before discussing this classification it is appropriate to remind ourselves, the operation of the computer program. A computer program resides in the primary memory where it is represented as set of machine instructions which in turn are represented as sequence of binary digits. At any point in time the computer is said to be in a particular state. A central feature of the state is the instruction pointer which points to the next machine instruction to be executed. The execution sequence of a group of machine instructions is known as the flow of control.

Machine Code is a program running on a computer is simply a sequence of bits. A program in this format is said to be machine code. We can write programs in machine code:

2	3	f	c	0000	0001	0000	0040
0	c	b	g	0000	0009	0000	0040
6	e	o	c				
0	6	b	g	0000	0001	0000	0040
6	0	e	8				

Assembly Language was our first attempt at producing a mechanism for writing programs that was more palatable to our selves. Of course a program written code, in order to "run" must first be translated (assembled) into machine code. Let us see an assembly language program, which prints "Hello assembly":

```
data segment
    msg db "Hello assembly $"
data ends
code segment
    Assume cs: Code, c/s: data
start proc far
    mov Ax, data
    mov Ds, Ax
    mov Ah, 0gh
    mov Dx, offset msg
    Int 21 h
    mov Ax, 4 cooh
    Int 21 h
    start end p
code ends
end start
```

High Level Languages are introduced to overcome the difficulties of machine and assembly languages. The advent of higher level languages, commencing with the introduction of "Auto codes" and going on to Algol, Fortran, Pascal, Basic, Ada, C. Etc.

1.3.2 Chronological Classification of Programming Languages

In this type of classification, we can see the phase to phase development of programming languages.

1940s

Prelingual phase: Machine code is the example.

1950s

Exploiting machine power: Assemble code, Autocodes, first version of Fortran 1960s. are the examples.

Increasing expressive power: Cobal, Lisp, Algol 60, Basic, PL/I are the examples.

1970s

Fighting the "software crisis": The phrase software crisis alludes to set of problems encountered in the development of computer software during the 1960s when attempting to build large and larger software system using existing development technique. As a result:

1. Schedule and cost estimates were often grossly inaccurate.
2. Productivity of programming could not keep up with demand.
3. Poor quality software was produced.

To address above Problems the discipline of software engineering came into picture are as follows:

- (a) *Structured programming* concept devised in late 1960s (Dijkstra and others). Emphasizes programming using sequence, condition and repetition (no jumps and "go to" statement). Constructs which have a predictable logic, that is, they are entered at the top and exited at the bottom. Structured programming enhances readability and hence maintainability.
- (b) *Modular programming* concept was first taken seriously in the early 1970s. Modular programming is concerned with sub-devision of programs into manageable "chunks". The concept also encompasses ideas about levels of abstraction-modules are usually arranged in a hierarchy of abstraction. Modularity also espouses the concept of information hiding.
- (c) *Information Hiding* espouses the idea that the attention of programmers should not be distracted by details not central to their current programming task, that is programmers should not require to be concerned with unnecessary details. In terms of modular programming modules should be designed so that information contained within those modules is "hidden" from other modules, that is implementation details of functions.

Examples of this phase are pascal, Algol 68 and C

1980s

Reducing complexity: Object orientation, functional programming.

1990s

Exploiting Parallel and distributed hardware: Various parallel extension to existing languages and dedicated parallel language such as occam were developed in this phase.

8 Programming Languages—Principles and Paradigms

2000s

Genetic Programming languages, DNA computing and bio-computing are focus area in this era.

1.3.3 Language Generation

The classification of generations are as follows:

Generation	Classification
1st	Machine languages
2nd	Assembly languages
3rd	Procedural languages
5th	AI technique, inference languages
6th	Neural network

1.3.4 Language Levels of Abstraction

According to the abstraction level classification is as follows:

Level	Instructions	Memory handling
Low level languages	Simple machine-like instructions	Direct memory access and allocation
High level languages	Expression and explicit flow of control	Memory access and allocation through operations
Very high level languages	Fully abstract machine	Fully hidden memory access and automatic allocation

1.3.5 Declarative v Non-declarative Programming

Languages can also be classified by the emphasis they put on "what is to be achieved" against "how it is to be achieved". The first are said to be declarative (that is functional and logic languages). The second is said to be non-declarative (that is imperative languages).

1.4 PRINCIPLES OF PROGRAMMING LANGUAGES

Like any other language programming languages are essential communication media:

- (i) human to machine
- (ii) machine to machine, and
- (iii) machine to machine.

The principles of programming languages should be defined in such a manner, so that languages can be better tool for above discussed communications. Let us discuss some of the principles as follows:

1. Abstraction:

The principles of programming languages should be able to maintain the level of abstraction so that languages may factor out the recurring patterns and avoid requiring something to be stated more than once.

2. Automation:

Language should be designed in such a manner, So that they may be able to automate mechanical, tedious or error-prone activities.

3. Defence in Depth:

Language should have the facility to caught the errors as early as possible. If an error gets through one line of defence, then it should be caught by the next line of defence.

4. Information hiding:

Modules should be designed so that:

- (a) The user has all the information needed to use the module correctly, and nothing more.
- (b) The implementor has all the information needed to implement the module correctly, and nothing more.

5. Labeling:

The design of programming languages should be so effecient so that we should not require the user to know the absolute position of an item in a list. Instead, we should associate labels with any position that most be refrenced else where.

6. Localized Cost:

The design principles of porgramming language should allowe to user to only pay for what he uses; avoid distributed costs.

7. Manifest Interface:

All interfaces must be apparent (manifest) in the syntax that is clear to use without any side-effects.

8. Orthogonality:

The design principles should avoid dependencies, that is independent functions should be controlled by independent mechanism. Logic and control of any function should not be dependent.

9. Portability:

The design principles should increase the portability of programming languages. we should avoid features or facilities that are dependent on a particular machine or a small class of machines.

10. Preservation of Information:

The language should allow the representation of information that the user knows and that the compiler will need.

11. Regularity:

The design principles of programming languages should based on unambiguous regular rules which are:

- (a) Without exceptions
- (b) Easier to learn
- (c) Easier to use
- (d) Easier to describe and
- (e) Easy to implement.

12. Security:

No program that violates the definition of the language, or its own intended structure, should escape detection.

13. Simplicity:

A language should be as simple as possible. There should be a minimum number of concepts with simple rules for their combination.

14. Syntactic Consistency:

Things which look similar should be similar and things which look different should be different.

1.5 CHARACTERISTICS OF PROGRAMMING LANGUAGES

We have seen in previous sections that there exist various paradigm and languages. Programmers generally prefers to use one language over to other due to its attributes and as per their need.

A good programming language should have following attributes as language design criterion:

1. Simplicity of Languages:

According to great Einstein

"Everything should be as simple as possible, but not simpler".

Above statement fits of programming language design, language should be easy to master. A language should provide a simple framework to express and program a given algorithm. Development of new algorithm requires clear, simple and unified set of concepts, so language design should be able to fulfill it.

2. Uniformity:

Statements with same syntax in any Programming language should have same semantics, in other words language design must support unambiguity to a large extent.

3. Expressiveness:

The programming language should be easy to express, so that it may provide user friendly style of writing program. Expressiveness of any programming language depends on the level of abstraction.

Finally success of any programming language depends on its application towards industry.

4. Orthogonality and Generality:

The programming language should support maximum independence. Orthogonality refers to the attribute of being able to combine various features of a language in all possible combinations, with every combination being meaningful.

The language should support maximum generality that is no special restrictions should be imposed.

For example small talk, Eiffel, other pure object-oriented languages do it well.

5. *Clear, Unambiguous Syntactic and Semantic Description:*

The syntactic and semantic structure of any programming language should be clear and unambiguous. Ambiguity may confuse the translator and same syntactic structure may landup with different semantics (meanings).

6. *Readability – Modifiability and maintenance:*

Hoare consider readability more important than writeability. The design of programming language should support maximum readability. Readability discouraged syntactical and logical (semantic) errors. By increasing readability these errors also can be discovered very soon and without any difficulty.

Readability decides the measure of the quality of the program and programming language. If a programming language is fairly readable then modifications in softwares, which are programmed by using that particular programming language, are simple and costeffective. Maintenance is also easy and cost effective for a good readable programming language.

7. *Documentation:*

Documentation is another aspect through which a programming language can be hit in the market. A programming language should be well-documented and should have it's own commenting conventions. Programming languages must have a well-documented and easy to read manual, so that programmers can understand the language with in short time.

8. *Writability:*

Writability is a measure of how easily a language can be used to create programs for a chosen problem domain. A programming language should have less number of effective constructs so programmars may be familiar with all of them.

Programming language support combining of smaller number of primitive constructs and a consistent set of rules to find out large number of primitives.

9. *Abstraction support:*

Abstraction is the key factor of modern programming language, which allows to use complicated structures and operation without knowing much detail about them.

By the support of abstraction attribute programming languages may be factor out the recurring patterns and avoid requiring something to be stated more than once.

Languages like C++ and Java support high level of abstraction.

10. *Reliability:*

Reliability is very much related to writeability, readability and modifiability. If a language is reliable then it should discoursed and easily discovered the syntactic and logical (semantic) errors.

A programming language is said to be more reliable if it supports strict type checking and exception handling technique.

For example the current version of C language support type checking as it type checked to all the parameters.

Language like Ada C++, Java intercept run-time errors by using the exception handling technique.

12 Programming Languages–Principles and Paradigms

11. Portability:

Portability of programming languages is another area of concern. The good programming language should support transportability of programs from the computers on which they are developed to other computer systems.

Languages like C++ and Java support this concept efficiently, since they have standardized definitions allowing for portable programs to be implemented.

12. Simplicity of compiler Implementation:

A good programming language is always simple for compiler implementation.

According to Wirth:

"A language that is simple to parse for the compiler is also simple to parse for human programmer, and that can only be an asset"

13. Machine Independence:

Machine Independence is very important feature of modern programming languages and this is most concerned area.

It is very hard to achieve completely machine independence.

For example language like Java is said to be machine independent to some extent.

14. Cost of Use

Cost of factor plays an important role in the evaluation of programming languages. Cost factor may influence in several ways. Let us discuss some of them.

- (i) It can be very easily understood by our knowledge of software engineering that, it is not easy task to develop the large software, which are cost effective. So always programmer wants to use the language, which can minimize the cost of program creation testing and use.
- (ii) We have observed, during the compilation of C program that it required to compile the program many times to debug it but we only execute the program once. So clearly we need fast compiler to minimize the cost of translation.
- (iii) We know that space and time complexity always matter in computer science. So execution should be faster. Cost of program execution can be optimize by using effective register-allocation technique and by providing run-time support mechanism.
- (iv) As we used software programs over the years, they may require some maintenance due to some errors, hardware specification changes and user's need. So always design of programming language should support easy maintenance or optimize the maintenance cost.

1.6 PROGRAMMING LANGUAGE DESIGN ISSUES

It is very challenging task to develop a new programming language, so designer of programming languages must be very careful about the design of language and its application towards industry.

Following are some important issues, which must be addressed during the designing process:

- (i) Programming languages must solve the problems, without causing new ones.
- (ii) The design of programming languages should be simple and efficiently implemented
- (iii) The language should have carefully specify semantics
- (iv) Design should be especially wary of new features
- (v) Design of programming languages should be ready to make modifications to solve minor problems.
- (vi) Design of programming languages should support it's implementation on several machines.
- (vii) Languages should be prepared to sell it to customers.
- (viii) Design of programming languages should not include untried ideas.
- (ix) Programming languages must have written manuals and texts.
- (x) Design group of programming languages should be as small as possible.
- (xi) Design group should not afraid to revise the design since, once distributed, too hard to change.
- (xii) Syntax of programming languages should increase the readability.

1.7 FACTORS INFLUENCING THE EVALUTION OF PROGRAMMING LANGUAGES

We have seen that designing of programming languages is very innovative task and it is influenced by several factors but computer architecture and design methodologies are the two most important factors.

1.7.1 Computer Architecture

In 1945, John Von Neuman developed two important concepts that directly affected the path of computer programming languages. The first was known as "shared program technique". This technique stated that the actual computer hardware should be simple and not need to be hand-wired for each program. Instead computer instruction should be used to control the hardware allowing it to be reprogrammed much faster.

The second concept was extremely important to development of programming languages. Von Neumann called it "conditional control transfer". This idea gave rise to the notion of subroutines, or small block of code that could be jumped to in any order, instead of a single set of chronological order steps for the computer to take. The central idea of Von Nuaman architecture was that "Both data and programs are stored in the same mamory. The central processing unit (CPU), which actually executes instruction, is seprate from the memory." *This stucture is mainly responsible for evolution of imperative languages.*

The overall strcuture of a von Neumann Computer is shown in Fig. 1.1.

Iteration was very fast on this achitecture because instruction are stored in adjacent cells of memory.

Functional languages mainly works without assignment statement and without iteration, but functional languages supports computation by passed parameters. Parallel architecture was the next, that have appeared over the past 20 years hold same promise for speeding the execution of functional programs.

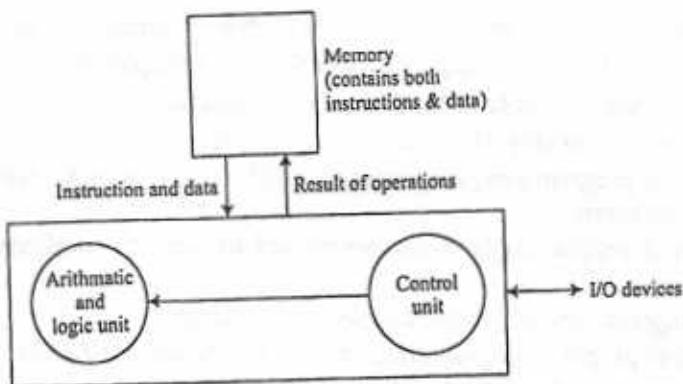


Fig. 1.1

1.7.2 Programming Methodologies

The main task of developing the new language was to increase the productivity of programmers and to decrease the cost of hardware. New programming methodologies also plays important role in the development of programming languages. Let us discuss the some of the programming methodologies, which are mainly responsible for the development of new programming languages:

(i) Structured Programming:

The 1960s saw remarkable development in the field of electronics with the result that more and more powerful hardware components could be produced more and more cheaply. In parallel it was assumed that it would similarly be possible to construct ever large and more complex software. This assumption proved to be quite wrong - programs failed to be ready on time greatly exceeded their budget, contained many errors and did not fulfill costromers expectations. This phenomenon became known as the *software crisis*.

Among the reason for crisis poor project management, and fact that many programmers considered their porgrams to be their property. Thus many individual and curiou porgramming styles developed, and it proved difficult to create error-free programs. In order to remedy this, the concept of structured porgramming was devised in the late 1960s (by Dijkstra and others). The aim of structured programming is to create programs that are:

- Understandable (and hence maintainable), and
- So far as possible, error free.

Structured programming can be said to be set of rules and recommendations for how "good" programs should be written. It emphasises programming using sequences, condition and repetition (not jumps and goto statement). These are cell constructs which have predictable logic, that is they are entered at the top and exited at the bottom. The introduction of the concept of structured programming quickly indicated this need that initiated the development of what became known as *structured* language such as C and pascal.

(ii) Modular Programming:

During the 1970s it became clear that even well-structured programs were not enough for mastering the complexity involved in developing a large program system. It was al-

recognised that it was necessary to support the division of the program into well-defined parts or modules, that could be developed and tested independently of one another, so the several people work together within one large programming object modular programming is thus concerned with the subdivision of programs into manageable "chunks". The concept encompasses ideas about level of abstraction modules are usually arranged in a hierarchy of abstraction. It also describes that information hiding modules should be designed so that information contained within a module is not accessible to other modules that have no need for such information.

(iii) *Object-Oriented methodology:*

It begins in the early 1980s. Object-oriented methodology begins with data abstraction, information hiding, inheritance and dynamic method binding. Java, C++, Ada 95 and prolog++ are some examples of object-oriented programming languages.

(iv) *Concurrency:*

Till the concept of concurrency was developed, the traditional computer follow the concept of one processor one piece of data at a time, in other words single instruction single data (SISD). But now a days concurrency supports single instruction and multiple data (SIMD) concepts. The most widely used SIMD machines are category of machines called *vector processor*.

We can view concurrency at following levels of programming:

- (a) Instruction level (Two or more instructions are executed simultaneously)
- (b) Statement level (Two or more statements are executed simultaneously)
- (c) Subprogram level (Two or more subprograms executed simultaneously)
- (d) Program level (Two or more programs are executed simultaneously).

Java threads and C# threads are examples concurrent programming.

1.8 LANGUAGE PROCESSING

The machine language of computer is its set of macroinstruction and it is not easy to write the program by using machine languages without support of any other software.

The most more practical processing of programming languages is through interface programs in higher-level languages, which works as interface between high-level languages and machine language.

A language implementation system can not be only software or computer but also required large collection of utility programs, called operating system. Operating system provides higher-level primitives for various jobs like resource management, file management, editors, and input-output operations.

Programming language translation can be viewed in two ways:

(i) *Hardware Computers:*

We know that computers generally have a low-level machine language and when the target language of any translation is only machine language we are realizing a computer in hardware only. In other words, in this view a high-level language is directly translated to machine language.

(ii) *Software simulated computer:*

We can also view a computer which have another high-level language as its machine language and it is called software simulated computer. In other words theoretically it is possible to design a computer with a particular high-level language as its machine language, but it would be very complex expensive and highly inflexible (because it would be difficult to use it with other high-level languages).

To understand the translation, we must know the structure of the program.

1.9 THE STRUCTURE OF THE PROGRAM

We can analyse a computer program on 4 levels:

1. Lexical Structure Level
2. Syntactic Structure Level
3. Contextual Structure Level
4. Semantic structure Level

1.9.1 Lexical Structure

The lexical level of any program is lowest level. At this level computer programs are viewed as simple sequence of lexical items called tokens. In fact at this level programs are considered as different groups of strings that makes sense. Such a group is called a token. A token may be composed of single character or a sequence of characters.

We can classify tokens as being either:

Identifier:-

In a particular language names chosen to represent data items, functions and procedures, etc are called identifiers. Considerations for the identifier may differ from language to language, for example, some computer language may support case sensitivity and others may not. Number of characters in identifiers are also depend on the design of the computer language.

Keywords:-

Keywords are the names chosen by the language designer to represent facts of particular language constructs which can not be used as identifiers. (sometimes referred to as reserved words).

Let us see example of language C

```
int id;
here id is identifier.
struct id1 {
    int a;
    char b;
}
```

here struct is keyword and id1 is identifier.

Operators:-

In some language, special "keywords" used to identify operations to be performed on operands, that is math operators.

For Example in language C:

- (a) Arithmathematical operators (+, -, *, /, %)
- (b) Logical operators (>, = >, <, <=, ==, !=)

Separators:-

These are punctuation marks used to group together sequences of tokens that have a "unit" meaning.

When outputting text it is often desirable to include punctuation, where these are also used (within the language) as separators we must precede the punctuation character with what is called an *escape character* (usually a back slash '\')

Literals:-

Some languages support literals which denote direct values, can be

- Number that is 1, -123, 3.14, 6.02 e23.
- Characters that is 'a'
- string, that is "some text"

Comments:-

We know that a good program is one that is understandable. We can increase understandability by including meaningful comments into our code.

Comments are omitted during processing. The start of a comment is typically indicated by a key word (that is comment) or a separator, and may also be ended by a separator.

1.9.2 Syntactic Structure

The syntactic level describes the way that program statements are constructed from tokens. This is always very precisely defined in terms of a context free grammars.

The best known examples are BNF (Backus Naur form) or EBNF (Extended Backus Naur Form). Syntax may also be described using a syntax tree.

Let us see an EBNF example:-

```
(sum)      :: = <operand>
              <operator>
              <operand>
<operand> :: = <number/ (<sum>)
<operator> :: = + / -
```

The syntax tree for above EBNF will be as in Fig. 1.2.

1.9.3 Contextual Structure

The contextual level of analysis is concerned with the "context" in which program statements occur. Program statements usually contain identifiers whose value is dictated by earlier statements (especially in the case of the imperative or object oriented paradigms).

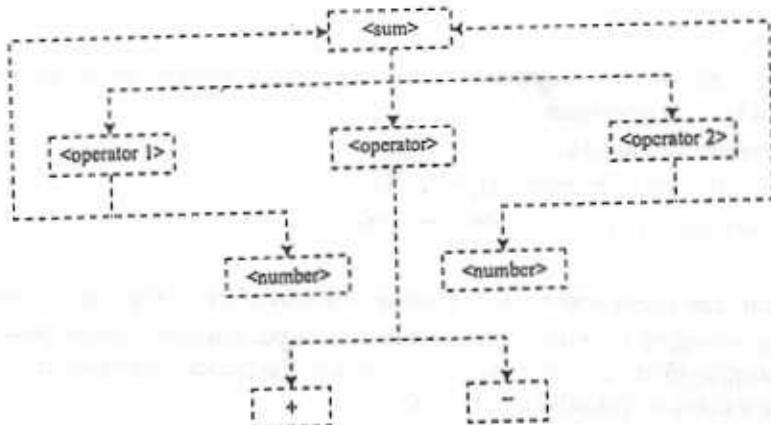


Fig. 1.2

Consequently the meaning of a statement is dependent on "what has gone before", that its context. Context also determines whether a statement is "legal" or not (context conditions - a data item must "exist before it can be used").

Let us see an example of language C.

```

int C = a + b; } not valid, since a and b are not defined
int a;
int b;
int C = a + b; } This is valid.
  
```

1.10 INTRODUCTION OF TRANSLATORS

Let us consider the translation from French to English. When translating from French to English, the translator must comprehend each French sentence and produce a corresponding sentence in English. Human translator receives a text that is written in the source language (here it is French) and he decomposes it in sentences and word using his knowledge of the language. This "knowledge" is stored in his "internal memory", that is his brain, and it consists of the vocabulary of the language (used to recognize the words) and the grammar of the language (used to analyze the correctness of the sentences). Once the translator recognizes all the words and understands the meaning of the sentence, he starts the process of translating. For any translator, the most difficult part comes from the fact that he or she can not translate word by word because the two languages not have the same sentence patterns, and in spite of this the meaning of the sentence in both languages must be the same.

A translator in computer science follows same sequence of operation that is performed by a human translator. Fortunately for the construction of a translators, for the computer languages are far similar than any spoken language, hence the task of writing a program that translates from source language to the target language is by far more simple than translating from French to English. So finally we can define translators as follows:

"Translators are the software programs which can translate programs written in to source language to target language".

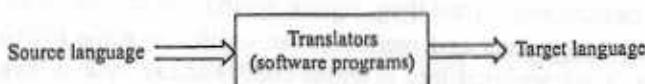


Fig. 1.3

Let us discuss some translators, such as Assemblers interpreters, and compilers.

1.10.1 Assembler

An Assembler is translator for an assembly language of a computer.

An assembly language is the lowest level programming language for a computer. It is peculiar to a certain computer system and is hence machine-dependent.

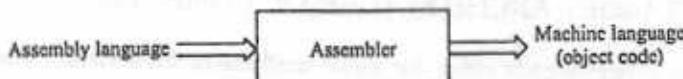


Fig. 1.4

1.10.2 Compilers

Compilers are the translators, which translates a program written in high-level language (like C++, FORTRAN, COBOL, PASCAL) in to machine code for some computer architecture (such as an intel pentium architecture). The generated code can be later executed many times against different data each time.

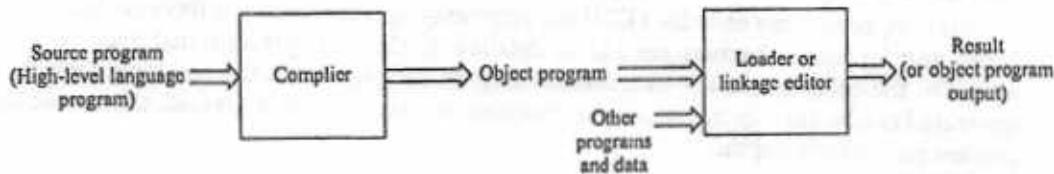


Fig. 1.5

In above figure compiler performs of converting a program written in one programming language (that is HLL program) in to object code. Now another software processor (which is called loader, also known as linkage editor) Performs some very low-level processing of object code in order to convert it into a ready-to-run program in the machine language. Loader is the program from which actually runs on the computer, reading input data, if any and producing results.

1.10.3 Interpreter

An interpreter reads an source program written in high-level programming language as well as data for this program, and it runs the program against the data to produce some results.

Note that both interpreters and compilers are written in some high-level programming language and they are translated into machine code. For example, a Java interpreter can be completely written in pascal, or even Java. The interpreter source program is machine independent since it does not generate machine code. An interpreter is generally slower than a compiler because it processes and interprets each statement in a program as many times as the number of the evaluation of this statement. For example, when a forloop is interpreted, the statements inside the for-loop body will be analyzed and evaluated on every loop step. Some languages, such as Java and LISP, come with both an interpreter and a compiler. Java source programs (java classes with Java extension). The Java interpreter, java, called the Java Virtual machine (JVM), may actually interpret byte codes directly or may internally compile them to machine code and then execute that code.

1.11 THE TARGET ARCHITECTURE OF COMPILER

A compiler can generate code either for a real machine or for a virtual machine (VM). A real machine is a piece of hardware equipment, like a microprocessor, capable of understanding an assembly code and executing it. *A virtual machine is a software program that acts like a real machine.* There are advantages and disadvantages, which result from using each one of these machines.

Real machine Architecture:-

Real machine has the advantage of working very fast: as fast as the hardware of the microprocessor works. On the other hand it has the disadvantage of being just as complicated. Often, generating and formatting instructions for these machine is very technical, tedious and time-consuming. In addition, the programmer has to become familiar with several very technical features of the operating system.

In short, generating code for a CPU is a very nessy operation because there are hundreds of codes, and for each code there are a lot of details at the bit and byte level that must be considered. On the other hand, the instructions may be very tiny, and the resulting code that is generated can be very long, making the compiler more complicated. Overall, the instructional process can be very painful.

1.11.1 Virtual Machine Architecture

In case of a virtual of a virtual machine, instructions are different than those of a real machine. In software, things like registers do not necessarily speed UP the execution process. Also instructions must necessarily be simple, since the overhead of decoding each instructions prior to execution becomes quite significant. On one hand, the higher the level of instructions, the less code the compiler has to generate. On the other hand, the simpler the instruction are, the quicker they can be executed by the VM. Fortunately, the demands of two are not mutually exclusive. Instruction must be both high-level, and quick for the software to decode. In addition to this, an extra benefit can be added, that the virtual machine instructions can be designed in accordance with the needs of the compiler. This can really simplify the process of

generating object code. The code generated for virtual machine best fit the needs of the compiler could be measurable shorter than code generated for a general-purpose computer.

Another advantage that results from using a virtual machine is the ease with which the compiler can be ported on other machines. *The best example is with the Java language*, although there have been others, such as the original pascal compiler, which generated code for a virtual machines called P-code. Usually, proting the compiler to another machine requires that the entire code generation process be rewritten. In the case of virtual machine, porting the compiler to another machine is a matter of resolving compatibility issue with the new compiler for the host architecture, that is the compiler that compiler the language being ported. Once a compiler has been ported to another architecture, it can be left largely the same as it was on the previous architecture, and it will still generate code that executes properly on our VM. Where the real work lies in porting a Vm language to another machines them selves are usually quite simple in nature. In that were not the case, they would be very slow.

The price that is paid these andvantages spoken of so far is the lack of speed at run time. A virtual machine, sometimes called an interpreter, is always slower than a real machine.

1.11.2 Pros and Cons of Both the Architecture

	<i>Hardware CPU</i>	<i>Virtual machine</i>
Pros:	<ul style="list-style-type: none"> • Very, very fast by comparision • Generalized architecture and Versatile • Low-level instruction set for doing low-level tasks 	<ul style="list-style-type: none"> • Some what simpler • Portable, all VM features are standardized • Able to have features that are expensive in hardware • Higher-level instruction set.
Cons:	<ul style="list-style-type: none"> • Complex and tedious to work with • Non portable. Each CPU is unique not all CPUS have the same features • Lower level • Complex instruction formatting. • Requires some knowledge of the US, as well 	<ul style="list-style-type: none"> • Very, very slow by Comparison

1.12 IMPORTANCE OF IMPRATIVE LANGUAGES

Out of the five principal porgramming language paradigm the imperative is the most popular, The importance can be viewed by following points.

- The imperative paradigm is the most established paradigm.
- It is much more in tune with the computer community's way of thinking.
- Imperative programs tend to run much faster than many other types of program.
- Programmers are prepared to sacrifice some of the advanced features and programming convenience generally associated with higher level languages in exchange for speed of execution.

The imperative languages can be defined, according to the characteristic that they display:

- By default, statement (commands) of imprective languages are executed in a step-wise, sequential, manner.
- As a result order of execution is crucial

- Destructive assignment- the effect of allocating a value to a variable has the effect of destroying any value that the variable might have held previously.
- Control is the responsibility of the programmer - programmers must explicitly concern themselves with issues such as memory allocation and deallocation of variables.

1.12.1 Growth of Imperative Programming Languages

All modern imperative languages can trace their origins back to three imperative languages: FORTRAN, ALGOL 60 and COBOL. consequently these influential languages are often described as foundation language.

Fortran, the IBM mathematical Formula translating system can be viewed as follows:

- Designed in 1955 by the backus and team.
- It was the first successful attempt to improve on "assembly languages".
- Fortran has been revised to take account of ideas on structured programming, however is decreasing in popularity.

ALGOL 60 ALGO rhythmic language can be viewed as follows:

- It was in 1950 s through a joint European- American committee.
- ALGOL was the first block structure language.
- ALGOL was the first language whose syntax was defined using BNF.
- It was the direct ancestor of most modern imperative languages.

COBOL Common Business oriented language can be viewed as follows:

- Developed in 1950s through a committee consisting mainly of U.S. computer manufacturers.
- COBOL was designed to process files and is therefore the most extensively used language for data processing.
- Since its intial introduction the language broadly unchanged.

Relationship between a number of common imperative (and other) programming languages are shown in the "writing diagram" of following figure:

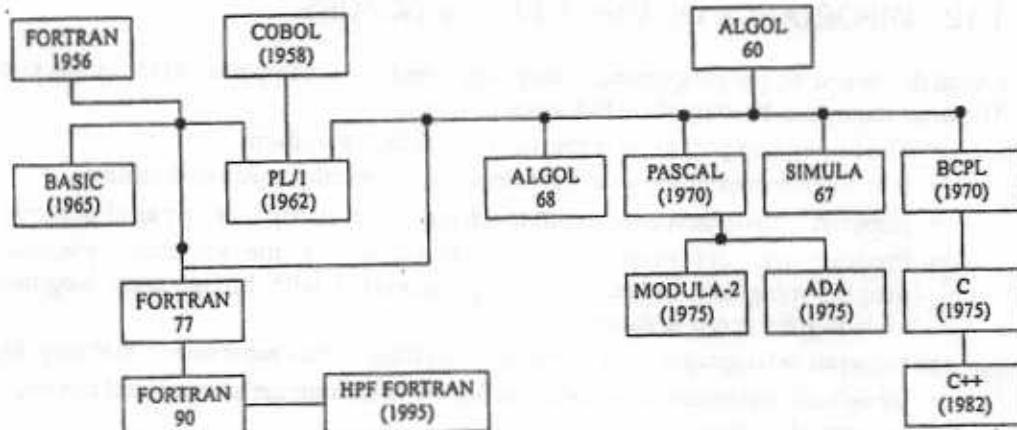


Fig. 1.6 Relation between a number of common imperative programming languages

A large number of modern imperative languages follow the style originated in Algol 60. These languages include Ada and pascal, and can thus be described as Algol style languages. There is also a significant group of languages that can be described as C style languages, these include the object oriented languages C++ and Java.

The most widely used imperative language found in industry is currently (the year 2000) C, which tends to dominate the market (with C++ and Java catching UP)

1.12.2 Features of Imperative languages

Following features of the imperative programming languages, made them important between all paradigm of programming languages:

1. They are usually "typed". Broadly we can identify two categories of imperative data type:
 - (i) Basic data types.
 - (ii) Compound data types
2. Imperative languages comprises the following components.
 - (i) Data declaration
 - (ii) Expression which yield values.
 - (iii) Statements which carry out some operation that is assignment statement, conditional statements, and program constructs.
3. I/O and error handling mechanism.
4. A method of grouping all of the above into a complete program.

1.13 THE CONCEPT OF BINDING AND BINDING TIME

"A binding is an association between two things, such as a name and the thing it names"

Binding may be between an attribute and an entity or between operation and a symbol.

For example `int x;`

When this statement is compiled, x is bound to a memory space.

Binding Time is the time at which the association between two items is created- There are many different binding times that can be implemented:

1. Language Design Time:

We known that most of the structures of the programming languages takes place at design time only. For example control flow constructs, primitives and other semantics. The binding of reserved words that is if, else, for, and while etc takes place at design time. The binding of asterisk symbol (*) is decided at design time, as it is bound to the multiplication operation.

2. Language Implementation Time:

Some binging may takes place at implementation time. Such as I/O couplings, system dependant things such as max heap and stack size and range of variables consider the example of C language variable

```
int x;
```

The range of possible values is decided at language implementation time

3. Program Writing Time:

Programmer also have some freedom to decide about the binding. For example the programmer selects data structures and names.

For example in C language:

```
Struct x {
    int y;
    int = [10];
}
```

The identification of above defined structure is x and it is decided by programmer only, as it may ram, shyam whatever.

4. Link Time :

We compile a program then certain things we did not write get compiled and bound. For example we are not supposed to compile the inbuilt functions of C language, like pring () and scarf () but we have to only link them by using the corresponding header files.

5. Load Time:

Load time binding is used for the primitive operating system.

In the link time binding, we have seen the binding of the inbuilt function of C language. The translator typically binds variables to addresses within the storage specified for each subprograms or inbuilt functions. However, this storage must be allocated actual addresses within the physical computer that will execute the program and this occurs during load time.

6. Run Time:

Run time binding is very broad term, it covers span of execution.

For example the values to variables binded at run time and in c++ and C, the binding of formal to actual parameters.

7. Translation Time:

Translator specification is also responsible to some bindings, for example the relative location of data objects in storage is decided by the translator only.

Now let us discuss a example to make the concept clear.

Consider the following statement of C language

```
int i;
-----
i = i +1;
```

Some of the bindings and their binding times for the parts of this assignment are as follows:

- The type (int) of variable is bound at compile time.
- The set of possible values of count is bound at language implementation time.
- The internal representation of literal is bound at design time.
- The meaning of the '+' operator is bound at compile time.
- The value of 2 is bound at execution time with this statement.

From the above discussion, it is very clear that binding and binding time can be known only when semantics of the program is clear.

1.13.1 Static Vs. Dynamic Bindings

The concept of static scope is almost similar to static binding, if a binding occurs before execution and remains unchanged throughout the program then it is said to be static binding.

Static bindings some time refers as early bindings and can be handled with greater efficiency.

If bindings occurs during run time then, said to be *dynamic bindings or some time late bindings.*

Every binding has its own advantages and disadvantages. For example early bindings provides efficiency and late bindings provides greater flexibility. But dynamic bindings has two more important disadvantage also, first, the error detection capability is diminished relative to static type bindings. Second greatest disadvantage of dynamic binding is cost.

Pascal has static bindings of types and dynamic bindings of values to variables, but LISP has dynamic binding of both values and types.

EXERCISE

1. There is possibility of one language, which can do all the jobs, which are performed by different language? Support your thoughts with arguments.
2. What are the pros and cons of the languages, which supports case sensitivity in their design?
3. Describe some design trade-offs between efficiency and safety in some language you know.
4. Evaluate C and C++ languages, using the criteria described in the chapter.
5. What major features would a perfect programming language include, in your opinion?
6. Give a brief description of the attributes a good programming languages.
7. What is a virtual computer? How are the languages implemented on a virtual computer.
8. A program consists of 400 statements of 100 statements are actually visited during execution. During initial phase of program development and test, the interpreter is certainly superior to the compiler, justify the statement
9. What is programming language? What is most important feature of a programming language?
10. What is binding? What are the features of language affected by late/early binding?
11. For the following statement of a programming language discuss various types of binding and the time when these bindings are done.

$X = X + 2 * Y - 5;$

12. Write major factors that influence development of programming languages and give brief overview of programming language development.

CHAPTER 2

Syntax, Semantic and Translation

2.1 SYNTAX Vs SEMANTICS OF THE LANGUAGES

It is very important for any programming language that it is easily understandable to the programmers and they are able to determine how expressions, statements and program module of the language formed. Programmers must be able to write the software by the help of manual of programming language.

So syntax of a programming language is the form of its expression, statements and program modules on the other hand the semantic of the programming language is the meaning given to the various syntactic structure.

For example in C language, we can take the decision by the following statement.

```
if (conditional expression)
{
    ...
    statements;
    ...
}
```

The meaning of above syntactical structure is that if conditional expression have true value the statements of the blocks will be execute.

2.2 SEMANTICS, THE MEANING OF THE PROGRAM

Obviously, anyone who can write a program in some programming language has at least an informal understanding of what that program means. There are, however, some problems raised by such in formal semantics:

- How can we be sure that what the programmer thinks it about the program?
- How can we prove (or give a rigorous argument) that the program does what it's meant to do?

Formal semantics provides a way of reasoning rigorously about program: one way of describing the meaning of the program is as a mathematical object; its mathematical properties can then be used to show that it meets its specification, or that it is equivalent to another program. Alternatively, a more direct route to proving the correctness of a program is to describe its semantics in terms of the logical properties it satisfies, and from those properties infer that it meets its specification.

There are three main kinds of semantics:

(a) *Operational semantics*

Operational semantics describes the meaning of the program in terms of how it is executed, for example by specifying the individual steps in its computation, or by describing an abstract machine that evaluates the program.

(b) *Denotational Semantics*

This kind of semantics describes the meaning of the program as a mathematical object, typically as a function that takes a state of a computer (its initial state before the program executes) and returns an updated state (the state of the computer that results from executing the program.)

(c) *Axiomatic semantics*

It describes the logical properties of programs; typically, an axiomatic semantics says that if some property holds before a program is run, then some other property will hold after the program has run.

For example, the semantics of a program to compute the minimum of two numbers p and q might state that if p is less than q then (on completion of the program) the value stored in a variable \min is p .

More generally, we can speak of the semantics not just of programs, but of a programming language. By this we mean that each syntactic constructs in the language can be given a formal semantics (that is, there would be a formal semantics for assignment, conditions, loops, etc.). The semantics of a program written in the language is then derived from semantics of its component parts (that is its assignment, loops, and so on). When the semantics of all programs are described in terms of the semantics of their components, we call this *compositional semantics*.

A related issue is that of programming language design, and it is here that formal semantics can play an important role. A formal semantics for a language can help separate the various elements that play a role in the language design. A key example is given by the programming language Algol, whose design was recorded in the Algol Report, a document that was groundbreaking in that it described both the syntax and the semantics of the language, and also clearly separated the syntax of the language from its semantics. One of the great contributions of the Algol Report was the introduction of Backus-Naur Form (named after John Backus and Peter Naur, two of the authors of the Algol Report) to give a formal description of the syntax of the language.)

We will use Backus-Naur Form (BNF) to describe the syntax of a simple programming language whose semantics we consider later. Now let us list the following examples of the uses of formal semantics:

- Some programming languages (such as Ada) support *parallel assignment*, where two or more variables can be assigned to simultaneously. For example in Ada,

$x, y := 0, 1$

has the effect of assigning 0 to x and 1 to y . Moreover, these assignments are done all at once: 0 is assigned to x at the same time as 1 is assigned to y . For example,

$x, y := y, x$

has the effect of swapping the values of x and y , which is clearly different from

$x := y;$

$y := x;$

Which simply sets both x and y to the initial values of y . It is fairly clear what parallel assignment does, and even from this brief description. It is often possible, however, to think up examples that test this informal understanding to the limit; for example, what are we to make of

$x, x := 0, 1$

Clearly, something is wrong here, and the program cannot set x to both 0 and 1. May be the program should assign either 0 or 1 to x , although there's no good reason to prefer one possibility or the other. Probably the best solution is to say that this is syntactically incorrect, have the compiler flag this as a syntax error, and but in the absence of formal syntactic and semantic description of the language, this choice is rather arbitrary.

The programming language Java was designed to be a plate form - independent language suitable for writing programs that could be downloaded across the internet and run in a browser (as Applets). Of course, no language can be run on some physical machine, but Java has a high degree of plate form-independence that is largely achieved through the use of the Java Virtual Machine. Java source code is compiled, but the target language of the compilation process is not the machine code of the some particular chip-set, but the machine code (so-called "byte code") of an abstract machine (a stack automation). This process of compiling to an abstract machine effectively provides Java with an operational semantics. Moreover, this semantics is useful in proving properties of the language such as the enforcement of Java's security model.

2.3 GRAMMARS OF PROGRAMMING LANGUAGES

Initially linguists were trying to define precisely valid sentences and to give structural description for these sentences. They tried to define rules for natural languages like Hindi, English, etc. Noam Chomsky gave a mathematical model for the grammars in 1956. Although it was useless to describe natural languages but it becomes very useful for computer languages.

The original motivation for grammars was the description of natural languages. We can write rules for the grammar of natural languages as follows:

```
<Sentence> → <noun phrase> <verb phrase>
<noun phrase> → <article> <noun>
```

According to above set of rules "the girl eats" is valid sentence.

It can be easily observed that, some words in the grammar works as termination for the sentence, these symbols are called terminal symbol. Rest of the symbols of the vocabulary are called non-terminals. So it is clear that terminals and non terminals forms vocabulary for any language.

Let us now formalise the idea of a grammar and how it is used. There are four important components in a grammatical description of language.

1. There is a finite set of symbols that form the strings of the language being defined. We call these alphabets terminal symbols. Represented by V_t
2. There is a finite set of variables, also called sometimes non terminals or syntactic categories. Each variable represents a language; that is, a set of strings represented by V_n .
3. One of the variable represents the language being defined; it is called the start. Generally it is denoted by S .
4. There is a finite set of rules or productions that represents the recursive definition of a language. Each production consists of:
 - (a) A variable that is being (partially) defined by the production. This variable is often called the head of the production.
 - (b) The production symbol \rightarrow
 - (c) A string of zero or more terminals and variables. This string, called the body of production, represents one way to form string in the languages of the variable of the head. So, we leave terminals unchanged and substitute for each variable of the body any string that is known to be in the language of that variable.

After this discussion, we able to define a mathematical model for the grammar

$$G = (V_n, V_t, P, S)$$

G : is the notation for the grammar

V_n : A finite set of non-terminals, generally represented by capital letters, A, B, C, D ...

V_t : A finite set of terminals, generally represented by small letters, like a, b, c ...

S : Starting non-terminal, called start symbol of the grammar. S belongs to V_n .

P : Set of rules or productions in CFG.

All the production in P have the form:

$$\alpha \rightarrow \beta$$

where $\alpha \in (V_n \cup V_t)^*$

and $\beta \in (V_n \cup V_t)^*$

These kind of grammars are known as unrestricted grammar, or phase - structure grammar.

2.3.1 Derivations and Their Variations

- Now we will define the notations to represent a derivation. First we define two notations $\xrightarrow[G]{\alpha} \beta$ and $\stackrel{i}{\xrightarrow[G]} \beta$. If $\alpha \rightarrow \beta$ is a production of P in CFG and a and b are strings in $(V_n \cup V_t)^*$, then

$$a \alpha b \xrightarrow[G]{\alpha} a \beta b.$$

We say that the production $\alpha \rightarrow \beta$ is applied to the string $a \alpha b$ to obtain $a \beta b$ or we say that $a \alpha b$ directly drives a βb .

- Now suppose $\alpha_1, \alpha_2, \alpha_3, \dots, \alpha_m$ are strings in $(V_n \cup V_t)^*$, $m \geq 1$ and $\alpha_1 \xrightarrow[G]{\alpha_2} \alpha_2, \alpha_2 \xrightarrow[G]{\alpha_3} \alpha_3, \dots, \alpha_{m-1} \xrightarrow[G]{\alpha_m} \alpha_m$. Then we say that $\alpha_1 \stackrel{i}{\xrightarrow[G]} \alpha_m$, that is, we say α_1 drives α_m in grammar G. If α_1 drives β by exactly i steps, we say $\alpha \stackrel{i}{\xrightarrow[G]} \beta$.
- In terms of notation, a *sentential form* is the goal or start symbol, or any string that can be derived from it, that is, any string σ such that $s \xrightarrow[G] \sigma$ where $\sigma \in (V_n \cup V_t)^*$.
- A grammar is called *recursive* if it permits derivations of the form $A \xrightarrow[G] w_1 Aw_2$ (where $A \in V_n$, and $w_1, w_2 \in (V_n \cup V_t)^*$). More specifically, it is called left recursive if $A \xrightarrow[G] Aw$ and right recursive if $A \xrightarrow[G] wA$.
- A grammar is *self-embedding* if it permits derivations of the form $A \xrightarrow[G] w_1 Aw_2$ (where $A \in V_n$, and where $w_1, w_2 \in (V_n \cup V_t)^*$, but where w_1 or w_2 contains at least one terminal (that is $(w_1 \cap V_t) \cup (w_2 \cap V_t) \neq \emptyset$).
- Formally we can define a language $L(G)$ produced by a grammar G by the relation

$$L(G) = \{w / w \in V_t^*; s \xrightarrow[G] w\}.$$

2.3.2 Classic BNF Notation for Productions

As we have remarked, a production is a rule relating to a pair of strings, say α and β , specifying how one may be transformed into the other. This may be denoted $\alpha \rightarrow \beta$, and for simple theoretical grammars use is often made of this notation, using the conventions about the use of upper case letters for non-terminals and lower case ones for terminals. For more realistic grammars, such as those used to specify programming languages, the most common way of specifying productions for many years to use alternative notations invented by Naur were largely responsible for the Algol 60 report (Naur, 1960 and 1963), which was the first major attempt to specify the syntax of a programming language using this notation. Regardless of what the acronym really stands for, the notation is now universally known as BNF.

In BNF, productions have the form

$$\text{left side} \rightarrow \text{definition}$$

where left side $\in (V_n \cup V_t)^+$ and definition $\in (V_n \cup V_t)^*$ although we must be more restrictive than that, for left side contain one non terminal, so that we must also have

$$\text{left side} \cap V_n = \emptyset$$

Frequently we find several productions with the same left side, and these are often abbreviated by listing the definitions as a set of one or more alternatives, separated by vertical bar symbol '|'.

Let production for any grammar are:

$$\begin{aligned}s &\rightarrow a \mid a \\ s &\rightarrow b \mid b \\ s &\rightarrow c\end{aligned}$$

then in BNF we can represents, above production as follows:

$$s \rightarrow asa \mid bsb \mid c$$

2.3.3 Context-Free Grammars

Mathematically context - free grammar is defined as follows:

"A grammar $G = (V_n, V_t, P, S)$ is said to be context - free"

where V_n : A finite set of non-terminals, generally represented by capital letters, A, B, C, D, ...

V_t : A finite set of terminals, generally represented by small letters, like a, b, c, d, e, f, ...

S: starting non-terminal, called start symbol of the grammar. S belong to V_n .

P: Set of rules or productions in CFG.

All production in P have the form

$$\alpha \rightarrow \beta$$

where $\alpha \in V_n$, $\beta \in (V_n \cup V_t)^*$ and $|\alpha| = 1$.

Context-free grammars derive their name from the fact that the substitution of the variable on the left of a production can be made any time, such a variable appears in the sentential form. It does not depend on the symbols in the rest of the sentential form (the context). This feature is the consequences of allowing only a single variable on the left side of production.

Example 2.1 Write a CFG, which generates strings of balanced parenthesis.

Solution: This grammar will accept the balanced right and left parenthesis. For example () () is acceptable, ((())) is also accepted.

Let us design the CFG for this

Let CFG be

$$G = (V_n, V_t, P, S) \text{ where}$$

$$V_n = \text{set of non-terminals} = \{S\}$$

$$V_t = \text{set of terminals} = \{(,)\}$$

and set of production p is given by

$$S \rightarrow SS$$

$$S \rightarrow (S)$$

$$S \rightarrow t$$

Now we see what this grammar generates:

$$\begin{aligned}s &\Rightarrow ss \\ &\Rightarrow (s) s \\ &\Rightarrow (s) ss \\ &\Rightarrow (s) (s) s \\ &\Rightarrow (s) (s) (s) \\ &\Rightarrow () (s) (s) \\ &\Rightarrow () () (s) \\ &\Rightarrow () () ()\end{aligned}$$

thus the above grammar generates balanced pairs of parenthesis.

2.3.4 Left Most and Right Most Derivations

In order to restrict the number of choices of replacement of variables, if at each step we replace the "left most" variable by one of its production bodies such a derivation is called a "left most derivation", and we indicated that a derivation is left most by using relation \Rightarrow_{lm} and $\dot{\Rightarrow}_{lm}$ for one or many steps, respectively.

Similarly, it is possible to require that each step the "right most" variable is replaced by one of its bodies. If so, we call the derivation "right most" and use symbols \Rightarrow_{rm} and $\dot{\Rightarrow}_{rm}$ to indicate one or many right most derivation steps, respectively.

2.3.5 Parse Tree

There is a tree representation for derivation that has proved extremely useful. It is the second way of showing derivations, independent of the order in which production are used, is also called "derivation tree".

"A parse tree is an ordered tree in which nodes are labeled with the left sides of productions and in which the children of a node represent its corresponding right sides".

Let $G = (V_n, V_t, P, S)$ be a context-free grammar. An ordered tree for this CFG, G is a derivation tree if and only if it has the following properties.

- (a) The root is labeled by the starting non-terminal of the CFG that is S .
- (b) Every leaf of the ordered tree has a label from $V_t \cup \{\epsilon\}$.
- (c) Every internal node of ordered tree has a label from V_n .
- (d) Let us assume that a vertex has label $X \in V_n$, and its children's are labeled (from left to right) $y_1, y_2, y_3, \dots, y_n$, then production must contain a production of the form

$$X \rightarrow y_1, y_2, \dots, y_n$$

- (e) A leaf labeled ϵ has no siblings, that is, vertex with a child labeled ϵ can have no other children. Clearly if the leaf is labeled ϵ , then it must be the only child of its parent.

If we look at the leaves of any parse tree and can concate them from the left, we get a string called the *yield of the tree*, which is always a string of special importance.

Example 2.2 Consider the CFG

$$S \rightarrow XX$$

$$X \rightarrow XXX/bX/Xb/a$$

Find the parse tree for the string bb aa aa b.

Solution: Given CFG is

$$S \rightarrow XX$$

$$S \rightarrow XXX/bX/Xb/a$$

and string is $w = bb\ aa\ aa\ b$

We begin with S and apply the production $S \rightarrow XX$

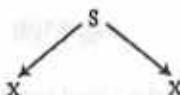


Fig. 2.1(a)

To the left hand X, let us apply the production $X \rightarrow bX$.

To the right hand X, let us apply $X \rightarrow XXX$.

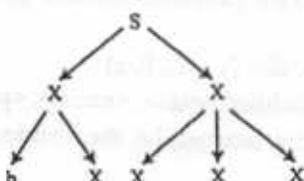


Fig. 2.1(b)

The b that we have on the bottom line is a terminal, so it does not descend further. In the terminology of trees, it is called a "terminal node". Let the four X's left to right, undergo the production $X \rightarrow bX$, $X \rightarrow a$, $X \rightarrow a$ and $X \rightarrow Xb$ respectively. Now we have

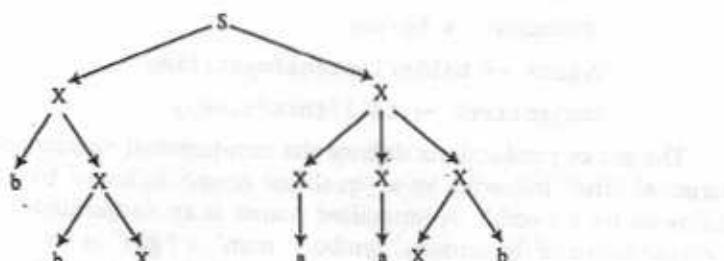


Fig. 2.1(c)

Let us finish off the generation of a word with the production $X \rightarrow a$ and $X \rightarrow a:$

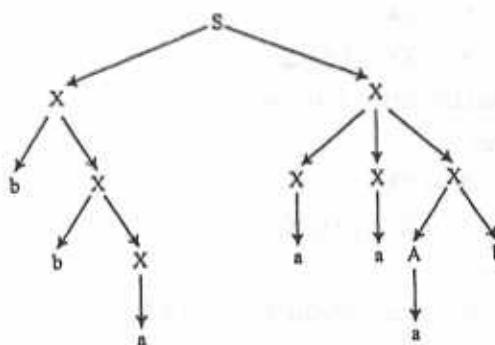


Fig. 2.1(d)

Reading from left to right, we see the word we have produced is bb aa aa b.

These tree diagrams are called "syntax trees", "parse trees", "generation trees", production trees and derivation trees.

Example 2.3 Write a context-free grammar which is able to generate "the thin boy talks" English sentence, also give the parse tree.

Solution: We carefully reads the problem then it is clear that we have to write a CFG for a tiny subset of English itself.

Let us assume that CFG be $G = (V_n, V_t, P, S)$

where $V_n = \{\langle \text{Sentence} \rangle, \langle \text{qualified noun} \rangle, \langle \text{noun} \rangle, \langle \text{pronoun} \rangle, \langle \text{Verb} \rangle, \langle \text{adjective} \rangle\}$.

$V_t = \{\text{the}, \text{man}, \text{girl}, \text{boy}, \text{lecturer}, \text{he}, \text{she}, \text{drinks}, \text{sleeps}, \text{mystifies}, \text{tall}, \text{thin}, \text{thirsty}\}$

$S = \langle \text{sentence} \rangle$

$P = \{$

$\langle \text{Sentence} \rangle \rightarrow \text{the } \langle \text{qualified noun} \rangle \langle \text{Verb} \rangle \quad \dots(1)$

$\langle \text{Sentence} \rangle \rightarrow \langle \text{Pronoun} \rangle \langle \text{Verb} \rangle \quad \dots(2)$

$\langle \text{qualified noun} \rangle \rightarrow \langle \text{adjective} \rangle \langle \text{noun} \rangle \quad \dots(3)$

$\langle \text{noun} \rangle \rightarrow \text{man} / \text{girl} / \text{boy} / \text{lecturer} \quad \dots(4)$

$\langle \text{Pronoun} \rangle \rightarrow \text{he/she} \quad \dots(5)$

$\langle \text{Verb} \rangle \rightarrow \text{talks/listens/mystifies} \quad \dots(6)$

$\langle \text{adjective} \rangle \rightarrow \text{tall/thin/sleepy} \quad \dots(7)$

The set of productions defines the non-terminal $\langle \text{sentence} \rangle$ as consisting of either the terminal "the" followed by a $\langle \text{qualified noun} \rangle$ followed by a $\langle \text{verb} \rangle$, or as a $\langle \text{pronoun} \rangle$ followed by a $\langle \text{verb} \rangle$. A $\langle \text{qualified noun} \rangle$ is an $\langle \text{adjective} \rangle$ followed by a $\langle \text{noun} \rangle$, and a $\langle \text{noun} \rangle$ is one of the terminal symbols "man" or "girl" or "boy" or "lecturer".

A $\langle \text{Pronoun} \rangle$ is either of the terminals "he" or "she", while a $\langle \text{verb} \rangle$ is either "talks" or "listens" or "mystifies". Here $\langle \text{Sentence} \rangle, \langle \text{noun} \rangle, \langle \text{qualified noun} \rangle, \langle \text{pronoun} \rangle, \langle \text{adjective} \rangle$

and $\langle \text{verb} \rangle$ are non terminals. These do not appear in any sentence of the language, which includes such majestic prose as

```
he talks
the thin lecturer mystifies
the sleepy boy listens
```

From a grammar, one non-terminal is singled out as the so-called *start symbol*. If we want to generate an arbitrary sentence we start with the start symbol and successively replace each non-terminal on the right of the production defining that non-terminal, until all non-terminals have been removed. In the above example the symbol $\langle \text{sentence} \rangle$ is, as one would expect, the start symbol.

Thus for example, we could start with $\langle \text{Sentence} \rangle$ and from this derive the sentential form

```
the <qualified noun> <verb>
```

Now let us generate "the thin boy talks".

```
 $\langle \text{sentence} \rangle \rightarrow \text{the } \langle \text{qualified noun} \rangle \langle \text{verb} \rangle$ 
      → the <adjective> <noun> <verb>
      → the thin <noun> <verb>
      → the thin boy <verb>
      → the thin boy talks.
```

Parse tree for above derivations is as follows:

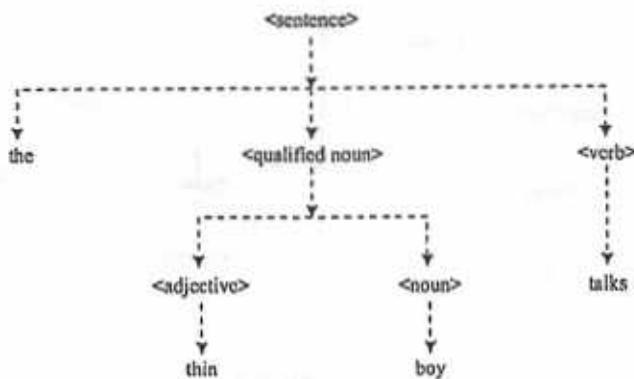


Fig. 2.2

Example 2.4 Design a context - free grammar which generate the mathematical expressions for '+' and '-' operators.

Solution: Let CFG be $G = (V_n, V_t, S, P)$

where $V_n = \{\langle \text{goal} \rangle, \langle \text{expression} \rangle, \langle \text{term} \rangle, \langle \text{factor} \rangle\}$

$V_t = \{a, b, c, +, -\}$

$S = \langle \text{goal} \rangle$

and production are defined as follows:

```

<goal> → <expression>
<expression> → <term>/<expression> - <term>
<term> → <factor>/<term> * <factor>
<factor> → a/b/c
    
```

Let us apply above grammar on algebraic expression $a - b \times c$.

```

<goal> → <expression>
        → <expression> - <term>
        → <term> - <term>
        → <factor> - <term>
        → a - <term>
        → a - <term> × <factor>
        → a - <factor> × <factor>
        → a - b × <factor>
        → a - b × c
    
```

The parse tree for above string is as follows:

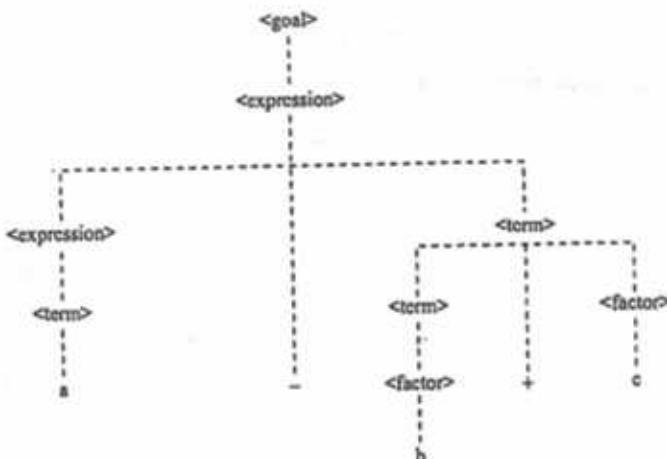


Fig. 2.3

2.3.6 Extensions to BNF

Various simple extensions are often employed with BNF notation for the sake of increased readability and for the elimination of unnecessary recursion (which has a strange habit of confusing people brought up on iteration). Recursion is often employed in BNF as a means of specifying simple repetition, as for example.

```

<unsigned integer> → <digit>/<digit><unsigned integer>
(which uses right recursion) or
<unsigned integer> → <digit>/<unsigned integer> <digit>

```

(which uses left recursion).

Then we often find several production used to denote alternatives which are very similar, for example.

```

<integer> → <unsigned integer>/<sign><unsigned integer>
<unsigned integer> → <digit>/<digit><unsigned integer>
<sign> → +/-

```

Using six productions (besides the omitted obvious ones for <digit>) to specify the form an <integer>.

The extension introduced to simplify these construction lead to what is known as EBNF (Extended BNF). There have been many variations on this, most of them by the meta symbols used for regular expression. Thus we might find the use of the kleene closure operations to denote repetition of a symbol zero or more times, and the use of round brackets or parentheses () to group items together.

Using these ideas we might define as integer by

```

<integer> → <sign> <unsigned integer>
<unsigned integer> → <digit> (<digit>) '*'
<sign> → +/-/ε

```

or even by

```
<integer> → (+/-/ε) <digit> (<digit>) '*'
```

which is, of course, nothing other than a regular expression anyway. In fact, a language that can be expressed as regular expression can always be expressed in a single EBNF expression.

2.3.7 Wirth's EBNF Notation

In defining Pascal and Modula-2, Wirth came up with one of these many variations on BNF which has now become rather widely used (Wirth, 1977). Further meta symbols are used, so as to express more succinctly the many situations that otherwise require combinations of the kleene closure operators and the ε string. In addition, further simplification are required to facilitate the automatic processing of production by parser generators such as we shall discuss in later section. In this notation for EBNF.

- Non-terminal* : are written as single words.
- Terminals* : are all written in quotes, as in "BEGIN".
- / : is used, as before to denote alternatives
- () : (parentheses) are used to denote the optional appearance of a symbol or group of symbols.
- [] : (brackets) are used to denote optional repetition of a symbol or group of symbols

- = : is used in place of the → symbol
- * : is used to denote the end of each production.
- (**) : are used in some extensions to allow comments.
- ε : can be handled by using the [] notation
- spaces : are essentially insignificant.

For example

```

Integer      = Sign Unsigned Integer.
Unsigned Integer = digit {digit}.
Sign          = ["+" / "-"].
digit         = "0"/"1"/"2"/"3"/"4"/"5"/"6"/"7"/"8"/"9".

```

A variation on the use of braces allows the (otherwise impossible) specification of limit on the number of times a symbol may be repeated - for example to express that an identifier in Fortran may have a maximum of six characters. This is done by writing the lower and upper limits as sub - and super - scripts to the right of the curly braces, as for example.

$$\text{Fortran Identifier} \rightarrow \text{letter} \{ \text{letter/digit} \}_0^5$$

2.3.8 The British Standard for EBNF

The British standard Institute has a published standard for EBNF (B56154 of 1981). The BSI standard notation is noisier than Wirth's one: elements of the production are separated by commas, productions are terminated by semicolons, and spaces become insignificant. This means that compound words like ConstIdentifier are unnecessary, and can be written as separate words. An example in BSI notation follows:

```

Constant Declaration = "CONST"
                      constant Identifier, "=",
                      Constant Expression, ";"
                      {constant Identifier, "=",
                      constant Expression, ";"};

Constant Identifier = identifier;

Constant Expression = Expression;

A → B
B → C
C → A

```

Not only is the useless in this new sense, it is highly undesirable from the point of obtaining a unique parse, and so all parsing technique require a grammar to be cycle-free, it should not permit a derivation of the form $A \xrightarrow{*} A$

2.3.9 Ambiguous Grammar

An important property which one looks for in programming languages is that every sentence that can be generated by the language should have a *unique parse tree*, or, equivalently, a *unique left (or right) Canonical parse*.

If a sentence produced by a grammar has two or more parse tree then the grammar is said to be ambiguous. An example of ambiguity is provided by another attempt at writing a grammar for simple algebraic expression.

```

S      → Expression
Expression → Expression - Expression
           / Expression × Expression
           / Factor
Factor   → a/b/c
  
```

with this grammar the sentence $a - b \times c$ has two distinct parse trees and two canonical derivations. We refer to the number to show the derivation steps.

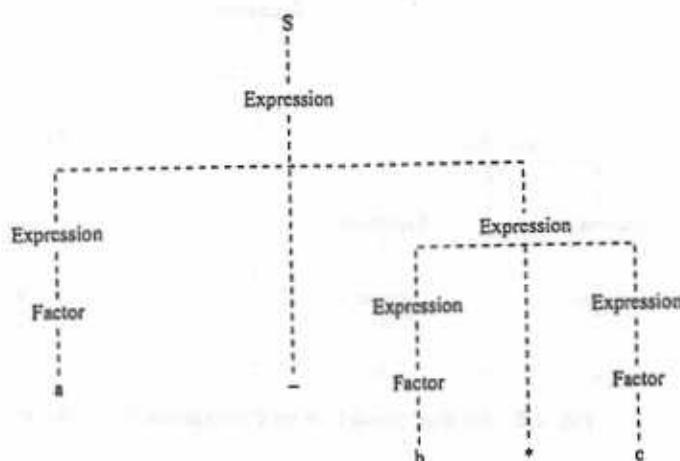


Fig. 2.4 One parse tree for the expression $a - b \times c$

The parse tree shown in Fig 2.4 corresponds to the derivation

```

S → Expression
→ Expression - Expression
→ Factor - Expression
→ a - Expression
→ a - Expression × Expression
→ a - Factor × Expression
→ a - b × Expression
→ a - b × Factor
→ a - b × c
  
```

while the second derivation

```

S → Expression
→ Expression - Expression × Expression
  
```

$\rightarrow \text{Factor} - \text{Expression} \times \text{Expression}$
 $\rightarrow a - \text{Expression} \times \text{Expression}$
 $\rightarrow a - \text{Expression} \times \text{Expression}$
 $\rightarrow a - \text{Factor} \times \text{Expression}$
 $\rightarrow a - b \times \text{Expression}$
 $\rightarrow a - b \times \text{Factor}$
 $\rightarrow a - b \times c$

Corresponds to the parse tree depicted in Fig. 4.8.

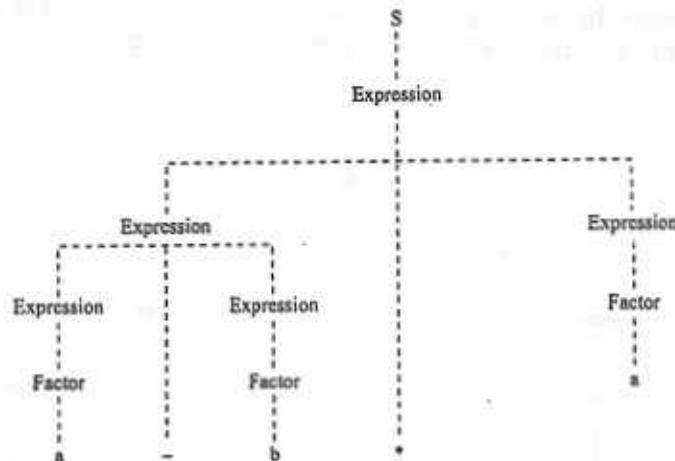


Fig. 2.5 Another parse tree for the expression $a - b \times c$

If the only use for grammars was to determine whether a string belonged to the language, ambiguity would be of little consequence. However, if the meaning of a program is to be tied to its syntactic structure, then ambiguity must be avoided. In the example above, the two trees correspond to two different evolution sequences for the operators \times and $-$.

In the first case the "meaning" would be the usual mathematical one, namely $a - (b \times c)$, but in the second case the meaning would effectively be $(a - b) \times c$.

The most famous example of an ambiguous grammar probably relates to the IF ... THEN ... ELSE statement in simple Algol-like language. Let us demonstrate this by defining a simple grammar for such a construct.

Program	\rightarrow	Statement.
Statement	\rightarrow	Assignment/ IfStatement.
Assignment	\rightarrow	Variable ":" Expression.
Expression	\rightarrow	Variable.
Variable	\rightarrow	"i"/"j"/"k"/"a"/"b"/"c".
IfStatement	\rightarrow	"IF" condition "THEN" statement "/"IF" condition "THEN" statement "ELSE" statement.

Condition → Expression “=” Expression
 / Expression “≠” Expression

In this grammar the string

IF i = j THEN IF i = k THEN a : = b ELSE a : = c

has two possible trees. The essential point is that we can parse the string to correspond either to

IF i = j THEN (IF i = k THEN a : = b ELSE a : = c)
 ELSE (nothing)

or to

IF i = j THEN (IF i = k THEN a : = b ELSE nothing)
 ELSE (a : = c)

Any language which allows a sentence such as this may be inherently ambiguous unless certain restrictions are imposed on it, for example, on the part following the THEN of an If statement, as was done in Algol (Naur, 1963). In Pascal and C++, as is hopefully well known, an ELSE is deemed to be attached to the most recent unmatched THEN, and the problem is avoided that way. In other languages it is problem is avoided that way. In other languages it is avoided by introducing closing tokens like ENDIF and ELSEIF. It is, however, possible to write production that are unambiguous:

Statement → Matched / Unmatched.

Matched → “IF” condition “THEN” Matched “ELSE”
 matched/other statement.

Unmatched → “IF” condition “THEN” statement
 / “IF” condition “THEN” matched “ELSE” unmatched.

In the general case, unfortunately, no algorithm exists (or can exist) that can take an arbitrary grammar and determine with certainty and a finite amount of time whether it is ambiguous or not.

2.3.10 Context Sensitivity

Some potential ambiguities belong to a class which is usually termed *context-sensitive*. Spoken and written language is full of such examples, which the average person parse with ease. For example, the sentences

Time flies like an arrow.

and

Fruit flies like apple.

in one sense have identical construction

Noun Verb Adverbial phase

but, unless we were preoccupied with aerodynamics, in listening to them we would probably subconsciously parse the second along the lines of

Adjective Noun Verb Noun phrase

Example like this can be found in programming language too. Until now all our practical examples of productions have a single non-terminal on the left side, although grammars may be more general than that. Based on pioneering work by a linguist (Chomsky, 1959), computer scientist now recognize four classes of grammar. The classification depends on the format of the productions. This classification is called Chomsky Hierarchy. According to the classification grammars can be of four types, as follows:

- (i) Type - 0 Grammar (Unrestricted grammar)
- (ii) Type - 1 Grammar (Context sensitive grammar)
- (iii) Type - 2 Grammar (Context - free grammar)
- (iv) Type - 3 Grammar (Regular grammar)

We have already seen type - 2 (context - free grammar) in detail, now we will study type - 0, type - 1 and type - 3.

2.3.10.1 Type - 0 Grammar (Unrestricted Grammar)

The unrestricted grammar is defined as

$$G = (V_n, V_t, P, S)$$

where V_n = a finite set of non-terminals

V_t = a finite set of terminals

S = starting non-terminal, $S \in V_n$

and P is the set of productions of the following form

$$\alpha \rightarrow \beta$$

where α and β are arbitrary string of grammar symbols with $\alpha \neq \epsilon$. These grammar are known as type - 0, phrase - structure or unrestricted grammars.

This type is so rare in computer applications that we shall consider it no further here. Practical grammars need to be far more restricted if we are to base translators on them.

2.3.10.2 Type - 1 Grammars (Context-sensitive)

If we impose the restriction on a type - 0 grammar that the number of symbols in the string on the left of any production is less than or equal to the number of symbols on the right side of that production, we get the subset of grammars known as type-1 or context-sensitive. In fact, to qualify for being of type 1 rather than of a yet more restricted type, it is necessary for the grammar to contain at least one production with a left side longer than one symbol.

Production in type 1 grammar of the general form

$$\alpha \rightarrow \beta$$

with $|\alpha| \leq |\beta|, \alpha \in (V_n \cup V_t)^* V_n (V_n \cup V_t)^*$ and

$$\beta \in (V_n \cup V_t)^+$$

Strictly, it follows that the null string would not be allowed as a right side of any production. However, this is sometimes overlooked, as ϵ -productions are often needed to terminate recursive definition. Indeed, the exact definition of "context - sensitive" differs from author. In another definition, productions are required to be limited to the form

$$\alpha A \beta \rightarrow \alpha v \beta \text{ with } \alpha_1 \beta \in (V_n \cup V_t), A \in V_n^+, v \in (V_n \cup V_t)^*$$

Here we can see the meaning of context-sensitive more clearly-A may be replaced by v when A is found in the context of (that is, surrounded by) α and β .

A much quoted simple example of such a grammar is as follows:

$$G = \{V_n, V_t, S, P\}$$

$$V_n = \{A, B, C\}$$

$$V_t = \{a, b, c\}$$

$$S = \{A\}$$

$$P =$$

$$A \rightarrow aABC/abc \quad \text{say P1}$$

$$CB \rightarrow BC \quad \text{say P2}$$

$$bB \rightarrow bb \quad \text{say P3}$$

$$bC \rightarrow bc \quad \text{say P4}$$

$$cC \rightarrow CC \quad \text{say P5}$$

Let us derive a sentence using this grammar. A is the start string:

Let us choose to apply P_1

$$A \rightarrow aABC$$

$$A \rightarrow aabCBC \quad \text{By P2}$$

$$A \rightarrow aabBCC \quad \text{by P3}$$

$$A \rightarrow aa bb CC \quad \text{by P4}$$

$$A \rightarrow aa bb cC \quad \text{by P5}$$

$$A \rightarrow aa bb cc \quad \text{by P6}$$

However, with this grammar it is possible to derive a sentential form which no further production can be applied. For example after deriving the sentential form

$$aabCBC$$

if we were apply P_5 instead P_3 we would obtain

$$aabcBC$$

but no further production can be applied to this string. The consequence of such a failure to obtain a terminal string is simply that we must try other possibility's until we find those that yield terminal strings.

The consequences for the reverse problem, namely parsing, are that we may have to resort to considerable backtracking to decide whether a string is a sentence in the language.

2.3.10.3 Type 3 Grammars (Regular, Right-liner or Left-liner)

A regular grammar may be left liner or right liner. "If all production of a CFG are of the form $A \rightarrow wB$ or $A \rightarrow w$, where A and B are variables and $w \in V_t^*$, then we say that grammar is right liner".

"If all the production of a CFG are of the form $A \rightarrow Bw$ or $A \rightarrow w$, we call it left liner".

Example 2.5 Write the left liner and right liner grammar for the regular expression

$$r = 0(10)^*$$

Solution: Given regular expression $r = 0(10)^*$, it means any string which starts with 0 followed by any number of 10's is in regular expression.

Left liner: Let grammar be $G_1 = (V_n, V_v, P, S)$

here

$$V_n = \{S\}$$

$$V_t = \{0, 1\}$$

and P is defined as follows:

$$S \rightarrow S10/0$$

Right liner: Let grammar be $G_2 = (V_n, V_v, P, S)$ where

$$V_n = \{S, A\}$$

$$V_t = \{0, 1\}$$

and p is defined as follows

$$S \rightarrow 0A$$

$$A \rightarrow 10A/\epsilon$$

2.3.10.4 The Relationship between Grammar-type and Language-type

It should be clear from the above that type - 3 grammars are a subset of type 2 grammars, which themselves from a subset type 1 grammars, which in turn from a subset of type 0 grammars

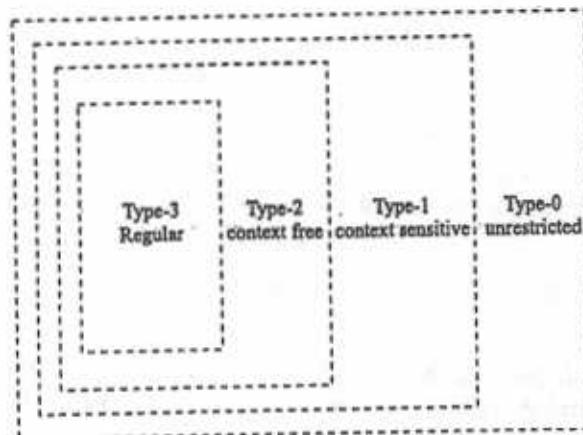


Fig. 2.6

A language $L(G)$ is said to be of type k if it can be generated by a tape k grammar. Thus for example, a language is said to be context - free if a context - free grammar may be used to define it.

Grammars for modern programming languages are usually largely Context - free, with some unavoidable context sensitive features, which are usually handled with a few extra adhoc rules and by using so-called attribute grammar, rather than by engaging on the far more difficult task of finding suitable context - sensitive grammars.

2.4 THE SYNTAX OF A SIMPLE IMPERATIVE LANGUAGE

There are five syntactic classes in our imperative language:

- $\langle \text{zz} \rangle$ - numbers in standard decimal notation, for example 4, 58, etc.
- $\langle \text{Var} \rangle$ - variable (or identifier) names.
- $\langle \text{Exp} \rangle$ - arithmetic expressions, built up from numbers and variables using operation for addition, subtraction and multiplication.
- $\langle \text{Tst} \rangle$ - Boolean expressions; these include tests on two arithmetic expressions and testing whether the value of one arithmetic expression is less than or equal to another. There are also the logical "and", "or" and "not" operations.
- $\langle \text{pgm} \rangle$ - Program ("commands") including assignment, sequential composition, conditionals and while - loops.

These syntactic classes are defined by the following BNF rules.

```

<Exp> → <zz>/<var>/<Exp> + <Exp>/<Exp> - <Exp>/<Exp> × <Exp>
<Tst> → true/false/<Exp> = <Exp>/<Exp> <= <Exp>
        /not <Tst>/<Tst> and <Tst>/<Tst> or <Tst>
<pgm> → skip /<var> : = <Exp>/<pgm>; <pgm>
        /if <Tst> then <pgm> else <pgm> fi
        / while <Tst> do <pgm> od
    
```

The program `skip` is a program that does nothing: it simply terminates immediately. It is useful, for example, in defining on "if - then" construct the "if - then - else - if" conditional:

$$\text{if } x \text{ then } y = \text{if } x \text{ then } Y \text{ else skip } f_2$$

2.4.1 Denotational Semantics

Denotational Semantics describes the meaning of a program as some mathematical object: this is the program's denotation. Typically, the denotation of a program is a function that maps states to states. Given a program p , its denotation, $[] P$, is the function that takes a state s as argument and maps it to the state that results from running p in the states, provided that program P terminates in the state s .

The syntax of our language is described by the following BNF definitions:

```

<VAR> → a/b/ ... /z/A/B/ ... /aa/ ...
<Num> → 0/1/2 ... /9
<Exp> → <VAR>/<Num>/ - <Exp>/<Exp> + <Exp>
        /<Exp> × <Exp>
<Tst> → <Exp><<Exp>/<Exp> is <Exp>/ true / false / not <Test>/ <Tst> and
        <Tst>/<Tst> or <Tst>
<pgm> → skip/<var> : = <Exp>/<pgm>; <pgm>/ if <Tst> then <pgm>
        else <pgm> fi/while <Tst> do <pgm> od
    
```

Now let us learn, what we mean by a state.

State

In any imperative language, assignment is the most basic form of program: the other linguistic constructs (conditional and loops) are simply ways of organizing assignment into series that are executed in a particular order. For example, think of how the following program is evaluated:

```
x: = 0;
f: = 1;
while x <= 2
do
    x: = x + 1;
    f: = f * x
```

On each pass through the loop, the value of *x* is increased by 1, and the body of the loop is gone through three times (while *x* has values 0, 1, then 2). Thus, we could “unfold” the program above to the following series of assignments:

```
x: = 0;
f: = 1;
x = x + 1;
f: = f * x;
x : = f + 1;
f : = f * x;
x: = x + 1;
f : = f * x
```

(one of the benefits of the formal semantics is that we could give rigorous argument that these two programs are indeed equivalent).

Assignment, therefore, lie at the heart of imperative programming languages. In particular, they suggest that any semantics should be based on the notation of *state* (or *storage*), by which we mean that particular values that are associated with a program’s variables. When a program is being executed, we can think of the computer that is running the program as being in a certain state. This state is determined by the values stored by the program’s variables, and these values are updated by assignments to the variables. For example, after running the program above, the computer will be in a state where the variable *x* has the value 3, and the variable *f* has the value 6.

We can think of a state as being a table that tells us the value associated with any given variable (in our language, variables are not declared, and have no scope, so the state should tell us the value associated with any variable. This would give us an infinitely long table, and it is more convenient to think of a state as a function that takes a variable as argument and returns the value stored in that variable more formally, a state function

Var → *zz*

where *Var* is the set of variable names, and *zz* is the set of numbers (in our language, all variables store numbers). For example after running a program

```
x := 8;
y := x + 1;
z := y + 2;
```

We obtain a state s , with

- $s(x) = 8$,
- $s(y) = 9$, and
- $s(z) = 11$.

(we don't know what values gives to variables other than x , y or z , but presumably those values are not changed by the program.)

The semantics, which will be given us, will describe how assignment (and other programs) update states. Before we do this, note that an assignment like

$$y := x + 1;$$

depends on the state of the computer before the assignment is executed: the value assigned to y depends on the value of x in this "initial" state (that is if the value of x in the initial state is 8, then y is assigned the value 9).

2.4.2 The Denotational Semantics of Arithmetic Expressions

Given an arithmetic expression such as

$$2 \times (x + y)$$

we want to describe its denotation as a number, but clearly the value of such an expression depends on the values stored in x and y . In other words the value of an arithmetic expression depends on the state of the computer when the expression is evaluated. We therefore describes the denotation of an arithmetic expression e as a function $[e]$: state \rightarrow zz.

This function is defined inductively as follows:

1. For a number n , clearly the value should just be n , independent of the state:

$$[[n]](s) = n.$$
2. For a variable x , the value should simply be the value in the given state:

$$[[x]](s) = s(x).$$
3. When e has form $e_1 + e_2$, the value should be the sum of the values of e_1 and e_2 :

$$[[e_1 + e_2]](s) = [[e_1]](s) + [[e_2]](s).$$
4. The operations for "unary minus" and multiplication are treated similarly.

2.4.3 The Denotational Semantics of Program

The denotational semantics for programs is given by defining, for a program p , its denotation $[[p]]$. From our intuitive understanding of programs (strengthened by having seen the operational semantics), we know that executing a program has the effect of updating the state in which execution of the program begins. Thus, the denotation $[[p]]$ will be a function that takes a state S as argument and returns the state that result from executing p in the state s . of course, there is the possibility that p fails to terminate when it is executed in s , in which case $[[p]](s)$ shouldn't return any state: in other words, $[[p]](s)$ is undefined. This mean that, in

general, $[[p]]$ is a partial function from states to states. (The denotation functions $[[e]]$ and $[[t]]$ for expressions and tests are both total functions from states to number and truth-values, respectively, as evaluating arithmetic and Boolean expression always terminates).

The function $[[p]]$ is defined inductively as follows:

1. $[[\text{skip}]](s) = s$
2. $[[x := e]](s) = s[n/x]$, where $n = [[e]](s)$
3. $[[p_1; p_2]](s) = [[p_2]]([[p_1]](s))$.
4. $[[\text{if } T \text{ then } p_1 \text{ else } p_2]](s) = [[p_1]](s)$
if $[[T]](s) = \text{true}$.
5. $[[\text{if } T \text{ then } p_1 \text{ else } p_2]](s) = [[p_2]](s)$
if $[[T]](s) = \text{false}$
6. $[[\text{while } T \text{ do } p]](s) = s$,
if $[[T]](s) = \text{false}$.
7. $[[\text{while } T \text{ do } p]](s) = [[\text{while } T \text{ do } p]]([[p]](s))$,
if $[[T]](s) = \text{true}$.

2.5 INTRODUCTION TO REGULAR EXPRESSION

Regular expression offer something that automata do not: a declarative way to express the strings, we want to accept. Thus regular expression as the input language for many system that string. For example, lexical-analyser generators, such as Lex or Flex.

Definition of regular expression

The set of regular expression is defined by the following rules:

1. Every letter of Σ can be made in to regular expression, null string, ϵ itself is a regular expression.
2. if r_1 and r_2 are two regular expression, then

(i) (r_1)	(ii) $r_1 r_2$
(iii) $r_1 + r_2$	(iv) r_1^*
(v) r_1^+ are also regular expression.	
3. Nothing else is a regular expression.

2.5.1 Building Regular Expression

The algebra of regular expressions follows this path using constants and variables that denote languages, and operators for three operations of union, dot and star. we can describe the regular expression recursively, as follow: In this definition, we not only describe what the legal regular expression are, but for each regular expression r , we describe the language it represent, which we denote $L(r)$. We can consider some facts:

1. The constant ϵ (null string) and ϕ (empty set) are regular expression, denoting the language $\{\epsilon\}$ and ϕ , respectively, that is $L(\epsilon) = \{\epsilon\}$, and $L(\phi) = \phi$.
2. If 'a' is any symbol, then 'a' is regular expression. This expression denotes the language $\{a\}$. That is $L(a) = \{a\}$.
3. A variable, usually capitalized and italic such as L is a variable, representing any Language.

2.5.2 Inductive Step

There are four parts to the inductive step, one for each of three operators and one for the introduction of parentheses.

1. If r_1 and r_2 are regular expressions, then $r_1 + r_2$ is a regular expression denoting the union of $L(r_1)$ and $L(r_2)$.
That is $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
2. If r_1 and r_2 are regular expressions, then $r_1 \cdot r_2$ is a regular expression denoting the concate of $L(r_1)$ and $L(r_2)$. That is, $L(r_1 \cdot r_2) = L(r_1) \cdot L(r_2)$
3. If r is a regular expression then r^* is a regular expression denoting the closure of $L(r)$, that is $L(r^*) = (L(r))^*$
4. If r is a regular expression, then (r) , a parenthesized r , is also a regular expression, denoting the same language are r .

Formaly $L((r)) = L(r)$

Now let us discuss some example of building regular expression for better understanding.

Example 2.6 Write the regular expression over alphabet (a, b, c) containing at least one a and at least one b.

Solution: Let us first analyse the language of regular expression.

Problem is very simple and the language will set of string, like ab, abbbb, abab, aaaab, baaaa ... So strings are 1 a 2 b 3, and any combination of a's, b's and c's is possible at the places 1, 2, and 3 any combination of a, b, c means $(a + b + c)^*$.

So regular expression is

$$r = (a + b + c)^* a (a + b + c)^* b (a + b + c)^* + (a + b + c)^* b (a + b + c)^* a (a + b + c)^*$$

If $r = r_1 + r_2$ (say)

then minimum string of r_1 is ab and minimum string of r_2 is ba.

Example 2.7 Write the regular expression for the set of strings of 0's and 1's whose tenth symbol from the right end is f.

Solution: $\Sigma = \{0, 1\}$ given the set of string, according to the required regular expression are 111000000000, 10011010101010 So clearly we concern about only tenth position and it must be 1, rest 9 places from the right are either 0 or 1. Similarly position 11th on ward from right end may be either 0 to 1.

So regular expression is

$$r = (0 + 1)^* 1 (0 + 1) (0 + 1) (0 + 1) (0 + 1) (0 + 1) (0 + 1) (0 + 1) (0 + 1)$$

$$r = (0 + 1)^* (0 + 1)^9$$

Example 2.8 Write the regular expression for the set of strings of an equal number of 0's and 1's such that in every prefix 1 the number of 0's differs from the number of 1's by at most 1.

Solution: Clearly the set of strings is having strings like 010101010, 10101010, 0110, 1001 ...

So regular expression will be

$$r = (01 + 10)^*$$

Example 2.9 Write a regular expression for the set of strings of 0's and 1's not containing 101 as a substring.

Solution: When we analyse the set of strings then it becomes clear that string may start with '0' and '0' may be repeated, wherever one is encountered then no single 0 must follow it, so strings are like 0000010, 00000111111, 1001001...

$$\text{Regular expression is } r = (0^* 1^* 00) 0^* 1^*$$

Example 2.10 Write a regular expression over alphabet {a, b} for the set of strings with even number of a's followed by odd number of b's that for the language

$$L = \{a^{2n} b^{2m+1} : n \geq 0, m \geq 0\}$$

Solution: Clearly every string of language L will start with even number of a's that is at least two a's should be there in the beginning of the string. String should end with at least three b's (that is odd number of b's)

$$\text{Regular expression is } r = (aa)^* (bb)^* b.$$

Example 2.11 Write the regular expression for the language. $L = \{w \in \{0, 1\}^*: w \text{ has no pair of consecutive zeros}\}$.

Solution: Clearly whenever a '0' occurs, it must be followed by a 1. such a substring may be proceeded and followed by any number of 1's, that is $(1^* 0 1 1^*)^*$. But strings ending in 0 or consisting of all 1's are unaccounted here, that is $1^*(0 + \epsilon)$

$$\text{So regular expression is } r_1 = (1^* 0 1 1^*) (0 + \epsilon) + 1^* (0 + \epsilon)$$

$$\text{Another possible solution is } r_2 = (1 + 01)^* (0 + \epsilon)$$

"Although the two expression look different both answers are correct, as they denote the same language. Generally, there are an unlimited number of regular expression for any given language."

Example 2.12 Write the regular expression for the language

$$L = \{a^n b^m : n \geq 4, m \leq 3\}.$$

Solution: Language contain the set of strings such that every string start with atleast 4 a's and end with atmost 3b's.

$$\text{So regular expression will be } r = a^4 a^* (\epsilon + b + b^2 + b^3)$$

Example 2.13 Write the regular expression for the language

$$L = \{a^n b^m : (n + m) \text{ is even}\}.$$

Solution: We know $(n + m)$ can be even in two cases

Case 1: n and m both are even

So in this case regular expression will be $r_1 = (aa)^* (bb)^*$

Case 2: n and m both are odd

In this case regular expression is $r_2 = (aa)^* a (bb)^* b$

Let regular expression for the language L is r, so

$$\begin{aligned} r &= r_1 + r_2 \\ &= (aa)^* (bb)^* + (aa)^* a (bb)^* b \end{aligned}$$

Example 2.14 Write the regular expression for the language

$$L = \{ab^n w : n \geq 3, w \in (a, b)^*\}.$$

Solution: Clearly every string of language L starts with 'a' followed by atleast three b's, followed by at least one 'a' or one 'b', that is strings are like $ab^3a, bbbbbbbb, abbbb\dots a$.

So regular expression is: $r = ab^3b^* (a + b)^*$

Here + is a positive closure that is $(a + b)^+ = (a + b)^* - \epsilon$

Example 2.15 Write a regular expression for the language

$$L = \{w : |w| \bmod 3 = 0\}, w \in (a, b)^*$$

Solution: Let us first understand the meaning of $|w| \bmod 3 = 0$, when length of string belongs to the language L, is divided by 3 then remainder should be 0, that is length of w must be 0, 3, 6, 9 ... clearly multiple of three.

Regular expression is $r = ((a + b)^3)^*$

Example 2.16 Write the regular expression for the language

$$L = \{w \in (a, b)^*: n_a(w) \bmod 3 = 0\}.$$

Solution: $n_a(w) \bmod 3 = 0$ means, number of a's in string should be 0, 3, 6, 9

So regular expression is: $r = (b^* ab^* ab^* ab^*)^*$.

2.5.3 Comparative Study of Regular Expression, Regular Sets and Finite Automata

Regular expression

\emptyset

Regular set

{}

Finite Automata



Fig. 2.7

ϵ

{ ϵ }



Fig. 2.8

Every a in Σ is a
Regular expression

{a}

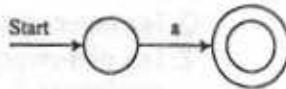


Fig. 2.9

If r_1 and r_2 are regular expressions then $(r_1 + r_2)$ or r_1/r_2 is also regular expression.

$R_1 \cup R_2$ (where R_1 and R_2 are regular set corresponding to r_1 and r_2 respectively)

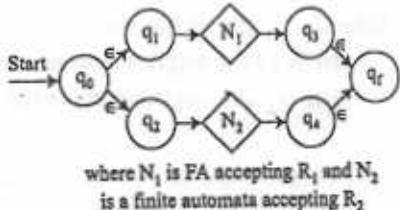


Fig. 2.10

r_1r_2 is a regular expression. R_1R_2 is a regular set

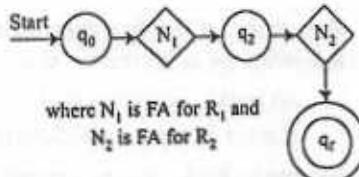


Fig. 2.11

r^* is a regular expression if r is regular expression

R^* (where R is a regular set corresponding to r)

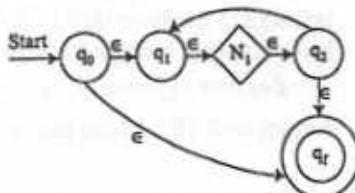


Fig. 2.12

2.6 DETERMINISTIC FINITE AUTOMATA

We know that computer is deterministic, by which we mean that, on reading one particular input instruction, the machine converts it self from the state it was into some particular other state, where the resultant state is completely determined by the prior state and the input instruction. Some sequence of instruction may lead to success and some may not.

Finite automata is called "finite" because number of possible states and number of letter in the alphabet are both finite, and "automation" because the change of the state is totally governed by the input. It is deterministic, since, what is next is automatic not will-full, just as the motion of the hands of the clock is automatic while the motion of hands of a human is presumably the result of desire and thought.

When we analyse above discussion mathematically the following formal version obtained.
A deterministic finite automata is a quintuple

$$M = (Q, \Sigma, \delta, q_0, F)$$

where

Q : Is a non-empty finite set of states presents in the finite control ($q_0, q_1, q_2 \dots$)

Σ : Is a non-empty finite set of input symbols which can be passed to the finite state machine (a, b, c, d, e ...)

q_0 : Is a starting state, one of the state in Q.

F: Is a non-empty set of final states or accepting states, set of final states belongs to Q.

δ : Is a function called transition function that takes two arguments a state and a input symbol, if returns a single state.

Let 'q' is the state and 'a' be input symbol passed to the transition function as:

$$\delta(q, a) = q'$$

q' is the output of the function that is a single state only, q' may be q.

Some people prefer to call this just a finite acceptor because it sole job is to accept certain input strings and rejects others. It does not do anything like printing machine. Some time it is also referred as language recognizer because FA merely recognizes whether the input strings in the language, as we can also recognize that some body is speaking Russian or English without necessarily understanding what it means.

2.6.1 Processing of Strings by DFA

Suppose $a_1, a_2, a_3 \dots, a_n$ is a sequence of input symbols. We start out with deterministic finite automata having $q_0, q_1, q_2, \dots, q_n$ states where q_0 is the intial state and q_n is the final state and transition function and processed as

$$\delta(q_0, a_1) = q_1$$

$$\delta(q_1, a_2) = q_2$$

$$\delta(q_2, a_3) = q_3$$

$$\vdots$$

$$\vdots$$

$$\delta(q_{n-1}, q_n) = q_n$$

Input $a_1, a_2 \dots, a_n$ is said to be accepted since q_n is the number of the final states, and if not then it is "rejected". By adopting this approach we can define the language of a particular DFA, and can also check that particular string is accepted or rejected by given FA.

So a string is said to be accepted by the given FA only when it is processed by transition function δ in such a manner that it ends at the final state.

2.6.2 Simpler Notations for Dfa's

There are two preferred notations for describing automata:

1. Transition Diagram
2. Transition Table.

1. *Transition Diagram Notations*: A transition diagram for DFA,

$M = (Q, \Sigma, \delta, q_0, F)$ is a graph defined as follows:

- (a) For each state in Q there is a node represented by the circle.
- (b) For each state in Q and each input symbol a in Σ , let $\delta(q, a) = P$. If there are several input symbols that cause transition from q to p, then the transition diagram can have one arc, labelled by the list of these symbols as:

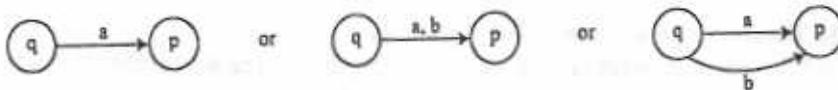


Fig. 2.13

- (c) If any state $q \in Q$ is the starting then it is represented by the circle with arrow as



Fig. 2.14

- (d) Nodes corresponding to accepting states (those are in F) are marked by a double circle. States not in F have a single circle.



Fig. 2.15

2. Transition Tables: A transition table is a conventional, tabular representation of a function like δ that takes two arguments and returns a state. The rows of the table correspond to the states and the columns correspond to the input. The entry for one row corresponding to state q and the column corresponding to input a is the state $\delta(q, a)$. For example:

δ/Σ	a	b
$\rightarrow q_0$	q_1	q_2
q_1	q_1	q_0
$* q_2$	q_2	q_2

Fig. 2.16

q_0 is the starting state and q_2 is the final state,

$$Q = \{q_0, q_1, q_2\}$$

$$F = \{q_2\}$$

$$\delta(q_0, a) = q_1, \quad \delta(q_0, b) = q_2$$

$$\delta(q_1, a) = q_1, \quad \delta(q_1, b) = q_0$$

$$\delta(q_2, a) = q_2, \quad \delta(q_2, b) = q_2$$

Let us draw the transition diagram for the transition table showing Fig. 2.17.

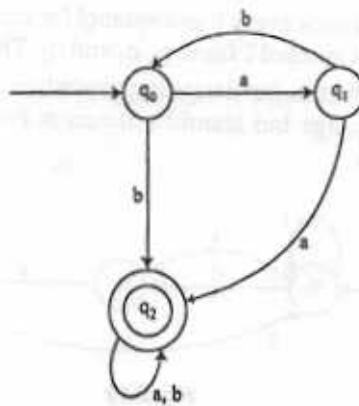


Fig. 2.17

2.6.3 The Language of A Dfa

It have been explained informally that DFA defines a language: The set of all strings that result in a sequence of state transition from start state to an accepting state.

Now we can define the language of a DFA $M = (Q, \Sigma, \delta, q_0, F)$. This language is denoted by $L(m)$, and defined by

$$L(m) = \{w / \hat{\delta}(q_0, w) \text{ is in } F\}$$

$\hat{\delta}$ is extended transition function.

That is the language of m is the set of strings (w) that take the start q_0 to one of the accepting states. If L is the language of any finite automation, then we say that L is a regular language.

Now for practical understanding of FA in lexical analysis, let us discuss the some examples.

Example 2.17 Design a FA that accepts set of strings such that every string ends in 00 over alphabet {0, 1}.

Solution: Let DFA be $M = (Q, \Sigma, \delta, q_0, F)$

q_0 : intial state and

$\Sigma : \{0, 1\}$ is given

According to the given language, we have to design a FA which accepts the strings $W = W'00$

where W' can be any combination of 0's and 1's for example 011100, 111100, 00, 100, 01010100 etc.

Now first we will decide the approach to design FA. it is not an easy task to think FA as a whole, so we have to fulfill minimum condition that is every string end in 00.

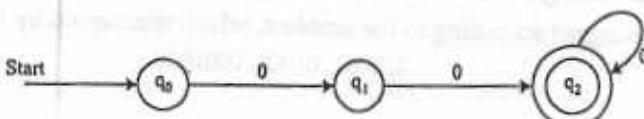


Fig. 2.18

Being the DFA, we must check every input symbol for output state from every state. So we have to decide output state at symbol 1 from q_0 , q_1 and q_2 . Then it will be complete FA.

It is important to know that in the designing procedure, FA must accept all these strings which are in the given language but should not accept those strings which are not in the language of finite Automata.

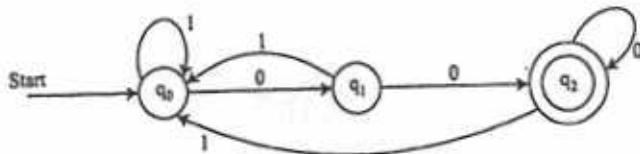


Fig. 2.19

So Fig. 2.18 satisfies the minimum condition and Fig. 2.19 is the transition diagram of required FA.

$$Q = \{q_0, q_1, q_2\}$$

$$F = \{q_2\}.$$

Example 2.18: Design a FA which accepts set of strings containing four 1's in every string over alphabet

$$\Sigma = \{0, 1\}.$$

Solution: DFA should accept the strings 1111, 0101011, 0110110000

Let Dfa be $M = (Q, \Sigma, \delta, q_0, F)$

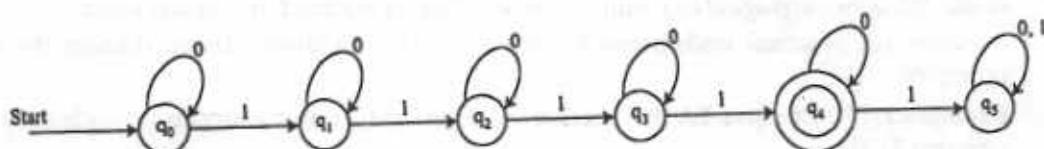


Fig. 2.20

$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$$

$$F = \{q_4\}$$

From q_0 to q_4 four 1's are already there and q_4 is final state so 1 from q_4 should go on trap path, that is why q_5 is there so q_5 is called trap or dead state.

"Dead states are those non final states which transits in itself for all the input symbol. So q_5 is the example of dead state".

Example 2.19 Design a FA that accepts strings containing exactly 1 over alphabet {0, 1}.

Solution: String set according to the problem, which is accepted by the FA is like.

$$1, 0001, 01000, 0001000, \dots$$

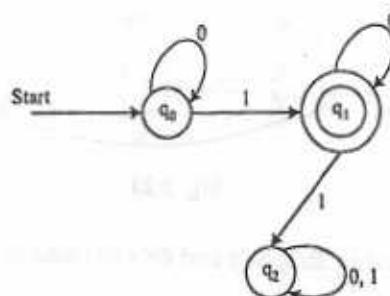


Fig. 2.21

Example 2.20 Design FA for the language

$$L = \{(01)^i 1^j / i \geq 1, j \geq 1\}.$$

Solution: By analysis of language L , it is clear that FA will accept strings start with any number of (not empty) and end with even number of 1's. Let FA be

$$M = (Q, \Sigma, \delta, q_0, F)$$

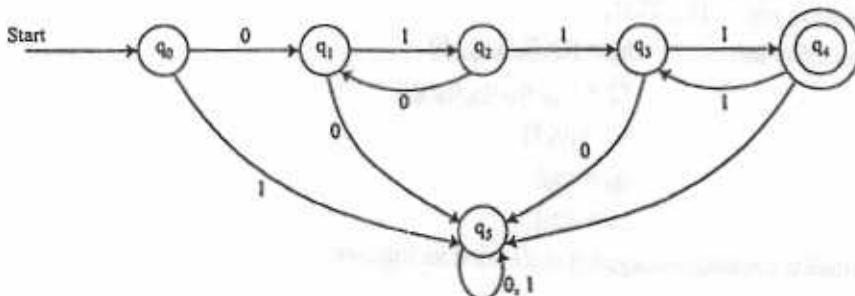


Fig. 2.22

Here

$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}, F = \{q_4\}, \Sigma = \{0, 1\}$$

Example 2.21 Design a DFA for the language

$$L = \{w \in (a, b)^*/ n_b(w) \bmod 3 > 1\}.$$

Solution: Let us first analyse the language L .

$n_b(w)$: number of b's in every string of language L . $n_b(w) \bmod 3$: number of b's in each string of L is divided by three and remainder is result of $n_b(w) \bmod 3$.

$n_b(w) \bmod 3 > 1$: mean that remainder is only 2 since when number is divided by 3 then possible remainder are 0, 1, and 2.

Let required DFA be $m = (Q, \Sigma, \delta, q_0, F)$

$$\Sigma = (a, b) \text{ given.}$$

and transition diagram is as follows

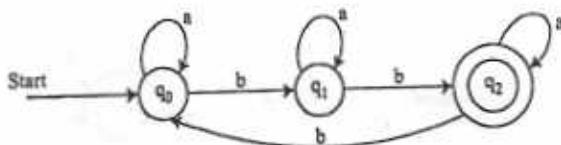


Fig. 2.23

it will accept strings, having 2b's, 5b's and 8b's so remainder is always 2.

$$Q = \{q_0, q_1, q_2\}$$

$$F = \{q_2\}$$

Example 2.22 Construct a DFA that accepts strings on $\{0, 1\}$, if and only if the value of the strings interpreted as binary representations of an integer, is zero modulo five. For example, 0101 and 11, representing the integers 5 and 15, respectively, are to be accepted.

Solution: Let us understand problem first. We have to design a automation which accepts those binary string, when interpreted as integers then those integers are divisible by 5 that means after dividing 5 remainder is 0, if is the only meaning of the “zero module five”. So clearly integer interpretation should be 0, 5, 15, 20, 25 ... so binary strings which are accepted by automation are 0, 101, 1111,

Let DFA be

$$M = (Q, \Sigma, \delta, q_0, F)$$

$$Q = \{q_0, q_1, q_2, q_3, q_4\}$$

$$\Sigma = \{0, 1\}$$

$$q_0 = \{q_0\}$$

$$F = \{q_0\}$$

and finally transition diagram is defined as follows:

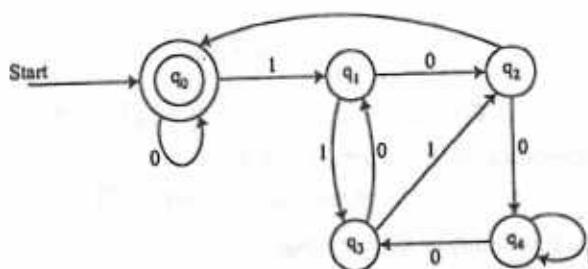


Fig. 2.24

2.7 WHY STUDY COMPILERS

You may never write a commercial compiler, but that's not why we study compilers. We study compiler construction for the following reasons:

- Writing a compiler gives a student experience with largescale applications development. Your compiler program may be the largest program you write as a

student. Experience working with really big data structure and complex interactions between algorithms will help you out on your next big programming projects.

- Compiler writing is one of the shining triumphs of CS theory. It demonstrates the value of theory over the impulse to just "check up" a solution.
- Compiler writing is a basic elements of programming language research. Many language researchers write compilers for the language they design.
- Many applications have similar properties to one or more phases of a compiler, and compiler expertise and tools can help an application programmer working on other projects beside compilers.

2.7.1 What is the Challenge in Compiler Design?

Compiler writing is not an easy job. It is very challenging and you must have lot of knowledge about various field of computer science. Let us discuss some of challenges:

(a) *Many variations:*

- Many programming Languages (FORTRAN, C++, Java)
- Many programming paradigm (that is object oriented, functional, logical)
- Many computer architectures (that is MIPS, SPARC, Intel, alpha)
- many operating system (that is linux, solaris, windows)

(b) *Qualities of a Good compiler (in order of importance):*

- The compiler itself must be bug-free
- It must generate correct machine code
- The generated machine code must run fast
- The compiler it self must run fast (compilation time must be proportional to program size)
- The compiler must be portable (that is modular, supporting separate compilation)
- It must print good diagnostics and error messages.
- The generated code must work well with existing debuggers.
- must have consistent and predictable optimization.

(c) *Building a compiler requires knowledge of:*

- programming languages (parameter passing, variable scoping, memory allocation etc.)
- Theory of automata, context free languages etc.
- Algorithms and data structure (hash tables, graph algorithms, dynamic programming etc).
- Computer architecture (assembly programming)
- Software engineering.

2.7.2 The Components of a Compiler

A compiler is composed of several components also called phases, each performing one specific task. Each of these components performs at a slightly higher Language level than the

previous. The compiler must take an arbitrarily – formed body of text, and translate it into a binary stream that computer can understand.

The complete compilation procedure can be divided in to six phases and these phase can be regrouped in two parts, as follows:

1. Analysis:

In this part source program is broken into constituent pieces and creates an intermediate representation. Analysis can be done in three phases

- (a) Lexical analysis
- (b) Syntax analysis
- (c) Semantic analysis

2. Synthesis

Synthesis constructs the desired target program from intermediate representation. Synthesis can be done in following three phases:

- (a) Intermediate code generation
- (b) code optimization
- (c) code generator.

Every phase of compilation can interact with a special data structure called symbol table and with error handle.

Now let us draw the block diagram of compiler.

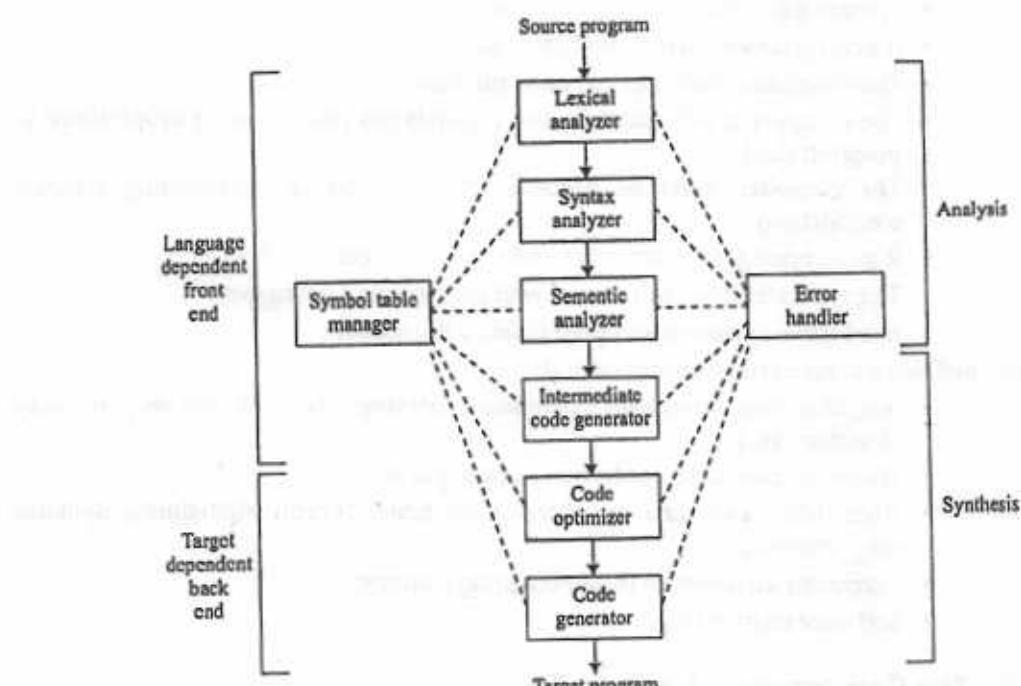


Fig. 2.25

Sometimes we divide synthesis as follow:

- (a) Intermediate code generation
 - (b) code optimization.
 - (c) storage allocation
 - (d) code generation

Here we are looking storage allocation as a different phase, but in previous assumption it was the part of code generation.

Now Block diagram can be drawn as follows:

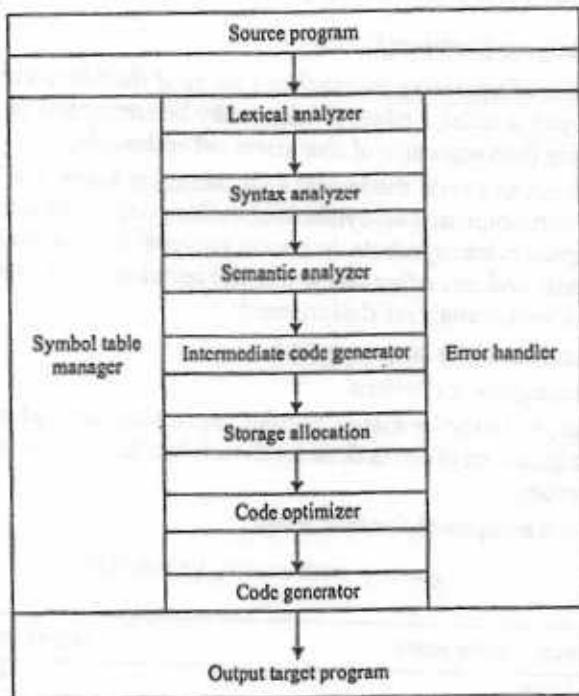


Fig. 2.26

2.7.2.1 Front and Back Ends

The front end of a compiler includes all analysis phases and the intermediate code generator while the back end includes the code optimization and final code generation phases: the front end analyzes the source program and produces intermediate code while the back end synthesizes the target program from the intermediate code.

It is an advantage to use the same intermediate code representation for several different source languages and/or several different target languages: One can then combine any front end with any back end. For example metroworks developed a compiler with four front ends to analyze source programs written in pascal, C, C++ or Java and four back end to synthesize

machine code for a macintosh with a 68k processor, a macintosh with a power processor, (PC), a PC using windows 95, or a PC using windows NT: it's really 16 different compilers even though only 4 front end with 4 back end are written modern compilers eliminate the storage problems using syntax-directed translation to interleave the actions of phases. The syntax analyzes directs the whole process the lexical analyzer is a subroutine that produces just one token each time it is called by the syntax analyzer, and the action of the semantic analyzer and the intermediate code generator are built inside the syntax analyzer. At any time only a small fraction of the syntax tree is stored in side the computer so even a very large source program can be compiled without the need for temporary disk files.

Now we will study these phases of complier design in detail as follows:

2.7.2.2 Lexical Analyzer (Scanner)

This module has the task of separating the continuous string of characters into distinctive groups that make sense. Such a group is called a token. A token may be composed of a single character or a sequence of characters (this sequence of characters called lexeme).

In order to perform this task, the lexical analyzer must know the key words, identifiers, operators, delimiters and punctuation symbols of the language to be implemented. At the same time as forming characters into symbols the lexical analyzer should deal with (multiple) spaces and remove comments and any other characters not relevant to the later stages of analysis. So now we can say that lexical analyzer design most:

- (a) Specify the token of the language, and
- (b) efficiently recognize the tokens.

To specify the token of the language, regular expression concept from automata theory is used and for recognition of token is done by deterministic finite automata (we will see these concepts in text chapter).

Now let us do the analysis of the following:

Sum: = old sum - value/100

Lexeme (collection of characters)	Token (category of lexeme)
sum	identifier
-	assignment operator
oldsum	identifier
-	subtraction operator
value	identifier
/	division operator
100	integer constant

Let see another example

Position : = initial + rate × 70

This statement grouped in to 7 tokes as follows:

Lexeme	Token
Position	identifier
:=	assignment operator
initial	identifier
+	addition operator
rate	identifier
*	multiplication operator
> 0	integer constant

Fig. 2.27

After recognizing the tokens lexical analyzer pass them to syntax analyzer. Another way of looking at it is that the lexical analyzer usually has no context to work with. As it is processing one symbol it has no knowledge of any of the symbols that preceded or will follow this symbol. There are widely available tools that can be used for producing a lexical analyzer for a language (we will see some of them).

2.7.2.3 Syntax Analyser (Parser)

Syntax analyzer is the module in which the overall structure is identified, and involves an understanding of the order in which the symbols in a program may appear.

We know that it was the scanner's duty to recognize individual words, or tokens of the language. The lexical analyzer does not, however recognize if these words have been used correctly. The main task of parser is to group the tokens in to sentences, that is to determine if the sequence of tokens that have been extracted by the syntax analyzer are in the correct order or not. In other words, until the syntax analyzer (parser) is reached the tokens have been collected with no regard to the whole context of the program as a whole. The parser analyzes the context of each token and groups the token in declarations, statements and control statement. *In the process of analyzing each sentence, the parser builds abstract tree structures.*

A tree is a useful representation of a program segment because it facilitates transformations of the program that may lead to possible minimization of the machine instructions that are needed to carry out the required operations that is optimization.

Let see the procedure by an example:

sum: = old sum - value/100

There are 7 tokens, sum (identifier), := (assignment operator), old sum (identifier), '-' (subtraction operator), value (identifier), /(division operator) and 100 (integer constant) as output of lexical analysis. Now these tokens works as input to the parser.

Parser will produce syntax tree as shown in Fig. 2.28.

So we can say that syntax analysis is not only the key phase of the analysis stage of compilation but provides the frame work for the compiler as a whole. It drives the lexical analysis phase and builds the structure upon which semantic analysis is performed. Syntax analysis also provides a frame work for a whole range of source code analysis tools, including measuring tools of various sorts, cross-references, layout tools and so on.

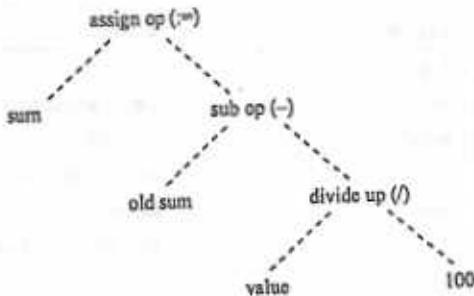


Fig. 2.28 Syntax tree as output of parser

2.7.2.4 Semantic Analyzer

Some Features of programming cannot be checked in a single left to right scan of the source code without setting up arbitrary sized tables, to provide access to information that may be an arbitrary distance away, when it required. For example, information concerning the type and scope of variables falls in to this category.

The semantic analyzer gathers type information and checks the tree produced by the syntax analyzer for semantic errors. Let us see the statement; and consider rate is real.

sum: = old sum + rate × 60;

Here in this statement the semantic analyzer might add a type conversion node, say inttoreal, to the syntax tree to convert the integer to real quantity.

Output of syntax analysis:

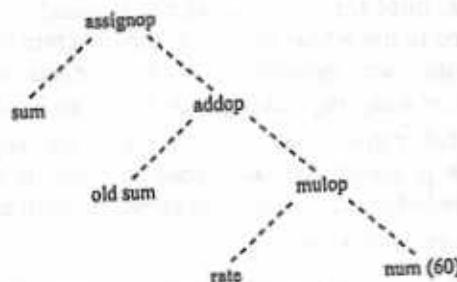


Fig. 2.29

Now semantic analyzer comes in the picture and output will be as shown in Fig. 2.30.

2.7.2.5 Symbol Table Manager

Though the parser determines the correct usage of tokens, and whether or not they appeared in the correct order, it still did not determine whether or not the program said anything that made sense. This type of checking occurs at the semantic level. In order to perform this task, the compiler makes use of a detailed system of lists, known as the symbol tables. The compiler needs a symbol table to record each identifier and collect information about it.

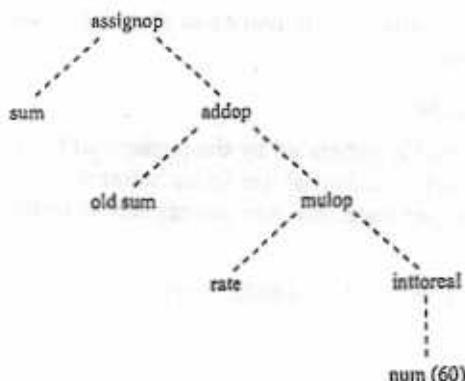


Fig. 2.30

For a variable the symbol table might record its type expression (integer, float, etc.), its scope (where the variable can be used), and its location in run-time storages, etc. For a procedure the symbol table might record the type-expression of its arguments and the type-expression of its returned value.

The symbol table manager has a FIND function that returns a pointer to the descriptor for an identifier (descriptor is a record which contains all the information about identifier) when given its lexeme. Compiler phases use this pointer to read and/or modify information about the identifier. FIND returns a NULL pointer if there is no record for a given lexeme. The INSERT function in the symbol table manager inserts a new record into the symbol table when given its lexeme.

2.7.2.6 Intermediate Code Generator

After semantic analysis many compilers generate an Intermediate representation of the source program that is both easy to produce and easy to translate in to the target program. There are a variety of forms used for intermediate code. One such form is called three-address code and looks like code for a memory-memory machine where every operation reads operands from memory and writes results into memory. The intermediate code generator usually has to create temporary locations to hold intermediate results.

sum := old sum + rate * 60;

After the compiler broke this stream of text down in to tokens, parsed the tokens and created the tree and finally performed the necessary type checking, it might create a sequence of three-address instructions. Three-address instructions contain no more than three operands (registers) per instruction, and in addition to an assignment, contain only one other operator. Three - address instructions also assume on unlimited supply of registers.

```

temp1: = inttoreal (60)
temp2: = rate * temp1
temp3: = oldsum + temp2
sum: = temp3
  
```

where temp , temp_2 and temp_3 are the names of three temporary locations created by the intermediate code generator.

2.7.2.7 Code Optimization

The structure of the tree that is generated by the parser can be rearranged to suit the needs of the machine produce an object code that run faster. what results is a simplified three that may be used to generate the object code. An example of such an optimization is the following FOR LOOP:

```
for (int = i = 0; i < 2000; i++)
{
    x: = y + 3;
    q: = a * 1;
}
```

In this loop there are instructions that are not related at all to the variables a of the for statement, such as $x: = y + 3$. In the above program this instruction is executed unnecessarily 2000 times. The object code can be generated such that the machine code for that line is moved outside the for's body, (either before or after the for command). The resulting object code is more efficient at run time because the loop has been significantly tightened, and the above mentioned instruction is executed only once, instead of 2000 times. This is called *loop optimization*.

In intermediate code generation phase, we have seen three-address instructions as follows:

```
temp1 := inttoreal (60)
temp2 := rate × temp1
temp3 := old sum + temp2
sum := temp3
```

We can optimize the code, by applying abstract algebraic rules to it. A subsequent optimization of the above code might be:

```
temp1 := rate × inttoreal (60)
sum := old sum + temp1
```

This is some time called *local optimization*. Once the intermediate code has been optimized, the compiler can generate the final code in its machine - readable form.

2.7.2.8 Storage-allocation

Every constant and variable appearing in the program must have storage space allocated for its value during the storage allocation phase. This storage space may be one of the following types:

(a) *Static storage*:

If the life time of the variable is life time of the program and the space of its value once allocated cannot later be released. This kind of allocation is called static storage allocation.

(b) *Dynamic storage:*

We follow Dynamic storage allocation if the lifetime is a particular block or functions or procedure in which it is allocated so that it may be released when the block or functions or procedure in which it is allocated is left.

(c) *Global storage:*

If its lifetime is unknown at compile time and it has to be allocated and deallocated at run time, the efficient control of such storage usually implies run - time overheads.

After space is allocated by the storage allocation phase, an address, containing as much as is known about its location at compile time, is passed to the code generator for its use.

2.7.2.9 Code Generation

This is the part of the compiler where native machine code is actually generated. If the target machine has registers the target program. For example for statement

sum := old sum + rate × 60

Now we can understand that sum, old sum and rate are identifier, let us say them id_1 , id_2 and id_3 .

That is $id_1 := id_2 + id_3 \times 60$
 MOVF id_3 , R₂
 MULF #60.0, R₂
 MOVE id_2 , R₁
 ADDF R₂, R₁
 MOVE R₁, id_1

During this step, the compiler has to map address names from the three - address intermediate code onto the very finite amount of registers had by the machine. the final result is piece of code that is (mostly) executable.

None of the remaining phases prior the actual execution of the program are a part of the compiler's responsibility. what remains to be done is to link the program and build a binary executable. In this phases, various libraries as well as the main executable portion of the program are combined to form a fully-fledged application. There are various types of linking most of which are done prior to runtime. The final phase that a program must pass through prior to execution is for it to be loaded in to memory. In this phase, the addresses in the binary code are translated from logical addresses, and any final runtime binding are performed.

2.7.2.10 Peephole Optimization

The object code can be further optimized by a peephole optimizer. This process is present in a more sophisticated compiler. The purpose is to produce a more efficient object program that can be executed in a shorter time. Optimization is performed at a local level as it takes advantage of certain operator properties such as commutativity, associativity, and distributivity. This topic is very advanced and still research is going on. To see a summary of the entire process of compilation from start to finish, refer to following figure.

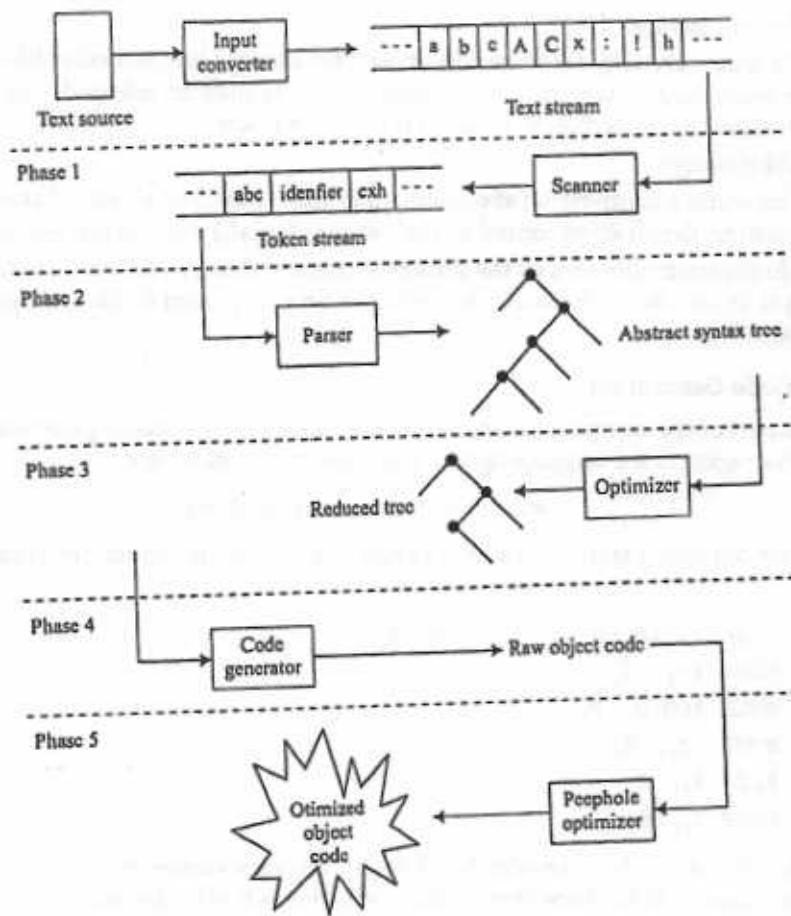


Fig. 2.31

2.7.2.11 Error Handler

Detection and reporting of errors in source program is main function of compiler. Error may occur at any phase of compilation. A good compiler must determine the line number of program exactly, where the errors have occurred.

Let us see the various errors which may occur at different levels of compilation.

- *The First of these are lexical (scanner) errors.* Some of the most common types here consist of illegal or unrecognized characters, mainly caused by typing errors. A common way for this to happen is for the programmer to type a character that is illegal in any instance in the language, and is never used. Finally the quite commonly, another type of error that the scanner may detect is an undetermined character or string constant. This happens whenever the programmer types some thing in quotes and forget the trailing quote. Again, these are mostly typing errors.

- The second class of error is syntactic in nature, and is caught by the parser. These errors are among the most common. The really difficult part is to decide from where to continue the syntactical analysis after an error has been fund. What happens if the parser is not carefully written, or if the error detection and recovery scheme is sloppy, the parser will lit one error and "mess up" after that, and cascade spurious error messages all throughout the rest of the program. In the case of an error, what one would like to see happening, is to have the compiler skip any improper tokens, and continue to detect errors without generating error messages that are not really an error but a consequence of the first error. This aspect is so important that some compilers are categorized based on how good their error detection system is.
- The third type of error is semantic in nature. The semantic that are used in computer languages are by far simpler than the semantics that are uses in spoken languages. This is the case because in computer languages every thing is very much defined, there are no nuances implied or used. The semantic errors that may occur in a program are related to the fact that some statements may be correct from the syntactical point of view, but they make no sense and there is no code that can be generated to carry out the meaning of the statement.
- The fourth type of error may encounter during code optimization for example in control flow analysis, there may exists some statements, which can never be reached.
- The fifth type of error may occur during code generation, since in code generation architecture of computer also play an important role. For example there may exist a constant that is too large to fit in a word of the target machine.
- The sixth type of error may encounter when compiler try to make symbol table entries, for example there may exist an identifier that has been multiple declaration with contradictory attributes.

2.7.3 The Number of Compile Passes

We have seen the structure of compiler. The structure of the compiler is divided in to six phases. Each phase have it's own unique job to perform. The output of one phase is used as input to another phase. We have already seen that phases can be viewed as analysis phase and synthesis phases. On the basis of regrouping of phase compilers may multipass or one pass compilers.

2.7.3.1 Multi-Pass Compilers

We have already studies a compiler that scan the input source once, produces a first modified form, then scan the first-modified form and produce a second-modified form and so on, until the object form is produced. Such a compiler is called a multi pass compiler. In the case of the multi-pass compiler each function of the compiler can be performed by one pass of the compiler. For instance the first pass can read the input source, scan and extract the pass can read the input source, scan and extract the tokens and store the result in an output file. The second pass can read the file was produced in the first pass, do the syntactical analysis by building a syntactical tree, and associate all the information relating to each node of the tree. The output of the second pass, then is a file containing the syntactical tree. The third pass can read the output file produced by the second pass and perform the optimization by restructuring the tree structure.

2.7.3.2 One-pass Compilers

In a one-pass compiler, when a line source is processed, it is scanned and tokens are extracted. Then the syntax of the line is analyzed and the tree structure and some tables containing information about each token are built. Finally, after the semantical part is checked for correctness, the code is generated. The same process is repeated for each line of code until the entire program is compiled. Usually the entire compiler is built around the parser, which will call procedures that will perform different functions.

2.7.3.3 Advantages and Disadvantages for Both Single and Multi Pass Compilers

- A one pass compiler is fast, since all the compiler code is loaded in the memory at once. It can process the source text without the overhead of the operating system having to shut down one process and start another. Also, the output of each pass of a multi-pass compiler is stored on disk, and must be read in each time the next pass starts.
- On the other hand, a one pass tends to impose some restrictions upon the program: constants, types, variables, and procedures must be defined before they are used. A multi pass compiler does not impose this type of restrictions upon the user. Operation that can not be performed because of the lack of information can be deferred to the next pass of the compiler when the text has been traversed and the needed information made available.
- The components of a one-pass compiler are interrelated much closer than the components of a multi pass compiler. This requires all the programmers working on the project to have knowledge about the entire project. A multi pass compiler can be decomposed into passes that can relatively independent, hence a team of programmers can work on the project with little interaction among them. Each pass of the compiler can be regarded as a mini-compiler, having an input source written in one intermediary language, and producing an output written in another intermediary language.

EXERCISE

1. Define syntax and semantic with example of C, C++ and Java Programming language.
2. Using the following unambiguous grammar:

```

<assign> → <id> = <expr>
<id> → A/B/C
<expr> → <expr> + <term>/<term>
<term> → <term> × <factor>/<factor>
<factor> → (<expr>)/<id>
    
```

Show a parse tree and a leftmost derivation for each of the following statement:

- (i) $A = A \times (B + (C \times A))$
- (ii) $B = C \times (A \times C + B)$
- (iii) $A = A \times (B + (C))$

3. Prove that the following grammar is ambiguous.

$$\begin{aligned} <S> &\rightarrow <X> \\ <X> &\rightarrow <X> + <X>/<\text{id}> \\ <\text{id}> &\rightarrow a/b/c. \end{aligned}$$

4. Consider the following grammar:

$$\begin{aligned} <S> &\rightarrow <A> \ a \ \ b \\ <A> &\rightarrow <A> \ b/b \\ &\rightarrow a/a. \end{aligned}$$

Which of the following sentences are in the language generated by this grammar?

- (i) baab
- (ii) bbbbab
- (iii) bbaab

5. Write denotational semantics mapping function for C switch?

6. Design the FA for the following C languages:

- (i) identifier
- (ii) constant
- (iii) relops.

7. Write a C program for the FA of constant.

8. What is the use of DFA and regular expression in lexical analysis?

10. Consider the following grammar:

$$\begin{aligned} S &\rightarrow (L)/a \\ L &\rightarrow L, \ s/s \end{aligned}$$

- (i) What are the terminals, nonterminals and start symbol?
- (ii) Find parse trees for the following sentences:
 - (a) (a, a)
 - (b) (a, (a, a))
 - (c) (a, (a, a), (a, a))

11. Give generalized structure of a compiler and discuss the syntax analysis phase in brief.

Basic Parsing Techniques

3.1 INTRODUCTION OF PARSERS

A parser for any grammar is a program (and parsing is a technique) that takes as input string W and produces as output either a parse tree for W , if W is a valid sentence of grammar, or an error message indicating that W is not a valid sentence of given grammar.

The goal of parsing is to determine the syntactic validity of a source string. If the string is valid, a tree is built for use by subsequent phases of the computer. The tree reflects the sequence of derivations or reduction used during the parse. Hence it is called parse tree. If a string is invalid, the parser has to issue diagnostic message identifying the nature and cause of the error(s) in the string.

Every elementary subtree in the parse tree corresponds to a production of the grammar. There are two ways of identifying an elementary subtree-

- By deriving a string from a Nonterminal, or
- By reducing a string of symbols to an Nonterminal.

Based on these, two fundamental approaches to parsing, namely *top down parsing* and *bottom up parsing*, are defined.

There are universal parsing methods that will parse any grammar but they are too inefficient to use in compilers. Almost all programming languages have such simple grammars that an efficient top-down or bottom-up parser can parse a source program with a single left-to-right scan of the input.

For example The production rules of grammar G is:

```
list → list + digit/list - digit / digit  
digit → 0/1/ ... / 9
```

Given token is 9 - 5 + 2

Parse tree is shown in Fig. 3.1.

Each node in the parse tree is labeled by a grammar symbol. The interior node corresponds to the left side of the production. The children of the interior node corresponds to the right side of production.

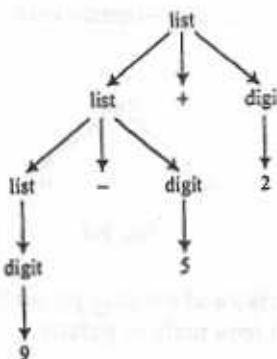


Fig. 3.1

The language defined by above parse tree is lists of digits separated by plus and minus signs.

For any language that can be described by CFG, the parsing requires $O(n^3)$ time to parse string of n token. However, most programming languages are so simple that a parser requires just $O(n)$ time with a single left-to-right scan over the input string of n tokens.

3.2 TOP-DOWN-PARSING

For a given input string, top-down-parsing attempt to derive a string identical to it by successive application of grammar rules to the grammar's distinguished symbol. When such a string is obtained, a tree representing its derivation would be the syntax tree of the input string. Thus if w is the input string, a top-down parser determines a derivation sequence.

$$S \Rightarrow \dots \Rightarrow \dots \Rightarrow w$$

where S is starting Non-terminal in given grammar,

Basically Top-down parsing attempts to find the left-most derivations for the input string w , since string w is scanned by the parser left to right, one symbol/token at a time, and the left-most derivations generate the leaves of the parse tree in left-to-right order, which matches the input scan order.

Example 3.1 Consider the following grammar.

$$V_N = \{\text{expr, term, rest}\}$$

$$V_t = \{+, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, t\}$$

$$\begin{aligned} \text{expr} &\rightarrow \text{term rest} \\ \text{rest} &\rightarrow + \text{ term rest} / - \text{ term rest} / t \\ \text{term} &\rightarrow 0/1/2/3/4/5/6/7/8/9 \end{aligned}$$

and show the construction of the parse tree for the input string: $9 - 5 + 2$.

Solution: Initialization: The root of the parse tree must be starting symbol (Non-terminal) of the grammar, expr .

$$\begin{array}{c} \downarrow \\ \text{expr} \end{array}$$

Step 1: The only production for expr is $\text{expr} \rightarrow \text{term rest}$ so the root node must have a *term node* and a *rest node* as children.

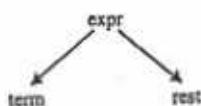


Fig. 3.2

Step 2: The first token in the input is 9 and the only production in the grammar containing a 9 is: $\text{term} \rightarrow 9$ so 9 must be a leaf with *term node* as parent.

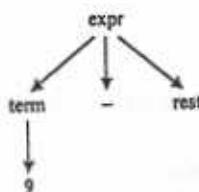


Fig. 3.3

Step 3: The next token in the input is the minus-sign and the only production in the grammar containing a minus-sign is: $\text{rest} \rightarrow \text{term rest}$. The rest node must have a minus-sign leaf, a *term node* and a *rest node* as children.

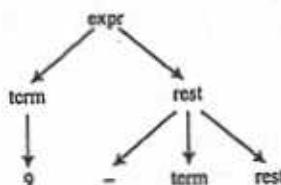


Fig. 3.4

Step 4: The next token in the input is 5 and the only production in the grammar containing a 5 is: $\text{term} \rightarrow 5$ so 5 must be a leaf with a *term node* as parent.

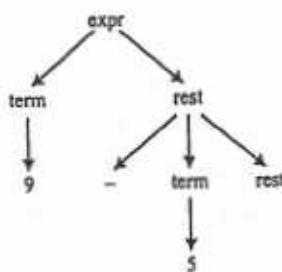


Fig. 3.5

Step 5: The next token in the input is the plus-sign and the only production in the grammar containing a plus-sign is: $rest \rightarrow + term rest$. A *rest* node must have a plus-sign leaf, a *term* node and a *rest* node as children.

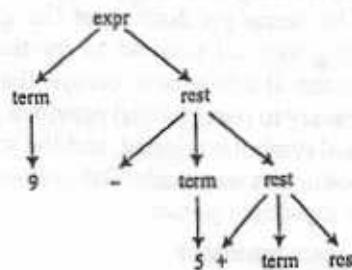


Fig. 3.6

Step 6: The next token in the input is 2 and the only production in the grammar containing a 2 is: $term \rightarrow 2$ so 2 must be a leaf with a *term* node as a parent.

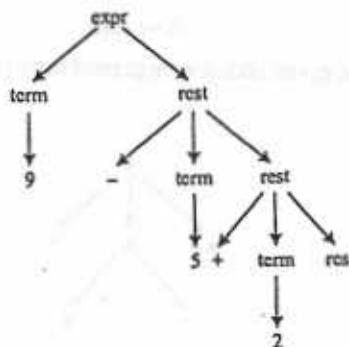


Fig. 3.7

Step 7: The whole input has been absorbed but the parse tree still has a *rest* node with no children the $rest \rightarrow production$ must now be used to give the *rest* node the empty string as a child.

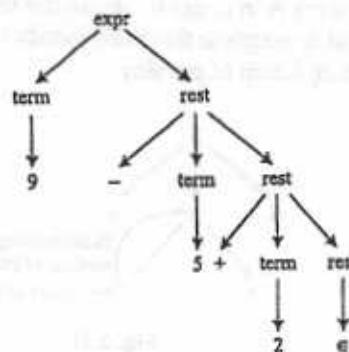


Fig. 3.8

3.2.1 Backtracking a Problem of Top-down Problem

Now by the study of concept of top-down parsing, it is clear that top-down parsing attempts to find the left-most derivations for input string. Basically in top-down mechanism, every terminal symbol generated by some production of the grammar (which is predicted) is matched with the input string symbol pointed to by the string marker. If the match is successful, the parse can continue. If a mismatch occurs, then predictions have obviously gone wrong. At this stage, it is necessary to reject (some) previous predictions. The prediction which led to the mismatching terminal symbol is rejected, and the string marker (pointer) is reset to its position when the rejected production was made. This is known as *backtracking*. Backtracking is one of the major drawback of top-down parser.

Example 3.2 Consider the given grammar

$$S \rightarrow aAb$$

$$A \rightarrow cd/c$$

and show the backtracking for string $W = acb$.

Solution: Let us start with

$$S \rightarrow aAb$$

'a' of aAb matches with 'a' of given string w and next input symbol is 'c', so use the production.
 $A \rightarrow cd$

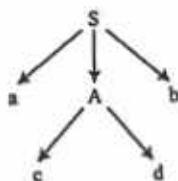


Fig. 3.9

next input symbol is b in string w and parser now compares b to the label of next leaf. If the label does not match d, it reports failure and goes back to A, the parser will also reset the input pointer to the second input symbol – the position it had when the parser encountered A, and it will try a second alternative for A in order to obtain the tree. In the leaf 'c' matches the second symbol, and if the next leaf b matches the third symbol of w, then the parser will halt and announce the successful completion of parsing.

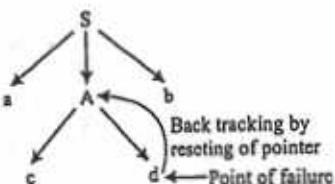


Fig. 3.10

Now finally correct alternative is as follows:

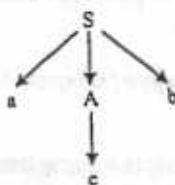


Fig. 3.11

yield is acb which matches to $w = acb$.

Example 3.3 Consider the following grammar

Statement \rightarrow if expression then statement else statement
statement \rightarrow if expression then statement

Now alternate the production so that it may free from the backtracking.

Solution: When the input token is an *if token* should a top-down parser use the first or second production? The parser would have to guess which one to use, continue parsing, and later on, if the guess is wrong, go back to the *if taken* and try the other production.

Usually one can modify the grammar so a predictive top-down parser can be used: the parser always pick the correct production in each step of the parse so it never has to backtrack. To allow the use of a predictive parser, one replaces the two productions of the given grammar with:

Statement \rightarrow if expression then statement A
A \rightarrow else statement/ ϵ

3.2.2 Issues of CFG for the Programming Languages

Issues to resolve when writing a CFG for a programming languages.

- (a) Ambiguity
- (b) Left recursion
- (c) Indirect Left Recursion
- (d) Left Factoring

CFG is very important in compiler design and it can be discussed as follows:

- (i) CFG gives a precise and easy to understand syntactic specification of a programming language.
- (ii) Efficient parser can be automatically constructed from a CFG.
- (iii) A carefully defined grammar can make the detection of errors and code generation easier.
- (iv) compilers based on CFG's are easy to modify.

3.2.2.1 Ambiguity

We have already studied the concept in previous chapter but let us refresh the concept again. Consider the following grammar:

$$\begin{aligned} \langle \text{expr} \rangle &\rightarrow \langle \text{expr} \rangle + \langle \text{expr} \rangle / \langle \text{expr} \rangle \times \langle \text{expr} \rangle / (\langle \text{expr} \rangle) / \langle \text{id} \rangle \\ \langle \text{id} \rangle &\rightarrow a/b/c \end{aligned}$$

The general result of syntax analysis is a parse tree description of the statements of a source program.

Let given string be $w = a + b \times c$ then there exist two parse trees as follows:

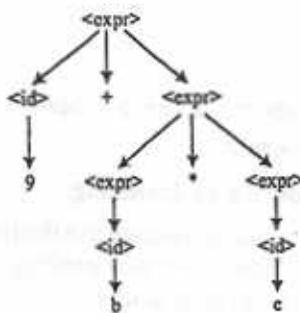


Fig 3.12 Parse tree 1

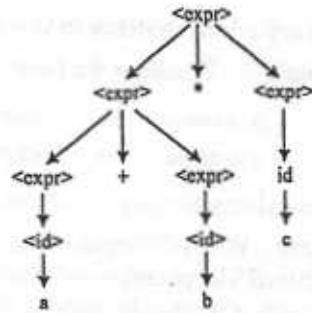


Fig 3.13 Parse tree 2

yield of both the parse tree is $a + b \times c$ so given grammar is ambiguous.

Now we can resolve the ambiguity by re-writing the grammar:

$$\begin{aligned} \langle \text{expr} \rangle &\rightarrow \langle \text{term} \rangle + \langle \text{expr} \rangle / \langle \text{term} \rangle \\ \langle \text{term} \rangle &\rightarrow \langle \text{term} \rangle \times \langle \text{factor} \rangle / \langle \text{factor} \rangle \\ \langle \text{factor} \rangle &\rightarrow (\langle \text{expr} \rangle) / \langle \text{id} \rangle \\ \langle \text{id} \rangle &\rightarrow a/b/c \end{aligned}$$

Now only one parse tree is possible for same string $a + b \times c$

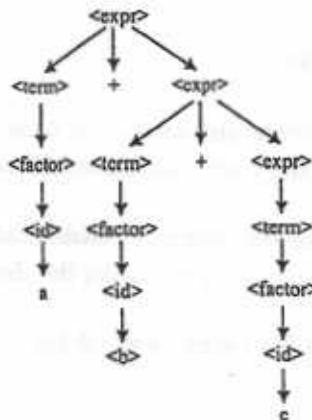


Fig 3.14 Parse tree for $a + b \times c$

3.2.2.2 Elimination of Left-Recursion

The production is left-recursive if the left most symbol on the right side is the same as the non-terminal on the left side. For example,

$$\text{expr} \rightarrow \text{expr} + \text{term}$$

If one were to code this production in a recursive-decent parser the parser would go in an infinite loop.

We can eliminate the left-recursion by introducing new nonterminals and new production rules.

If we have left-recursive pair of production

$$A \rightarrow A\alpha / \beta \text{ (where } \beta \text{ does not begin with } A.)$$

then we can eliminate left-recursion by replacing this pair of production with following pair.

$$A \rightarrow BA'$$

$$A \rightarrow \alpha A' / \epsilon$$

General algorithm for removing left recursion.

$$A \rightarrow A\alpha_1 / A\alpha_2 \dots / A\alpha_m / \beta_1 / \beta_2 / \dots \beta_n$$

where no β_i begins with an A

Now we can write above production as follows:

$$A \rightarrow \beta_1 A' / \beta_2 A' / \dots / \beta_n A'$$

$$A \rightarrow \alpha_1 A' / \alpha_2 A' / \dots / \alpha_m A' / \epsilon$$

Example 3.4 Consider the following CFG

$$E \rightarrow E + T / T$$

$$T \rightarrow T \times F / F$$

$$F \rightarrow (E) / I$$

$$I \rightarrow a / b / c$$

and remove the left recursion.

Solution Given CPG is

$$E \rightarrow E + T / T : P1$$

$$T \rightarrow T \times F / F : P2$$

$$F \rightarrow (E) / I : P3$$

$$I \rightarrow a / b / c : P4$$

Only P_1 and P_2 are left recursive production so let us introduce another non-terminals A in P_1 and B in P_2 as follows:

$$E \rightarrow TA$$

$$A \rightarrow + TA / E$$

$$T \rightarrow FB$$

$$B \rightarrow \times FB / \epsilon$$

$$F \rightarrow (E)/I$$

$$I \rightarrow a/b/c$$

Now all the productions are free from left recursion.

Example 3.5 Write an algorithm to remove the indirect left recursion.

Solution: *Removing Indirect Left-Recursion:*

Getting rid of *immediate left recursion* is not enough, one must get rid of *indirect left recursion*, where two or more nonterminals are *mutually left-recursive*. One can write any CFG. to remove left recursion.

```
for i/: = 1 to n do
    for j/: = 1 to i - 1 do
        begin
            replace each  $A_i \rightarrow A_s \gamma$  with productions
             $A_i \rightarrow \delta_{i1} \gamma / \delta_{i2} \gamma$ .
        end.
```

eliminate immediate left recursion.

Example 3.6 Remove the left recursion from the following CFG.

$$\text{expr} \rightarrow \text{expr} + \text{term} / \text{expr} - \text{term} / \text{term}$$

Solution: Given CFG

$$\text{expr} \rightarrow \text{expr} + \text{term} / \text{expr} - \text{term} / \text{term}$$

can be re-written so it will be free from left recursion.

$$\begin{aligned} \text{expr} &\rightarrow \text{term rest} \\ \text{rest} &\rightarrow + \text{term rest} / - \text{term rest} / t \end{aligned}$$

Example 3.7 Consider the following CFG.

$$S \rightarrow Bb/a$$

$$B \rightarrow Bc/Sd/e$$

Find out the left recursion and remove it.

Solution: Given CFG is

$$S \rightarrow Bb/a$$

$$B \rightarrow Bc/Sd/e$$

We can also write above CFG as follows:

$$S \rightarrow Bb/a$$

$$B \rightarrow Bc/Bbd/ad/e$$

Now we will remove immediate left recursion as follows:

$$S \rightarrow Bb/a$$

$$B \rightarrow adB'/eB'$$

$$B' \rightarrow cB'/bdB'/\epsilon$$

Now above CFG is free from left recursion.

3.2.2.3 Left Factoring

Left factoring is useful for producing a grammar suitable for a predictive parser.

Let us consider an production.

```
<statement> → if <expr> then <statement> else <statement>
          /if <expr> then <statement>
```

Now we don't know which production rule to follow without knowing if there is an "else".

We can introduce another non-terminal S, as follows:

```
<statement> → if <expr> then <statement> S
          S → else <statement> ∈
```

So we can finally if situation of the productions are as follows:

$A \rightarrow \alpha\beta_1/\alpha\beta_2$ then it is difficult to decide whether to expand A to $\alpha\beta_1$ or to $\alpha\beta_2$.

Now we will rewrite the production:

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1/\beta_2$$

Now after seeing the input derived from α , we expand A' to β_1 or to β_2 .

Example 3.8 Consider the following grammar & remove the left-factoring from it.

$$A \rightarrow xBy A/xBy AzA/a$$

$$B \rightarrow b$$

Solution: The given CFG is as follows:

$$A \rightarrow xByA/xByAzA/a$$

$$B \rightarrow b$$

Left-factored, this grammar becomes.

$$A \rightarrow xBy AA'/a$$

$$A' \rightarrow zA/\epsilon$$

$$B \rightarrow b$$

Example 3.9 Consider the following CFG.

```
Statement → variable ASSIGNOP expr/Procedure-call/
           block/if expr THEN statement else statement
           /while expr do statement.
```

Three of the production for *statement* begins with nonterminals:

$$\text{Variable} \rightarrow \text{ID}/\text{ID } (\text{expr})$$

$$\text{Procedure call} \rightarrow \text{ID}/\text{ID } (\text{expr - list})$$

$$\text{block} \rightarrow \text{Begin opt-statements End.}$$

Now remove the left factoring from the given grammar.

Solution: In the productions for statement we replace nonterminals Variable, procedure call, and block, by the right-sides of their production to obtain:

```

Statement → ID ASSIGNOP expr/
           ID (expr) ASSIGNOP expr/
           ID/
           ID (expr-list) /
           Begin opt-statement End/
           If expr THEN statement ELSE statement/
           while expr do statement

```

Now every production for *statement* begins with a terminal but four of the productions begin with the same terminal, ID, so we add a new nonterminal, *statement-rest*, to the grammar and left factor ID out of these four productions to obtain:

```

Statement → ID statement-rest
           Begin opt-statements End/
           If expr THEN statement ELSE statement/
           while expr do statement

Statement-rest → ASSIGNOP expr/
                  (expr) ASSIGNOP expr/
                  (expr-list) /ε

```

3.3 RECURSIVE-DECENT PARSING

"A top-down parser that executes a set of recursive procedures to process the input without backtracking is called recursive-decent parser and parsing is called recursive-decent parsing".

The recursive procedures can be easy to write and fairly efficient if written in a language that implements the procedure call efficiently.

There is a procedure for each nonterminal in the grammar. We assume a global variable, *lookhead*, holding the current input token and a procedure *match* (*Expected token*) is the action of recognizing the next token in the parsing process and advancing the input stream pointer, such that *lookhead* points to the next token to be parsed, *match* () is effectively a call to the lexical analyzer to get the next token.

For example input stream is: a + b \$

```

then lookhead == a
      match ()
      lookhead == +
      match ()
      lookhead == b
      ...
      ...

```

In this manner parsing can be done.

Consider the following grammar:

$$\begin{aligned} E &\rightarrow TA \\ A &\rightarrow +\text{TA}/\epsilon \\ T &\rightarrow FB \\ B &\rightarrow \times FB/\epsilon \\ F &\rightarrow (E)/\text{id} \end{aligned}$$

Now let us write the algorithm of recursive-decent presser for given grammar.

```

E( )
{
    T( );
    A( );
}.
T( )
{
    F( );
    B( );
}.
B( )
{
    if (lookhead == [ast])
    {
        match();
        F( );
        B( );
    };
}.
A( )
{
    if (lookhead == [plus])
    {
        match();
        T( );
        A( );
    };
}.
F( )
{
    if (lookhead == [id])

```

```

    {
        match( );
    }
    else if (lookhead == [lparen])
    {
        match( );
        E( );
        if (lookhead == [rparen])
        {
            match( );
        }
        else ERROR
    }
    else ERROR
} // end of F( ).
```

If there any ϵ -production for a nonterminal then the procedure for that nonterminal selects it whenever none of the other productions are suitable. If there is no production for a non-terminal and none of its productions are suitable then the procedure should report a syntax error.

But one of the major draw back of recursive decent parsing is that, it can be implemented only for those language which supports recursive procedure call, and it suffers with the problem of left recursion.

Example 3.10 Write a code for the Recursive-decent parsing of the following grammar.

```

expr → term rest
rest → + term rest / - term rest / ε
term → 0/1/ .... / 9
```

Solution: We know that in a recursive-decent parsing, we write code for each nonterminal of a grammar. In the case of above grammar, we should have three procedure, correspond to nonterminal expr, rest, and term.

Since there is only one production for nonterminal expr, the procedure expr is:

```

expr( )
{
    term( );
    rest( );
    return
}
```

Since there are three productions for rest, procedure rest uses a global variable, 'lookhead', to select the correct production or simply selects "no action" that is, ϵ — production, indicating at lookhead variable is neither + nor (neither plus nor minus).

```

rest( )
{
    If (lookhead == '+')
    {
        match();
        term();
        rest();
        return;
    }
    else if (lookhead == '-')
    {
        match();
        term();
        rest();
        return;
    }
    else
    {
        return;
    }
}.

```

The procedure *term* checks whether global variable is a digit.

```

term()
{
    if (is digit (lookhead))
    {
        match();
        return;
    }
    else
    {
        Error();
    }
}.

```

After loading first input token into variable 'lookhead' predictive parser is started by calling starting symbol, *expr*.

If the input is error free, the parser conducts a depth first traversal of the parse tree and return to caller routine through *expr*.

3.4 TOP-DOWN PREDICTIVE PARSERS

A predictive parser is an efficient way of implementing recursive-descent parsing since a stack is maintained in predictive parsing for handling the activation records. In top-down predictive parsers, the grammar will be able to predict the right alternative for the expansion of a nonterminal during the parsing process, and hence, it need no backtrack.

Let $B \rightarrow \alpha_1 / \alpha_2 \dots / \alpha_m$ are the production in the grammar, then during the parsing process parser has to decide that B is to be expanded or not. In predictive technique parser can exactly decided that which production of B should be used to expand B. Basically predictive parser looks at the next input symbol and find out which of the α_i derives a string that start with the terminal symbol comes next in the input.

The predictive parser program maintains a stock of grammar symbols and uses a two-dimensional M-table created from the grammar. A special symbol, \$, marks the bottom of the stack and also the end of the input string. *The parser is initialized with the start symbol of grammar on the stack top and the input pointing to the first token.* Pictures view of parser program can be viewed as follows:

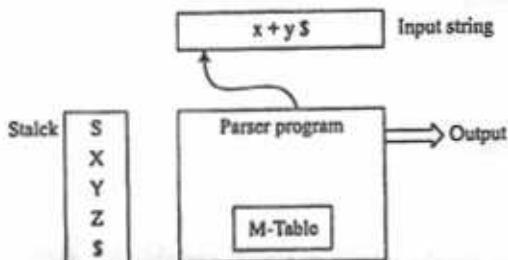


Fig. 3.15 Predictive parser

The action of the parser depends on the grammar symbol on the top of the stack, Y and the current input token, b:

- If $Y = b = \$$ then passing halts and parser announces successful completion of the parsing.
- $Y = b$ but doesn't equal \$ then parser pops Y off the stack and advances the input to the next token.
- If Y is a terminal not equal to b then parser announces an error.
- If Y is a Non-terminal then parser consults entry $M[Y, b]$ in the M-table.

If the $M[Y, b]$ entry is a production for Y then the parser pops Y off the stack and pushes the symbols on the right-side of the production on to the stack (pushing the right most symbol of the right-side last.)

If the $M[Y, b]$ is an entry then the announces the error and calls an error recovery routine.
Let us consider the following example to make the concept clear.

$$E \rightarrow TE'$$

$$E \rightarrow + TE' / \epsilon$$

$$T \rightarrow FT'$$

$$T \rightarrow \times FT' / \epsilon$$

$$F \rightarrow id / (E)$$

The table given below shows the moves made by the parser when the input is $id + id \times id$. In the table the stack is shown with the bottom on the left and the top on the right.

Stack	Input string	Production used
ES	$id + id \times id \$$	
TE' \$	$id + id \times id \$$	$E \rightarrow TE'$
FT'E' \$	$id + id \times id \$$	$T \rightarrow FT'$
idTE' \$	$id + id \times id \$$	$F \rightarrow id$
TE' \$	$+ id \times id \$$	
E' \$	$+ id \times id \$$	$T' \rightarrow \epsilon$
+TE' \$	$+ id \times id \$$	$E' \rightarrow + TE'$
TE' \$	$id \times id \$$	
FT'E' \$	$id \times id \$$	$T \rightarrow FT'$
idTE' \$	$id \times id \$$	$F \rightarrow id$
TE' \$	$\times id \$$	
$\times FT'E' \$$	$\times id \$$	$T \rightarrow \times FT'$
FT'E' \$	$id \$$	
idTE' \$	$id \$\$$	$F \rightarrow id$
TE' \$	$\$$	
E' \$	$\$$	$T' \rightarrow t$
\$	$\$$	$E' \rightarrow t$

3.4.1 First and Follow

First and follow are two function associated with a grammar that help us fill in the entries of an M-table.

First (): is a function which gives the set of terminals that begin the strings derived from the production rule.

FOLLOW: is a function which gives the set of terminals that can appear immediately to the right of a given terminal symbol.

Benefits of First () and Follow():

1. Can be used to prove the LL(k) characteristic of a grammar.
2. Can be used to aid in the construction of predictive parsing tables.
3. Provides selection information for recursive descent parsers.

3.4.2 Following Algorithm can be used to Compute First (X) for a Grammar Symbol X

- (i) If X is a terminal, First (X) is {X}.
- (ii) If $X \rightarrow \epsilon$ is a production, then add ϵ to the set of First (X).

(iii) If X is a nonterminal

(a) If $X \rightarrow Y$, $\text{First}(Y)$ is an element of the set of $\text{First}(X)$.

(b) If $\text{First}(Y)$ has ϵ as an element and $X \rightarrow YZ$ then $\text{First}(X) = \text{First}(Y) - \{\epsilon\} \cup \text{First}(Z)$.
 $\text{First}(Z)$ can be calculated in identical manner.

For example if given grammar is

$$\begin{aligned} E &\rightarrow TA \\ A &\rightarrow +TA/\epsilon \\ T &\rightarrow FB \\ B &\rightarrow x FB/\epsilon \\ F &\rightarrow (E)/id \end{aligned}$$

then let us calculate $\text{First}(E)$, $\text{first}(A)$, $\text{first}(T)$, $\text{first}(B)$ and $\text{First}(F)$.

$$\begin{aligned} \text{First}(E) &= \text{First}(TA) && (\text{since } E \rightarrow TA) \\ &= \text{First}(T) && (\text{since } T \rightarrow \epsilon \text{ is not in the grammar}) \\ &= \text{First}(FB) && (\text{since } T \rightarrow FB) \\ &= \text{First}(F) && (\text{since } F \rightarrow \epsilon \text{ is not there in the given grammar}) \\ &= \{C, id\} && (\text{since } F \rightarrow (E)/id, \text{ so terminals are (and id) the}) \\ \text{First}(A) &= \{+, \epsilon\} && (\text{since } A \rightarrow + TA/\epsilon \text{ so terminals are + and } \epsilon) \\ \text{First}(T) &= \{C, id\} && (\text{as above}) \\ \text{First}(B) &= \{x, \epsilon\} && (\text{since } B \rightarrow x FB/\epsilon, \text{ so terminals are } x \text{ and } \epsilon) \\ \text{First}(F) &= \{C, id\}. \end{aligned}$$

So for the given grammar $\text{First}(E) = \{C, id\}$, $\text{First}(A) = \{+, \epsilon\}$

$$\text{First}(T) = \{C, id\}, \text{First}(B) = \{x, \epsilon\}$$

and

$$\text{First}(F) = \{C, id\}.$$

3.4.3 Algorithm to Calculate the $\text{Follow}(X)$ for a Non terminal X

- (i) $\$$ is an element of $\text{follow}(s)$, where s is the start symbol and $\$$ indicates the end of the input
- (ii) If $A \rightarrow \alpha XY$ is a production, the set $\text{First}(Y)$ is in the set $\text{follow}(X)$, excluding ϵ .
- (iii) If $A \rightarrow \alpha X$ or $A \rightarrow \alpha XY$ are productions, and $\text{First}(Y)$ has an element ϵ , the set $\text{follow}(X)$ is in the set $\text{follow}(A)$.

Let us consider the same example as follows to calculate the grammar:

$$\begin{aligned} E &\rightarrow TA & \text{First}(E) &= \{C, id\} \\ A &\rightarrow + TA/\epsilon & \text{First}(A) &= \{+, \epsilon\} \\ T &\rightarrow FB & \text{First}(T) &= \{C, id\} \\ B &\rightarrow x FB/\epsilon & \text{First}(B) &= \{x, \epsilon\} \\ F &\rightarrow (E)/id & \text{First}(F) &= \{C, id\}. \end{aligned}$$

$\text{Follow}(E) = \{\$, \}\}$ (Here \$ is in $\text{Follow}(E)$ since E is starting nonterminal and) is for production $F \rightarrow (E)/\text{id}$.

$\text{Follow}(A) = \{\$, \}\}$ (Since $E \rightarrow TA$ and $A \rightarrow \epsilon$ so follow E is the follow (A)).

$\text{Follow}(T) = \text{First}(A)$ by rule 2 (since $E \rightarrow TA$) $\cup \text{Follow}(E)$ by rule 3 on production $E \rightarrow TA$ (since $\text{First}(A)$ contains ϵ).

$$\begin{aligned} &= \{+, \epsilon\} - \epsilon \cup \{\$\}\} \\ &= \{+, \$\}\} \end{aligned}$$

$\text{Follow}(B) = \text{Follow}(T)$ by the rule 3 on production $T \rightarrow FB = \{+, \}, \$\}$

$$\begin{aligned} \text{Follow}(F) &= (\text{First}(B) \text{ by rule 2}) \cup (\text{Follow}(T) \text{ by rule 3 on } T \rightarrow FB) \\ &\quad \cup (\text{Follow}(B) \text{ by rule 3 on } B \rightarrow Fb) \\ &= \{x\} \cup \{+, \}, \$\} \cup \{+, \$\} \\ &= \{+, \}, x, \$\}. \end{aligned}$$

3.4.4 Constructing M-table/Predictive Parsing Table

Many books follows various different- different technique to construct the M-table, here we will discuss another approach, by calculating a new function select () .

Select (): is a function which can be defined by the First () and follow () functions and by using select () function, we can easily construct the M-table.

Let $A \rightarrow w$ is any production then

$$\text{Select}(A \rightarrow W) = \text{First}(\text{First}(W) \text{ follow}(A))$$

Let us continue with previous example.

$$\begin{array}{lll} E \rightarrow TA & \text{First}(E) = \{(), \text{id}\} & \text{Follow}(F) = \{\$\}\} \\ A \rightarrow + TA / \epsilon & \text{First}(A) = \{+, \epsilon\} & \text{Follow}(A) = \{\$\}\} \\ T \rightarrow FB & \text{First}(T) = \{(), \text{id}\} & \text{Follow}(T) = \{+, \}, \$\} \\ B \rightarrow x FB / \epsilon & \text{First}(B) = \{x, \epsilon\} & \text{Follow}(B) = \{+, \}, \$\} \\ F \rightarrow (E)/\text{id} & \text{First}(F) = \{(), \text{id}\} & \text{Follow}(F) = \{+, \}, x, \$\}. \end{array}$$

$$\begin{aligned} \text{Select}(E \rightarrow TA) &= \text{First}(\text{First}(TA) \text{ Follow}(E)) \\ &= \text{First}(\{(), \text{id}\} \{\$\}\}) \\ &= \text{First}(\{(\$, (), \text{id}), \$, \text{id}\}) \\ &= \{(), \text{id}\} // \text{first symbol of each pair.} \end{aligned}$$

$$\begin{aligned} \text{Select}(A \rightarrow + TA) &= \text{First}(\text{First}(+ TA) \text{ Follow}(A)) \\ &= \text{First}(\{+\}, \{\$\}\}) \\ &= \text{First}(\pm \$, \pm\}) \\ &= \{+\} \end{aligned}$$

$$\begin{aligned} \text{Select}(A \rightarrow \epsilon) &= \text{First}(\text{First}(\epsilon) \text{ Follow}(A)) \\ &= \text{First}(\{\epsilon\} \{\$\}\}) \\ &= \text{First}(\$\}) \\ &= \{\$\}\} \end{aligned}$$

$$\begin{aligned}
 \text{Select } (T \rightarrow FB) &= \text{First}(\text{First}(FB) \text{ Follow}(T)) \\
 &= \text{First}(\{(, id\} \{ +, \}, \$\}) \\
 &= \text{First}(\{ (, +, ,), (\$, id +, id), id\$ \}) \\
 &= \{ (, id\}
 \end{aligned}$$

$$\begin{aligned}
 \text{Select } (B \rightarrow x FB) &= \text{First}(\text{First}(x FB) \text{ Follow}(B)) \\
 &= \text{First}(\{x\}, \{ +, \}, \$\}) \\
 &= \{x\}
 \end{aligned}$$

$$\begin{aligned}
 \text{Select } (B \rightarrow \epsilon) &= \text{First}(\text{First}(\epsilon) \text{ Follow}(B)) \\
 &= \text{First}(\{\epsilon\} \{ +, \}, \$\}) \\
 &= \text{First}(\{ +, \}, \$\}) \\
 &= \{ +, \}, \$\}
 \end{aligned}$$

$$\begin{aligned}
 \text{Select } (F \rightarrow (E)) &= \text{First}(\text{First}(E) \text{ Follow}(F)) \\
 &= \text{First}(\{\()\} \{ +, \}, x, \$\}) \\
 &= \{\()
 \end{aligned}$$

$$\begin{aligned}
 \text{Select } (F \rightarrow id) &= \text{First}(\text{First}(id) \text{ Follow}(F)) \\
 &= \text{First}(\{id\} \{ +, \}, x, \$\}) \\
 &= \text{First}(id +, id), idx, id\$ \\
 &= id
 \end{aligned}$$

So for predictive Parsing table:

$E \rightarrow TA$	$\text{Select}(E) = \{ (, id\}$
$A \rightarrow TA$	$\text{Select}(A) = \{ +\}$
$A \rightarrow t$	$\text{Select}(A) = \{ \$,)\}$
$T \rightarrow FB$	$\text{Select}(T) = \{ C, id\}$
$B \rightarrow x FB$	$\text{Select}(B) = \{ +, \}, \$\}$
$B \rightarrow \epsilon$	$\text{Select}(B) = \{ +, \}, \$\}$
$F \rightarrow (E)$	$\text{Select}(F) = \{x\}$
$F \rightarrow id$	$\text{Select}(F) = \{id\}$

Finally With the help of these select values we can construct M-table as follows:

Nonterminals	Look Ahead Token					
	(id	+	x)	\$
E	TA	TA				
A			+TA		ε	ε
T	FB	FB				
B			ε	xFB	ε	ε
F	(E)	id				

Construction is clear now since we have the values of select () for all non terminals.

For example if for $T \rightarrow FB$ select (T) is $\{(), id\}$ then corresponding entry for $M[T]$, () will be EB, and for $M[T, id]$ will be FB.

Similar for $A \rightarrow \epsilon$ Select is $\{(), \$\}$ so $M[A, ()]$ and $M[A, \$]$ will be ϵ and ϵ respectively.

In the similar fashion the complete parsing table is constructed.

3.5 LL(1) GRAMMARS

Now we have fair idea about predictive parsing technique and we know that in *some of the situation M-table (predictive parsing table) may have same entry for two or more different productions, it simply means that the grammar is ambiguous and/or left recursive and/or not left factored.*

"A grammar is an LL(1)- grammar if and only if its M-table has no entries that are multiply defined."

We can also define LL(1) grammar mathematically as follows:

"If each non-terminal of given grammar which appears on the left hand side of more than one production have the LL(1) property then grammar is said to be LL(1)".

(If a nonterminal appears on the left-side of more than one production and select () for those production are disjoint then, it has LL(1) property).

For example If $A \rightarrow ai$ and $A \rightarrow aj$

If $\text{select}(A \rightarrow ai) \cap \text{select}(aj) = \emptyset$ then A has the LL(1) Property.

Let us consider the same grammar $F \rightarrow TA$

$$A \rightarrow +TA/\epsilon$$

$$T \rightarrow FB$$

$$B \rightarrow xFB/\epsilon$$

$$F \rightarrow (E)/id$$

Here only A and, B and F are present in more than one production.

$$A \rightarrow +TA, \quad A \rightarrow \epsilon$$

$$B \rightarrow xFB, \quad B \rightarrow \epsilon$$

$$F \rightarrow (E), \quad F \rightarrow id$$

$$\text{Select}(E \rightarrow TA) = \{(), id\}$$

$$\text{Select}(A \rightarrow +TA) = \{+\}$$

$$\text{Select}(A \rightarrow \epsilon) = \{\$\}$$

$$\text{Select}(T \rightarrow FB) = \{(), id\}$$

$$\text{Select}(B \rightarrow xFB) = \{x\}$$

$$\text{Select}(B \rightarrow t) = \{+, (), \$\}$$

$$\text{Select}(F \rightarrow (E)) = \{(()\}$$

$$\text{Select}(F \rightarrow id) = \{id\}$$

Now

For $A \rightarrow +TA$, $A \rightarrow \epsilon$, select $(A \rightarrow +TA) \cap \text{select}(A \rightarrow \epsilon) = \emptyset$

For $B \rightarrow x FB$, $B \rightarrow \epsilon$, select $(B \rightarrow x FB) \cap \text{select}(B \rightarrow \epsilon) = \emptyset$

For $F \rightarrow (E)$, $F \rightarrow \text{id}$, select $(F \rightarrow (E)) \cap \text{select}(F \rightarrow \text{id}) = \emptyset$

So A, B and F have LL(1) property.

So finally we can say that given grammar is LL(1).

3.5.1 Advantages and Disadvantages of LL(1) Parsing

The reason why LL(1) parsing by recursive descent is appealing to many people is its naturalness, and the method is extensively taught as an introduction to, and a framework for compiling programming language. In addition, it is easy to implement and to have confidence in its correctness. As well as performing the parsing process, the functions written may also contain actions to perform type checking and other forms of checking, as well as synthesis actions such as storage allocation and code generation.

However, we have also identified some disadvantages of recursive descent parsing, such as the inefficiency of function calls and the need to perform grammar transformations, without even knowing whether suitable transformations exist or not. The problem is not just finding the transformation but ensuring that they are performed correctly. There are therefore good reasons for using tools to perform approach. The disadvantages of recursive descent parsing are:

- (i) The very large parsers often produced.
- (ii) The tendency for actions which are part of different phases of compilation to appear in the same function bodies.

The following are the requirements for effective use of recursive descent:

- A good grammar transform that will usually be able to transform a grammar into LL(1) form
- The ability to represent the equivalent of the recursive descent parser in tabular form. This means that the parser, instead of entering and leaving functions as it checks the input text, will instead move about the tabular equivalent of the grammar, stacking return addresses as necessary.

Example 3.11 Consider the following grammar

$$S \rightarrow AaAb / BbBa$$

$$A \rightarrow \epsilon$$

$$B \rightarrow \epsilon$$

Test whether the grammar is LL(1) or not, and construct a predictive parsing table for it.

Solution: Given grammar is as follows:

$$S \rightarrow AaAb / BbBa$$

$$A \rightarrow \epsilon$$

$$B \rightarrow \epsilon$$

Let us calculate first and follow:

$$\text{First}(s) = \text{First}(AaAb) \cup \text{First}(BbBa)$$

$$\text{First}(AaAb) = \text{First}(A) - \{\epsilon\} \cup \text{First}(aAb) = \{a\}$$

$$\text{First}(BbBa) = \text{First}(B) - \{\epsilon\} \cup \text{First}(bBa) = \{b\}$$

$$\text{So } \text{First}(S) = \{a\} \cup \{b\} = \{a, b\}$$

$$\text{First}(A) = \{\epsilon\}$$

$$\text{First}(B) = \{\epsilon\}$$

$$\text{Follow}(S) = \$$$

$\text{Follow}(A) = \{a, b\}$ (since in $S \rightarrow AaAb$, A is followed by both a and b)

$\text{Follow}(B) = \{b, a\}$ (Since in $S \rightarrow BbBa$, B is followed by both b and a)

Now let us calculate select () for different productions:

$$\begin{aligned} \text{Select}(S \rightarrow AaAb) &= \text{First}(\text{First}(AaAb) \text{Follow}(S)) \\ &= \text{First}(\{a\} \{\$\}) = a \end{aligned}$$

$$\begin{aligned} \text{Select}(S \rightarrow BbBa) &= \text{First}(\text{First}(BbBa) \text{Follow}(S)) \\ &= \text{First}(\{b\} \{\$\}) = \text{First}(\{\$\}) = b \end{aligned}$$

$$\begin{aligned} \text{Select}(A \rightarrow \epsilon) &= \text{First}(\text{First}(t) \text{Follow}(A)) \\ &= \text{First}(\{\epsilon\} \{a, b\}) = \text{First}(\{a, b\}) \\ &= \{a, b\} \end{aligned}$$

$$\text{Select}(B \rightarrow \epsilon) = \{a, b\}$$

Here S (Nonterminal) is in the left hand side of two productions as:

$$S \rightarrow AaAb/BbBa$$

$$\text{Select}(S \rightarrow AaAb) \cap (S \rightarrow BbBa)$$

Now let us make

$$= a \cap B$$

$$= \emptyset$$

So we can say that S follows LL(1) properties. Now finally we can say that given grammar is LL(1).

Now it is easy to construct the parse table by the help of select () .

	a	b	\$
S	$S \rightarrow AaAb$	$S \rightarrow BbBa$	
A	$A \rightarrow \epsilon$	$A \rightarrow \epsilon$	
B	$B \rightarrow \epsilon$	$B \rightarrow \epsilon$	

Since Select $(S \rightarrow AaAb) = a$

Select $(S \rightarrow BbBa) = b$

$$\text{Select } (A \rightarrow \epsilon) = \{a, b\}$$

$$\text{Select } (B \rightarrow \epsilon) = \{a, b\}$$

Example 3.12 Consider the following grammar

$$S \rightarrow 1AB/\epsilon$$

$$A \rightarrow 1 AC / OC$$

$$B \rightarrow OS$$

$$C \rightarrow 1$$

and test that whether the following grammar is LL(1) or not.

Solution: Let us calculate First and Follow:

$$\text{First}(S) = \text{First}(1AB) \cup \text{First}(\epsilon) = \{1, \epsilon\}$$

$$\text{First}(A) = \{1, 0\}$$

$$\text{First}(B) = \{0\}$$

$$\text{First}(C) = \{y\}$$

$$\text{Follow}(S) = \{\$\}$$

$$\begin{aligned}\text{Follow}(A) &= \text{First}(A) \cup \text{First}(B) = \{1, 0\} \cup \{0\} \\ &= \{1, 0\}\end{aligned}$$

$$\text{Follow}(B) = \text{Follow}(S) = \$$$

Now only S and A derives two production as

$$S \rightarrow 1AB, \quad S \rightarrow \epsilon$$

and

$$A \rightarrow 1AC, \quad A \rightarrow OC$$

$$\begin{aligned}\text{Select}(S \rightarrow 1AB) &= \text{First}(\text{First}(1AB) \text{Follow}(S)) \\ &= \text{First}(\{1\} \{\$\}) = \{1\}\end{aligned}$$

$$\begin{aligned}\text{Select}(S \rightarrow \epsilon) &= \text{First}(\text{First}(\epsilon) \text{Follow}(S)) \\ &= \text{First}(\epsilon \$) = \{\$\}\end{aligned}$$

$$\begin{aligned}\text{Select}(A \rightarrow 1AC) &= \text{First}(\text{First}(1AC) \text{Follow}(A)) \\ &= \text{First}(\{1\}, \{1, 0\}) \\ &= \text{First}(11, 1, 0) \\ &= \{1\}\end{aligned}$$

$$\begin{aligned}\text{Select}(A \rightarrow OC) &= \text{First}(OC) \text{Follow}(A) \\ &= \text{First}(\{0\} \{1, 0\})\end{aligned}$$

$$\text{First}(01, 00) = \{0\}$$

Now following condition should hold for LL(1) grammar:

$$\begin{aligned}\text{Select}(S \rightarrow 1AB) \cap \text{Select}(S \rightarrow \epsilon) \\ = \{1\} \cap \{\$\} = \emptyset\end{aligned}$$

$$\text{Select } (A \rightarrow 1AC) \cap \text{Select } (A \rightarrow 0C)$$

$$= \{1\} \cap \{0\} = \emptyset$$

So Here both S and A are following LL(1) property so we can say that grammar is LL(1).

3.6 INTRODUCTION TO BOTTOM-UP PARSING

Consider the grammar

$$S \rightarrow aTUE$$

$$T \rightarrow Tbc/b$$

$$U \rightarrow d$$

and let us find out the rightmost derivation of the sentence for the string abb cde

$$S \Rightarrow aTUE$$

$$\Rightarrow aTde$$

$$\Rightarrow aTbc de$$

$$\Rightarrow abbcde$$

Note let us try to read abbcde from left-to-right and performs the right most derivation in the reverse order, it can be done in four steps as follows:-

- (i) Choose the first b and reduce it to the left-side of the $T \rightarrow b$ production to produce the sentential form: aTbcde.
- (ii) Now reduce Tbc by the left-hand side of the production $T \rightarrow Tbc$ to produce the sentential form aTde.
- (iii) Now reduce d by left-hand side of the production, $U \rightarrow d$ to produce aTUE
- (iv) finally aTUE can be replaced by the left-hand side of $S \rightarrow aTUE$ production get S.

If we carefully analyze the above discussion than it is very clear that in each step, we are applying the productions in reverse, replacing the right-side with the left-side, so we use the word reduce instead of produce.

"So Bottom-up parsing is an attempt to reduce the input string w to the start symbol of a grammar by tracing out the right-most derivations of w in reverse. This is equivalent to constructing a parse tree for the input string w in the reverse order."

Bottom-up parsing involves the selection of a substring that matches the right side of the production, whose reduction to the nonterminal on the left side of the production represents one step along the reverse of a right-most derivation. That is it leads to the generation of the previous right most derivation. Clearly selecting a substring that matches the right side of production is not enough; the position of this substring in the sentential form is also important.

3.6.1 Handle of a Right Sentential Form

From the definition of Bottom-up parsing it is clear that finding a substring that matches the right side of production, as well as its position in the current sentential form, are both equally important. We will define a new term "handle" to address both these problems simultaneously.

"A handle is a substring that matches a right hand side of production rule in the grammar and whose reduction to the nonterminal on the left hand side of that grammar rule is a step along the reverse of a rightmost derivation".

For example for the grammar

$$S \rightarrow aTUb$$

$$T \rightarrow Tbc/b$$

$$U \rightarrow d$$

the string "abbcde" can be based bottom-up by the following reduction steps:

- (i) aTbcde
- (ii) aTde
- (iii) aTUe
- (iv) S

So here in this example the handle are:

1. The first (b) in (abbcde)
2. The (Tbc) substring in (aTbcde)
3. The (d) in (aTde)
4. The whole string, (aTUe)

In step 1 and step 2 of the example the parser has three possible handles to choose from: if the parser chooses the wrong handle it won't be able to complete the reverse-ordered right most derivation. The main task of a bottom-up parser is to choose the correct handle at each step of the parse. there could be many choices on any step, that is the empty string can be inserted in to the string of n symbols in any nth different locations so just a single ϵ -production in a grammar will give us many possible handles to choose from.

Example 3.13 Find the handle of the sentential forms occurring in the derivation of string id + id \times id

$$E \rightarrow E + E / E \times E / id$$

Solution: Given grammar is $E \rightarrow E + E$

$$E \rightarrow E \times E$$

$$E \rightarrow id$$

The string can be reduced to E in following steps

- | | |
|-------------------------|---|
| (i) id + id \times id | (First id is handle of this sentential form) |
| (ii) E + id \times id | (How second id is handle) |
| (iii) E + E \times id | (Now third id is handle for this sentential form) |
| (iv) E + E \times E | (Noe E \times E is handle) |
| (v) E + E | (Now E + E is handle) |
| (vi) E | (Reduced to starting symbol.) |

Example 3.14 Consider the following grammar and show the handle of each right sentential form for string $(a, (a, a))$

$$S \rightarrow (L)/a$$

$$L \rightarrow L, S/s$$

Solution: The following sentential form will occur in reduction of $(a, (a, a))$ to S .

(i) $(a, (a, a))$	(handle id first a)
(ii) $(s, (a, a))$	(Now S is the handle)
(iii) $(L, (a, a))$	(Now first a is handle)
(iv) $(L (S, a))$	(S is handle again)
(v) $(L (L, a))$	(Now a is handle)
(vi) $(L, (L, S))$	(Now L, S is handle)
(vii) $(L(L))$	((L) is handle)
(viii) (L, S)	(again L, S is handle)
(ix) (L)	((L) is handle)
(x) S	(finally string is reduced to starting nonterminal).

3.6.2 Shift Reduce Parsing

Most bottom-up parsers are implemented as shift-reduce parsers. Such a parser uses a stack to hold grammar symbols. A parser goes on shifting the input symbols onto the stack until a handle comes on the top of the stack. When a handle appears on the top of the stack, it performs reduction.

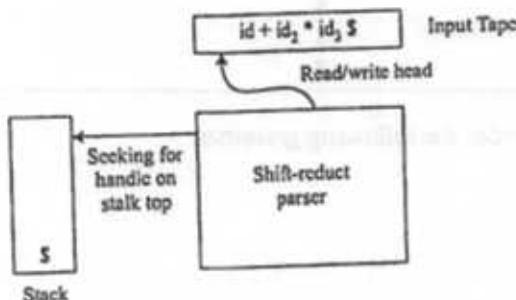


Fig. 3.16 Shift reduce parser

The implementation makes use of a stack to hold grammar and an input Tape to hold string w to be parsed, which is terminated by the right end marker $\$$, the same symbol used to mark the bottom of the stack. Parser may have following four possible actions:

- *Shift*: Move the next input symbol on to the top of the stack.
- *Reduce*: Reduce a handle on the right-most part of the stack by popping it off the stack and pushing the left-side of the production on to the right-end of the stack.
- *Accept*: Announce successful completion of parsing.
- *Error*: Signal discovery of a syntax error.

We use \$ to mark the left-end (bottom) of the stack and also the end of the input string. Initially the stack is empty, parsing ends successfully when the input is empty and stack contains only the start symbol.

As an example we use the following grammar:

$$\begin{aligned} E &\rightarrow E + E \\ E &\rightarrow E \times E \\ E &\rightarrow (E) \\ E &\rightarrow id \end{aligned}$$

Now action of shift-reduce parser to parse the input string $id_1 \times (id_2 + id_3)$ can be shown by following figure:

Stack contents	Input string	Actions
\$	$id_1 \times (id_2 + id_3) \$$	Shift id_1
\$ id_1	$\times (id_2 + id_3) \$$	Reduce by $E \rightarrow id$
\$ E	$\times (id_2 + id_3) \$$	Shift \times
\$ $E \times$	$(id_2 + id_3) \$$	Shift (
\$ $E \times ($	$(id_2 + id_3) \$$	Shift id_2
\$ $E \times (id_2$	$+ id_3) \$$	Reduce by $E \rightarrow id$
\$ $E \times (E$	$+ id_3) \$$	Shift +
\$ $E \times (E +$	$+ id_3) \$$	Shift id_3
\$ $E \times (E + id_3$) \$	Reduce by $E \rightarrow id$
\$ $E \times (E + E$) \$	Shift)
\$ $E \times (E + E)$	\$	Reduce by $E \rightarrow E + E$
\$ $E \times (E)$	\$	Reduce by $E \rightarrow (E)$
\$ $E \times E$	\$	Reduce by $E \rightarrow E \times E$
\$ E	\$	Accept

Example 3.15 Consider the following grammar

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $E \rightarrow T \times F$
4. $T \rightarrow E$
5. $F \rightarrow (E)$
6. $F \rightarrow y$

Show the shift reduce parse action for the string $y + y + y \times y$

Solution: For the given grammar

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $E \rightarrow T \times F$
4. $T \rightarrow E$
5. $F \rightarrow (E)$
6. $F \rightarrow y$

For the string $y + y + y \times y$, following parsing table can be constructed:

Stack contents	Input strings	Actions
\$	$y + y + y \times y \$$	shift y
\$y	$+ y + y y \$$	reduce by $F \rightarrow y$
\$F	$+ y + y x y \$$	reduce by $T \rightarrow F$
\$T	$+ y + y x y \$$	reduce by $E \rightarrow T$
\$E	$+ y + y x y \$$	shift +
\$E+	$y + y x y \$$	shift y
\$E+y	$+ y x y \$$	reduce by $F \rightarrow y$
\$E+F	$+ y x y \$$	reduce by $T \rightarrow F$
\$E+T	$+ y x y \$$	reduce by $E \rightarrow T$
\$E+E	$+ y x y \$$	reduce by $E \rightarrow E + E$
\$E	$+ y x y \$$	shift +
\$E+	$y x y \$$	shift y
\$E+y	$x y \$$	reduce by $E \rightarrow y$
\$E+F	$x y \$$	reduce by $T \rightarrow F$
\$E+T	$x y \$$	reduce by $E \rightarrow T$
\$E+E	$x y \$$	reduce by $E \rightarrow E + E$
\$E	$x y \$$	shift $\rightarrow x$
\$Ex	$y \$$	shift y
\$Exy	$\$$	reduce by $F \rightarrow y$
\$ExF	$\$$	reduce by $T \rightarrow F$
\$ExT	$\$$	reduce by $E \rightarrow T$
\$ExE	$\$$	reduce by $E \rightarrow E \times E$
\$E	$\$$	Accept

3.6.2.1 Short Comings of Shift-reduce Parsing

Shift-reduce parsing does not tell anything about the finding out the handles. Shift-reduce parsing is very efficient, when handle is identified. Handle can be easily managed with technique of shift-reduce parsing.

We can use several techniques to find out the handle and on the basis of these techniques we can define different shift-reduce parsers, such as shift-reduce parsers with precedence relationship and various LR parsers.

3.6.3 LR Parsers

A large class of grammar can be parsed using LR(K) parsers: the "L" stands for left-to-right scanning of the input, the "R" stands for constructing a rightmost derivation in reverse, and K is the number of input symbols of lookahead used to make parsing decision. When (K) is omitted, K is assumed to equal 1. LR parsing has several advantages:

- It can recognize virtually all programming language constructs for which grammars can be written.
- It is the most general nonbacktracking shift-reduce parsing method known.
- It can be implemented as efficiently as other shift-reduce methods.

- It can parse only grammar that a predictive parser can parse plus other grammars.
- It can detect a syntax error as early as possible on a left-to-right scan of the input.

The main disadvantage of LR parsing is that is too much work to construct a parser by hand: one needs a specialized tool—an LR parser generator.

3.6.3.1 The LR Parsing Algorithm

If we see the block diagram of an LR parser in Fig 3.17, it have an input, an output, a stack, a driver program, and a parsing table with two parts (action and go to). The driver program is the same for all LR parser: it reads the input string one symbol at a time and maintains a stack.

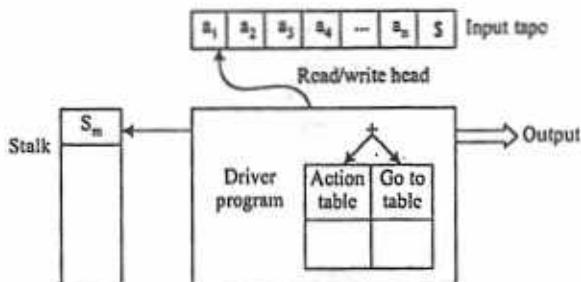


Fig. 3.17 Block diagram of LR parser

The stack is always maintain the following form:

$$S_0 X_1 S_1 X_2 S_2 \dots X_{m-1} S_{m-1} X_m S_m$$

Where each X_i is a grammar symbol, each S_i is the state, and S_m state is top of the stack.

The action of the driver program depends on action $[S_m, a_i]$ where a_i is the current input symbol. Following action are possible:

- **Shift:** If action $[S_m, a_i] = \text{shift } s$, the parser shifts the input symbol, a_i , onto the stack, and then stacks state s . Now current input symbol becomes a_{i+1}

StackInput

$$\text{So } X_1 S_1 X_2 \dots X_{m-1} S_{m-1} X_m S_m a_i s \qquad a_{i+1}, a_{i+2} \dots a_n \$$$

- **Reduce:** If action $[s_m, a_i] = \text{reduce } A \rightarrow B$, the parser executes a reduce move using the $A \rightarrow \beta$ production of the grammar. If β has r grammar symbols, first $2r$ symbols are popped off the stack (r state symbol and r grammar symbol). So the top of the stack now becomes S_{m-r} , then A is pushed on the stack, and then state go to $[S_{m-r}, A]$ is pushed on the stack. The current input symbol is still a_i .

StackInput

$$\text{So } X_1 S_1 X_2 \dots X_{m-r} S_{m-r} AS \qquad a_i, a_{i+1} \dots a_n \$$$

where $S = \text{Goto } [S_{m-r}, A]$

- If action $[S_m, a_i] = \text{accept}$, parsing is completed.
- If action $[S_m, a_i] = \text{error}$, the parser has discovered a syntax error.

For Example for the given grammar:

- (1) $E \rightarrow E + T$
- (2) $E \rightarrow (T)$
- (3) $T \rightarrow T \times F$
- (4) $T \rightarrow F$
- (5) $F \rightarrow (E)$
- (6) $F \rightarrow id$

and given parsing table:

State	action						Goto		
	<i>id</i>	+	\times	()	<i>s</i>	<i>E</i>	<i>T</i>	<i>F</i>
0	S_5			S_4			1	2	3
1		S_6				accept			
2		r_1	S_7		r_2	r_2			
3		r_4	r_4		r_4	r_4			
4	S_5			S_4			8	2	3
5		r_6	r_6		r_6	r_6		9	3
6	S_5			S_4					10
7	S_5			S_4					
8		S_6			S_{11}				
9		r_1	S_7		r_1	r_1			
10		r_3	r_3		r_3	r_3			
11		r_5	r_5		r_5	r_5			

Where S_i means shift state *i* on stack, r_j means reduce by using production numbered *j*, and finally blank means error.

Now show the parsing of string $id \times id + id \$$

Stack	Input string	Action
O	$id \times id + id \$$	S_5
O id_5	$\times id + id \$$	r_6
O $F3$	$\times id + id \$$	r_4
O $T2$	$\times id + id \$$	S_7
O $T2 \times 7$	$id + id \$$	S_5
O $T2 \times 7 id_5$	$+ id \$$	r_6
O $T2 \times 7 F 10$	$+ id \$$	r_3
O $T2$	$+ id \$$	r_2
O $E1$	$+ id \$$	S_4
O $E1 + 6$	$id \$$	S_5
O $E1 + 6 id 5$	$\$$	r_6
O $E1 + 6 F3$	$\$$	r_4
O $E1 + 6 T9$	$\$$	r_1
OE 1	$\$$	Accept

EXERCISE

1. Consider the following grammar:

$$S \rightarrow AaAB/ BbBa$$

$$A \rightarrow \epsilon$$

$$B \rightarrow \epsilon$$

is the grammar LL(1)?

2. Given a grammar:

$$E \rightarrow E + T/T$$

$$T \rightarrow T \times F/F$$

$$F \rightarrow id$$

which is a set of valid items for a valid prefix E^+ .

3. (i) Eliminate the left recursion from the following grammar:

$$S \rightarrow (L)/a$$

$$L \rightarrow L, S/a.$$

- (ii) Construct the predictive parser for given grammar in (i).

- (iii) Show the behaviour of parser on the string:

$$(a) (a, a)$$

$$(b) (a, (a, a))$$

4. State the problem associated with Top-Down parsing, state and eliminate the problem associated with the following grammar for Top-Down parsing:

$$E \rightarrow E + T/T$$

$$T \rightarrow T \times F/F$$

$$F \rightarrow (E)/id$$

5. After computing First and Follow functions, construct predictive parsing table for the following grammar:

$$E \rightarrow TE'$$

$$E' \rightarrow + TE'/\epsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow \times FT'/\epsilon$$

$$E \rightarrow (E)/id.$$

6. Obtain the canonical collection of sets of LR (1) items for the following grammar:

$$S \rightarrow SB/ca$$

$$B \rightarrow Sb/\epsilon$$

$$C \rightarrow aB/c$$

7. Find the LR(1) parsing table for the following grammar:

$$T \rightarrow int$$

$$L \rightarrow L, id/id$$

8. Construct an SLR(1) parsing table for the following grammar:

$D \rightarrow L \cdot T$
 $L \rightarrow L, \text{id}/\text{id}$
 $T \rightarrow \text{integer}.$

Elementry Data Types

4.1 INTRODUCTION TO DATA

Computer programs produce results by manipulating data.

A data item has a number of attributes:

1. An *address*.
2. A *value* which in turn has:
 - An *internal representation* comprising one or more bits and
 - An *interpretation* of that representation.
3. A set of *Operations* that may be performed on it.
4. A *name* to tie the above together (also referred to as a label or identifier).

Address

- The address or reference of a data item is the physical location in memory (computer store) of that data item.
- The amount of memory required by a particular data item is dictated by its type that is integer, character, etc.

Value

- The binary number stored at an address associated with a data item is its value.
- The required interpretation is dictated by the type of the data item.
- Consider 01 011 010, this can be interpreted as:
 - (a) The Decimal integer 90
 - (b) the Hexadecimal integer 5A
 - (c) The Octal integer 132
 - (d) The ASCII character (capital) Z, etc.
- The set of symbols used to express the value of a given data item is sometimes referred to as a literal.

Operations

The type of a data item also dictates the operations that can be performed on it. For example numeric data types can have the standard mathematical operations performed on them while string data types cannot.

Names

To do anything useful with a data item all the above must be tried together by giving each data item an identifying name (some time referred to as a label or an identifier).

Let us discuss some of the attributes of names:

- Names are usually made up of alpha-numeric characters.
- White space is usually not allowed but algol 60 is exception language.
- Some languages (Ada and C included) allow underscores.
- Some languages restrict the number of characters, for example first 31 characters in ANSI C.
- Some allow any length but treat the first N characters as significant.
- Case may be significant, For example in Algol 60, C and Modula 2 case matters but in pascal and Ada it does not.
- In any programming languages key/ reserved words can not be used as names.

It is always useful to adopt some sort of naming convention, for example ending all constant data item names with "const".

We can also view attributes by the help of following figure:

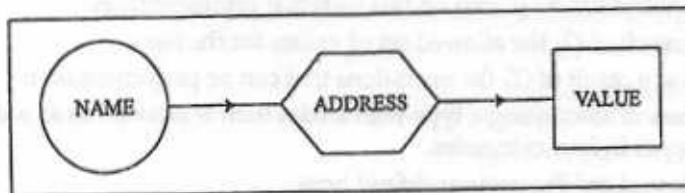


Fig 4.1 Components of a data item

4.1.1 Data Types

We all know that computer program produce results by manipulating data. An important factor in determining the ease with which they can perform this task is how well the data types match the real-world problem space. It is therefore crucial that a language support an appropriate variety of data types and structures.

The type of data item dictates:

- The range of possible values.
- The amount of storage required for its internal representation.
- The interpretation of the internal representation.
- The operations that can be performed on the item.

Data items can be classified as being either global or local data items, and as being either variables or constants. The nature of a data item is defined through a data declaration.

According to Bal and Grune (1994), inter-relationship between the different data item attributes can be summarised using a diagram of the form presented in following Fig. 4.2.

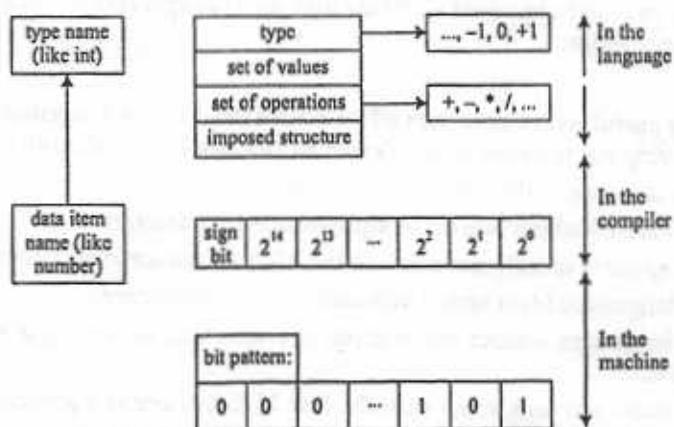


Fig. 4.2 The inter-relationship between data item attributes

4.1.2 Categorisation of Types

The type of a data item defines:

1. The nature of its internal representation
2. The interpretation placed on this internal representation.
3. As a result of (2) the allowed set of values for the item.
4. Also as a result of (2) the operations that can be performed on it.

The process of associating a type with a data item is referred to as a data declaration. We divide data types in four categories.

1. *Pre-defined and Programmer-defined types:*

Pre-defined types immediately available to the user (they are integral to the language). Programmer-defined types derived by the programmer using existing types.

2. *Scalar and compound types:*

Scalar types define data items that can be expressed as single values (For example numbers and characters). Compound types (also referred to as composite or complex type) define data items that comprise several individual values. Scalar types are generally pre-defined, while compound types are often programmer-defined.

3. *Discrete and Non-discrete types:*

Discrete types (also referred to as linear types) are types for which each value (except its minimum and maximum) has a predecessor and a successor value. The opposite are referred to as non-discrete types.

4. *Basic and Higher Level types:*

Basic types (also referred to as simple types or type primitives) are the standard scalar predefined types that one would expect to find ready for immediate use in any

imperative programming language. Higher level types are then made up from such basic types or other existing higher level types. Higher level types are not necessarily programmer define - for example many imperative languages include a string high level type.

4.1.3 Basic Operations on Data Items

Computer programs produces its results by performing some operations on the given data. Let us discuss some of the basic operation:

1. Data Declaration:

A data declaration introduces a data item. This typically achieved by associating the item (identified by a name) with a data type.

Let us see examples of different languages declarations.

For example in C languages

```
int number
```

here int is data type and number is name.

In pascal:

```
number : integer
```

here integer is data type and number is name.

In Ada:

```
NUMBER : integer
```

here integer is type and NUMBER is name.

Note that by convention, in Ada and pascal (which not case sensitive), we distinguish between reserved words and user defined labels by writing one or the other using upper-case characters.

In some imperative languages we can specify the possible values for a data item, in which case the compiler may deduce the type of a data item according to the nature of these values.

For example in pascal:

```
number: 1 ... 10;
letter: 'A' .. 'C';
```

Here the first item, number, will be considered to be an integer (because the range of possible value for the item are integers); latter, will be considered to be a character (because the range of possible values for the item are characters).

2. Assignment:

Assignment is the process of associating a value with a data item. Usually achieved using an assignment operator.

For example in C:

```
number = 3;
```

In pascal:

```
number: = 3;
```

In Ada:

```
Number: = 3;
```

Here we have assigned the value 3 to a data item called "number". Note that the assignment operator in Ada and pascal is := and in C is =. This is a common distinction between the Algol and C style of programming language.

3. Multiple Assignment:

Many imperative languages (Ada and pascal, but not C) support the concept of multiple assignment where a number of variables can be assigned a value (or values) using a single assignment statement.

For example in Ada:

```
NUMBER 10, NUMBER 2: = 2;
```

In pascal we do not have to assign the same value to each variable when using a multiple assignment statement:

```
number 1, number 2,: = 2, 4;
```

In pascal we can also write:

```
number1, number 2: = number 2, number 1;
```

which has the effect of "doing a swap".

4. Initialisation:

Initialisation is the process of assigning an initial value to a data item. Some imperative languages allow this to be done on declaration (that is C and Ada) others do not (that is pascal).

For example in C:

```
int number = 2;
```

In Ada:

```
NUMBER: integer: = 2;
```

In case of pascal we must first declare a data item and then assign a value to it, consequently the concept of initialisation does not exist in pascal.

5. Ordering of declarations:

In some imperative languages (notably pascal) declaration must be presented in a certain order, that is constants before variables.

In pascal constants are grouped together in a CONST section, followed by variables grouped together in a VAR section.

6. Positioning of declaration statements:

In languages such as C, Ada and pascal data declarations are expected to be found at the start of a procedure or function definition (with the exception of C global data items).

Some languages allow declarations to occur anywhere within procedure or function definition (for example the object oriented languages Java and C++). whatever the case remember that:

- A data item can not be used until it is declared.
- Good software engineering practice dictates that declarations should be arranged in some sensible and systematic manner, for example immediately before the first time they are used.

4.1.4 Variables

Given that we can always replace a particular bit pattern with another, the value of a data item can always be changed. Imperative languages support such change. Data items that are intended to be used in this way are referred to as variables.

Let us discuss some facts about variable:

- The value associated with a variable can be changed using an assignment operation.
- In most imperative languages (including Ada and C) data items are assumed to be variables by default (in pascal we must include the key word VAR in the declaration section).
- It is possible to declare a data item that has no value (other than an arbitrary bit pattern). Such a data is referred to as an *uninitialised variable*. But uninitialised variables are dangerous.
- Individual imperative languages treat the phenomena of uninitialised variables differently.

Common approaches include:

1. Ignore the problem and leave it up to the programmer not to use them.
 2. Consider their usage to represent an error.
 3. Allocate an appropriate value of such variables.
- Most imperative languages either ignore the problem completely (Modula II), or at least to a large extent (C and Ada).
- Ada initialises pointers to NULL and otherwise ignores the problem.
C initialises global variables to some appropriate "zero" value and otherwise ignores the problem.

4.1.5 Constants

It is sometimes desirable to define a data item whose value cannot be changed. Such data items are referred to as *constants* and are usually by incorporating a predefined key word in to declaration.

For example in C language.

```
const int label = 3;
```

In Ada:

```
LABEL: constant: = = 3;
```

Note that in Ada we do not declare the data type of the constant because this can be deduced from its value.

In Pascal we declare constants by grouping them together in a const section:

```
Const
  label = 2;
  x = 3.14159;
VAR
  number integer;
```

We can think of a constant as a data item comprising only a name and a value, that is no address.

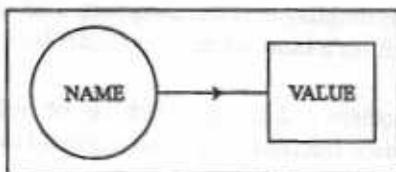


Fig. 4.3 A constant data item

However it should be appreciated that what we are doing here is telling the compiler to "flag" the data item as a constant (that is we are only instigating software protection). Theoretically we can still change the bit pattern representing the value.

4.1.6 Global and Local Data

Every data items have associated with following two things:

1. *A lifetime*: The period during the running of a program when they can be said to exist.
2. *A visibility*: The parts of the program from where they can be accessed ("seen").

The nature of the lifetime and visibility of a data item is dictated by what are referred to as scoping rules. In this respect there are two types of data:

1. Global data, which has a lifetime to an entire program and visible from anywhere within that program.
2. Local data, whose lifetime and visibility is in some way limited.

Now at this stage of discussion, it is appropriate to present a number of example programs that feature different kinds of data.

Example 4.1 We are discussing a C program that has two global data items, a variable global and a constant global-const. Note that:

- (a) Global data items are declared at the top of a program outside of any functions or procedures.

- (b) As in Java, C program must include a function main where execution commences.
- (c) The program output four data items, preforms some (mixed mode) arithmetic, and then output the same data items.

```
# include <stdio.h>
int global_var = 0;
const int global_const = 1;
/* Main function */
int main (void)
{
    int localvar = 2;
    const int localconst = 3;
    Print f ("%d, %f, %d\n", global_var, global_const,
    localvar, localconst);
    global_var = global_const + (localvar * localconst);
    Print f ("%d, %d, %f, %d\n", global_const, localvar, localconst);
}
.
```

Example 4.2 Here we are discussing the pascal program with same functionality as of C program in example 4.1. Note here:

- (a) The reference to output is a reference to library files
- (b) Data items are order in a declaration section.
- (c) Initialisation is not permitted.
- (d) Pascal is block nested language where global data items are defined in the *outermost* level.

```
Program myprog (output);
  {Example program}
  const
    globalconst = 1;
  var
    globalvar: integer;
    procedure proc 1;
      {second level procedure}
      const
        localconst = 3;
      var
        localvar : integer;
      begin (proc-1)
        localvar := 2;
        write (globalvar);
        write (',', globalconst);
```

```

        write (;; localvar);
        writeln (localconst);
        {sum}
        globalvar: = globalconst + (localvar * localconst);
        {out write}
        write (globalvar);
        write (;; globalconst);
        write (;; localconst);
        end; {pro{1}
begin {my prog}
    globalvar: = 1;
    Pro 1;
end. {my prog}

```

4.2 ANONYMOUS DATA ITEMS

Often we use data items in a program without giving them a name: such data items are referred to as anonymous data items.

Example include (in C);

$$\begin{aligned} & 4 + (5 \times 6) \\ & \text{number} + (5 \times 6) \end{aligned}$$

Here the arithmetic sub-expression, (5×6) , is processed first and the result stored somewhere as a data item which has a value and an address. However it has no name, consequently such a data item is known as an anonymous data item. The significance is that anonymous data items cannot be changed or used again in other parts of a program. Conceptually we can think of an anonymous data item as a data item without a name, consequently we can not access its address (or its value).

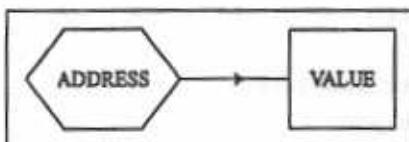


Fig. 4.4 An anonymous data items

After discussing data item thoroughly we can summarise the above as follows:

- Data items can be variable or constants.
- Variable or constants can be global or local.
- Data items are “introduced” using a declaration statement.
- Data items have an “initial” value associated with them through a process known as initialisation.
- The value associated with a variable can be changed using an assignment operation.

4.3 RENAMING (ALIASING)

It is sometimes useful, given a particular application, to rename (alias) particular data items, that provide a second access path to it.

This is supported by some imperative languages such Ada, but this concept is not supported by languages like C or pascal.

For example in Ada:

```
ITEM: integer = 1;
ONE: integer renames ITEM;
```

Here we have declared a data item ITEM and then allocated a second name (ONE) to the item.

Thus, conceptually, renaming creates a data item with more than one name (but not one address and consequently only one value), that is we have more than one access route to the address.

Let us viewed the concept with the following figure:

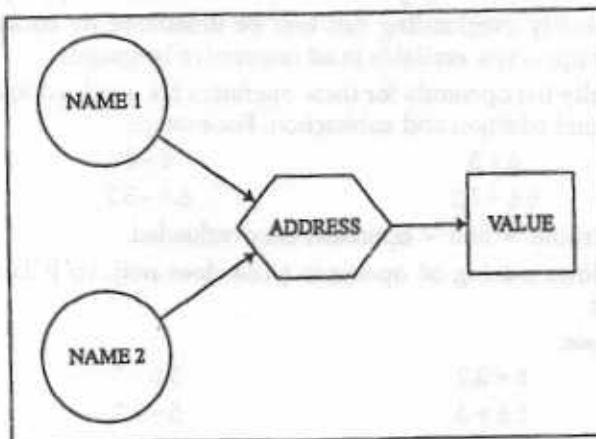


Fig. 4.5 Aliasing

It is dangerous to rename variables, since any change made to one data item results in an identical change to renamed data item. This is because both names represent the same data item.

4.4 CONCEPT OF OVERLOADING

In case of renaming we binds more than one name to one data item; When, we binds two or more data items (usually of different types) to one name then this concept of binding is said to be overloading.

Following Fig. 4.6 make the concept more clear.

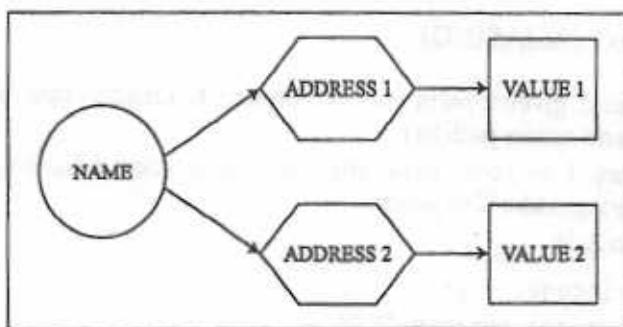


Fig. 4.6 Overloading

Now let us study some facts about overloading.

- Neither Ada or C explicitly support overloading, however:
 1. In C overloading can be implemented by declaring a compound type known as a *union*.
 2. In Ada the same effect can be contrived using an appropriate defined *record*.
- More generally overloading can best be illustrated by considering the '+' and '-' arithmetic operators available in all imperative languages.
- Traditionally the operands for these operators are overload to allow both integer and floating point addition and subtraction. For example:

$$\begin{array}{r} 6 + 3 \\ 6.6 + 3.2 \end{array}$$

$$\begin{array}{r} 6 - 3 \\ 6.6 - 3.2 \end{array}$$

- We say that the '+' and '-' operators are overloaded.
- C also allows mixing of operands (Ada does not), so it is said to be *mixed mode* arithmetic

For example:

$$\begin{array}{r} 5 + 3.2 \\ 5.6 + 3 \end{array}$$

$$\begin{array}{r} 5.6 - 3 \\ 5 - 3.2 \end{array}$$

4.5 TYPE CHECKING

The activity of ensuring that the operands of an operator are of compatible type, is said to be type checking. Here compatibility means that either legal for the operator or is allowed under language rules to be implicitly converted by compiler-generated code to a legal type.

Every expression require their operands to be of a certain type. Imperative compilers will thus check that this is the case. This is called *equivalence testing* or *equivalence checking*. Let us focus on some of the main points of type checking:

- Strongly typed languages require exact equivalence.
- Broadly we can identify two types equivalence testing:
 1. Structural equivalence (Algol 68)
structural equivalence requires that two have the same value set and the same operators.

2. Name equivalence (Ada)

Name equivalence requires that two type have the same name.

- C uses name equivalence for structures and unions, and structural equivalence for everything else.
- If language supports only compile time binding, then type checking can always be *done statically*, that is at compile time.
- If language supports runtime binding then it requires type checking at run time, which is called *dynamic type checking*. For example languages like Java script and APL supports dynamic type checking.
- Some times Programs written in any specific language are implemented in such a way that each name has a single type associated with it, and type is known at compile time then it is called strong typing.

For example C and C++ are not strongly type languages because both include union types, which are not type checked. Ada is nearly strongly typed.

4.5.1 Coercion

In some cases it may be appropriate, when a compiler finds an operand that is "not quite" of the same type to alter the operand so that it is of the right type. This is called *coercion* language C supports coercion but Ada does not. A special example of coercion is *voiding* which takes an arbitrary value and convert it to be of type void.

4.5.2 Casting

It is sometime necessary for a program to force a coercion and this forcive coercion is called a *cast*.

Casting is supported by C (but not Ada). Let us see a programming example of C:

```
int main (void)
{
    float j = 2.0;
    int i;
    i = (int) (j * 0.5); /* casting takes place here */;
}
```

4.5.3 Conversion

Now we know that there are certain languages, which do not support casting then in that case the most common alternative is *explicit conversion*. This is supported by.

Let us consider an example of Ada language: T is the name of a numeric type and N has another numeric type then the result of T(N) is of type T.

The distinction between conversion and casting is that the latter relies on the compiler's coercion mechanism to achieve the end result, while a conversion obtains the desired result explicitly through a call to a conversion function.

4.6 PRIMITIVE DATA TYPES

Data types that are not defined in terms of other types are called *primitive data types*. Nearly all programming provide a set of primitive data types. So of the primitive data types only depend on hardware specification of machine for example, integer types.

The primitive data types of any language are used, along with one or more type constructors, to provide structured types.

Every imperative languages support a number of standard basic data types. Let us study the basic data types of some of the languages as follows:

<i>Data Item</i>	<i>Ada</i>	<i>C</i>	<i>Pascal</i>	<i>Modula-2</i>
Character (integer in the range n ... m)	character	char	char	CHAR
Integer (range n ... m)	integer	int	integer	INTEGER
Natural Number (Range 0 ... m)				CARDINAL
Real or Floating point number	float	float	real	REAL
Logical type	boolean		boolean	BOOLEAN
Character String	string			text
Void		void		

Note that type void supported by C indicates a type whose value corresponds to "nothing".

To make the concept of primitive data type, let us study the Java language primitive data types as example.

The following table lists, by key word, all the primitive data type supported by *Java platform*, their sizes and formats, an a brief description of each.

<i>Key word</i>	<i>Description</i>	<i>Size/Formal</i>
Integers		
byte	Byte-length integer	8-bit two's complement
short	short interger	16-bit two's complement
int	Integer	32-bit two's complement
long	Long integer	64-bit two's complement
Real numbers		
float	Single-precision floating point	32-bit IEEE 754
double	Double-precision floating point	64-bit IEEE 754
other types		
char	A single character	16-bit unicode character
boolean	A boolean value (true or false)	true or false

In other programming languages (other than Java), the format and the size of primitive data type can depend on the system on which the program is running. In contrast, the Java programming language specifies the size and the format of its primitive data types. Hence we don't have to worry about system dependencies.

4.6.1 Numeric Data Types

Numeric data types play an important role in all the programming languages. Numbers are so important in Java that 6 of the 8 primitive types are numeric.

Let us discuss some important facets about numeric data types;

- The term *range* refers to the minimum and maximum value that numeric types can take.
- The term *precision* refers to the number of significant figures with which a numeric type is stored.
- All numeric types have default precision and/or ranges associated with them.
- Different imperative languages specify ranges and precision in different ways, but there are three principal styles:
 1. Predefined (cannot be influenced by the programmer)
 2. Adjectives (For example C language)
 3. Explicit specification (For example Ada, pascal)

Adjectives are used to specify classes of numeric types. Some C examples are given below:

Adjectives and Type	Size in Bits	Values
Unsigned short int	16	0 to 65532
Signed short int, short int, short	16	-32768 to 32767
Unsigned long int	32	0 to 9294967296
Signed long int, long int, long, int	32	-214748368 to 2147483647
Float	32	Real number to approx 6 sig. figs.
Double	64	Real number to approx 15 sig. figs.
long double	128	Real number to approx 33 sig. figs.

Explicit specification is most high level approach to defining range/precision. Ada and pascal support this concept. In Ada this is achieved using a general type declaration statement as follows:

```
type SCORE is range 0 ... 100;
type PERCENTAGE is range 0.0 ... 100.0;
```

4.6.1.1 Integer

The most common primitive numeric data type is integer. Many computers now support integers of different sizes, and these capabilities are reflected in programming languages. For example Java supports byte, short, int and long.

Integer Primitive Data Types of Java

Type	Size	Range
byte	8 bits	-128 to +127
short	16 bits	-32,768 to +32,767
int	32 bits	(about) - 2 billion to + 2 billion
long	64 bits	(about) - 10 E 18 to +10 E 18

Every integer is represented by bit pattern in computer, and typically the leftmost bit representing the sign. In a negative integer, leftmost bit represents negative and the remainder of the bit string represents the absolute value of the number.

Negative integers can be represented by either one complement notation or by two complement notation. Now a days most of computers follows two complement notation since one complement notation has two representation of zero. Two complement notation is convenient for addition and subtraction.

4.6.1.2 Floating Point Types

If we use the literal 197.0 in a program, the decimal point tells the compiler to represent the value using floating point primitive data type. The bit pattern used for floating point 197.0 is very much different than used for integer 197. Most languages include two floating point types

(i) float

(ii) double.

- Let us see the example of Java floating point:

Floating point Primitive Data Types In Java

Type	Size	Range
float	32 bits	-3.4 E + 38 to +3.4 E + 38
double	64 bits	-1.7E + 308 to 1.7E + 308

Now a days almost all new machines use the IEEE Floating point standard 754 format.

Data type float is sometimes called "single-precision floating point". Data type double has twice as many bits and is sometimes called "double-precision floating point". Let us analyse the concept by following Fig. 4.7.

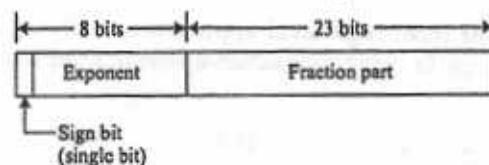


Fig 4.7(a) single precision (total 32 bits required)

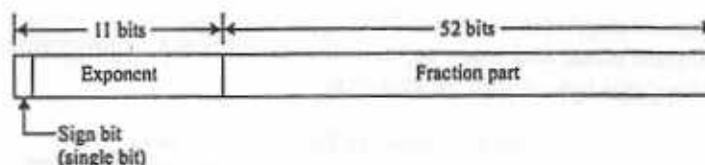


Fig 4.7(b) Double precision (Total 64 bits required)

Sometimes we need to explicitly ask for a single-precision float literal. We can do this by putting a lower case 'f' or upper case 'F' at the end, like this:

123.0f -123.5F -198234.234f 0.00000381F

Sometimes we need to explicitly ask for a double-precision float literal. We can do this by putting a lower case 'd' or upper case 'D' at the end, like this:

123.0d -123.5D -198234.234d 0.00000381D

Remember, that without any letter at the end, a floating point literal will automatically be of type double.

On paper integers have no decimal point, and floating point type do. But in main memory there are no decimal points. Even floating point values are represented with bit patterns.

4.6.1.3 Decimal

The computer which support large business application, have hardware support for decimal data types. Decimal data types generally represented by binary coded decimal (BCD) and stored very much like character strings.

Language like COBOL and C# supports this concept. During the memory allocation process to the decimal data type a fixed number of decimal digits are required and decimal point is at fixed position in value.

4.6.2 Character Types

Character are very common in computer. The primitive data type used has the name char to save typing. There are 16 bits used to represent characters using the char type in Java language.

In many programming languages, only 8 bits are used for this purpose. C is the example of language which use 8 bits for character.

In Java, 16 bits are used in order to represent characters from human languages other than English. The method used is called *Unicode*. For example, here is a 16 bit pattern

000000000110011

These 16 bits are of data type char and if we look in a table then these bits represents 'g'. But these 16 bits also represent the integer 103 if they are regarded as data type short.

In a program, a character literal is surrounded with an apostrophe on both sides:

'm' 'y' 'A'

In a program, control characters are represented with several characters inside the apostrophes:

'm' '\t' '\377'

Each of these is what we do in a program to get a *single* char. In Java language, first one represents the 16 bit newline character, the second one represents the tabulation character, and last one is the delete character.

4.6.3 Boolean Type

Another of the primitive data types is the type boolean. It is used to represent a single true/false value. A boolean value can have only one of two values:

true false

The concept was introduced in ALGOL 60. C is the exception, in which numerical expressions can be used as conditionals. In such expressions, all operands with non zero values are considered true and zero is considered as false.

C++ and Java supports boolean type. The data type boolean is named after George Boole, a nineteenth century mathematician, who discovered that a great many things can be done with true/false values (otherwise known as bits).

4.6.4 Pointers and Access Values

Pointers are variable that have as their value an address, that is a reference to another data object.

Let us make the concept clear through following figure:

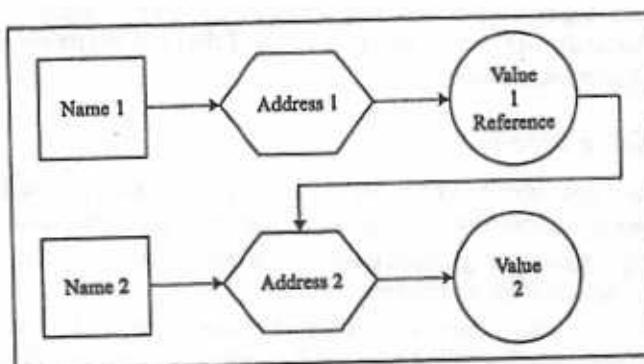


Fig. 4.8 C Pointer

The consequence of above figure is that we have a second "access" route to value 2 in Fig. 4.8; the standard route through name 2 and a second route by dereferencing the value 1 accessed through Name 2.

A pointer can take a special value which indicates that the pointer is pointing nowhere. Given a pointer we can access the value "pointed" to through a process known as dereferencing.

Let us consider the *pointers in C language*:

C makes extensive use of pointer and uses two prefix operators:

(i) * (Asterisk)

(ii) & (Ampersand)

- * When used in declaration is read as "pointer to a data type". For example;

```
int * numberptr;
```

- * When used elsewhere is interpreted as "the value contained in the reference/address pointed at" (that is the pointer is dereferenced). For example:

$$\ast \text{numberptr} = \ast \text{numberptr} + 1,$$
- & when used anywhere is read as the "reference/address of". For Example

$$\text{number ptr} = \& \text{x}$$

In this case x is referred to as a reference variable.

Now let us consider the following C example:

```
# include <stdio.h>
Void main (void)
{
    int n, * nptr;
    n = 2,
    nptr = &n;
    Printf ("n = % d", n);
    Printf ("address of n = %d", &n);
    Printf ("nptr = %d/n", nptr);
    Printf("address of nptr = %d/n", & nptr);
    Printf("value pointed at = % d/n", * nptr);

    n = 4
    Printf("n = %d/n", n);
    Printf("address of n = %d/n" &n);
    Printf("nptr = % d/n", nptr);
    Printf("address of nptr = % d/n", & nptr);
    Printf("value pointed at = % d/n", * nptr);
}
```

The output will be following:

```
n = 2
address of n = 206 308 8352
nptr = 2063808352
address of nptr = 2063808356
value pointed at = 2
n = 4
address of n = 2063808352
nptr = 2063808352
address of nptr = 206 380 8356
value pointed at = 4.
```

Note that the address of number and the value of nptr are the same. we can illustrate this as shown in Fig. 4.9.

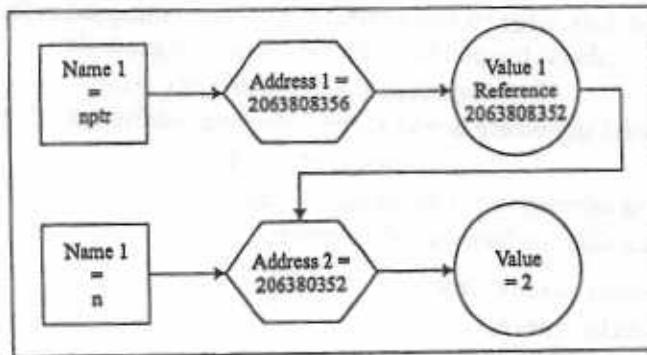


Fig. 4.9 C pointer example

4.6.4.1 Operations on Pointers

Basically operation on pointers is the issue related to the language design, so different languages permitted different type of operations. Let us discuss some operations:

1. *Dereferencing*: By the dereferencing process, we can access the value "pointed" by a pointer.

For example in C: `* nptr`

In pascal: `^nptr`

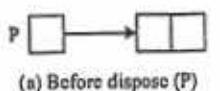
2. *Dynamic allocation on the heap*: By applying this operation we can allocate the memory for new data objects in heap area.

For example, in C language, it can be done by `malloc()`, and `calloc()` functions. In pascal `new(p)` leaves pointer P pointing to a newly allocated data structure on the heap.

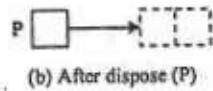
3. *Deallocation*: A dynamic data structure exist until it is explicitly released.

For example, in pascal it can be done by `dispose(p)`, here `dispose(p)` deallocates the cell that points to.

Let us see it by Fig. 4.10 as follows:



(a) Before dispose (P)



(b) After dispose (P)

Fig. 4.10

4. *Assignment*: Assignments are permitted between pointers of the same type. Let us see the example of C language:

```
int * p;
```

```
int * q;
```

```
p = q; /* since p & q are the pointer of same type */
```

5. *Equality testing*: The equality relation = tests if two pointers of the same type point to the same data structure. An inequality ≠ is allowed as well.

4.6.4.2 Insecurities of Pointers

Let us analyse a number of case of insecurities that may arise, during the pointer use and possible ways of controlling them.

1. Type-checking:

Some languages, such as pascal or Ada, requires pointer to be typed. For example a C variable `ptr` declared of type `int *ptr` is restricted to point to objects of type `integer`. This allows the compiler to type-check the correct use of pointers and object pointed to by pointers. On the other hand, other languages, such as PL/I, treat pointers as untyped data objects, that is, they allow a pointer to address any memory allocation, no matter what the contents of that location are. In these cases, dynamic type checking should be performed to avoid manipulation of the object via nonsensical operation.

2. Arithmetic operations:

Some language allows arithmetic operations on pointers. For example C allows the arithmetic operations:

```
int * ptr;
```

Now one can write `ptr = ptr + j;` where `j` is an `int` variable. This would make `ptr` refer to the memory location that is `j` `integer` objects beyond the one `ptr` is currently pointing to. Now it is up to the programmer to guarantee that object pointed by `ptr` is an `integer`.

3. Dangling pointers and Garbage:

A dangling pointer is a pointer to storage area that is being used for another purpose; typically the storage has been deallocated.

Dangling pointers may arise in languages, such as C, that allow the address of any variable to be obtained and assigned to a pointer.

Let see a program of C language:

```
int * ptr;
void dang ( )
{
    int x; /* allocates x*/
    ...
    ...
    ptr = &x; /* address of x is assigned to ptr */
    ...
    return; /* deallocate the memory of x */
}
main ( )
{
    ...
    dang (); /* after this call ptr is dangling pointer */
}
```

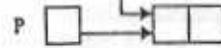
Storage that is allocated but is inaccessible is called garbage. Program module that create the garbage is said to have *memory leaks*. Assignment to pointers can lead to memory leaks

Let *ptr and *p are two pointers and pointer assignment is as follows

$\text{ptr} = \text{p}$



(a) Before assignment



(b) After assignment

Fig. 4.11

In Fig. 4.11 (a) pointer *ptr and *p are pointing two different memory cell and after assignment ptr is assigned with address of p but cell that ptr pointed to is still in memory, but it is inaccessible so it is the case of memory leak.

4. Pointers to union's Components:

Language that allow pointers to be components of a union may cause further insecurities.

Let us consider an example:

```
Union bug {
    int i;
    int * p;
}
```

Now let us declare a variable type bug, x can be assigned an integer value, which is then interpreted as a pointer to access some unpredictable location.

To resolve the problems associated with pointers the Java language has eliminated the notion of explicit pointer completely.

EXERCISE

- For an elementary data type in C language, do the following:
 - Describe the set of values that data objects of that type may contain.
 - Determine the storage representation for values of that type.
 - Define the syntactic representation used for constant of that type.
 - Determine any attribute that a data object of that type may have other than its data type.
- For a language with which you are familiar and uses static type checking, give example of construct that can not be checked statically.
- What are the elements of specification of a data object? Give the general syntax of specification of an operation.

4. Explain the difference between the type itself, variable of type and constant of the type for language of your choice.
5. Investigate the handling of type equivalence of C language.
6. With reference to storage management and data structures explain the following term:-
 - (i) Access path
 - (ii) Garbage
 - (iii) Dangling references
7. For a languages C, C++:
Write a program segment that generates garbage, and dangling referencee.
8. Write of difference between:
 - (i) malloc(), and free in C language
 - (ii) new and dispose in C++ language
9. The concept of multiple assignment is supported by Ada, discuss advantages and disadvantages of it.
10. Write the difference between following, with example of any known programming language.
 - (i) Lifetime and visibility
 - (ii) Global and local data.
11. Some languages supports anonymous data type, disadvantages of this concepts.

CHAPTER 5

Structured Data Types

5.1 INTRODUCTION TO STRUCTURED DATA TYPES

A data type that contains other data objects as its components, is said to be structured data types.

Some times structured data types also said to be *higher level* types. They may be "scalar" or "composite". Higher level types are (usually) user defined data types made up of basic types (and/or other existing higher level types).

Let us discuss some of the *attributes of structured data types*:

1. *The number of component* in a structured data type shows an important attribute of data type. If data type is of fixed size then number of components in given structured data type are fixed, for example arrays and records.
On the other hand stack, link list, and sets are examples of variable size types.
2. A data structure is said to be *homogenous* if all the components of data structure are of same type, otherwise it is said to be *heterogenous*. Clearly each type of components plays an important role in structured data type.
3. Some data structure itself restrict the *maximum number* of components in it. For example stack specified maximum size for the structure in terms of components.
4. *Accessing the each component* of a structure is another important issue. A proper data structure is required to identify the each individual component of the structure.
5. *The storage representation* for data structure is very important attribute of structured data types. The storage representation includes both *descriptor* to store all attributes of data structure and components of given structure.

There exist two basic representation.

- (a) *Sequential representation* stores data structure in a single contiguous block of memory. It can be shown in following figure.

Descriptor	Component ₁	Component ₂	...	Component _n
------------	------------------------	------------------------	-----	------------------------

Fig. 5.1 Contiguous allocation

- (b) *Linked representation* provides the freedom to store data structure in several non-contiguous blocks of storage. Every block contains the address of next block.

Let us analyse the concept through the Fig. 5.2.

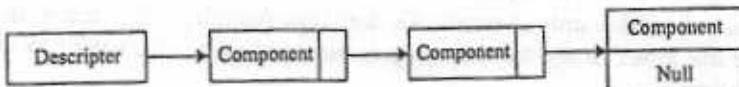


Fig. 5.2 Liked representation

There are some standard higher level types available in most imperative languages. These includes:

1. Arrays
2. Enumerated types.
3. strings
4. records/structures
5. unions

Note that all of the above are composite types, except enumeration which are scalar types.

5.2 ARRAYS AND THEIR DECLARATION

The most straight forward (and oldest) form of higher level data type is the array. An array is simply a series of data items, all of the same type, stored in a series of locations in memory such that a single value is held at each location.

Features of arrays are as follows:

1. Items in an array are called elements
2. Specific element in an array can be identified through the use of an index
3. The first index in an array is called the *lower bound* and the last the *upper bound*.
4. In some imperative languages the lower bound is always 0 (For example C), in other it can be specified (For example Ada).
5. Some time a one dimensional or linear array is called *vector*.

When declaring arrays we are doing two things:

1. Declaring the type of the array.
2. Declaring the nature of index.

The form of these declarations and the versatility with which indices can be specified is very much language dependent. Broadly we can identify a number of methods of defining indices:

1. Assume index is an integer type and:
 - Define only the upper bound (assume the lower bound is fixed)
 - Define both the lower and upper bound (assumes lower bound is not fixed, or has a default value that can be overridden).

In both the cases the index value can be expressed directly as a single integer, or indirectly in the form of some arithmetic operation.

2. Specify the data type of the index (or override the default type) and provide lower and upper bounds as appropriate.
3. Define the index only in terms of a data type (usually programmer defined), in which case the index range for the array is assumed to be equivalent to the range type in question.

To illustrate the above it is useful to consider some examples:

Example 5.1 Let us see the memory map of an linear array.

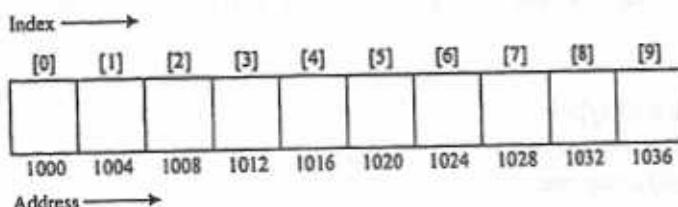


Fig. 5.3 Array structure

Example 5.2 Let us see the array declaration of C Language:

In C the facilities with which to declare arrays are much more succinct/limited compared to Ada. Array indices are always assumed to be integers, and the lower bound is always fixed at 0.

<Type> <NAME> [<UPPER_BOUND>];

For example:

int array1[10];

This declares an array of ten elements with indices ranging from 0 to 9 inclusive.

Example 5.3 In Ada, to declare arrays, we use a declaration statement of the form:

<NAME> : array <index - definition> of <TYPE>;

The key word array used here is often referred to as a type constructor in that it "constructs" a particular type. Indices in Ada can then be defined in a number of ways as follows:

1. In terms of a lower and upper bound only (assume integer index type).

Format is:

(<LOWER_BOUND> ... <UPPER_BOUND>)

Example declarations:

ARRAY1: array (1 ... 9) of integer;

START: integer;

ARRAY 2: array (START ... START + 4) of integer;

2. In terms of the index type supported by a lower and upper bound. Format is:

```
<INDEX _ TYPE> range <LOWER_BOUND> ... <UPPER_BOUND>
```

Example declaration:

```
ARRAY3: array (CHARACTER range a ... z) of integer;
```

3. In terms of a particular type. Format is:

```
(<INDEX _ TYPE>)
```

Example declaration:

```
type ITEMS_T is range 0_100;
```

```
ARRAY4: array (ITEMS_T) of integer;
```

5.2.1 Assignment of Values to Arrays Elements

Given knowledge of the index for an array element we can assign a value to that element (or change its value) using an "assignment" operation.

In Ada language assignment can be done as follows:

```
LIST1 (1) := 1;
```

In C Language:

```
list1 [0] = 1;
```

5.2.2 Compound Values

Some imperative languages support the concept of *compound values* which allow all the elements of an array to be assigned to simultaneously. Both C and Ada support compound values (but in Ada they are called aggregates).

Note that in C their usage is limited to initialisation, while in Ada array aggregates can be used in any assignment statement (not just initialisation).

Let us see an example of Ada:

```
MY_ARRAY: array (0-10) of integer  
:= (9, 8, 7, 6, 5, 4, 3, 2, 1, 0);
```

Ada also support an alternative to the above:

```
MY_ARRAY := (5, 4, 3, 2, 1, other => 0);
```

The same statement can be written in C as follow:

```
int myarray [10] = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
```

5.2.3 Operation on Complete Arrays

Some programming languages, such as PL/1 (but not Ada or C), allow operations on complete arrays.

For example in PL/1 we can write:

```
. IA = 0;
```

This will make all elements in the array IA equal to 0. The same effect can be achieved in Ada by writing:

```
IA: = (other => 0);
```

Alternatively, some imperative languages (pascal) support assignment of one array to another:

```
ARRAY 1:= ARRAY 2;
```

where ARRAY 1 and ARRAY 2 are two arrays of precisely the same type (such a capability supported by Ada and pascal, but not C language).

5.2.4 Implementation of One Dimensional Array/vector

Implementing arrays requires more compile time effort than does implementing simple types, such as integer. The code to allow accessing of array elements must be generated at compile time. At run time, this code must be executed to produce element address. Accesses to array elements, especially in arrays with several subscripts, are more expensive than necessary if the access code is not carefully designed. This is true regardless of whether the array is statically or dynamically bound to memory.

Let us analyse the storage area for one dimensional array:

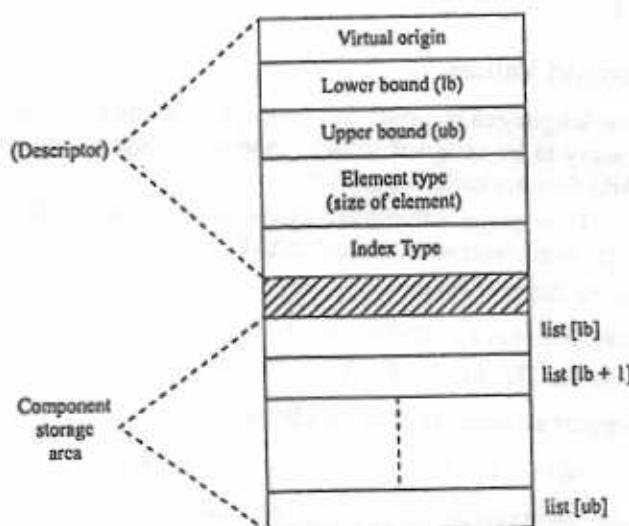


Fig. 5.4 Memory map for one dimensional array

From the above figure, it is clear that a single dimensional array is list of adjacent memory cells. The descriptor includes information required to construct the access function. If the run time checking of index ranges is not done and the attributes are all static, then only the access function is required during execution and no descriptor is needed. If run time checking of Index ranges is done, then those index ranges may need to be stored in a run time descriptor.

If the subscript ranges of a particular array type are static the ranges may be incorporated into the code that does the checking thus eliminating the need for the run time descriptor. If any descriptor entries are dynamically bound, then those parts of the descriptor must be maintained at run time.

Let a one dimensional array is list [K] and the array list is defined to have subscript range lower bound 1.

The access function for list can be written as follows:

$$\text{address } (\text{list}[i]) = \text{address } (\text{list}[1]) + (K - 1) * \text{element size}$$

here address (list[i]) represent the address of ith element of the list, which we want to access. address (list[1]) represents the base address of the list so above equation can be written as follows:

$$\text{address } (\text{list}[i]) = \alpha + (i - 1) * \text{element size}$$

where α is base address.

This simplifies to

$$\text{address } (\text{list}[i]) = (\alpha - \text{element size}) + (i * \text{element size})$$

where the first operand of the addition ($\alpha - \text{element size}$) is the constant part of the access function, and the second is the variable part.

If the element type is statically bound and the array is statically bound to storage, then the value of the constant part can be computed before run time. Only the addition and multiplication operations remain to be one at run time. If the base, or the beginning address, of the array is not known until run time, the subtraction must be done when the array is allocated.

If the lower bound is lb then generalization of the access function is as follows:

$$\text{address } (\text{list}[i]) = \text{address } (\text{list}[lb]) + (i - lb) * \text{element size}$$

This simplifies to

$$\text{address } (\text{list}[i]) = (\alpha - lb * \text{element size}) + (i * \text{element size})$$

If element size is e then,

$$\text{address } (\text{list}[i]) = (\alpha - lb * e) + (i * e)$$

For example in C char arrays, e is 1 and lb is always 0, the equivalent C accessing function will be:

$$\boxed{\text{address } (\text{list}[i]) = (\alpha - 0) + (i * 1) = \alpha + i}$$

Let us now address the element with subscript 0 of our list, then accessing function will be as follows:

$$\begin{aligned} \text{address } (\text{list}[0]) &= (\alpha - lb * e) + (0 * e) \\ &= (\alpha - lb * e) \end{aligned}$$

It is the base address and said to be virtual origin of the list, so

$$\boxed{\text{virtual origin} = \alpha - lb * e}$$

So if we want to access i th component of the list then address can also be written as follows:

address (list [i]) = virtual origin + $i \times e$
--

5.2.5 Constrained and Unconstrained Array

A constrained array is an array where the index is specified (and hence the number of component is specified), we say that the bounds are static, hence constrained arrays are sometimes referred to as static arrays.

Many imperative languages (including Ada) support the concept of unconstrained arrays. C and Pascal do not support this concept, however, C does provide facilities where such arrays can be simulated.

Ada makes use of the symbol $\langle\rangle$ to indicate an unconstrained array:

```
type TYPE _ NAME = array (T1 range <>) of T2;
DATA_ITEM_NAME: TYPE_NAME (START... END);
```

where T₁ and T₂ are type identifiers (not necessarily of the same type), and the data items START and END are of type T₁.

Example declaration:

```
type LIST is array (INTEGER range <>) of FLOAT;
L1 : LIST1 (START ... END);
```

5.2.6 C Dynamic Arrays

Although C does not support the concept of unconstrained arrays it does provide facilities to delay the declaration of an upper bound of an array till run time, that is the upper bound is declared dynamically hence such an array is referred to as a dynamic array. To achieve this two library functions malloc and free are used. The malloc (<size>) function obtains a block of memory (for the array in this case) according to the parameter <size>. The type of this parameter is system dependent not usually is an int or unsigned int. The free function releases the memory when it is no longer required.

Let us see an example of C:

```
# include <stdlib.h>
# include <stdio.h>
void main (void)
{
    int num, index = 0;
    int * numptr = NULL;

/* Get size of array and create space (malloc returns NULL if no space available). */

    Scanf ("%d", & num);
    numptr = (int *) malloc (size of (int) * num);
    if (numptr == NULL)
```

```

{
    Printf("Not enough memory\n");
    exit (1);
}
/* Initialise */
while (index <= num)
{
    numptr [index] = index;
    index++;
}
/* output */
index = 0;
while (index <= num)
{
    Printf ("%d", numptr [index]);
    index++;
}
Printf("/n");
/* End */
free (numptr);
}

```

5.2.7 TYPES OF ARRAY

Other than the standard array forms described above identify a number of alternative "types" of array which are a feature of particular imperative languages.

These include:

1. Flexible arrays
2. Multi-dimentional arrays
3. Lists
4. Arrays of arrays
5. Slices.
6. Sets
7. Bags

5.2.7.1 *Flexible Arrays*

A powerful facility associated with some imperative languages is the ability to dynamically "resize" an array after it has been declared, that is during run-time. Such an array is referred to as a flexible array.

Neither Ada or C supports flexible arrays but Algol' 68 does. However, a similar effect can be achieved in C using the built - in function malloc and calloc. The latter dynamically extends or contracts the memory space available for a previously declared array.

Let us analyse this concept by a C example:

```
# include <stdlib.h>
# include <stdio.h>
int num = 4;
void main (void)
{
    int *numptr = NULL;
    /* Allocate memory and initialise */
    numptr = (int *) malloc (size of (int) * num);
    numptr [0] = 2;
    numptr [1] = 4;
    numptr [2] = 6;
    numptr [3] = 8;
    /* we can write output code here */
    /* Reallocate memory and reinitialize */
    num = 3;
    numptr = (int *) realloc (numptr, size of (int) * num);
    numptr [0] = 1;
    numptr [1] = 3;
    numptr [2] = 5;
    /* again we can write the output code */
    free (numptr);
}
```

5.2.7.2 Multi-dimensional Arrays

Multi-dimensional arrays are arrays where the elements have more than one index. In the case of two-dimensional arrays these can be envisaged as tables comprising a number of rows and column such that the row number represents one index and the column number a second index.

For example multi-dimensional array can be declared in pascal as follows:

```
X : array [1 - 10, 1 - 5] of real;
```

here X is the array having 10 rows and five columns and data objects stored in array are real type.

Declaration in C is as follows:

```
int X [10][5];
```

This declaration also having the same meaning but here data type are integer type.

Let us analyse a C Program, which shows the implementation of two dimensional array:

```
void main (void)
{
    int twodim [3][2] = {{1, 2}, {3, 4}, {5, 6}};
    Printf ("size of array = %d (bytes)\n", size of (twodim));
    Printf ("Num elements = % d\n", size of (twodim)/size of
           (twodim [0][0]));
    Printf("array comprises = (%d, %d), (%d, %d), (%d, %d)\n",
           twodim [0][0], twodim [0][1], twodim [1][0]
           twodim [1][1], twodim [2][0], twodim [2][1]);
}
```

Three dimensional arrays can then be envisaged as comprising a book of tables such that the third index represents page number.

Four dimensional arrays can then be envisaged as a set of books of tables and so on.

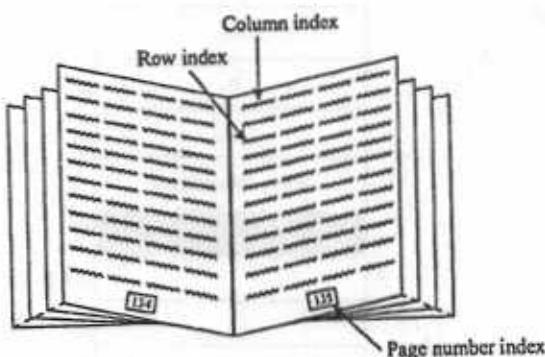


Fig. 5.5 Conceptualisation of a three-dimensional array through a book

5.2.7.2.1 Implementation of Multi-dimensional Array

Implementation of multidimensional array will decide their memory map in storage area.

Multidimensional arrays are more complex to implement than single dimensional arrays, although the extension the more dimensions is fairly straight forward. The memory implementation is linear – it is usually a simple sequence of bytes. So values of data types that have two or more dimensions must be mapped onto single dimensioned memory. There are two common ways in which multidimensional arrays can be mapped to one dimension:

1. *Row major order:*

In row major order, the elements of the array that have as their first subscript the lower bound value of that subscript are stored first followed by the elements of the second value of the first subscript and so forth (In other word, first row is stored first, then second row and so forth).

For example, if the matrix had the values

3	4	7
6	2	5
1	3	8

it would be stored in row major order as:

3, 4, 7, 6, 2, 5, 1, 3, 8

The storage representation of the row major order can be analyse by the Fig. 5.6 as follows:

In figure, we are taking above discussed matrix as example for storage representation.

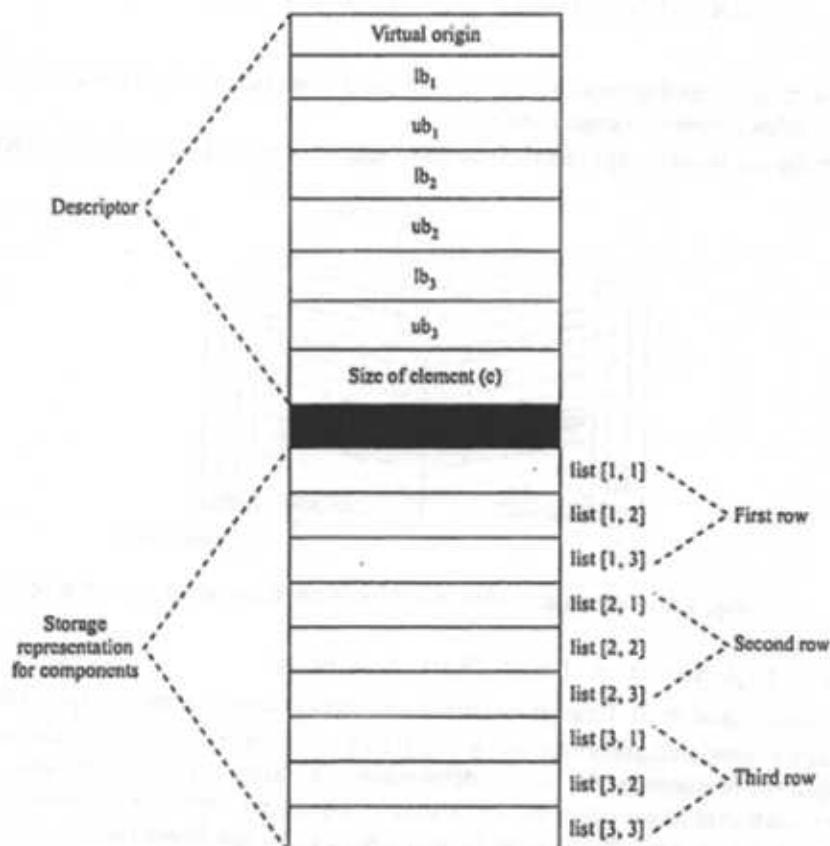


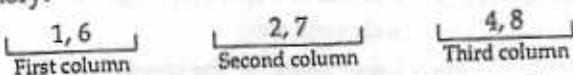
Fig. 5.6 Memory representation for above given multidimensional array

2. Column major

In column major order, first column is stored first, then second column and so forth . If the array is following matrix:

1	2	4
6	7	8

An matrix above, were stored in column major order, it would have the following order in memory:



Column major order is used in FORTRAN, but the other languages use row major order.

Memory map can be design in the same manner as we did in row major order.

5.2.7.2.2 Access Function for Multidimensional Array

The access function for a multidimensional array is the mapping of its base address and a set of index values to the address in the memory of the element specified by the index values.

The access function for two dimensional arrays stored in row major order can be developed as follows.

In general, the address of an element is the base address of the structure plus the element size times the number of elements that precede it in the structure. For a matrix in row major order, the number of elements that precedes on element is the number of rows above the element times the size of row, plus the number of element to the left of the element.

Let us make the simplifying assumption that subscript lower bounds are all 1.

To get an actual address value the number of elements that precedes the desired element must be multiplied by the element size. Now the access function can be written as

$$\begin{aligned} \text{address}(\text{list}[i, j]) &= \text{address of list}[1, 1] + (((\text{number of rows above the } i\text{th row}) \\ &\quad \times (\text{size of a row}) + (\text{number of elements left of the } j\text{th column})) \\ &\quad \times \text{element size}) \end{aligned}$$

Because the number of row above the i th row is $(i - 1)$ and the number of elements to the left of the i th column is $(j - 1)$.

$$\begin{aligned} \text{Base address } (\alpha) &= \text{address of first element} \\ &= \text{address}(\text{list}[1, 1]) \end{aligned}$$

$$\text{size of an element} = e$$

$$\text{number of element per row} = n$$

above access function can be written as follows:

$$\text{address}(\text{list}[i, j]) = \alpha + (((i - 1) \times n) + (j - 1)) \times e$$

Let us simplify the expression.

$$\begin{aligned} \text{address}(\text{list}[i, j]) &= \alpha + ((i \times n - n + j - 1) \times e) \\ &= \alpha + ((i \times n + j) \times e - (n + 1) \times e) \\ &= \alpha - (n + 1) \times e + (i \times n + j) \times e \end{aligned}$$

$$\text{virtual origin (VO)} = \alpha - (n + 1) \times e$$

So finally $\text{address}(\text{list}[i, j]) = \text{virtual origin} + (i \times n + j) \times e$

$$\text{address}(\text{list}[i, j]) = \text{VO} + (i \times n + j) \times e$$

The generalization of this arbitrary lower bound results in the following access function:

$$\text{address}(\text{list}[i, j]) = \text{address of a}[rlb, clb] + (((i - rlb) \times n) + (i - clb)) \times \text{element size}$$

Here

rlb = lower bound of the rows

clb = lower bound of the columns

α = address($\text{list}[rlb, clb]$) = Base address

n = number of the element in a row

e = size of a element

then access function of element located at address ($\text{list}[i, j]$) can be rearranged as follows:

$$\begin{aligned}\text{address}(\text{list}[i, j]) &= \alpha (((i - rlb) \times n) + (j - clb)) \times e \\ &= \alpha + ((i \times n - rlb \times n + j - clb) \times e \\ &= \alpha - (rlb \times n + clb) \times e + (i \times n + j) \times e\end{aligned}$$

Here

$$VO = \alpha - (rlb \times n + clb) \times e$$

$$\boxed{\text{So } \text{address}(\text{list}[i, j]) = VO + (i \times n + j) \times e}$$

So clearly for each dimension of an array, one add and one multiply instruction is required for the access function.

5.2.7.3 Lists

Lists (or sequence) can be considered to be special types of arrays but:

- with an unknown number of elements, and
- without the indexing capability.

List are popular in logic and function languages but not in imperative languages.

5.2.7.4 Arrays of Arrays

Some imperative language (including Ada and C) support arrays of arrays:

Let us see on example Ada language.

With CS – IO; Use CS – IO;

Procedure ARRAY_OF_ARRAY is

```
type ARRAY_T is array (1.. 2) of integer;
type TWO_D_ARRAY-T is array (1.. 3) of ARRAY_T;
A1: ARRAY_T: = (1, 2);
A2: ARRAY_T: = (3, 4);
A3: ARRAY_T: = (5, 6);
IA: TWO_O_ARRAY-T:= (A1, A2, A3);
begin
    put (IA (1)(1));
    put (IA(1) (2)); new-line;
    put (IA(2) (1));
```

```

put (IA(2)(2)); new - line;
put (IA(3)(1));
put (IA(3)(2)); new - line;
end ARRAY_OF_ARRAY;

```

In above Ada programming example TWO_D_ARRAY_T is array of arrays.

Note that in Ada the "sub-arrays" must all be of the same type (that is size and length), consequently the distinction between arrays of arrays and two-dimensional arrays can be blurred.

Arrays of arrays in C are created in similar manner.

5.2.7.5 Slices

A slice is a sub_array of an array. The process of slicing returns a sub_array. This concept is supported by Ada language but not supported by C language.

Let us see example of Ada languages.

A: array (integer range START .. START + 4) of integer := (2, 4, 6, 8, 10)

or

A: array (integer range START ... START + 4) of character := ('a', 'b', 'c', 'd', 'e');

The statement

A_SUB := A (START + 1.. START + 3)

would assign the sub array (4, 6, 8) or ('b', 'c', 'd') to the variable A_SUB.

From the above discussion, it is clear that slice of an array is some substructure of the array. For example, if B is a matrix, the first row of B is one possible slice, as are the last row and the first column.

It is important to realize that a slice is not a new data type. Rather, it is a mechanism for referencing part of an array as a unit.

If arrays cannot be manipulate as units in a language, that language has use for slice.

A slice of a STRING type is called substring reference.

5.2.7.6 Sets

A set type is one whose variable can store unordered collection of distinct clues from some ordinal type called its base types.

So set is a group of (distinct) elements, all of the same type, which are all possible values of some other type referred to as the base type. The relationship is similar that of an Ada sub-type to its "super-type". Neither Ada or C features sets, however pascal and modula-2 do.

Let us see a pascal set declaration example

```

Program SET_EXAMPLE (output);
type
  SOMEBASE_T = 0..10;
  SOMESET_T = set of SOMEBASE_T;

```

```

Var
    SET 1, SET 2: SOMESET_T;
begin
    SET 1: = [1, 2, 5]
    SET 2: = [6, 8, 9];
    ...
    ...
end.

```

Here we have defined a base type `SOMEBASE_T` and a set `SOMESET_T` over this base. The number of elements in a set is referred to as its cardinality. The only operations that can be performed on set are set operations, for example member, union, intersection etc.

If we compare arrays with set then it is very clear that arrays are, of course, far more flexible than sets; they allow many more operations, more complex shape, and more options for element types. In fact, if arrays were restricted to maximum length of 32, as are sets in many pascal implementations, users would not consider them acceptable.

As far as *implementation* concern, set are stored as bit strings in memory.

For example, if a set has the ordinal base type

`['b' .. 'k']`

then variables of this set type can use the first 10 bits of a machine word, with each set bit (1) representing a present element, and each clear bit (0) representing an absent element.

Using this scheme, the set value.

`['b', 'c', 'h']`

would be represented as :

`1100001000`

The pay off in this approach is that a typical operation such as set union can be computed as a single instruction, a logical OR. Set membership can also be done in a single instruction when the set cardinality is less than or equal to the machine's word size. Let us do membership test for 'C', the process could be done with AND operation between the bit string representations of the two operands.

5.2.7.7 Bags

Bags (multisets) are similar to sets except that they can contain an element more than once and record how many times a value has been inserted. The primary operations on bags are "insert value" and "remove value" (as opposed to the set operations found in sets). Very few imperative languages feature bags.

5.3 ENUMERATED TYPES

An enumerated type is a user-defined scalar type where the possible values for the type are itemised.

In Ada enumerated data types are declared as follows:

```
type DAYS is (MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY,
SUNDAY); NEWDAY: DAYS := WEDNESDAY;
```

Here we have created an "enumerated" type DAYS and declared a variable, NEWDAY, of that type. Note that the values in an enumerated type are "ordered" so that the value listed first is least and the value listed last is greatest. Conceptually at least, each value for an enumeration also has an ordinal value. This allows comparison of enumerated type values using operators such as <(less than) and > (greater than).

Ada provides a number of attributes (constants) for enumerations:

<i>First</i>	<i>Gives First value of the enumeration</i>
<i>LAST</i>	<i>Gives last value of the enumeration</i>
<i>Pred(x)</i>	<i>Gives the predecessor of the enumerated value x</i>
<i>Succ(x)</i>	<i>Gives the successor of the enumerated value x</i>

Note that value used in conjunction with the pred and succ attributes must not be the first or last values respectively. Some imperative languages that support the constant/attribute concept include an *ord(x)* attribute which returns the ordinal value of x (Pascal supports this).

C language also supports this concept, let us see an example program:

```
# include <stdlib.h>
# include <stdio.h>
void main (void)
{
    enum days { Monday = 1, Tuesday, Wednesday, Thursday, Friday,
                Saturday, Sunday};

    enum days newday = Monday;
    int n;
    Scanf ("%d", &n);
    Printf ("newDay = %d\n", newDay);
    newDay = Monday + n;
    printf ("newDay + %d = %d \n", n, newDay);
}
```

Let us see another C example with "typedef" statement.

```
void main (void)
{
    typedef enum days { Monday = 1, Tuesday, Wednesday, Thursday,
                        Friday, Saturday, Sunday} days_t;
    days_t newday1 = Monday, newday2;
    Printf ("newday1 = % d \n", newday2);
```

```

    Printf ("successor to newday1 = %d \n", newday1 + 1);
    newday2 = newday1 + 2;
    printf ("newday2 = % d \n", newday2);
    if (newday2 == 3)
        Printf ("newday2 = Wednesday \n");
    }
}

```

Let us see some more example of enumerated type:

A well known enumerated type (although not often recognised as such) is the Boolean type. This may be specified in Ada and C respectively as follows:

```

BOOLEAN_T is (FALSE, TRUE);
TypeDef enum {false, true} boolean_t;

```

Note that type Boolean is predefined (enumerated) type in some imperative language (Ada, pascal). It is also possible to consider any discrete type (that is integer, character) to be enumerated type.

5.4 CHARACTER STRINGS

A further type of array is the string. A string is a sequence of character usually enclosed in double (Ada, C) or single quotes (pascal) – for this reason strings are sometimes referred to as a character arrays.

The two most important design issues that are specific to character string types are the following

- strings are character array or a primitive type?
- strings supports static and dynamic length?

The declarations of strings depends on language implementation. In C a 10 character string is declared as follows:

```
Char list [10];
```

Other languages provides a specific predefined data type for character arrays, for example string (Ada), or text (Pascal). Ada example:

```
NAME : string (1- 10);
```

5.4.1 Operations of Strings

Operations on the string depends on the implementation, that, string is character array or primitive data type. Languages like pascal, C, C++ and Ada considers strings as character arrays. In pascal, although string are not a primitive type, char arrays that have packed attribute can be assigned and compared with the relational operators.

Language like C and C++ provides some library functions to perform operation on the strings.

- *Strcpy*, which moves strings
- *Strcat*, concatenates one given string on to another.
- *strcmp*, lexicographically compares (by the order of their codes) two given strings.
- *Strlen*, which return the number of characters, not counting the null.

Language like FORTRAN 77, FORTRAN 90 and BASIC treat strings as a primitive type and provides operations like assignment, relational operators, concatenation, and substring reference operation for them.

In modern languages, Java also supports strings as primitive type by the `String` class whose values are constant strings and the `StringBuffer` class, whose values are changeable and are more like arrays of single characters. Subscripting is allowed on `String` variables. Pattern matching is another fundamental character string operation. It is often provided by a library function rather than as an operation in the language. SNOBOL 4 is probably the ultimate string manipulation language.

5.4.2 Static Vs. Dynamic Length Strings

According to the variations in the length of the strings, we can divide them in three types:

1. *Static length strings* have static and specified length in the declaration. Languages like FORTRAN 77, FORTRAN 90, COBOL, Pascal and Ada languages are the example of static length strings.
Static length strings are always full, if a shorter string is assigned to a string variable, the empty characters are usually set to blanks.
2. *Limited dynamic length strings* are strings in which we allow strings to have varying length up to a declared and fixed minimum set by the variable's definition. Languages like C and C++ are the examples of these kind of strings. C and C++ use a special to indicate the end of string's characters rather than maintaining the string length.
3. *Dynamic length strings* are allowed to have varying length with no maximum. Language like perl and SNOBOLU are the examples of these type of arrays. The major disadvantage is overhead for storage allocation and deallocation but provides maximum flexibility.

5.4.3 Implementation of Character String Types

Implementation of character strings in storage area depends on the type of strings, for example if the string is of static length then descriptor of string is required at only at compile time and will have only three fields:

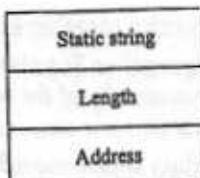


Fig. 5.7 Descriptor of static length string.

Limited dynamic strings requires a run time descriptor to store both the fixed minimum length and the current length as follows:

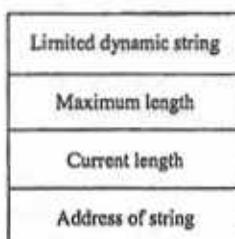


Fig. 5.8 Descriptor of limited dynamic strings

Only exception are strings of C and C++, which do not require run time descriptors because the end of a string is marked with null character.

Dynamic length strings require a simple run time descriptor, with out maximum length.

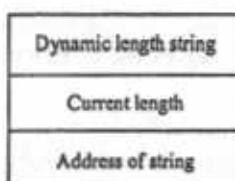


Fig. 5.9 Descriptor of Dynamic length string

The length of string and therefore the storage to which it is bound, must grow and shrink dynamically.

Dynamic allocation can be done by following two means:

1. We can use linked list for dynamic allocation/deallocation of memory from heap area. But large overhead (in terms of memory) is the big disadvantage of this method, since large memory is required to list representation.
2. We can use adjacent cells to reduce the memory over head but when string grows then method get failed.

5.5 RECORDS AND STRUCTURES

We know that arrays are used to group together groups of data item of same type.

A record (as we say in Ada language) or Structure (of C language) is used to group together a "fixed" number of data items (not necessarily of the same type) into a single data item. The item in records are called fields or components (or some time members).

Unlike array or enumerated data types, records can be recursive, that is, they can contain field of the same type as the record in which the field is contained.

Let us see the record declaration in Ada:

```
Type DATA_T is record
    DAY: integer;
    MONTH: string (1 - 10);
    YEAR: integer
end record;
```

Structure in C language:

```
Struct Employee
{
    int id;
    int age;
    char name [15];
    float salary;
    char dept;
};
```

5.5.1 Accessing Component of Records/structures

The fields of a record/structure are accessed using the record name and the field name (used in this way the latter is referred to as a selector), linked by a member operator.

In Ada, Pascal and C this operator is a '.' (dot). However, in C, if a pointer to a structure is used instead of its name an "arrow" (→) operator is used

5.5.2 Comparing Records/structures

Some imperative languages (Ada, Pascal) provide an operator to carry out comparison of records.

For example, Ada uses = and /= operators. C language does not support such comparison.

Let us see following two example program to make the concept more clear:

Ada records example

```
With CS_IO; use CS_IO;
Procedure RECORD_EXAMPLE_1 is
    type DATE_T is record
        DAY : integer;
        MONTH : string (1 - 9);
        YEAR : integer;
    end record;
    BIRTHDAY : DATE_T := (15, "May", 1993);

begin
    put line ("Birthday =");
    put (Birthday . DAY);
```

```

        put (" ");
        put (Birthday . MONTH);
        put (Birthday . YEAR);
        new_line;
    end RECORD_EXAMPLE_1;

```

C structure example

```

# include <Stdio.h>
# include <string.h>
typedef struct date {
    int day;
    char month [9];
    int year;
} DATE_T, * DATE_PTR_T;
Void output_structure (DATE_PTR_T);
Void main (void)
{
    DATE_T nationalDay;
    DATE_T birthday;
/* national day */
    nationalDay . day = 26;
    strcpy (nationalDay . month, "January");
    nationalDay . year = 2006;
    output_structure (& nationalDay);
/* Birthday */
    Scanf ("%d", & birthday . day);
    Scanf ("%d", & birthday . month);
    Scanf ("%d", & birthday . year);
    output_structure (& birthday);
}
Void output_structure (DATE_PTR_T dateptr)
{
    Printf ("%d", dateptr -> day);
    printf ("%d \n", dateptr -> month, dateptr -> year);
}

```

5.5.3 Arrays of records/structures

It is often necessary, given a particular application, to create a number of records/structures of the same type in which to store data, that is an array of structures.

Let us analyse a example of C languages:

```

# include <stdio.h>
# include <string.h>
typedef struct date {
    int day; char month [9];
    int year;
} DATE_T, *DATE_PTR_T;

/* main */
Void main (void) {
    DATE_T STRUCTARRAY [3];
    /* load the element en array */
    assign tostructure (StructArray 0, 15, "October", 1991);
    assign tostructure (StructArray 1, 15, "may", 1993);
    assign tostructure (StructArray 2, 9, "August", 1997);
    /* output array */
    output (structarray, 0);
    output (structarray, 1);
    output (structarray, 2);
}
/* Assign to structure */
Void assign Tostructure (DATE_PTR_T newArray, int index, int newday,
                        char * newmonth, int newyear).
{
    newArray [index]. day = newDay;
    strcpy (newArray [index]. month, newmonth);
    newArray [index]. year = newyear;
}
/* output */
Void output (DATE_PTR_T newArray, int index)
{
    Print f ("%d %5 %d\n", newArray [index]. day, newArray [index].
              newArray [index]. year month, );
}.

```

In many cases the number of elements in such an array are known in advance, and consequently the amount of memory required is known in advance. We refer to such an array as a static array of structures/records. In other cases we do not know how many elements are to be the array until run time, that is a dynamic array of structures/record.

In this case we must create sufficient memory at run time. To create a dynamic data item (that is during run time) we must use an allocate or which reserves space in memory for the data item.

Example:

```
Ada allocator: new integer
C allocator: malloc (size of (integer))
```

here we have dynamically created space for an integer type. An allocator (such as new or malloc) returns a reference (address), referred to as an access value in Ada and a pointer in C, which indicates the first byte of the allocated memory.

In Ada, in addition to reserving memory for a particular type, it is also possible to add initialisations terms to the allocator so that a newly created object can be given a value. This is achieved using an "apostrophe" operator which is inserted between the new type name and the desired value. An example of Ada language is given below:

```
With CS_IO; use CS_IO;
procedure ALLOCATOR_EXAMPLE is
    type INT_PTR_T is access INTEGER;
    PI: INT_PTR_T;
begin PI:= new integer (2);
    put ("PI. all ="); put (PI. ALL);
    new_line;
end ALLOCATOR_EXAMPLE;
```

Here we have created an access value for an integer type. Note the use of the "dot" (member) operator. In the following (comprehensive) examples we will create a linked list of date records of the form defined earlier.

Let us see an example of C language:

```
# include < stdio.h>
# include <string.h>
typedef struct date
{
    int day;
    char month [9];
    int year;
} DATE_T, * DATE_PTR_T;
void main (void)
{
    int num;
    DATE_PTR_T struct array;
    Printf ("enter number of structure");
    scanf ("%d", & num);
    if ((struct Array = (DATE_PTR_T) (malloc (size of (DATE_T) *
        num))) = Null)
```

```

    {
        Printf ("Insufficient space \n");
        exit (-1);
    }
    Rest of code here

```

5.5.4 Implementation of Structures/records

The elements of structure/Records are stored in sequence and allocated sequential block of memory.

Let us discuss C struct:

```

struct employee
{
    int id;
    int age;
    char name [15];
    float salary;
    char deptt.
};

```

Here individual component may need descriptors to indicate their date type or other attributes, but in general no run-time descriptor for the structure is required.

The elements of above discussed structure are stored in sequential storage area and will occupied total 24 bytes of memory.

Now suppose we want to access i th element of a structure S , the accessing function will be

$$\begin{aligned} \text{address}(S.i) &= \text{Base address of storage block representing } S \\ &\quad + \text{memory occupied by } (i-1) \text{ elements} \end{aligned}$$

$$\text{address}(S.i) = \alpha + \sum_{j=1}^{i-1} (\text{size of s. } j)$$

where α is base address.

The declaration for the record also allows the size of each component and its position within the storage block to be determined during translation. So the off set of any component may be computed during compilation.

Clearly at the execution time only the base address for the storage block of structure is required.

5.5.5 Variant Records

We have seen in previous section that, the usual records represents objects with common properties, so all the records of the same type have the same field in common.

If records represents objects that have some but not all properties in common, then they are said to be variant records.

Variant records have a part common to all records of that type, and a variable part, specific to some subset of the records.

Let us consider a pascal declaration:

```

type Payclass = (Regular, contract);
var Employee: record
    ID : integer;
    Deptt : array [1 - 5] of char;
    Age : integer;
    Case empclass: Payclass of Regular : (monthlyrate: real;
                                              Joining date : integer);
    Contact : (Hourrate: real; overtime : integer; regis :
                integer)
end

```

The record always has components ID, Deptt, Age and empclass.

When the value of empclass = Regular, then the record also has monthly rate and Joining date, whereas if the value of empclass = contract, then it has component Hourrate, overtime and regis.

As far as implementation is concerned, it is very similar to usual record implementation. We know that during translation, the amount of storage required for the components each variant is determined, and storage is allocated in the record for the largest possible variant. In above discussed variant, if empclass = Regular then memory requirement is 15 byte, while if emp class = contract then 17 byte so the memory layout of above declared pascal variant will be as follows:

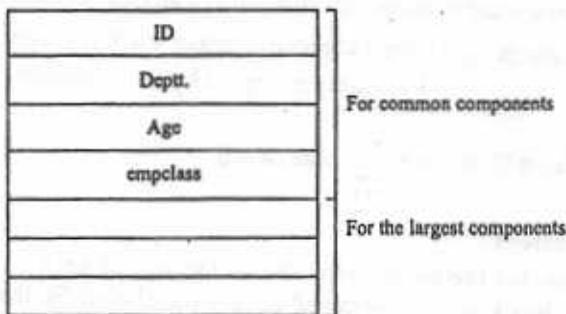


Fig. 5.10 Storage representation of variants

During execution, no special descriptor is needed for a variant record because the tag component is considered as another component of the record.

5.5.6 Union Types

A union is a type that may store different type that may store different types values at different times during program execution.

A union is a special case of a variant record, with an empty common part. During the designing of union type checking implementation plays an important role.

Designer of union should address following Design Issues:

- Type checking is necessary or not? If it is present the must be dynamic type checking.
- Union will be implemented as embedded in records or not?

According to type checking implementation, we can categorise union in two parts:

1. *Free unions* are the unions in which, there is no language supported type checking. For example language like C and C++ have free unions, the unions in these languages are called free union, because programmers are allowed complete freedom from type checking in their use.
2. *Discriminated unions* are the unions with type indicator, and such an indicator is called tag or discriminate. ALGOL 68 was the first language, which supports concept of discriminated union.

Let us consider an example to make the concept more clear:

Consider the following union:

```
Union (int, real) Un1;
int x;
...
Un1 := 50
...
...
x := Un1;
...
```

In above union, first assignment $Un_1 := 50$ is valid, but we can not be sure about second assignment,

$x := Un_1$, because system cannot statically check the type of Un_1 . So clearly compiler can not guarantee that Un_1 will actually contain an integer value.

To handle these problems ALGOL 68 provides conformity clauses for such reference:

```
Union (int, real) Un1;
int x;
real sum;
...
...
case Un1 in
(int y) : x := y,
(real z) : sum := z
esac.
```

This case statement executes the assignment that is currently valid—that is, the one in which the value type of the union variable Un_1 matches the type of the destination (x or sum). Therefore, the different are handled individually. The correct choice is made by testing a type

tag maintained by the run-time system for the variable. The dummy variables, *y* and *z*, and be thought of as implicitly declared variables whose scope is the statement following their specification.

Unions are not considered safe in programming languages. This is the reason, why FORTRAN, C, C++, Pascal and modula - 2 are not strongly typed. On the other hand, unions provide programming flexibility, for example their presence in pascal allows pointer arithmetic. We can design, so that they can be safely used, as in Ada.

Recently designed languages generally avoids unions, for example Java and Modula - 3 do not have union concept.

Implementation is very similar to variant records.

EXERCISE

1. Give the accessing formula for computing the location of component $A[i][j]$ of a matrix *A* declared as

`int A[n][m];`

where *n* and *m* are constants, and *A* is stored in column major order.

2. Choose any two programming languages and contrast their definition of Array data type.
3. With full justification derive accessing formula for computing location of component $A[i_1, i_2 \dots i_n]$ for an array declare as

`A : array [l1 ... u1, l2 ... u2, ... ln ... un]`

Assume that the array is stored in row major order. Take other suitable assumption.

5. What is record? discuss its implementation.
6. What is variant record? Discuss its implementation.
7. Discuss the implementation of multidimensional arrays, when they are stored as:
 - (i) In row major order
 - (ii) In column major order
8. What is enumerated data type? How they are helpful for the programmers.
9. Discuss the implementations of:
 - (i) Static length strings
 - (ii) Limited dynamic length strings
 - (iii) Dynamic length strings.
10. The union and structures are different justify with C language example.
11. Using union are not safe in programming languages, Justify it with a sample program.

CHAPTER 6

Encapsulation

6.1 EVOLUTION OF DATA TYPE CONCEPT

Higher level languages provide a set of basic date types, such as real, integer and character string. Type checking is provided to ensure that operations such as + and × are not applied to data of wrong type. The early notion of data type defines a type as a set of values that a variable might take on.

Pascal extended the concept and a type definition defines the structure of a data object with its possible value bindings. To get a particular data object of the defined type, a declaration requires only the variable name and the name of the type to be given.

In 1970s the storage representation of reals and integers is effectively *encapsulated* (hidden from the programmer). The programmer can use these data objects without knowing storage representation. All the programmer sees is the name of the type and the list of operations available for manipulating data objects of the type.

6.2 THE CONCEPT OF ABSTRACTION

To make the programmers life easy it is very important to create distinction between what and how. Abstraction is the concept which creates this distinction.

"Abstraction means the ability to define and then use complicated structures or operations in ways that allow many of the detail to be ignored."

Abstraction is a key concept in contemporary programming language design. The degree of abstraction allowed by a programming language and the naturalness of its extension are therefore very important to its writability. More support for abstraction, generally more expressive is language. Abstraction is a weapon against the complexity of programming and its purpose is to simplify the programming process. It is an effective weapon because it allows programmers to focus on essential attributes and ignore subordinate attributes.

The two fundamental kinds of abstraction in contemporary programming languages are process abstraction and date abstraction.

The concept of *process abstraction* among the oldest in programming language design. All subprograms are process abstractions because they provide a way for a program to specify that some process is to be done, without providing the details of how it is to be done.

Consider the example of a program in which sorting algorithm required many times. If we replicate the sort program every where then it will make our program too large, so abstraction is better way to tackle this problem. By using the process abstraction sort program can be used without knowing the detail.

The *data abstraction* is extension of encapsulation concept to programmer defined data. we can define data abstraction for data object and operations on data objects.

6.2.1 Abstract Data Types

The concept of abstract data types is the major thrust of the programming language design in 70's and a clear distinction was made between specification and implementation.

We define an abstract data type as

- (i) A set of data objects, ordinarily using one or more type definitions.
- (ii) A set of abstract operations on those data object

Abstract data type should be treated same way as built-in types since it is a mechanism to build new data types (extensible types). The representation should be hidden from the users and only operations of abstract data types are available for them.

The language with abstract data type should provide: method for defining data types and the operations on that type (all in same place). The definition should not depend on any implementation details. The definition of the operations should include a specification of their semantics. The language typically includes data structures (constant, types and variables accessible to user, with hidden details) and declarations of functions and procedures accessible to user.

For example:

```
Pop (Push(s, x)) = s,  
if not empty (s) then push (POP(s), top(s)) = s
```

here we are using push and Pop with knowing the details.

6.3 INFORMATION HIDING

A large program, or even one of moderate size, when all its details are considered at once then it becomes very difficult for a single person to grasp it. To construct a large program, a form of "divide and conquer" strategy must be used: the program is divided into a set of components, often called modules. Each module performs a limited set of operations on a limited amount of data. This keeps intellectual control over the program design process.

Abstraction is so pervasive in programming activities that is often goes without notice. The layer of software and hardware is another example of abstraction. A flow chart is an abstraction of the statement-level control structure of a program. Methods for designing programs, given various names such as stepwise refinement, structured programming, modular approach and top-down programming, are concerned with the design of abstractions.

In formation hiding is the term used for the principle in the design of programmer defined abstractions. Each such program component should hide as much information as possible from the users of the component. The language provided the square function is a successful abstract operation because it hides the details of the numbers represent rotation and the square computing algorithm from the user. Similarly a programmer-defined data type is a successful abstraction if it may be used without knowledge of the representation of objects of that type or the algorithms used by its operations.

When information is encapsulated in an abstraction, it means that the user of the abstraction:

1. Does not need to know hidden information related to implementation in order to use the abstraction, and
2. There is no permission to directly use or manipulate the hidden information even if desiring to do so.

For example consider the integer data type in FORTRAN or pascal, language hides the details of the integer number representation and effectively encapsulates the representation so that the programmer can not manipulate individual bits of the representation of an integer.

6.4 PROGRAMME HIERARCHY

We can distinguish four levels of programme hierarchy

1. Blocks
2. Routines and sub-programs
3. Scope rules
4. Modules, packages and task
5. Programs.

In this chapter we will discuss first two only.

6.4.1 Blocks

A block is the smallest grouping of *declarations* and *statements* that can still meaningfully exist within a program.

In Ada a block is delimited by the key words *begin* and *end*.

```
begin
  ...
  ...
  statements
  ...
  ...
end
```

In C it is delimited using the symbols {and}.

```
{ ...
  ... }
```

```

statements
...
...
}.

```

Block define the *local scope* of their data items (usually variables) that is their "visibility" variables declared within a block cannot be seen from outside that block and such variables are referred to as local variables that is variables *local to block*.

6.4.1.1 Block Structured Languages

Block structured programming languages allow nesting of blocks. The concept was first introduced in Algol 60 and thus refers to languages where declarations must be made at the start of a procedure or function. Ada, Algol 60 and pascal are all block structured in the sense that procedure/functions (the smallest possible grouping in these languages) can be nested within other procedures and functions. C is not a block structured language because procedure/functions can not be nested. In block structured languages blocks can inherit names declared in enclosing blocks. Variables declared in the outermost block are global variables that are said to have global scope (they can be "seen" from any were in the program).

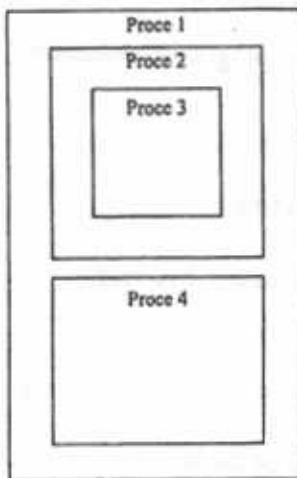


Fig. 6.1 Block structure

6.4.2 Sub-programs (Routines)

One of the most important issues that has to be addressed in the design of a programming language is the support it gives to control of complexity. An important technique is the division of a program in to "chunks", called routines or sub-programs, that will allow the programmer to think at a more abstract level. At its simplest a routine can be equivalent to a block. More specifically a routine can be defined as a collection of declaration and statements that must be invoked explicitly (*routine invocation*). Note that after invocation control is returned to the point immediately after the invocation. An additional advantage is that routine can be stored in libraries for reuse.

Further Issues involved include:

- The method of combining routines to form complete programs, and
- Parameter passing mechanisms between routines.

It should also be noted that although the concept remain the same, routines or sub programs are referred to by different names in different languages. In C, routines are referred to as *functions*. In pascal and Ada, they are called *procedures* and *functions* respectively. Modula-2 names them procedures (even if some of them are actually functions). COBOL uses the terms paragraph, while FORTRAN and BASIC refers. to them as sub-routines and functions.

6.4.2.1 Formal and Actual Parameters

The argument for a routine (if any) are referred to as its *formal parameters*. The values which are to be assigned to these formal parameters to as the *actual parameters*.

- Actual parameters can be variables, constants and expressions.
- Names chosen for actual parameters are completely independent of those chosen for the formal parameters.
- Most imperative languages (Ada and C included) use *positional parameters*, this means that any actual parameters supplied must agree with formal parameters in number, order and type
- Alternatives include *keywords* or *default parameters* (both supported by Ada).

For example in C language:

```
# include <stdio.h>
float meanValue (float, float);
void main (void)
{
    float num_x, num_y;
    Print f ("Enter two real numbers");
    Scan f ("%f %f", & num_x, & num_y);
    Print f ("MEAN VALUE FUNCTION \n");
    Print f ("...\n");
    Printf ("\n The mean is %f\n",
           meanValue (num_x, num_y));
}
                                ↑
                                actual parameters
float meanValue (float value1, float value2)
                                ↓
                                formal parameters
{
    return ((value1 + value2)/2.0);
}
```

In above program num_x num_y are actual parameters and value 1 and value 2 are formal parameters.

6.4.2.2 Procedures and Functions

We can identify two types of routine

1. Procedures
2. Functions

The distinction between a procedure and a function is that a function returns a value while a procedure does not. A function declaration must therefore be called as part of an assignment expression and incorporate a definition for its return value. Some imperative languages (Ada, FORTRAN, Pascal) make a clear distinction between procedures and functions. Other imperative languages (C, Algol 68) do not.

A procedure (function) comprises a head and a body. In the head the name of the routine and its formal parameters are specified. In the body the necessary code is given.

Functions and procedures are activated by a procedure or *function call* which names the routine and supplies the *actual parameters*. Most imperative languages (Ada and C included) use *positional parameters*. Examples in Ada and C:

```
AVERAGE := MEAN_VALUE (4.2, 9.6);
average = meanValue (4.2, 9.6);
```

Alternatives include keyword or default parameters.

Let us consider C procedure Example:

```
# include <stdio.h>
void meanValue (float, float);
void main (void)
{
    float num_x, num_y;
    scan f ("%f %f", & num_x, sum_y);
    mean value (num_x, num_y)
}
void mean value (float value 1, float value2)
{
    Print f ("MEAN VALUE PROCEDURE \n");
    Print f ("... \n");
    Print f ("\n the mean is % f", (value 1+ value2)/2.0);
}
```

In above example procedure not returning any value.

6.4.2.3 Order of Procedure Declaration In Block Structured Languages

In block structured languages the order in which procedures are declared also affects visibility. Ada (and other block structured language such as pascal) requires that any use of an identifier must be preceded by its declaration. Thus if two procedures are declared at the same level, their order of declaration will dictate "who can call whom".

The procedure currently being declared cannot see any following procedures. In Ada this can be overcome by using an *incomplete declaration*. Ordering of procedure declaration is not significant in C.

Note also that in languages such as Ada and pascal a following procedure cannot see a procedure or function nested within a previous procedure or function.

6.4.3 SCOPE Rules

Scope rules govern the visibility of data items (that is the parts of the program where they can be used). They also bind names to types. Where this is determined at compile time, this is called *static scoping*.

The opposite is *dynamic scoping*. Dynamic scoping is generally a feature of logic languages such as PROLOG and functional languages such as LISP. The advantages of static scoping is that allows type checking to be carried out at compile time. Most imperative languages use static scoping.

Generally speaking, in imperative languages, the scope of a declaration commences at the end of the declaration statement and continues to the end block.

6.4.3.1 C Scope Rules

The scope of a data item in C is governed by its *storage class*. Generally data items belong to one of two storage classes:

- Extern (external) : this storage class is used to define *global variables* and visible throughout a program.
- Auto (automatic): This storage class is define local variables (including formal parameters to function) and visible only inside a procedure (block). Local variables exist only while the block of code in which they are declared is executing. Note that where a local variable and a global variable share the same name variable and a global variable share the same name the local variable overrides the global variables.

C also supports two other storage classes *static* and *register*, but these will not be discussed here.

Let us consider a example of C program as follows;

```
/* ... Include statement * ... */
#include <stdio.h>
/* ...GLOBAL DECLARATION ... */
int a = 1, b;
type def double Temperature;
/* ... Function prototype ...*/
void Proc2 (int), Proc3 (temperature);
void Proc4 (int);

/* ... MAIN... */
main ( )
{
    b = a * 2;
    Print f ("In main : a = %d, b = %d \n", a, b);
    Proc2(3);
    proc4(4);
}
```

```
/* ... PROC 2 ... */

Void Proc2 (int x)
{
    double a = 11.1;
    Printf ("In proc 2 : a = %f1, b = %d, x = %d \n", a, b, x);
    Proc3 (99.99);
```

```
/* ... PROC 3 ... */
```

```
Void Proc 3 (Temperature C)
{
    double x = 22.22;
    Printf ("In Proc 3 : a = %d, b = %d, \n", a, b);
    printf ("C = %f1, x = %f1 \n", c, x);
}
```

```
/* ... PROC 4 ... */
```

```
Void Proc4 (int a)
{
    Temperature q = 88.88;
    Printf ("Proc 4: a = %d, b = %d, q = %f1\n", a, b, q);
    Proc2 (4);
}
```

Now let us analyse above interactive C program.

Main:

- The global variables a and b and type temperature are all visible to the procedure *main* (as well as all other procedures declared in the program).
- The procedure has no local variables.

Proc 2:

- The global variables b and type Temperature are both visible to the procedure 2.
- The global variable name a is reused to define a local variable a which will last for life time of the procedure.

Proc 3:

- The procedure has one formal parameter x, whose scope is local to the procedure.

Proc 3:

- The global variables a and b, and type temperature, are all visible to procedure Proc 3.
- The global type Temperature is used to declare a formal parameter, which is local to the procedure.

- A local variable x, is also declared in Proc3 which will exist for the duration of the procedure.

Proc 4:

- The global variables b and the type Temperature are both visible to the procedure Proc 4.
- The global variable name a is reused to define a formal parameter for the procedure Proc 4 which will last for the life time of the procedure.
- The procedure also contains a local declaration for a variable q which uses the global type declaration Temperature.

6.4.3.2 Ada Scope Rules

Let us discuss some of Ada scope Rules a follows:

1. A data item is visible from where it is declared to the end of the block in which the declaration is made.

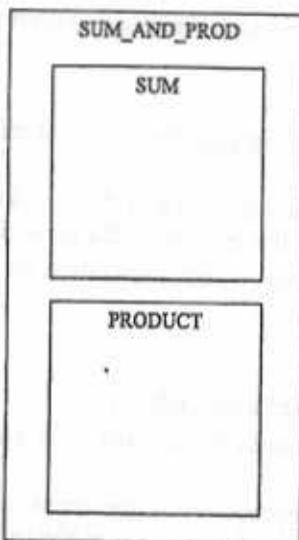
For example in following procedure the constants X_ITEM and Y_ITEM are visible from when they are declared till the end of the procedure. This is why X-ITEM can be used in the declaration of Y_ITEM (we could not do the reverse, or at least not without first reordering the declarations).

```
Procedure SUM_AND_PROD is
    X_ITEM : constant := 2;
    Y_ITEM : constant := X_ITEM * 2;
begin
    PUT (X_ITEM + Y_ITEM);
    NEW_LINE;
    PUT (X_ITEM * Y_ITEM);
    NEW_LINE;
end
```

2. A data item declared within a block can not be seen from outside that block, and is referred to as local data item, that is local to a block.

Consider the following example program (nesting illustrated in diagram to right):

```
with CS_IO;
use CI_IO;
Procedure
SUM_AND_PROD is
    X_ITEM : constant := 2;
    Y_ITEM : constant := X_ITEM * 2;
    .....
    .....
    .....
```

*Procedure*

```

      SUM (P_ITEM, Q_ITEM: INTEGER) is
          TOTAL : INTEGER;
      begin
          TOTAL := P_ITEM + Q_ITEM;
          PUT (TOTAL);
          NEW_LINE;
      end sum;
  
```

Procedure

```

      PRODUT (S_ITEM, T_ITEM: INTEGER) is
          PROD : INTEGER;
      begin
          PROD := S_ITEM × T_ITEM;
          PUT (PROD);
          NEW_LINE;
      end PRODUCT;
  
```

begin

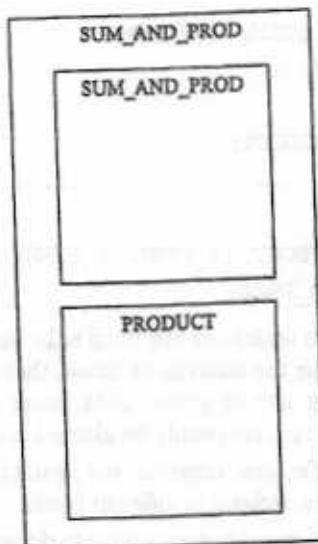
```

          SUM (X_TEIM, Y_ITEM);
          PRODUCT (X_ITEM, Y_ITEM);
      end SUM_AND_PROD;
  
```

The data items P_ITEM, Q_ITEM and TOTAL are local to the sum procedure (block), while data items S_ITEM and PROD are local to the PRODUCT Procedure (block).

3. Anything declared in a block is also visible in all enclosed blocks.

In the following example program (nesting given to the right) the data item ANSWER is visible to the two enclosed blocks (SUM and PRODUCT).



```

with CS - IO;
Use CI - IO;

Procedure
  SUM_AND_PROD is
    X_ITEM : constant := 2;
    Y_ITEM : constant := X_ITEM * 2;
    .....
    .....

Procedure
  SUM_AND_PRODUCT (P_ITEM, Q_ITEM: INTEGER) is
begin
  Answer := A_ITEM + B_ITEM
end SUM;
.....
Procedure PRODUCT (C_ITEM, O_ITEM: INTEGER) is
begin
  ANSWER := C_ITEM * O_ITEM;
end PRODUCT;
.....
.....
  
```

```

begin
    sum (P_ITEM, Q_ITEM);
    PUT (ANSWER);
    NEW_LINE;
    PRODUCT (P_ITEM, Q_ITEM);
    PUT (ANSWER);
    NEW_LINE;
end SUM_AND_PRODUCT;
.....
begin
    SUM_AND_PRODUCT (X_ITEM, Y_ITEM);
end SUM_AND_PROD;

```

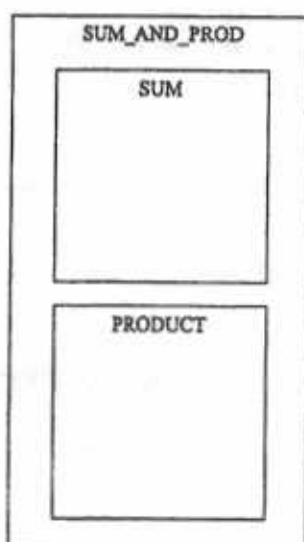
Global data items (items which are required to be visible from anywhere in a program) must be declared within the outermost block, that is level 1 or the global level. Note that wherever possible use of global data items should be avoided as the values associated with these items can easily be altered erroneously.

4. Several data items with the same name can not be used in same block, however several items with the same name can be declared in different blocks.

In this case the declared items have nothing to do with one another (other than sharing the same name).

5. Where a name is used in a block and reused in a sub-block nested within it, the sub-block declaration will override the super-block declaration during the life time of the sub-block.

This is referred to as *occlusion*. The identifier which is hidden of the inner declaration is said to be *occluded*. This is the case in the following example:



```

with CS_IO;
use CI_IO;

Procedure
  SUM_AND_PROD is
    X_ITEM : constant = 2;
    Y_ITEM : constant = X_ITEM * 2;
    .....
    .....
    .....

Procedure
  SUM (P_ITEM, Q_ITEM : INTEGER) is
    X_ITEM : INTEGER;
  begin
    X_ITEM := P_ITEM + Q_ITEM;
    PUT (X_ITEM);
    NEW_LINE;
  end SUM;
  .....
  .....

Procedure
  PRODUCT (X_ITEM, Y_ITEM : INTEGER) is
    PROD : INTEGER;
  begin
    PROD := X_ITEM * Y_ITEM;
    PUT (PROD);
    NEW_LINE;
  end PRODUCT;
  .....
  .....

begin
  SUM (X_ITEM, Y_ITEM);
  PRODUCT (X_ITEM, Y_ITEM);
end SUM_AND_PROD;

```

Here the global data item, X_ITEM is occluded in the SUM and PRODUCT Procedures. The data item, Y_ITEM is occluded in the PRODUCT Procedure in a similar way.

6.4.4 Modules

A number of related declarations of types, variables and routines can be grouped together into a module, also sometimes referred to as a package (Ada and Java) or task. The concept of modules (and modular programming) appeared in 1970's in response to the specification part

contains a description of the interface and the body the code that implements the interface. The specification part is accessible to the user, the implementation part is "hidden" (information hiding).

Let us see some of the example as follows:

- *Creating an Ada package:* An Ada Package comprise two parts.
 - (i) A specification part which gives the user information about the resources contained and how they are used.
 - (ii) A package body that details the resources. Note that the body is not visible to the user.

For example:

```
Package TIMES_TWO_PKG is
  ... SPECIFICATION
    function TIMES_TWO (NUMBER : integer return integer;)
  ...Body
    package body TIMES_TWO_PKG is
      function TIMES_TWO (NUMBER : integer return integer;)
      begin
        return (NUMBER * 2);
      end;
    end TIMES_TWO_pk;
```

- *Creating A C module*

```
/* specification */
int timesTwo (int);
/* BODY */
int timesTwo (int value)
{
  return (value * 2);
}
```

In cooperating the module into another program:

```
# include <stdio.h>
void main (void)
{
  int value = 4;
  printf ("Two times %d = %d \n", value, timestwo (value));
}
```

- *Creating A C Header File*

Whereas a C module is first compiled to create an object file (a.o file) and later linked in to the main program, a header file is as follows – first create a.h file:

```

int timesTwo (int);
int timeTwo (int x)
{
    return (x * 2);
}

```

Second, incorporate the .h file into program:

```

# include <stdio.h>
# include "timesTwo.h"
void main (void)
{
    int x = 4;
    printf ("two times %d = %d \n", x times two (x));
}

```

6.4.5 Programs

The concept of program has existed since the start of computer programming, with respect to C and Ada a program consists of one or more object files linked together. Note that individual object files are not independent. The main issue in program composition from object files is how to indicate to the compiler which objects are part of the desired program.

- *C program specification:* C assumes that the programmer will specify all necessary user object files in a specialized invocation of the system linker. This invocation searches the object files for a routine with the fixed external name main and constructs an executable binary program which, when called, will execute the routine main () .
- *ADA program specification:* Ada assumes that there exist a method outside the language which allows the programmer to specify the name of a procedure and which will then construct an executable binary program for that procedure containing all the necessary modules.

6.5 ENCAPSULATION BY SUBPROGRAM

A subprogram is an abstract operation defined by the programmer. Since a subprogram represents an abstract operation, we should be able to understand its specification without understanding how it is implemented.

The a primitive operation. It includes:

- The name of the subprogram.
- The signature (also called prototype) of subprogram.
- The action performed by the subprograms, that is a description of the function it computes.

For example in C languages.

Flout Fn (float x, int y) is specifies the signature.

Here specification also includes the names *x* and *y* by which the arguments may be referenced within the subprogram. In addition, some language also includes a keyword in the declaration such as *procedure* or *function* as in pascal:

```
Function FN(x : real; y: integer): real;
```

EXERCISE

1. Present a single argument against providing both static and dynamic local variables in subprograms.
2. "C Language Provides only function subprogram", give a suitable comment on this argument.
3. What is concept of abstraction and how abstract data type is useful?
4. Write a program in C++ to justify the concept of Information hiding.
5. "C is a block structure language", give a suitable and justified comment on this statement.
6. Write a C program to differentiate the formal and actual parameters.
7. What are the main difference between scope rules of C and Ada?
8. What is model? how can we create an Ada package?
9. Write a C program to show the Encapsulation by subprogram.
10. How C++ language supports the concept of Encapsulation?

CHAPTER 7

Sequence Control

7.1 INTRODUCTION TO SEQUENCE CONTROL

All programming language utilise program constructs. In imperative languages they are used to control the order (flow) in which statements are executed (or not executed). There are a number of recognised basic programming constructs that can be classified as follows:

1. Sequences.
2. Selection
3. Repetition
4. Routine Invocation.

A *sequence construct* tells the processor which statement is to be executed next. By default, in imperative languages, this is the statement following the current statement (or the First statement in the program). If we wish to "jump" to some other statement we can use a *goto* statement. This was very popular in the early days of computer programming (1950's to early 1960's).

We can categorized sequence-control structures in to four groups:

- The *sequence control of expressions* maintain the data manipulation and change of the program.
- The control flow from one program segment to another is maintained by sequence control of *statement* or *group of statements*. For example conditional and iterative statements.
- The logical programming model of prolog follows another sequence control method, which does not depends on the statements. It is said to be *declarative programming*.
- Another method of transferring control from one program segment to another is *subprogram*. Subprogram calls and coroutines, form a way to transfer control from one segment to another.

7.2 TYPES OF SEQUENCE CONTROL

In general we can divide sequence control according to the design of language, that is either language has to provide the control facility or programmer itself controls the sequence of program.

1. *Implicit Sequence Control*: is the facility provided by the language itself and this control structure will work until the programmer modified it through some structure. For example physical sequence of statements in the program.
2. *Explicit Sequence Control*: is the control facility written by the programmer in the program. Through explicit sequence control programmer can alter the sequence of the program statements, which is provided by the language itself. For example *if-else* statement of C language alter the sequence on the basis of condition.

7.3 EXPRESSION SEQUENCE CONTROL

Let us see the example of arithmetic operations, which is very similar to mathematical expression. In a programming language, arithmetic expression consists of operators, operands, parenthesis and function calls.

On the basis of number of operands, operators may be:

- **Unary operator**: Having only one operand.
- **Binary operator**: Having two operands.
- **Ternary operator**: Having three operands

The evolution of arithmetic expression is very simple, first the operands are fetched (generally from memory) then execution of operator takes place on these operands. But there are following issues which affects the evaluation process.

1. Precedence rules of operators.
2. Associativity rules of operators.
3. Order of the evaluation of operand.
4. Issue of operator overloading.
5. Issue of type mixing in expression.
6. Side effect of function call.
7. Errors in expression.
8. Short-circuit evaluation of Expressions.

Let us see one by one.

7.3.1 Precedence Rules of Operators

The operator precedence rules for expression evaluation define the order in which the operators of different precedence levels are evaluated. The precedence rules are decided at the design time of the language. Let us consider the example:

$$x + y * z.$$

Now let us assume that values of x is 5, value of y is 3, and the value of z is 2. How final evaluated value of $x + y * z$ depends as follows:

- If $x + z$ is calculated first then, evaluated value is $8 * 2 = 16$.
- If $y * z$ is evaluated first then, evaluated value is $5 + 6 = 11$.

So above two evaluation techniques are contradictory. This confusion can be removed if we know, which operator (either + or *) is evaluated first and it can be decided by precedence order of operators.

Let us see the precedence rules of C languages as follows:

Table 7.1 Precedence order in C

Precedence order	Operators
1	. (sequential evaluation)
2	=, +=, -=, *=, /=, % =, <<=, >>=, &=, ^=, 1=
3	? : (conditional operator)
4	
5	&&
6	
7	^
8	&
9	==, !=
10	<, >, <=, >=
11	<<, >>
12	+, -
13	*, /, %
14	type cast
15	~, -, size of, I, &, *
16	++, --
17	, ->

Programmers can alter the precedence and associativity rule by using the parenthesis in expressions. A parenthesized part of an expression has precedence over its adjacent unparenthesized parts. For example in following expression:

$(x + y) * c + d$

$(x + y)$ will be evaluated first.

7.3.2 Operand Evaluation Order

We know that variables in expressions are evaluated by fetching the values from memory. Constant may follow same rule and may be the part of the machine language instruction. If an operand is a parenthesized expression, the all operators it contains must be evaluated before its value can be used as an operand. Evaluation order only matter when evaluation of an operand does have side effects.

7.3.2.1 Side Effects

We know that a global variable is declared outside the function, so when function changes its parameter or global variable then it is said to the *side effect* of function call.

Let us consider the following code segment of C language:

```
int x = 2;
int demo1( )
{
    x = 10;
```

```

        return 3;
    } /* end of demo 1( ) */

```

Above all precedence rules, parenthesis evaluated first.

7.3.3 Associativity of Operators

Now if a expression have two or more then two operators of same precedence then again we face the problem in the evaluation of expression. For example consider the following expression:

$$a - b + c - 2$$

We know that + and - have same precedence order so which we have to evaluate first. This issue is resolved by *associativity rules* provided by language. An operator have two type of associativity:

1. Left to Right. or
2. Right to Left.

Associativity in common imperative language is from left to right, except some exception such as exponentiation operator.

Let us see some of the examples:

Language	Associativity Rules
C-based languages	(Left to Right: *, /, %, binary +, binary - Right to Left: ++, --, unary-, unary +)
Ada Language	(Left to Right: All except **, since ** is Non associative.)
Fortran	(Left to Right: *, /, +, - Right to Left: **)

```

void demo2( )
{
    x = x + demo1();
} /* end of demo2 */
void main()
{
    demo2();
} /* end of main */

```

The value computed for x in demo 2 depends on the order of evaluation of the operands in the expression $x + \text{demo1}()$. The value will be either 5 or 12.

Here language designer can overcome this side effect in following two manners:

- Designer should disallowing functional side effects.
- The second method of avoiding the problem is to state in the language definition that operands in expressions are to be evaluated in a particular order.

But above discussed solution are also not perfect since these methods reduces the flexibility in the languages.

7.3.4 Operator Overloading

Some language used operators in multiple ways, it is called *operator overloading*.

For example C++ and Java supports operators overloading. Java uses this concept for string concatenation.

The concept of operator overloading can be used till it does not affect the readability and reliability of the language.

The languages like Ada, C++, fortran 95, and C# allow programmer to further overload the operators. For example we wants to define the * operator between a scalar integer and an integer array to mean that each element of the array is to be multiplied by the scalar. This could be done by writing a function subprogram named could do this * that performs the new operations.

7.3.5 Type Conversion

Generally languages allows two type of type conversion:

1. Narrowing conversion
2. Widening conversion.

A Narrowing conversion is type conversion which can not store even approximation of all of the values of the original type, for example converting a double to float in Java.

Widening conversion is the type conversion which converts a value to a type that can include at least approximations of all of the values of the original type.

Widening conversion is consider as safe conversion then narrowing conversion, Narrowing conversion can leads towards inaccuracy. For example, Let us consider the situation where the language stores integer in 32 bits, which allows at least nine decimal digits of precision. But floating-point values are also stored in 32 bits, with only about seven decimal of precision (because of the space used for the exponent). So clearly integer to floating-point widening can result in the loss of two digits of precision.

Type conversion can be either explicit or implicit:

- *Implicit type conversion* is automatically done by the language itself. Language may have expressions in which an operator can have operands of different type. Languages that do allow such expressions, which are called *mixed-mode expression*, must define conventions for implicit operand type conversions called *coercions*.

Let us consider an example of C language:

```
int x;
float y, z;
x = 5;
y = 2.0
z = x/y;
```

If here we print z then it will be evaluated as $z = x/y = 5/2.0 = 2.5$. First x will be automatically converted to float then $z = \frac{5.0}{2.0} = 2.5$ comes.

- *Explicit type conversion* is done by the programmer. Most language provides both widening and narrowing.

In C-based languages it is said to be *type casts*.

```
(int) angle;
```

here angle will be converted to integer type, in spite of that, whatever the type of angle.
In Ada, the casts have the syntax of function calls.

For example

```
float (sum) .
```

7.3.6 Errors in Expressions

From the above discussion it is clear that number of errors can occur in expressions evaluation. The type errors occur when the language is not providing type checking mechanism.

Some times result of arithmetic operations can not be represented in the memory cell where it must be stored. This is called *overflow or underflow*, depending on whether the result was too large or too small. One limitation of arithmetic is that division by zero is disallowed. These kinds of errors are run-time errors, which are some times called exceptions.

7.3.7 Short-circuit Evaluation of Expression

A short-circuit evaluation of expression is one in which the result is determined without evaluating all operands and/or operators.

Let us consider an example:

$$(5 * x) * (y / 4 - 2)$$

Here, if $x = 0$ then, there is no need to evaluate the $(y / 4 - 2)$ since $5 * x = 0$ so the value of complete expression $0 * (y / 4 - 2)$ is 0 only, in any case. So this kind of evaluation is called short-circuit evaluation but it is very difficult to detect the short circuit during execution of arithmetic expressions. In the C-based language, the usual AND (`&&`) and OR (`||`) respectively are short circuit. However these languages also have bit wise AND (`&`) and OR (`|`) operators, that can be used on Boolean-valued operands and are not short circuit.

For example:

$$(x >= 0) \&\& (4 < 5)$$

If $x < 0$ then there is no need to evaluate the $(4 < 5)$.

The Ada language provides the best design because it provides the programmer flexibility of choosing short-circuit evaluation for any or all Boolean expressions.

7.4 SEQUENCE CONTROL BETWEEN STATEMENT

In previous section we have seen the sequence control by evaluating the expressions but it is not enough to provide great flexibility and power. Statements that provide these kinds of capabilities are called *control statements*. Many languages provides the facility of *control structure*, which is a control statement and the collection of statements whose execution it controls.

In this section we will discuss two type of control structures:

1. Selection
2. Repetition

Now there is a one design issue that is relevant to all of the selection and iteration controls statements, that whether the control statement can have multiple entries (Execution of these control statement always starts with first statement of them or not). multiple entries add file to the flexibility of a control constructor, relative to the decrease in readability caused by the increased complexity. Note that multiple entries are possible in only those languages that supports goto s and labels.

7.4.1 Selection

A selection statement provides for selection between alternatives we can identify three types of selection construct:

1. If statement
2. Case statement
3. Pattern matching

However, imperative languages do not generally make much use of pattern matching (logic and some modern functional languages do), thus we will limit the following discussion to the "if" and "case" forms of selection.

7.4.1.1 Two Way selection (If-else Statements)

An "if-else" statement (Fig. 7.1), sometimes referred to as a conditional, can be used in following two forms:

1. If condition then action 1
2. If condition then action 1 else action 2.

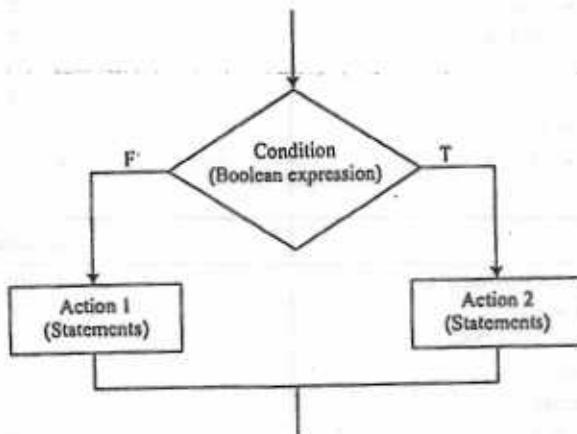


Fig. 7.1 If-else control flow diagram

Condition is a Boolean expression (or may be sequence of Boolean expressions), action 1 and action 2 are program statements of some description.

The Design issues of above control construct are as follows:

- Type of conditional expression may differ from language to language. Generally it is boolean expression. For example Ada, Java, and C# use only boolean expressions for control expressions. In C and C++ either arithmetic or boolean expressions can be used.
- Specification of then and else clause is very important issue in language design. In some cases there is a reserved word, so there is no need of parenthesis like in Ada. C-based languages never used then clause.
- Evaluation of Nested selectors are also taken in the consideration during the design time. Let us consider following code of Java language:

```
if (x == 0)
    if (y == 0)
        result = 0;
    else
        result = 1;
```

In above code segment else follows two if-statement, 1400 question arises this else will corresponds to which if. In Java, as in many other imperative languages, the static semantics of the language specify that the else clause is always paired with the nearest unpaired else clause. C, C++ and C# also follows same rules.

Now Let us see examples of various language like C, Ada, Pascal and modula-2.

Table 7.2

C	Ada
<pre>If (a < 0) { b = -a; C = 10 + a; } else { b = a; C = 10 - a; }</pre>	<pre>IF A < 0 then B: = -A; C: = 10 + A; else B: = A; C: = 10 - A; end if;</pre>
Pascal	Modula-2
<pre>if A < 0 then begin B: = -A; C: = 10 + A; end; else begin B: = A; C: = 10 - A; end;</pre>	<pre>IF (A < 0) THEN B: = -A; C: = 10 + A; ELSE B: = A; C: = 10 - A; END</pre>

We can also identify a number of variations of the above which are particular to individual programming languages. For example the Ada:

```
if condition_1 then action_1
elsif condition_2 then action_2
...
...
elsif condition_n then action_n
else action_n + 1
end if
```

7.4.1.2 Nested "If-else" Statements

There is also nothing that prevents us from nesting "if-else" statement, as shown in Fig 7.2.

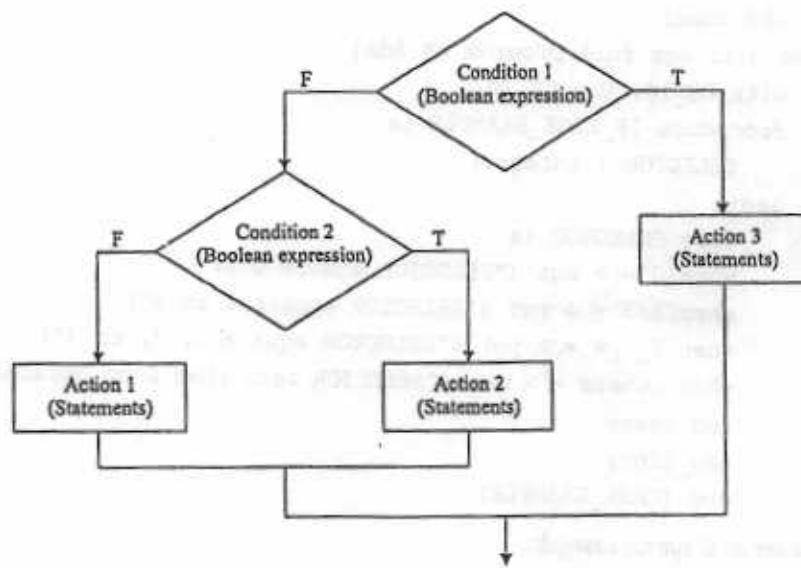


Fig. 7.2 Nested if-else

7.4.2 CASE Statements

Generally speaking an "if ... then ... else ..." statement supports selection from only two alternatives; we can of course nest such statements, but is usually more succinct to use a case statement. Case statements allow selection from many alternatives where each alternative is linked to a predicate, referred to as a selector, which when evaluated to true cause an associated program statement (or statements) to be executed. Selections may be made according to:

1. A distinct value of a given selector.
2. An expression involving the selector.
3. A default value.

Selectors must be of a discrete type (typically an integer, a character or an enumerated type). C only supports selection by distinct value or default. Ada also supports selection by distinct value and default (others), as well as selection according to:

1. A number of alternatives (each separated by a bar "|").
2. Ranges expressed using the "... " operator.
3. Selection by condition, that is <, = ,>.

Also note that where selection is by distinct value, in some cases, these must be presented in the control numeric order (this is the case in C).

Let us see an Ada syntax example:

```
Case <SELECTORS> is
    where <ALTERNATIVES> = > <STATEMENTS>
        ...
        ...
    end case;
```

Now we will see full program of Ada:

```
with CS_IO; Use CS_IO
Procedure IF_ELSE_EXAMPLE is
    SELECTOR : integer;
begin
    case SELECTOR is
        when 0 => put ("SELECTOR equals 0");
        when 1/5 => put ("SELECTOR equals 1 or 5");
        when 2 .. 4 => put ("SELECTOR equals 2, 3, or 4");
        when others => put ("SELECTOR less than 0 or greater than 5");
    end case;
    new_line;
end (CASE_EXAMPLE)
```

Let us see an C syntax example:

```
switch (<SELECTOR>) {
    Case <VALUE>: <STATEMENTS>
        ...
    default: <statements>
```

Finally we will see C program as follows:

```
# include <stdio. h>
void main (void)
{
    int selector;
    scanf ("%d", & selector);
    switch (selector) {
```

```

case 0:
    printf("selector equals 0\n");
case 1:
case 2:
case 3:
    Printf("selector equals 1, 2, or 3\n"); break;
default:
    Printf("selector less than 0 or greater than 3\n");
}
}

```

In C, when the selector is equal to a value (that is, 2), then the statements associated with that values and all following statements are evaluated until the end of the switch statement is reached, or until a break moves flow of control to the end of the switch statement. Note that C requires that selectors are presented according to their numeric order. Note also that neither Ada nor C support selection using expression involving selectors, that is a conditional expression or an arithmetic expression. This is supported by languages such as Pascal.

7.5 ITERATIVE STATEMENTS

An iterative statement is one that causes a statement or collection of statements to be executed zero, one, or more times. Every programming language has included some method of repeating the execution of segment of code.

A repetition construct causes a group of one or more program statements to be invoked repeatedly until some end condition is met. Typically such constructs are used to step through arrays or linked lists. We can identify two main forms of repetition:

1. *Fixed count loops*—repeat a predefined number of times.
2. *Variable count loops*—repeat an unspecified number of times.

Several categories of iteration control statements have been developed. The primary categories are defined by how designers answered two basic design questions:

- How is the iteration controlled?
- Where should the control mechanism appear in the loop?

7.5.1 Pre-test and Post-test Loops

Both *fixed* and *variable count loops* can be farther classified according to whether they are pretest or post-test loops. In the case of a pre-test loop (also referred to as an entrance-controlled loop) the end condition is tested for prior to each repetition, while in the case of a post-test loop (also referred to as an exit-controlled loop) the end condition is tested for after each repetition.

7.5.2 Fixed Count Loop

Fixed count loops (sometimes referred to as for loops) allow a statements to be repeated a "fixed" number of times as specified on entry into the loop. Fixed count loop tend to be pre-test loops, that is the end condition is tested prior to each repetition. Issues are as follows:

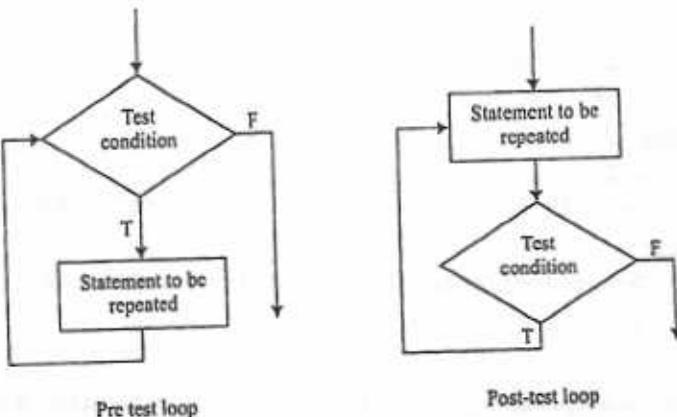


Fig. 7.3 Pre and post-test loops

1. Type of the control value.
2. Nature of start and end conditions.
3. Updating the control variable.
4. Reverse processing.
5. Nesting of loops.

7.5.2.1 *Type of the Control Variable*

- In some imperative languages (C, Pascal) the control variable is considered to be simply another data item which is declared in the normal manner. Thus the control variable can usually be of any discrete type (*integer* or *char* in Pascal).
- In other languages (Ada) the nature of the control variable is predefined-usually as an integer type.

7.5.2.2 *Nature of Start and End Conditions*

- The control variable, in a fixed count loop, must always have some start value, we refer to this value as start condition for the loop.
- In some imperative languages the start value is fixed.
- In most imperative languages (Ada, Pascal, C) the programmer is free to specify a start value, this is usually some discrete value which is assigned to the control variable. Alternatively in languages such as C and Pascal the start value must be defined by some assignment expression.
- Fixed count loops must also always have an end condition (if they are to terminate). This is usually a Boolean expression, or a sequence of such expression, involving the control variable.

7.5.2.3 *Updating the Control Variable*

- To avoid infinite repetition the control variable must be updated on each repetition.
- The most straight forward approach to updating is to simply increment the control variable by 1 on each repetition.

- In some imperative languages (Ada and Pascal) such incrementation is carried out automatically, in others (C) it must be specified by the programmer.
- Updating is much more flexible it can be defined by the programmer.

7.5.2.4 Reverse Processing

- In some languages where the control value is automatically incremented, to process a series of statements in reverse, an adjective such as *reverse* (Ada) or *down to* (pascal) must be included in the definition of the loop.
- In languages where the programmer controls incrementation, decrementation can be defined explicitly in the same manner that incrementation is defined.

7.5.2.5 C and Ada For-Loop Example

The following example programs initialize 10 element array and then processes (outputs) that array using a fixed count (for) loop construct.

Ada Example:

```
Format:
for <CONTROL_VALUE> in <START_CONDITON > ... <END_CONDITION>
loop
<STATEMENTS>
end loop;
```

Usage (processing an array):

```
With CS_IO; use CS_IO;
Procedure FOR_LOOP_EXAMPLE is
  DIGIT: array (integer range 0..9) of integer;
begin
  for INDEX in 0..9 loop
    put (DIGIT (INDEX));
    new_line;
  end loop;
  new_line;
end FOR_LOOP_EXAMPLE;
```

Note that: (1) the control variable INDEX is not declared or initialised by the program and (2) that the second loop processes the array in reverse.

C Example:

```
Format:
for (<CONTROL_VALUE>; <START_CONDITION>; <END_CONDITION>)
{
  <STATEMENTS>
}
```

```

Usage (processing an array):
# include <stdio.h>
void main <void>
{
    int index = 0, digit [10];
    for (index; index <10; index++)
    {
        digits [index] = index;
    }
    for (index = 9; index >= 0; index--)
    {
        printf("%d", digits [index]);
    }
    printf("\n")
}

```

Note that the way the C for-loop constructed is used here is fixed count; the construct could equally well be used to implement a variable count loop.

7.5.3 Nesting

All common imperative languages support nesting of for-loops as in figure 7.4

```

# include <stdio.h>
void main (void){
    int num 1, num 2,
    for (num = 4; num1 != 1; num --) {
        for (num2 = 2; num 2 < = 6; num 2 = num 2 + 2) {
            print f("%d\n" num 1 + num 2);
        }
        print f("\n");
    }
}

```



Nested loop structure

The next Fig. 7.4 will represent nested loop control flow diagram as follows:
Now we will see an example with more than control variable.

```

# include <stdio.h>
void.main (void)
{
    char letter;
    int number;
}

```

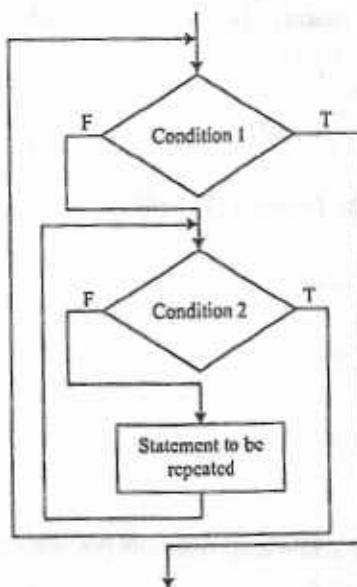


Fig. 7.4 Nested loop control flow diagram

```

for (letter = 'A', number = 2;
     letter < 'J', letter ++,
     number = number + 5)           ] more than
{                                     one control
  printf("% c (% d)\n", letter, letter, letter + number,
         letter + number);
  letter + number);
}
}
  
```

7.5.4 Repetition and Variable Count Loops

Variable count loops allow statements to be repeated an intermediate number of times. The repetition continues until a control condition (usually a Boolean expression) is no longer fulfilled. As such, variable count loops do not feature a control variable which is incremented and tested against some end condition on each loop. Variable count loops may be pre-test or post-test loops.

Pre-test variable count loops are sometimes referred to as a while loops (Ada, C, Pascal), and post-test variable count loops as a do-while (C) or repeat (Pascal) loops (Ada does not feature a post-test variable count loop construct).

Let us see Ada while loop example (variable count) as follows:

```
with CS_IO; use CS_IO;
Procedure EXAMPLE is
X: integer range 0..10;
begin
put_line ("Input Integer 0..10")
get(X);
while x <= 10 loop
Put(x);
put(x * x * x);
new_line;
X := x + y;
end loop;
end EXAMPLE;
```

Now finally we will see C while loop Example (variable count);

```
# include <stdio.h>
void main (void)
{
int x = 1;
printf("Input integer 0... 10\h");
scanf ("%d", &x);
while (x <= 10) {
print ("% d % d % d\h",
x, x * x, x * x * x);
x++;
}
}
```

7.5.4.1 For Loop or While Loop?

Both for and while loops are “pre-test” loop. In languages where “for” loops can only be used to describe fixed count loops (that is Ada, Pascal, BASIC) the answer is very straightforward:

- For statements for fixed count loops.
- While statements for variable count loops.

In C where for and while statements can be used to describe both fixed and variable count loops, it simply a matter of style.

The following C example shows how the same variable count program can be implemented using both a for and while construct.

```
void main (void) {
    int a;
    scanf ("%d", a);
    while (a <= 10) {
        printf ("%d", a);
        print ("%d", a * a);
        printf ("%d", a * a * a);
        a ++
    }
}
```

Using while loop

```
void main (void) {
    int a;
    scanf ("%d", a);
    for (; a <= 10; a++) {
        printf ("%d", a);
        printf ("%d", a * a);
        printf ("%d", a * a * a);
    }
}
```

Using for loop

7.5.5 Post-Test Loops (Do-while)

Where as for pre-test loops the control variable is tested prior to the start of each iteration, in post-test loops the control variable is tested at the end of each iteration.

Let us do the comparison of C pre-test (while) loops and post-test (do-while) loops for better understanding of loops:

```
void main (void)
{
    int number;
    Printf("Input an integer between
1 and 50\n");
    do {
        scanf ("%d", &number);
    } while (number < 1 || number > 50);
    Printf ("%\n", number);
}
```

```
void main (void)
{
    int number = 0;
    Printf("Input an integer
between 1 and 50\n");
    while (number < 1 || number > 50)
    {
        scanf ("%d", &number);
        printf ("%\n", number);
    }
}
```

7.5.6 Terminating A Loop

In some situation, it is convenient for a programmer to choose a location for loop control other than the top or bottom of the loop. Now question is wheather a single loop or several nested loops can be exited.

The design issues for such a mechanism are the following:

- Should the conditional mechanism be an integral part of the exit?
- Should only one loop body be exited, or can enclosing loops also be exited?

C and C++ have unconditional unlabeled exits (*break*). Java, perl and C# have unconditional labeled exits (*break* in Java and C#, *last* in perl).

Let us see the following C# example:

```
for (i = 0; i < 10; i++)
    for (j = 0; j < 5; j++) {
```

```

        sum += mat [i] [j]
        if (sum > 1000.0)
            break outer loop; labeled exit.
    }.

```

C and C++ have unlabeled control statement, *continue*. This is not an exit but rather a way to skip the rest of the loop statements on the current iteration without terminating the loop structure. For example, consider the following:

<pre> while (sum < 1000) { getnex (value); if (value < 0) continue; sum += value; } </pre>	<pre> while (sum < 1000) { guntent (value); if (value < 0) break; sum += value; } </pre>
if value is negative then sum += value; (assignment) is skipped	if value is negative loop is terminated

Java, perl, and C## have statements similar to *continue* except they can include labels that specify which loop is to be continued.

7.6 CONCURRENT PROGRAMMING

In a distributed computing system, it is natural to regard the nodes as independent processes that execute concurrently. In such a system, agents communicate with each other to coordinate their activities. Communication may be through *synchronization* or *message-passing*. Synchronization implies a simultaneous two-way exchange of information (like a telephone call) while message-passing involves one-way transfer that require acknowledgment for the sender to be sure that a message has reached the recipient (like a normal letter sent by post)

"However, even in a single processor system, it is often convenient to regard different components of a program as executing in parallel".

Consider, for instance, an interactive application like web browser. When we are downloading a page that takes a long time to retrieve, we can usually press the *stop* button to terminate the download. When programming a browse with such a capability, the most natural model is to regard the download component and the user-interface component (that reacts to button clicks) as separate processes. When the user-interface process detects that the *stop* button has been clicked, it notifies the download process that it can terminate its activity.

On a single processor, the run-time environment will allocate time-slices to each of these "logical" processes. These time-slices will be scheduled so that all the concurrent processes get a chance to execute. However, there is no guarantee about the relative speeds at which these processes will run. One process may get 10 more time-slices than another in a given interval.

Normally, each concurrent process comes with its own local variables. Thus, when the run-time environment switches from one process to another, it has to save the state of the first

process and load the suspended state of the second one. Often, however, it is simpler to assume that all the concurrent processes share the same set of variables. Thus, the processes interact via a global "shared memory". This makes it possible to switch from one process to another relatively easily, without an elaborate content switch involving all the variables defined in the processes.

In the literature, these kinds of processes that share a global memory often called "*threads*" (short for "threads of execution"). We shall study concurrent programming in the frame work of threads that operate with a global shored memory.

7.6.1 Processes

Process is the instance of program or program part that has been scheduled for independent execution. The execution state may be: *Executing*, *blocked* or *waiting*. The biggest problems of process execution are:

1. Process synchronization
2. Communicate data between processes.

In case of shared memory, the communication is done by shared memory but it suffers with the problem of mutual exclusion (reader-writer contention). In case of distribution memory the communication takes place by sending messages but it also suffers with problem of a synchronous sending and receiving messages.

To handle all these problems on operating system must have following mechanisms:

1. Create and destroy processes.
2. Manage processes by scheduling on one or more processes.
3. Implement mutual exclusion (for shared memory).
4. Create and maintain communication channels between processor (for distributed memory).

7.6.2 Language Mechanism Supporting Concurrency

Here our focus is on language mechanisms to support concurrency.

The major mechanisms are:

1. Race conditions
2. Protocol based on shared variables.
3. Programming primitives for mutual exclusion.
4. Monitors.
5. Java threads
6. Interrupts

7.6.2.1 Race Conditions

Shared variable provide a simple mechanism for communication between processes. For instance, in the example of the browser discussed earlier, we could have a Boolean variable that indicates whether the download should be interrupted. This variable is initially false. When the user-interface thread detects that the "stop" button has been clicked, it sets the variable to true. Periodically, the download thread examines the value of this variable. If finds that the variable

has been set to true, it aborts the current transfer. Once we have shared variables, it becomes important to incorporate a mechanism for consistent update of such variables by concurrent threads. Consider a system in which we have two threads that update a shared variable n , as follows:

Thread 1	Thread 2
....
$m = n;$	$k = n;$
$m++;$	$k++$
$n = m;$	$n = k;$
....

Under normal circumstances, after these two segments have executed, we would expect the value of n to have been incremented twice, once by each thread.

However, because of time-slicing, the order of execution of the statements may be as follows:

```

Thread 1:  $m = n;$ 
Thread 1:  $m++$ 
Thread 2:  $k = n;$  //  $k$  gets the original value of  $n$ 
Thread 2:  $k++$ 
Thread 1:  $n = m;$ 
Thread 2:  $n = k;$  // same value as that set by thread 1.

```

In this sequence, the increments performed by the two threads overlap and the final value of n is only one more than its initial value. This kind of inconsistent update, whose effect depends on the exact order in which concurrent threads execute is known as a *race condition*. This example can be formulated as instances of the familiar problem of *mutual exclusion* or *critical regions* of program code, which is well-studied, for instance, in the design of operating system.

7.6.2.2 Protocols Based on Shared Variables

One may solve the mutual exclusion problem is to develop a protocol using auxiliary shared variables. For instance, we could use a variable *turn* that takes value 1 and 2 to indicate whose *turn* it is to access the critical region, as follows:

<pre> Thread 1 while (turn != 1) { // "Busy" wait } // Enter critical section // Leave critical section turn = 2; </pre>	<pre> Thread 2 while (turn != 2) { // "Busy" wait } // Enter critical section // Leave the critical section turn = 1; </pre>
--	--

Let us assume that *turn* is initialized to either 1 or 2 (arbitrarily) and there is no other statement that updates the value *turn* other than the two assignments shown above. It is then clear that both Thread 1 and Thread 2 cannot simultaneously enter their critical sections—if Thread 1 has entered its critical section, the value of *turn* is 1 and hence Thread 2 is blocked until Thread 1 exits and sets *turn* to 2. A symmetric argument holds if thread 1 tries to enter the critical section while Thread 2 is already in its critical section. If both threads simultaneously try to access the critical section, exactly one will succeed, based on the current value of *turn*.

Notice that this solution does not depend on any atomicity assumptions regarding assigning a value to *turn* or testing the current value of *turn*. However, there is one serious flaw with this solution. When Thread 1 executes its critical section it sets *turn* to 2. The only way for Thread 1 to be able to re-enter the critical section is for Thread 2 to reset *turn* to 1. Thus, if only one thread is active, it cannot enter the critical section more than once. In this case, the thread is said to *starve*.

Another solution is to maintain two boolean variables, *request_1* and *request_2*, indicating that the corresponding thread wants to enter its critical section.

Thread 1	Thread 2
<pre> request_1 = true; while (request_2) { // "Busy" wait } // enter critical section ... // Leave the critical section. ... request_1 = false; </pre>	<pre> request_2 = true while request_1 { // "Busy" wait } // Enter critical section. ... // Leave critical section turn = 1; ... request_2 = false;</pre>

Here we assume that *request_1* and *request_2* are initialized to false and, as before, these variables are not modified in any other portion of the code. Once again, it is easy to argue that both threads cannot simultaneously be in their critical section – when Thread 1 is executing its critical section, for instance, *request_1* is true and hence Thread 2 is blocked. Also, if only one thread is alive, it is still possible for that thread to repeatedly enter and leave its critical section since the request flag for the other thread will be permanently false. However, if both threads simultaneously try to access the critical region, they will both set their request flags to true and there will be a *deadlock* where each thread is waiting for the other thread's flag to be reset to false.

Peterson [1981] found a clever way to combine these two ideas into a solution that is free from starvation and deadlock. Peterson's solution involves both the integer variable *turn* (which takes values 1 or 2) and the boolean variables *request_1* and *request_2*.

Thread 1	Thread 2
...
request_1 = true;	request_2 = true;
true = 2;	true = 1
while (request_2 && turn != 1)	while (request_1 && turn != 2)
{	{
// "Busy"wait	// "Busy"wait
}	// Enter critical section
// Enter critical section	...
// Leave the critical section	// Leave the critical section
request_1 = false;	request_2 = false;
...	...

If both threads try to access the critical section simultaneously, the variable *turn* determines which process goes through. If only one thread is alive, the request variable of the other thread is stuck at false and the value of *turn* is irrelevant. To check that mutual exclusion is guaranteed, suppose Thread 1 is already in its critical section. If Thread 2 then tries to enter, it first sets *turn* to 1. While Thread 1 is active, *request_1* is true. Thus, Thread 2 has to wait until Thread 1 finishes and turns off *request_1*. A symmetric argument holds when thread 2 is in the critical region and thread 1 tries to enter.

7.6.2.3 Programming Primitives for Mutual Exclusion

Peterson's algorithm does not generalize easily to a situation where there are more than two concurrent processes. For n processes, there are other solutions, such as Lamport's Bakery algorithm. However, the problem with these protocols is that, in general, they have to be designed anew for different situations and checked for correctness.

A better solution is to include support in the programming language for mutual exclusion. The first person to propose such primitive was Edsger Dijkstra, who suggested a new data type called a *semaphore*.

A *semaphore* is an integer variable that supports an atomic test-and-set operation. More specifically if a *s* is a variable of type semaphore, then two atomic operations are supported on *s*: *p(s)* and *v(s)*. (The letters *p* and *v* come from Dutch words *passeren*, to pass, and *vrygeven*, to release.) The operation *p(s)* achieves the following in an atomic manner:

```

if (s > 0)
    decrement s;
else
    wait for s to become positive;

```

The operation *V(s)* is defined as follows:

```

if (there are threads waiting for s to become positive)
    wake one of them up; // choice is nondeterministic
else
    increment s;

```

Using semaphore, we can now easily program mutual exclusion to critical section, as follows;

Thread 1	Thread 2
...
P(s);	P(s);
// Enter critical section	// Enter critical section
...	...
// Leave critical section V(s);	// Leave critical section V(s);
...	...

Mutual exclusion, starvation freedom and deadlock freedom are all guaranteed by the definition of a semaphore.

One problem with semaphores are that they are rather "low-level". The definition of a semaphore is orthogonal to the identification of the critical region. Thus, we connect critical regions across threads in a somewhat arbitrary fashion by using a shared semaphore. Correctness requires cooperation from all participating threads in resetting the semaphore. For instance, if any one thread forgets to execute V(s), all other threads get blocked. Further, it is not required that each V(s) match a preceding P(s). This could lead to a semaphore being "preloaded" to an unsafe value.

7.6.2.4 Monitors

Since critical regions arise the need to permit consistent updates to shared data, it seems logical that information about conflicting operations on data should be defined along with the data, using a class-like approach. This is the philosophy underlying monitors, a higher-level approach to providing programming support for mutual exclusion. Monitors were proposed by per Brinch Hansen and Tony Hoare.

Roughly speaking, a monitor is like a class description in a object-oriented language. At its heart, a monitor consists of a data definition, to which access should be restricted, along with a list of functions to update this data. The functions in this list are assumed to all be mutually exclusive - that is, at most one of them can be active at any time.

The monitor guarantees this mutual exclusion by making other function calls wait if one of the function is active. For instance, here is a monitor that handles the bank account example as follows:

```
monitor bank account {
    double accounts [100];
    // transfer "amount" from accounts [source] to accounts [target]
    boolean transfer (double amount, int source, int target)
    {
        if (accounts [source] < amount) {
            return false;
        }
        accounts [source] -= amount;
    }
}
```

```

        accounts [target] += amount
        return true;
    }
    // compute the total balance across all accounts
    double audit () {
        double balance = 0.0;
        for (int i = 0; i < 100; i++)
        {
            balance += accounts [i];
        }
        return balance;
    }
}

```

By definition, the functions transfer (...) and audit () are assumed to require mutually exclusive access to the array accounts []. Thus, if one process is engaged in a transfer and a second process wants to perform an audit, the second process is suspended until the first process finishes.

Thus, there is an implicit “external” queue associated with the monitor, where processes that are waiting for access are held. We use the word queue but, in practice, this is just a set. No assumption is made about the order in which waiting processes will get access to the monitor. The capabilities we have described so far for monitors are not as general as semaphores. We have to deal with a tricky problem: what happens when a notify () is issued? The notifying process is itself still in the monitor. Thus, the process that is woken up cannot directly execute.

At least three different types of notification mechanism have been identified:

- *Signal and exit*: In such a setup, a notifying process must immediately exit the monitor after notification and hand over control to the process that has been woken up. This means that *notify()* must be the last instruction that it executes. We shall see later that we might use multiple internal queues in a monitor each of which is notified independently. With signal and exit, it is not possible to notify multiple queues because the first call to *notify()* must exit the monitor .
- *Signal and wait*: In this setup, the notifying process and the waiting process swap rules, so the notifying process hands over control of the monitor and suspend itself. This is not very commonly used.
- *Signal and continue*: Here, the notifying process continues in the monitor until it completes its normal execution. The woken up process (es) shift from the internal to the external queues and are in contention to be scheduled when access to the monitor is relinquished by the notifying process. However, no guarantee is given that one of the newly awakened processes will be the next to enter the monitor – they may all be blocked out by another process that was already waiting in the external queue.

When a suspended process resumes execution, there is no guarantee that the condition it has been waiting for has been achieved.

Waiting condition should check the state of the data before proceeding because the data may still be in an undesirable state when data may still be in an undesirable state when it wakes up. To counter these problems, we have to extend the monitor definition to include a declaration of the internal queues (some times called condition variables) associated with the monitor. In the example below, we assume we can declare an array of queues. In practice, this may not be possible - we may have to explicitly assign a separate name for each queue.

```

monitor bank_account {
    double account [100];
    queue q[100]; // one internal queue for each account
    // transfer "amount" from accounts [source] to accounts [target]
    boolean transfer (double amount, int source, int target) {
        while (accounts [source] < amount) {
            q[source]. wait (); //wait in the queue associated with source.
        }
        accounts [source] -= amount;
        accounts [target] += amount;
        q[target]. notify (); // notify the queue associated with
                               target
        return true;
    }
    // compute the total balance across all accounts
    double audit () {...}
}

```

If we notify () a queue that is empty, it has no effect. This is more robust than the V(s) operation of a semaphore that increments s when nobody is waiting on S.

7.6.2.5 Monitors in Java

Java implements monitors with following characteristics:

- Java implements monitors with a single queue. This queue is "anonymous".
- Any class definition can be augmented to included monitor like behaviour. We add a qualifier *synchronized* to any method that requires exclusive access to the instance variables of the class. Thus, if we have two or more functions that are declared to be *synchronized* and one of them is currently being executed, any attempt to enter another *synchronized* method will be blocked and force the thread into the external queue.
- In Java terminology, there is a lock associated with the object. Only one thread can own the lock at one time and only the thread that owns the lock can execute a synchronized method. At the end of the synchronized method, the thread relinquishes the lock.
- Inside a synchronized method, the wait() statement behaves like the normal wait() statement in a monitor.
- In Java there are two makeup statement, notify() and notify all(). The function notify() signals only one of the waiting processes, at random. Instead, notify All() signals all

waiting processes (as one would expect) and this be used by default.

- Java monitors use the *signal and continue* mechanism, so the notifying process can continue execution after sending a wakeup signal and the processes that are woken up move to the external queue and are in contention to grab the object lock next becomes available.
- In addition to synchronized method, Java has slightly more flexible mechanism. Every object has a lock, so an arbitrary block of code can be synchronized with respect to any object, as follows:

```
Public class xyz {
    Objec o = new object ();
    public int f( ) {
        ...
        synchronized (o)
        {
            ...
        }
    }
    Public double g( ){
        ...
        synchronized (o)
        {
            ...
        }
    }
}
```

Now, f() and g() can both begin to execute in parallel in different threads, but only one thread at a time can grab the lock associated with o and enter the block enclosed by synchronized (o). The other will be placed in the equivalent of an "external queue" for the object o, while waiting for the object lock. In addition, there is a separate "internal queue" associated with o.

7.6.2.6 Java Threads

The simplest way to define concurrent threads in Java is to have a class extend the built-in class *Thread*. Suppose we have such a class, as follows, in which we define a function *run()* as shown:

```
Public int id;
public parallel (int i)
{
    //
    // Constructor
    id = i
```

```

    }
    public void run() {
        for (int j = 0; j < 100; j + 1)
            {
                system.out.println ("My id is " + id); try {
                    sleep (1000); // Go to sleep for 1000 ms
                }
                catch (InterruptedException e) {}
            }
    }
}

```

Then, we can created objects of type parallel and "start" them off in parallel, as follows:

```

public class Testparallel {
    public static void main (String [ ] args) {
        Parallel P[ ] = new parallel [5];
        for (int i = 0; i < 5; i++) {
            P[i] = new parallel (i);
            P[i]. start (); // start of P[i]. run() in concurrent thread
        }
    }
}

```

The call `P[i]. start()` initiates the function `P[i]. run()` in a separate thread. Notice the function `sleep(..)` used in `run()`. This is a static function defined in `Thread` that puts the current thread to sleep for the number of milli seconds specified in its argument. Any thread can put itself to sleep, not just one that extends `Thread`. In general, one would have to write `Thread.sleep(..)` if the current class does not extend `Thread`. Observe that `sleep(..)` throws `InterruptedException` (like `wait()`).

A thread can be in one of four states:

- *New*: when it has been created but not `start()` ed.
- *Runnable*: when it has been `start()` ed and is available to be scheduled. A runnable thread need not be "running" – it may be waiting to be scheduled. No guarantee is made about how scheduling is done but almost all Java Implementations now use time-slicing across running threads.
- *Blocked*: The thread is not available to run at the moment. This could happen for three reasons:
 1. The thread is in the middle of a `sleep(..)`. It will then get unblocked when the sleep time expires.
 2. The thread has suspended itself using a `wait()`. It will get unblocked by a `notify()` or `notifyAll()`.

3. *Dead*: This state is reached when the thread terminates.

7.6.2.7 Interrupts

A thread can be interrupted by another thread using the function `interrupt()`. Thus, in our earlier example, we can write

```
P[i]. interrupt();
```

to interrupt thread `P[i]`. This raises an `InterruptedException`, which, as we saw, we must catch if the thread is in `sleep()` or `wait()`.

However, if the thread is actually running when the interrupt arrives, no exception is raised. To identify that an interrupt arrived in such a situation, the thread can check its interrupt status using the function `interrupted()`, which returns a boolean and clears the interrupt status back to `false`. It is also possible to check another thread's interrupt status using `isInterrupted()`. Another useful function to spy on the state of another thread is `Alive()`. Java also supports a method `stop()` to kill another thread and `suspend()` and `resume()` to unilaterally suspend and resume a thread (which is not the same as the `wait()` and `notify()`/`notifyAll()` construction). However, these are not suspended to be used because they frequently lead to unpredictable conditions (when a thread dies abruptly) or deadlocks (when suspended threads are not properly resumed).

7.7 EXCEPTION HANDLING

"In programming language parlance, an exception is an unexpected event that disrupts the normal execution of a program".

Exception handling is a mechanism that allows two separately developed program component to communicate when a program anomaly, called an exception, is encountered during the execution of the program.

An illegal operation such as division by zero or accessing an array element beyond the bound of the array is almost surely an exception.

The key problem is to deal with exception in a sensible way. The aim is for the program to be able to recover and take corrective action if possible and abort only if there is no other option left. Further, it should be possible to report sensible information about the nature of the exception for diagnostic purposes. In C, a very rudimentary form of exception reporting is done by examining the return value of a function such as `main()`. Often, a return value `int` encodes the result of the computation of the function – minimally, a return value 0 may indicate no error while a non-zero value might indicate an error. The type of error might indicate malformed input while returning a value 2 might indicate the inability to open a file that was needed for the program to run.

7.7.1 Exception Handling In C++

The C++ language provides built-in language features to raise and handle exceptions. These language features activate a run-time mechanism used to communicate exceptions between two unrelated (often separately developed) portions of C++ programs.

When an exception is encountered in a C++ program the portion of the program that detect the exception can communicate that the exception has occurred by raising, or throwing, an exception.

Let us see a example of i stack:

```
# include <vector>
Class istack {
public:
    istack (int capacity)
        _stack (capacity), _top (0) {}
    bool pop (int &Stop_value);
    bool push (int value);
    bool full ();
    bool empty ();
    void display ();
    int size ();
Private:
    int _top;
    vector <int> _stack;
};
```

The stack is implemented using a vector of ints. When an istack object is created, the constructor of istack creates a vector of ints of the size specified with the intial value. The size is the maximum number of elements the i stack object can curtain. The following, for example, creates an i stack object called my stack that can contain as many as 20 values of type int:

```
istack mystack (20);
```

What can go wrong when we manipulate my stack? Here are two anomalies that may be encountered with our istack class:

1. A pop () operation is requested and stack is empty.
2. A push () operation is requested and the stack is full.

We decide that these anomalies should be communicated to the functions manipulating istack objects using exceptions. We will define here two simple classes to use as exceptions without istack class, and we place these class definitions in the header *stackExcp.h*:

```
// stack Excp. h
class PopOnEmpty /* ... */;
class PushonFull /* ... */;
```

We must change the definition of the member functions pop() and push () to throw these newly defined exceptions. An exception is thrown using a *throw expression*. An exception looks a great deal like a return statement. A throw expression is composed of the keyword throw followed by an expression whose type is that of the exception thrown. What does the throw expression in pop () look like? Let's try this:

```
// oops, not quite right
throw PopOnEmpty;
```

Unfortunately, this is not quite right. An exception is an object, and `pop()` must throw an object of class type. The expression in the `throw` expression cannot simply be a type. To create an object of class type, we need to call the class constructor. What does a `throw` expression that invokes a constructor look like? Here is the `throw` expression in `pop()`:

```
// expression is a constructor call
throw PopOnEmpty();
```

This `throw` expression creates an exception object of type `PopOnEmpty`.

Recall that the member functions `pop()` and `push()` were defined to return a value of type `bool`: a true return value indicates that the operation succeeded, on a *false* return value indicates that it failed. Because exceptions are now used to indicate the failure of the `pop()` and `push()` operation, the return values from these functions are now unnecessary. We now define these member functions with a `void` return type.

For example:

```
class istack {
public :
// ...
// no longer return a value
void pop (int & value);
void push (int value);
private:
//...
};
```

The function that use our `istack` class will now assume that everything is fine unless an exception is thrown; they no longer need to test the return value of the member function `pop()` or `push()` to see whether the operation succeeds. Now we will see how to define a function to handle exceptions in the next two sections.

We are now ready to provide the new implementations of `istack`'s `pop()` and `push()` member functions:

```
#include "stackExcp.h"
void istack :: pop (int & top_value)
{
    if (empty ())
        throw PopOnEmpty();
    top_value = - stack [--_top];
    cout << "istack :: pop (): " << top_value << endl;
}
void istack :: push (int value)
```

```

{
    cout << " istack :: push (" << value <<")\n";
    if (full ())
        throw PushOnFull ();
    _stack [_top ++] = value;
}.

```

Although exceptions are most often objects of class type, a throw expression can throw an object of any type. For example although it's unusual, the function *mathFunc ()* in the following code. Sample throws an exception object of enumeration type. This is valid C++ code:

```

enum EHstate { noErr, zeroOp, negativeOp, severeError};
int mathFunc (int i) {
    if (i == 0)
        throw zeroop; // exception of enumeration type
    // otherwise, normal processing continues
}

```

7.7.2 Exception Handling in Java

In case of Java language, when an error occurs within a method, the method creates an object and hand it off to the runtime system. The object, called an *exception object*, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object handing it to the runtime system is called *throwing an exception*.

After a method throw an exception, the runtime system attempts to find something to handle it. The set of possible "somethings" to handle the exception is the ordered list of methods that had been called to get to the method where these error occurred. Two list of methods is known as the call stack.

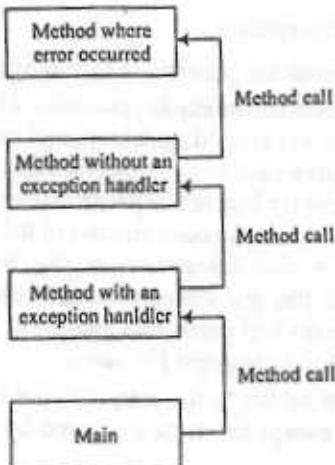


Fig. 7.5 The call stack

The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an *exception handler*.

The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the method were called. When an appropriate handler is founds the runtime system passes the exception to the handler.

An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler.

The exception handler chosen is said to catch the exception. If therun time system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, as shown in next figure, the runtime system (and, consequently, the program) terminates.

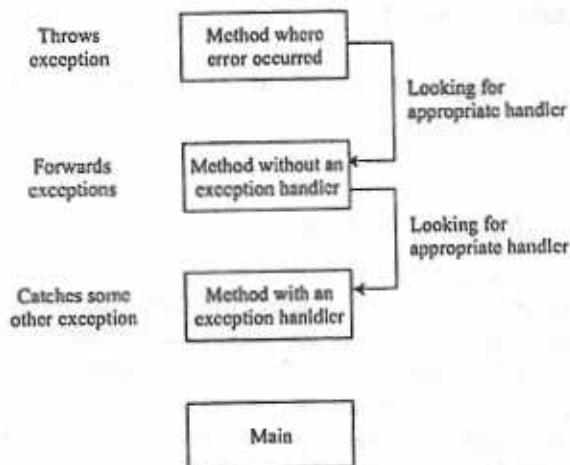


Fig. 7.6 Searching the call stack for the exception handler

7.7.2.1 The Three Kind of Exceptions

There are three type of exception are possible in Java language:

- The first kind of exception is the *checked exception*. These are exceptional conditions that a well-written application should anticipate and recover form. For example, suppose an application prompts a user for an input file name, then opens the file by passing the name to the constructor for `Java.io.FileReader`. Normally, the user provides the name of an existing, readable file, so the construction of the `File reader` object succeeds, and the execution of the application proceeds normally. But sometimes the user supplies the name of a nonexistent file, and the constructor throws `Java.io.File Not Found Exception`. A well-written program will catch this exception not notify the user of the mistake, possibly prompting for a corrected file name.

Check exceptions are *subject* to the catch or specify Requirement. All exceptions are checked exceptions, except for those indicated by `Error`, `Runtime Exception`, and their subclass.

- The second kind of exception is the *Error*. These are exceptional conditions that are external to the application, and that the application usually cannot anticipate or recover from.

For example, suppose that an application successfully opens a file for input, but is unable to read the file because of hardware or system malfunctions. The unsuccessful read will throw `Java.io.IOException`. An application might choose to catch this exception, in order to notify the user of the problem – but it also might make sense for the program to print a stack trace and exit. Errors are not subject to catch or specify requirement. Errors are those exceptions indicated by `Error` and its subclasses.

- The third kind of exception is the *runtime exception*. These are exceptional conditions that are internal to the application, and that the application usually cannot anticipate or recover from. These usually indicate programming bugs, such as logic errors or improper use of an API.

For example, consider the application described previously that passes a file name to the constructor for `FileReader`. If a logic errors cause a null to be passed to the constructor, the constructor will throw `NullPointerException`. The application can catch this exception, but it probably makes more sense to eliminate the bug that caused the exception to occur. Runtime exceptions are not subject to the catch or specify Requirement. Runtime exceptions are those indicated by `Runtime Exception` and its subclasses.

Note that *Errors and exceptions* are collectively known as *unchecked exceptions*.

7.7.2.2 Catching and Handllng Exceptions

In this section we will discuss that how to use the three exception handler component – the `try`, `catch`, and `finally` blocks – to write an exception handler.

- The first step in constructing an exception handler is to enclose the code that might throw an exception within a `try block`.

In general, a `try` block look like the following:

```
try
{
    code
}
catch and finally block ...
```

The segment in the example labeled `code` contains one or more legal lines of code that could throw an exception.

- We have to associate exception handlers with a `try` block by providing one or more `catch` blocks directly after the `try` block. No code can be between the end of the `try` block and the beginning of the first `catch` block.

```
try{
    } catch (ExceptionType name) {}
    catch (ExceptionType name) {
    }
}
```

Each catch block is an exception handler and handles the type of exception indicated by its argument. The argument type, *Exception Type*, declares the type of exception that the handler can handle and must be the name of a class that inherits from the *Throwable* class. The handler can refer to the exception with name.

The *catch* block contains code that is executed if and when the exception handler is invoked. The runtime system invokes the exception handler when the handler is the first one in the call stack whose *Exception Type* matches the type of the exception thrown.

The *finally* block always executes when the *try* block exists. This ensures that the *finally* block is executed even if an unexpected exception occurs. But *finally* is useful for more than just exception handling – it allows the programmer to avoid having cleanup code accidentally bypassed by a return, continue or break. Putting cleanup code in a *finally* block is always a good practice, even when no exceptions are anticipated.

7.7.2.3 Exception Throwing

Before we can catch an exception, some code somewhere must throw one. Any code can throw an exception: Your code, code from package written by some one else such as the package that come with the java platform, or the Java runtime environment. Regardless of what throw the exception, it's always throw with the *throw* statement.

Java platform provides numerous exception classes. All these classes are descendants of the *Throwable* class, and all allow programs to differentiate among the various types of exceptions that can occur during the execution of a program. We can also create our own exception class to represent problems that can occur within the classes we write. We can also create chained exceptions.

- *The throw statement:* All methods use throw statements to throw an exception. The *throw* statement requires a single argument: a throwable object. Throwables objects are instances of any sub class of the *Throwable* class.

Let look at the *throw* statement in content. The following *pop* method is taken from a class that implements a common stack object. The method removes the top element from the stack and returns the object.

```
Public Object pop() {
    object obj;
    if (size == 0) {
        throw new EmptyStackException();
    }
    obj = objectAt (size - 1);
    set objectAt (size - 1, null);
    size--;
    return obj;
}
```

The *pop* method checks to see whether any element are on the stack. If the stack is empty (its size is equal to 0), *pop* instantiate a new *Empty Stack Exception* object (a member of Java util) and throws it. We can throw only objects that inherit from the *java.lang.Throwable* class.

Note that the declaration of the *pop* method does not contain a *throws* clause. *Empty Stack Exception* is not a checked exception, so *pop* is not required to state that it might occur.

- *Throwable class and its subclasses:* The objects that inherit from the *Throwable* class include direct descendants (object that inherit directly from the throwable class) and indirect descendants (object that inherit from children or grand children of the throwable class). The figure below illustrate the class hierarchy of the *Throwable* class and its most significant subclasses. As we can see, *Throwable* has two direct descendants: *Error* and *Exception*.

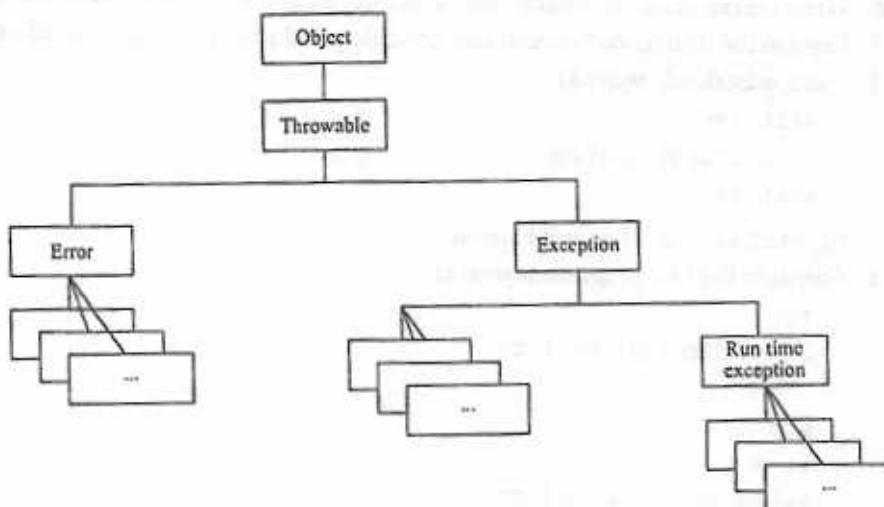


Fig. 7.7 The *Throwable* class

EXERCISE

1. Rewrite the following C code without *gos* or *breaks*:

```

j = - 3;
for (i = 0; i < 3; i++) {
    switch (j + 2) {
        case 3:
        case 2: j -=; break;
        case 0: j += 2; break;
        default: j = 0;
        if (j > 0) break;
        j = 3 - i
    }
}
  
```

2. What are the differences between the *break* statement of C++ and that of Java?

3. What is unusual about C's multiple selection statement? What design trade-off was made in this design?
4. What is a pretest loop statement? What is a posttest loop statement?
5. What are the difference between Implicit and Explicit type conversion. Explain it with the example of C.
6. What is the difference between C and Ada switch case structure.
7. What is the difference between pretest and post test loop, explain with the example of C.
8. Write the mechanisms, which should be there in concurrent programming languages?
9. Explain the difference between race condition and protocol based shared variables.
10. If one mistakenly write as

```
wait (s)
      critical section
wait (s)
```

In detail, explain what will happen.

11. Consider the two program segments.

```
for
i: = 1 to A(x) by 1 do
  s
end
i: = 1
while (i < = A (x)) do
  s
  i: = i + 1
end
```

Under what conditions are these two programs equivalent? Treat s as any sequence of statements.

CHAPTER 8

Sequence Control of Subprogram

8.1 INTRODUCTION

We have already seen the definition of subprogram and procedure. At the implementation level of any programming language it is very important to be very clear about the sub program sequence control. To understand the working of subprogram, we will discuss the implementation of subprogram in various languages. In this chapter we will discuss the inner working of subprogram, how they manage to pass data among themselves in a structured and efficient manner and how does one subprogram invoke another and then permit the called subprogram to return to the first.

8.2 ACTIVATION OF THE PROCEDURE AND ACTIVATION RECORD

Each execution of a procedure is referred to as an activation of the procedure. This is different from the procedure definition, which in its simplest form is the association of an identifier with a statement; the identifier is the name of the procedure and the statement is the body of the procedure. If a procedure is non-recursive then there exists only one activation of procedure at a time. Whereas procedure is recursive, several activation (one activation per call of procedure) of that procedure may be active at the same time.

The information needed by a single execution or a single activation of a procedure is managed using a contiguous block of storage called an "activation record" or "activation frame" consisting of the collection of fields. The activation record contains the following information:

1. Temporary values, used during expression evaluation.
2. Local data of procedure.
3. Saved machine status information (Program Counter registers, return address).
4. (Optional) access link, for access to non-local names
5. The actual parameters.
6. The returned value, used by called procedure to return a value of calling procedure.
7. (Optional) control link, points to the activation record of the caller. Some time it is also said to be *dynamic link*. In static-scoped languages, this link is used in the destruction of the current activation record instance when the procedure completes its execution.

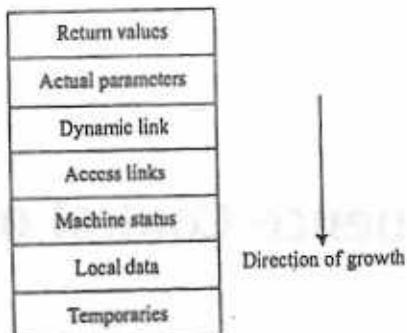


Fig. 8.1 A typical activation record

8.3 STACK AND SUBPROGRAM CALLS

We know that most of the time program and data elements requiring storage are tied to subprogram activations. When a subprogram is called, per call a new activation record is created on the top of stack and termination causes its deletion from the stack.

Let us consider the following program:

```
/* Find the sum of two integers */
void fnsumprint (int i, int j);
int main (int argc, char ** argv)
{
    int x = 10; y = 20;
    scanf ("%d", &x, &y);
    fnsumprint (x, y);
    return 0;
}
void fnsumprint (int i, int j)
{
    int result;
    iresult = i + j;
    printf ("% d", iresult);
}
```

Now we will see in Figure 8.2 that, how function call reflects on the stack.

Things can be easily understood by following steps:

1. Program stack will contain activation record of main () when executing it (through complete activation record is not shown here).
2. Program stack will contain activation of fnsumprint () at the top when main () is calling the function.

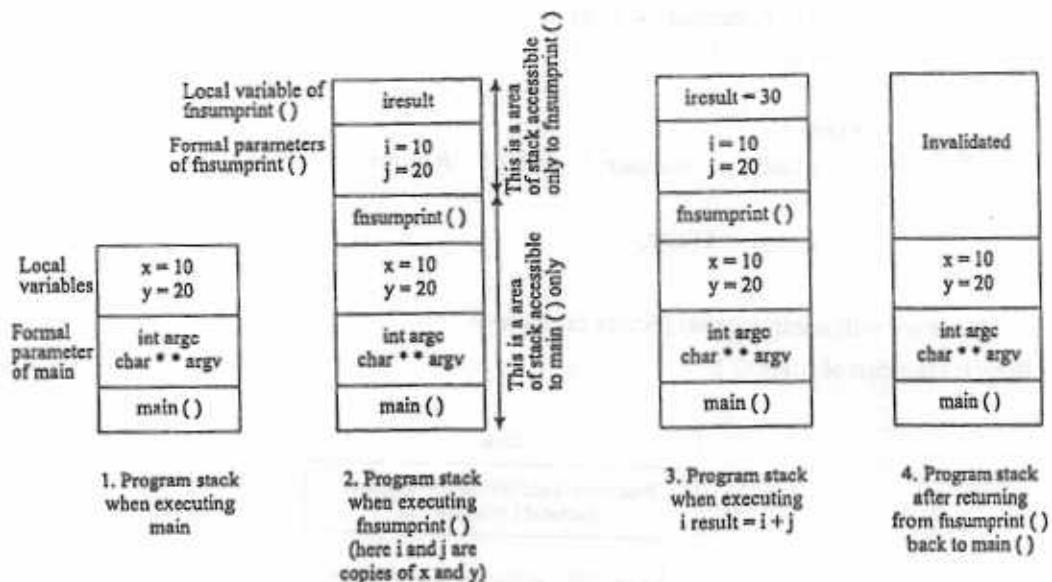


Fig. 8.2 Activation Record Instances

3. All internal calculation takes place and return value is set.
4. When control returns back to `main()` then activation record of `fnsumprint()` becomes invalidated.

8.3.1 Stack and Recursive Subprogram Calls

When a function calls itself it is called as Recursion. A recursive function definition has two parts:

Basic condition: When a function will terminate
Recursive condition: The invocation of a recursive call to the function.

Let us consider a c program to understand the program stack situation in case of recursive sub program execution.

```
/* Finding the factorial of an integer using a recursive function */
int fnFact (int i Number);
int main (int arg c, char ** argv) {
    int iFactorial;
    iFactorial = fnFact (u);
    Printf ("The factorial is % d\n\n", iFactorial);
    return 0;
}
int fnFact (int iNumber)
{
    int ifact;
```

```

        if (iNumber <= 1) {
            return 1;
        }
        else {
            iFact = iNumber * fnFact (number - 1);
        }
        return iFact;
    }
}

```

Now we will see the actual Picture of stack:

Step 1: First call of fnFact ()

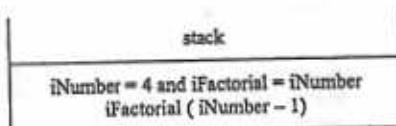


Fig 8.3(a) Instance of activation

Note that here we are concern only about calculation of functional through recursion and it does not depicting complete picture of activation.

Step 2: Second call of fnFact ()

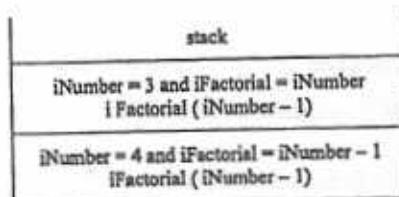


Fig 8.3(b) Instance of Activation after second call of fnFact ()

Step 3: Third call of fnFact ()

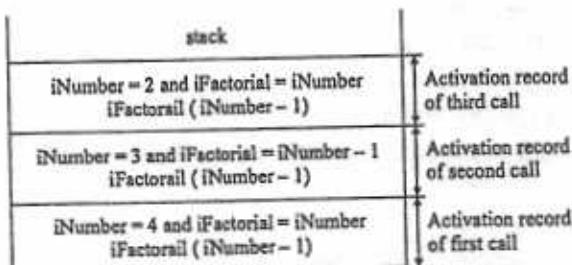


Fig. 8.3(c) Instance of Activation after third call of fnFact ()

Step 4: Invalidations of Activation record of third call

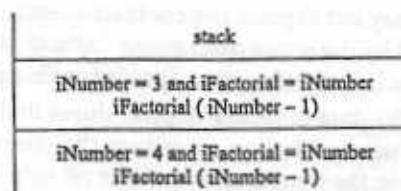


Fig. 8.3(d) Instance of activation after invalidation of third call

Step 5: Invalidation of activation record of second call.

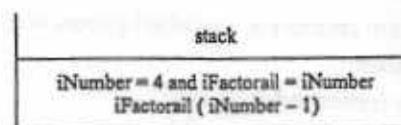


Fig. 8.3(e) Activation instance after invalidation of second call

Step 6: Finally control returns to main.

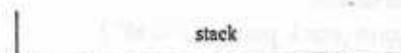


Fig. 8.3(f) Control returns to main

In brief we can view as follows:

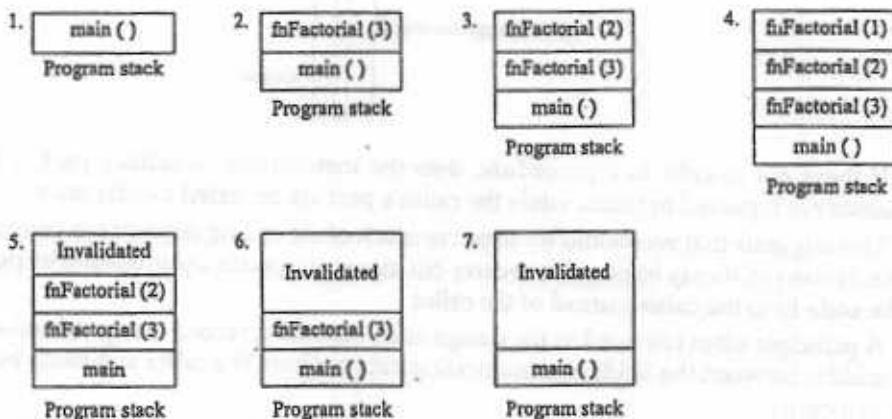


Fig. 8.4 A complete picture of activation of recursive sub program

8.4 HANDLING PROCEDURE CALLS AND RETRUNS

In practice, the stack frame may not expand and contract as each compound statement or block is entered and left, as implied by the above description. Instead, the maximum storage required for the frame may be allocated as each function / procedure is called.

Stack allocation is used for data in recursive procedures that may have multiple bindings. Each procedure call creates a new activation record on the control stack in which the actual parameters, the local variables, the temporaries, and the returned value of the activation can be stored. The activation record remains on the stack until the procedure activation returns to its caller. Stack-allocated data items are referenced using displacement from a register, top, pointing to the top of the control stack.

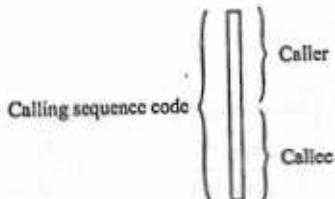
A *call sequence* is a sequence of machine instruction that is executed every time a procedure is called. It works as follows:

- Allocates an activation record for the called procedure;
- Loads actual parameter;
- Saves machine state (return address etc);
- Transfer control to callee.

A *return sequence* is a sequence of machine instruction that is executed every time a procedure returns control to its caller. It does following:

- Deallocates activation record of the called procedure.
- Sets up return value (if any)
- Restores machine state (stack pointer, PC etc.)

Activation records and calling sequences differ from machine to machine; the calling sequences are often divided up between the caller and the procedure being called (the callee).



If there are m calls to a procedure, then the instructions in caller's part of the calling sequence are repeated m times, while the callee's part are repeated exactly once.

This suggests that we should try to put as much of the calling sequence as possible in to the callee. However, it may be possible to carry out more call specific optimization by putting more of the code in to the caller instead of the callee.

A principle often followed in the design of an activation record is to put fixed-size fields in the middle between the fields communicating data to/from the caller and fields below follow this principle.

The following call sequence assumes there is a register ($top - sp$) holding a pointer to the start of local data in the current activation record:

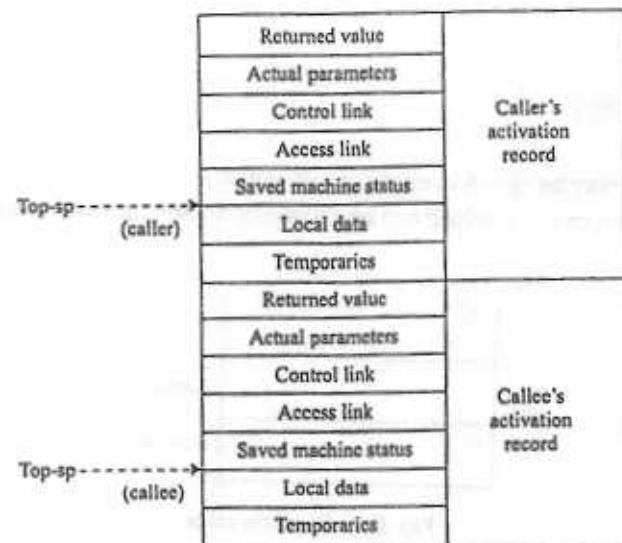


Fig. 8.5 Instance of activation records during calling sequence

1. The caller evaluates the actual parameters and inserts them in to the appropriate places of the callee's activation record.
2. The caller stores a return address and the value of its top - sp (stack pointer) register in to saved machine status of the callee's record.
3. The top - sp register is set to point to the start of the callee's local data and control is sent to the callee's code.
4. The callee saves other register values and status information.
5. The callee initializes its local data and starts execution.

A possible return sequence is:

1. The callee places a returned value at the start of its activation record.
2. Using the saved machine status in its record the callee restores top - sp and other registers and branches to the return address in the caller's code.
3. The caller resumes execution: The location of the returned value is a known displacement from top-sp.

8.4.1 C Language Function Calls and Returns

Now let us consider an example of c language to make the concept clear.

Consider the following c fragment:

```
main ( )
{
    first ( );
    second ( );
}
```

```

first ( )
{
    Second ( );
}

```

Here second () may be called in either of two ways:

(a) *Directory form main ()*, which can be shown by following run-time stack in Fig. 8.6.

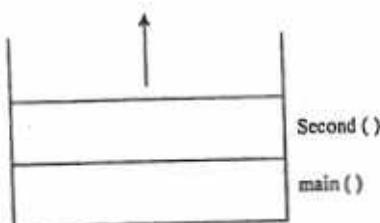


Fig. 8.6 Run time stack

(b) *From first ()*, which has been called from main (). It can be shown by run-time stack in Fig. 8.7.

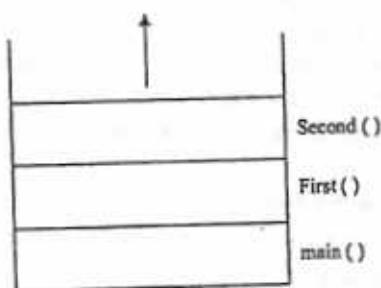


Fig. 8.7 run time stack

From the discussion above, the address of a variable with respect to the foot of the stack frame in which it is stored at compile time. However, as we have seen, the position of a stack frame with respect to the foot of the stack is not known, in general, at compile time, and has to be evaluated at run-time. C programs have the property that (a part from extern and global variables) only variables from a single function (the one currently active) can be accessed at run-time. Therefore, if a pointer to the start of the current stack frame is maintained at run-time, a knowledge of the value of this pointer, together with address of a variable within the section of stack (known at compile time) is sufficient to calculate the address of the variable, with respect to the foot of the stack.

The pointers to the start of each of the stack frames corresponding to functions currently active, are known as the set of *dynamic stack pointers*. When the function corresponding to the top stack frame is left, control return to the function corresponding to the stack frame below it, any of whose variables may be accessed. It is necessary, therefore, to retain the values of the

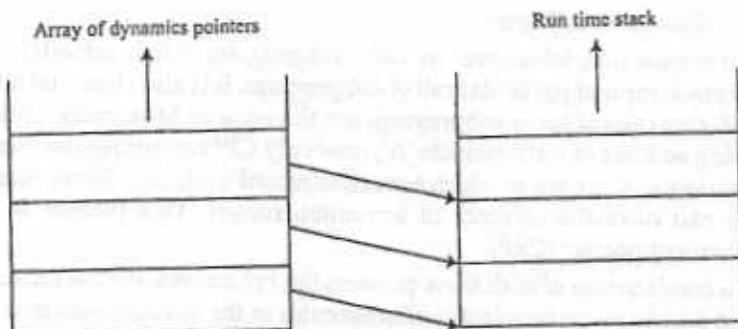


Fig. 8.8 Dynamic stack pointers maintenance

dynamic pointers to each of the stack frames currently on the stack. This may be done by means of an array of pointers as shown in following Fig. 8.8.

Maintenance of the array of dynamic pointers can be maintained (in similar fashion to stack) by the following actions at run-time.

- (a) Adding the start address of a new stack frame to the pointer array, each time a function is entered.
- (b) Removing the tail value of the pointer array each time, the function corresponding to the current stack frame is left.

8.5 IMPLEMENTATION OF CALL-RETURN OF SUBPROGRAMS

After going through the previous sections, we can clearly differentiate the subprogram definition and subprogram activation. The definition of the program, is which is translated in to template on other hand activation is created each time when subprogram calls takes place.

We know that each time a new activation record is created, when the call of the subprogram takes place and destroyed when the subprogram returns only this knowledge is not giving us clear picture of working of activation record and call-return sequence control, since during the execution of subprogram the content of activation records are constantly changing as assignment are made to local variables and other data objects.

We must be very clear that which statement is executing during activation of which subprogram activation. To make this thing clear three system-defined pointers variables are used.

1. Current Instruction Pointer:

We know the CPU can execute one instruction at a time so at any time hardware or software interpreter can execute only one instruction and that instruction is said to be *current instruction*. The pointer which maintain the address of current instruction is called *Current Instruction Pointer* (CIP). Basically CIP performs two jobs:

- (i) CIP fetches the instruction designated by it
- (ii) Updating CIP to point out next instructions in sequence.

2. Current Environment Pointer:-

Now it is clear that, whenever we call a subprogram, a new activation record is pushed on the stack for that particular call of subprogram. It is also clear that all the activation records for different calls of same subprogram use the same code segment. Now suppose CIP is having address of data variable 'A', now only CIP can not resolve the issue that this data variable 'A' is used in which activation record's reference. So we need some pointer which can have the reference of activation record. This pointer is called current environment pointer (CEP).

Now a combination of both these pointers (ip, cp) can resolve the referencing issue and we can decide that which instruction executes in the environment of which activation record.

3. Return Point: It is a system defined data object which contains the space for two pointer values, instruction pointer (ip) and environment pointer (ep), return point have (ip, ep).

Now let us see how program is executed with the help of CIP, CEP and return point. Whenever a program is executed, very first activation record of main is created and CEP has its address. CIP has the address of first instruction of main. Now only CIP will change and no change in the value of CEP till new subprogram is called in the main. As soon as a new subprogram has been called the CEP will have the address of new activation record of that particular subprogram and CIP will have address of first instruction of called subprogram. The old value of CEP and CIP are saved in return point before assigning the new value.

Now as the subprogram call is over, its activation record is destroyed, CEP and CIP is updated from the value of return point (ip, ep). So control is transferred at the instruction of caller subprogram and now again CEP will have the address of activation record of caller subprogram. Now it is to understand the multiple call of a subprogram since all the issues are resolved by the CIP and CEP.

8.6 REFRENCING ENVIRONMENT

We know that each subprogram has its own set of identifiers associations available for use in referencing during its execution. This set of identifier association is termed as the referencing environment of the subprogram. The referencing environment is setup during creation of activation record and it remains intact in the life time of that particular activation record. The referencing environment of a subprogram may have several components like.

- 1. Local Environment:** Formal parameters of subprogram, local variables and subprograms defined inside subprogram (if language design permits) creates the local referencing environment.
- 2. Non local referencing environment:** The identifiers which are not defined at the entry point of subprogram, but used within in subprogram, creates non local referencing environment of the subprogram. Basically these identifiers are defined outside the subprogram and used in subprogram.
- 3. Global referencing environment:** The association of these identifiers created at above main and used in subprogram then these identifiers creates global referencing environment. Basically global environment is part of the non local environment.

4. *Predefined referencing environment:* Some identifier are defined by the language (for example inbuilt function in C) it self. These identifier creates predefined referencing environment in subprogram activation.

8.6.1 Aliases for Data Objects

Some languages allows different names for the same data object. There may be several association in different referencing environment, each providing a different name for the data object. Assume a situation where a data object is passed as parameter and then same data object is used in same activation with different name (original name). In another situation several pointer variable may have the address of same data object may leads towards the use of same data object under different name.

In these situation when a data object is visible through more than one name in a single referencing environment, each of the names is termed as alias for the data object. But aliasing of data object in same referencing environment creates the problem for user and implementer.

Let us consider an pascal example:

```

Program main;
  var X, Y, Z : int;
Procedure sub1 (X : int);
  var P : int;
Procedure sub2 (Z : int);
  var p : int;
begin
  statements
  ...
  ...
  Z = Z + y;
  ...
  ...
end;
begin
  ...
  statements
  ...
  ...
Sub2 (y)
  ...
  statements
  ...
end;
begin
  ...

```

```

statements
...
...
Sub1 (X);
...
statements
...
end

```

Now let us analyse the referencing environment of all subprograms as follows:

1. Refrencing environment of sub2 : Local: Z, P Nonlocal: X, sub2 in sub1 y, sub1 in main
2. Refrencing environment of sub1 : Local: X, P sub2 Nonlocal: y, z, sub1 in main
3. Refrencing environment for main: Local: x, y, z, sub1

8.7 ACCESS TO NON-LOCAL NAMES

In a language with lexical scope and nested procedures that is pascal or scheme, how do we know where to find (at run-time) a variable declared in an enclosing scope?

Consider a program of pascal:

```

Procedure P (m: integer)
begin
    x : integer;
    function q(n : integer) : integer;
    begin
        if n > 0 then
            return 2 * q (n - 1);
        else
            return x + 1;
    end
    print q(m + 2);
end.

```

The variable x lives in p's activation record. But we don't know how deep in the stack this may be, since we don't know how many levels of recursion there will be in q at run-time.

The basic idea to encounter these problems, code can be generated to pass an access link at each procedure call.

Suppose a procedure p is nested immediately within procedure q, then the access link to a procedure p is a pointer to the activation record of the most recent activation of q.

The code to set up access links can be determined at compile time, using the idea of nesting depth.

Intutively:

- The outer most scope has nesting depth = 0
- The nesting depth increases by 1 each time we enter a new scope, decreased by 1 when we leave a scope.

Suppose a procedure p at nesting depth n_p refers to a non-local variable 'a' whose nesting depth is n_a : the storage for 'a' is given by the pair:

$\langle n_p - n_a, \text{offset of } 'a' \text{ in Activation Record} \rangle$

The pair is computed at compile time. The first number gives the number of access link to be traversed.

8.7.1 Static Chains

We have seen that non local variable can be accessed in two steps:

1. The first step of the access process is to find the instance of the activation record in the stack in which variable was allocated.
2. The second part is to use the local offset of the variable to access it.

The most common way to implement the static scooping in language that allows nested subprogram is static chaining. To implement the concept we define a new pointer a *static scope pointer* (static link), which points the bottom of the activation record instance of an activation of the static parent. Typically static link appears below the parameters in the activation record.

A *static chain* is a chain of static links that connect certain activation record instances in the stack.

The Fig. 8.9 makes the concept more clear. Suppose refrencing environment of an activation uses the identifier which is not find in same then it will searched in very next (adjacent) instance of activation, till the definition of identifier is not find, by the use of static chain of activations. It is very easy since the nestic scope is known at compile time, the compiler can determine not only that a refrence is non local but also the length of static chain that must be followed to reach the activation record instance that contains the non local object.

8.8 THE BLOCK STRUCTURE:

A block is a statement with its own local data declarations. In C-based language the syntax of a block is:

```
{
    declarations statements;
    .....
    .....
    .....
    .....
}
```

where the braces be limit the block. Let us consider the following program of c language.

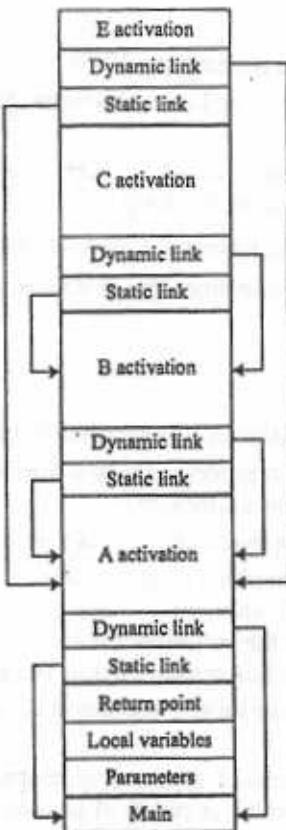


Fig. 8.9 Static chain of activation: The method for resolving non-local variables

```

main ( )
    /* start of block 0 */
    int x = 0;
    int y = 0;
    /* start of block 1 */
    int y = 1;
    /* start of block 2 */
    int x = 2;
    print f ("%d % d\n", x, y);
} /*end of block 2*/
/*start of block 3*/
    int y = 3
    print f ("% d %d\n", x, y);
} /*end of block 3*/
    print f ("% d %d\n", x, y);
}

```

```

} /*end of block 1*/
    print f ("%d% d\n", x, y);
} /*end of block 0*/

```

The scope of a declaration is given by the most closely nested rule:

1. The scope of a declaration in block B includes B.
2. If a name x is used in block B but is not declared in B then use a declaration of x in an enclosing block B, where B is more closely nested around B than any other enclosing block with a declaration of x.

Using this rule the four print statements of the example C program output:

2	1
0	3
0	1
0	0

Nesting of block structures can be treated at run time stack allocation: each time a new block is entered, space on a stack is allocated to hold the variables declared within that block and each time a block is exited its stack space is popped (*note that in the c example program above, x in block 2 and y in block 3 share the same stack allocation*).

Finally let us see activation of previous program as follows:

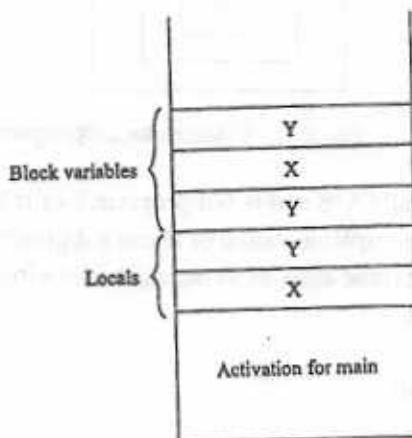


Fig. 8.10 Block variable activation record

8.9 IMPLEMENTATION OF DYNAMIC SCOPING

In chapter six we have seen the general rule of dynamic scoping, on that bases we can say that a typical dynamic scope rule states that the scope of an association created during an activation of subprogram x includes not only that activation but also any activation of a subprogram called by x, or called by a subprogram called by x, and so on, unless that later subprogram activation defines a new local association for the identifier that hides the original association.

According to this rule the dynamic association is tied to the dynamic chain of subprogram activation.

Dynamic-scoped language implements this concept in following two ways:

- (i) Deep Access.
- (ii) Shallow Access.

8.9.1 Deep Access

In case of dynamic-scoped language, the references to nonlocal variables can be resolved by searching through the activation record instances of the other subprograms that are currently activated. We can implement in similar fashion, as we did to access non local variable in a static-scoped language with nested subprograms, but in this case we follow dynamic chain instead of static chain. The dynamic chain link together all subprogram activation record instances in the reverse of the order in which they were activated.

Let us consider following example:

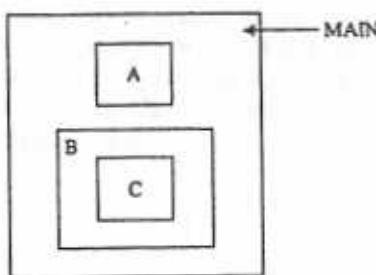


Fig. 8.11 Structure for subprogram call

In above Fig. Main calls A, B and B. Subprogram B calls C.

Figure 8.12 Show the implementation of above subprogram call structure.

To make the concept more clear let us consider following C program:

```
void main ( )
{
    int a, b;
}
void sub1( ){
    int a, c;
}
void sub2( ){
    int c, d;
}
void sub3( ){
    int d, e;
```

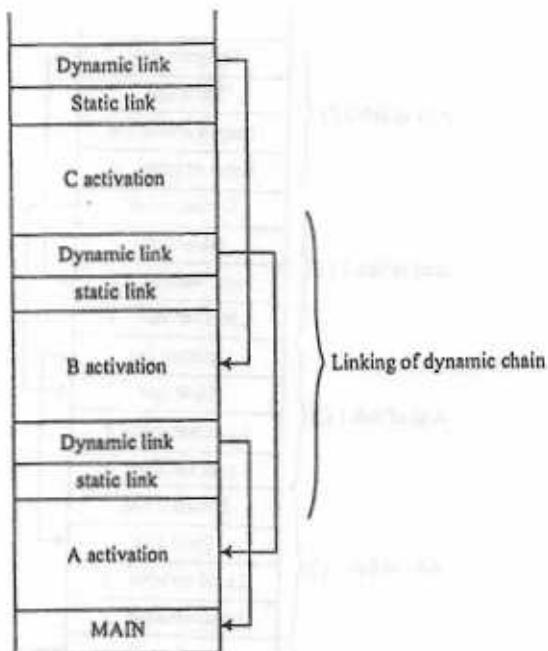


Fig. 8.12 Dynamic chain implementation

```
e = a + b;
}
```

Now let us consider the following calling sequence of above program:

```
main calls sub1
sub1 calls sub1
sub1 calls sub2
sub2 calls sub3
```

If we consider the references to variables 'e', 'a' and 'd' in sub3(). The reference of e is found in sub3() but references of a and d are not available in sub3(). Clearly the reference of a and d are searched as shown in Fig. 8.13.

Subprogram sub3() gets the reference of 'a' from the activation of sub1() (second activation of sub1()) and reference of b is found from the activation of main(), as it is searched from the activation of sub3 → sub2 → sub1 → sub1 → main and it is found in main().

8.9.1.1 Disadvantages of Deep Access

There are two main disadvantages of this access method:

1. The speed of execution of subprogram calling sequence becomes slow, since length of the chain is not calculated at compile time so overhead at runtime is more.
2. Activation records must store the names of variables for search process rather than value as in case of static implementation.

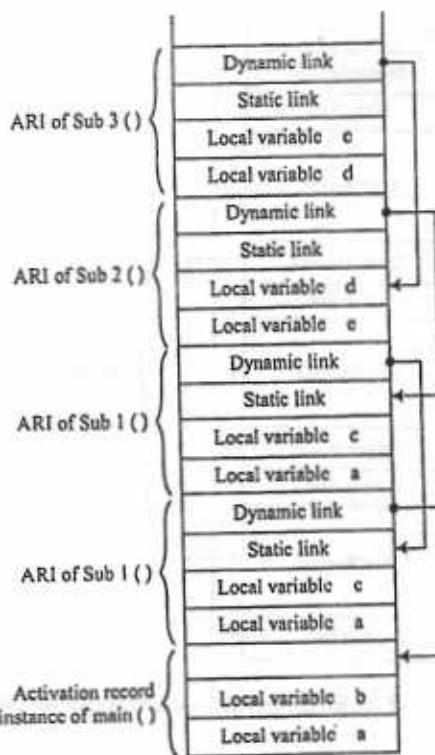


Fig. 8.13 Activation record instances of above sub program call sequences

8.9.2 Shallow Access

Now a days some programming languages like SNOBOL follows shallow access method with central table. Central table that has a location for each different variable name in a program. Along with each entry, a bit called *active* bit is maintained to show the binding of variable. In this case the offset of variable becomes static which makes the implementation faster. In case of this table method whenever a variable begins its life time, the active bit in its control table position must be set.

8.10 PARAMETER TRANSMISSION

We can established the communication between two or more then two subprogram if:

- We make the sharing of data/variable possible between different subprograms.
- We can established a correspondence, between actual and formal parameters.

The second approach is more applicable to make the execution of program efficient. We can established the correspondence by following two approaches

- Positional correspondence* is one method to established the relation between actual and formal parameters. In this case first actual parameter in paired with first formal parameter, and so on.

- Some languages like Ada established the *correspondance by explicit name* of formal and actual parameters.

For example in case of Ada:

```
sub (A => B, C => 27);
```

Here actual parameter B pairs with formal parameter A and actual parameter 27 with formal parameter C.

8.10.1 Methods for Transmitting Parameters

In this section we will discuss several methods of associating actual and formal parameters. like, call-by-value, call-by-reference, copy-restore, call-by-have, call-by-result. The left-value (l-value), the right-value (r-value), or the name of a variable may be passed to the called procedure.

1. Call-By-Value:

Usually, C and pascal use call-by-value: the r-values of actual parameters (values) are passed to the called procedure (to the formal parameters).

- A formal parameter is assigned a storage location in the activation record of the called procedure (just like a local variable).
- The caller evaluates the actual parameters and places their r-values (only values) in the storage locations of the formal parameters.

If the called procedure changes a formal parameter the change only occurs in the activation record of the called procedure. The change is lost when the record is deallocated so the actual parameter in the caller's record is not charged.

2. Call-by-reference:

Call-by-reference (or call-by-address of call-by-location) passes l-values to the called procedure:

- In an actual parameter is a name or an expression having an l-value, then that l-value is passed to the called procedure.
- If the actual parameter is an expression like $a + b$ or 2 which has no l-value, then the expression is evaluated in a new location, and the address of the location is passed to the called procedure.

The *var* keyword in the first line of the following pascal procedure causes call-by-reference instead of call-by-value:

```
Procedure swap (var x, y : integer);
  var temp: integer;
begin
  temp := x;
  x := y;
  y := temp;
end;
```

In the *swap* procedure, all reads and writes of the formal parameters, x and y, will read and write the associated actual parameters in the caller's activation record. Thus, *swap* will actually swap the actuals.

If the *swap* procedure above were called with *swap* (*i*, *a*[*i*]) then the following steps would occur:

1. The l-values of *i* and *a*[*i*] would be copied in to the activation record of *swap*.
2. *Temp* would be set to the r-value of *i*, say I_o .
3. *i* would be set to the r-value of *a*[I_o].
4. *a*[I_o] would be set to the r-value of *temp* or I_o .

Using call-by-reference, the r-values of the arguments are swaped correctly.

3. Copy-Restore:

Copy-restore is a hybrid of call-by-value and call-by reference:

- (i) The actual parameters are evaluated before the call and their r-value are passed to the called procedure as in call-by-values. But the caller also remember the l-values of those actual that have l-values.
- (ii) When control returns to the caller it copies back the r-values of the formal parameters into the l-values of the actuals.

Usually, *copy-restore* has the same effect as call-by-reference. It could have a different effect when the called procedure refers to one or more of the actual parameters as a nonlocal. The operation of the pascal program below depends on whether call-by-reference or copy-restore is used:

```
Program copyout (input, output);
  var a : integer;
  Procedure unsafe (var x: integer);
    begin x: = 2; a: =0 end;
  begin
    a: = 1; unsafe(a); writeln(a)
  end.
```

4. Call-by-Name::

Call-by-Name is used by Algol:

- (i) The procedure is treated as if it were a macro; that is its body is substituted for the call in the caller, with the actual parameters literally substituted for the formals.
- (ii) Local names in the procedure are kept distinct from names in the caller.
- (iii) Actual parameters are surrounded by parentheses if necessary to preserve their integrity.

If *swap* function (discussed in call-by-reference) is repeated with call-by-name instead of call-by reference:

1. The *swap* (*i*, *a*[*i*]) call in the caller is replaced with:
2. *temp*: =*i*)
3. *i*: = *a*[*i*];
4. *a*[*i*]: = *temp*
5. The first line copies the r-value of *a*[I_o] in to *temp*

6. The second line copies the r-value of $a[l_o]$ in to i .
7. The third line copies the r-value of $\text{temp}(l_o)$ into $[a[l_o]]$.

Using call-by-name, the r-values of the arguments are not swapped correctly.

5. Call-by Value-result:

If the parameter is transmitted by value-result:

- (i) The r-value (value) of the actual parameter is copied to formal parameter.
- (ii) During the execution of subprogram, each reference to the formal-parameter name is treated as an ordinary reference to a local variable, as with call by value.
- (iii) On the termination of subprogram the final content of formal parameter are copied to the actual parameters.
- (iv) In this case the actual parameters retain its original value until termination of subprogram takes place, when its new value is assigned as a result of the subprogram.

This type of concept is used in Algol-w language. To speed up execution, value-result transmission made all parameters local variable directly addressable by the current activation record pointer.

6. Call by constant value:

In this case:

- (i) Non change in the value of the formal parameter is allowed during program execution, since constant value is copied to the formal parameter.
- (ii) Formal parameter acts as local constant during execution of subprogram.
- (iii) Both call by-value and call by constant value protect the calling program from changes in the actual parameters.

7. Call by Result:

In this case:

- (i) Non transmission takes place from actual parameters to the formal parameters.
- (ii) Only result of subprogram execution is transmitted back from subprogram to actual parameters on the termination of subprogram.

EXERCISE

1. Describe the scope information for following program:

```

Program main
  var x, y: integer
Procedure P
  Variable x, a: Boolean;
Procedure Q
  variable x, y, z: real;

```

2. Consider the following program:

```
void foo()
{
    float a, b, c; /* level 0 declaration*/
    .....
    .....
    {
        float a, b; /* level '1x' declaration*/
        .....
        .....
    }
    {
        int d, e; /* level '1y' declaration */
        {
            int f; /* level '2' declaration */
            .....
            .....
        }
    }
}
```

Show the stack position offer execution of level 0, level 1x, level 1y and level2.

3. Consider the following program which computes the nth power of an integer p.

```
var p, r, s, y: integer
r := 1, x := p, y := n;
while (y ≠ 0) do
begin
    while (even) do
        y := y/2, x = x × x;
end
    y := y - 1; r = r * x;
end
```

Derive formula which describe the relationship between the variable at the beginning, end and middle of the program.

In case of error neous program, make suitable changes.

4. What is the difference between activation record and an activation record instance?

Memory Management

9.1 INTRODUCTION TO MEMORY MANAGEMENT

Any non-trivial program needs to allocate and free memory. Memory management techniques become more and more important as programs increase in complexity, size and performance. As far as allocating storage space for the life time of a variable, we can divide memory management in three types:

1. If the life time of a variable is the life time of the program and space for its value, *once allocated, can not be released later*. Such storage is referred to as static storage.
2. If the life time of variable is particular block, function or procedure, in which variable is declared. The storage allocated to variable may be deallocated when the execution of block, function or procedure is over. Such storage is referred to as *dynamic storage*.
3. Storage may be allocated for values, not necessarily associated with variables, at a particular point in the execution of a program not necessarily corresponding to start of a block or the entry to a procedure. The storage is then required from that point on until it is released either by language mechanism or through simply being no longer reachable from the program. However, the movement of this release, this space is only known at compile time. Such a storage is referred to as *global storage*.

At the end of introduction we can finally conclude that at run time following issues must be taken care:

- Managing the relationship between names in the source program and data objects that exit at run-time.
- Managing the allocation/de-allocation of, and access to, data objects at run-time.
- Controlling and keeping track of different procedure and their calls.
- managing the library function, provided by the language.

9.2 MAJOR RUN-TIME ELEMENTS REQUIRING STORAGE

The programmer tends to view storage management largely in terms of storage of data and translated programs. However, run-time storage management contains many other areas. Let us look at the major program and data element requiring storage during program execution.

1. *Code segments for translated user program:* A major block of storage in any system must be allocated to store the code segments representing the translated form of user programs, regardless of whether program are hardware or software interpreted.
2. *Run-Time Programs:* Another substantial block of memory during execution must be allocated to system programs that support the execution of the user programs. These may include from simply library routines, such as print and scanf functions in C language, software interpreters or translators present during execution. It also includes the routines that run-time storage management.
3. *Data structures and constants defined by users:-* Certainly space is required for all data structures and constants which are created by programmer in user's program.
4. *Coroutine resume points/subprogram return points:* We know that subprogram may be invoked from different points in same program. So clearly to maintain the caller and callee sequence information, such as sub-program return points, coroutine resume points, or event notices for scheduled subprograms must be allocated storage.
5. *Reference Environments:* Storage of referencing environment (identifier association) during execution may require substantial space.
6. *Temporary variable in expression evaluation:* we know that evaluation of expression requires the use of system-defined temporary storage for the intermediate result evaluation. For example let us see the evaluation of $(a + b) \times (c + d)$, the result of first addition may have to be stored in the temporary, while the second addition is performed. When expression contains recursive function calls, then unlimited number of temporary variable may be required to store partial result at each level of recursion.
7. *Temporary variables in parameter transmission:* When a sub-program is called, a list of actual parameters must be evaluated and the resulting values stored in temporary storage until evaluation of entire list is complete. Again in case of recursive function call unlimited number of variables may be required.
8. *Input-Output Buffers:* We know that input-output operations are performed through buffers, which is a temporary storage, where data are stored between the time of the actual physical transfer of the data to or from external and the input-output operations. According to the language requirement hundred of memory locations can be reserved for these buffers during execution.
9. *System defined data:* Every language implementation is required for various system data such as various tables, status information for input-output, reference count or garbage collection bits, and each of these system data required storage.

9.3 SUB-DIVISION OF RUN-TIME MEMORY

We assume that the compiler obtains a block of storage for the compiled program to run in. Run-time memory needs to be sub-divided to hold the different components of an executing program:

- Generated executable code
- Static data objects
- A structure to keep track of procedure activations. This can generally be considered to consist of two components:

- *The stack*: for objects whose life time do not exceed the life time of activation; and
- *The heap*: for the objects whose life time exceeds that of the activation.

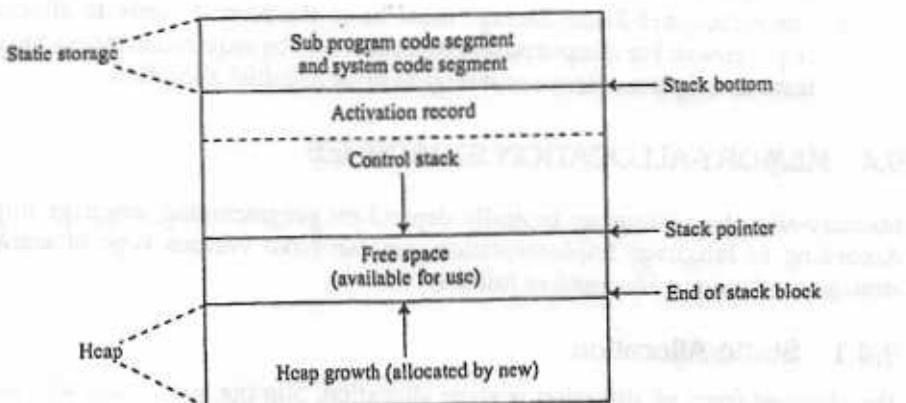


Fig. 9.1 Run-time division of memory

A typical subdivision might be:

9.3.1 Organization of Code Area

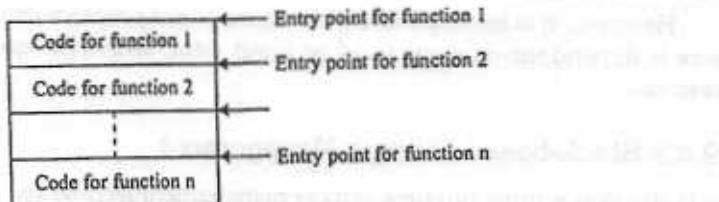


Fig. 9.2 Code area organization

Usually code is generated of a function at a time. Thus, the code area layout is of the form:

Within a function, the compiler has freedom to organize the code in any way. Careful layout of code within a function and careful attention to the order in which the functions are processed can improve cache utilization.

9.3.2 Memory Management Phases

Basically storage management process can be completed in three phases, which are as follows:

1. *Initial allocation*: At the start of execution each piece of storage may be either allocated for some use or free. If free initially, it is available for allocation dynamically as execution proceeds. Storage manage system must have some specific mechanism for keeping track of free storage as well as mechanism for allocation of free storage as the need arises during execution.

2. *Recovery*: Free storage must be recovered for further allocation. Storage manager must follow some mechanism, like repositioning of a stack pointer, or very complex as in garbage collection.
3. *Compaction and Reuse*: Storage must be in the format ready to allocated for some requirement. For this purpose compaction may be required to form a big piece of space from small pieces. Reuse of storage is same as initial allocation.

9.4 MEMORY-ALLOCATION STRATEGIES

Memory-allocation strategies basically depend on programming language implementation. According to language implementation, we can have various type of storage-allocation strategies, which are discussed as follows:

9.4.1 Static Allocation

The simplest form of allocation is static allocation. Storage space once allocated was never released, so a very simple storage allocation model that allocated storage, as required, from one end of the available space towards the other, was adequate.

The programming languages like Fortran, had static storage, the amount of which was known at compile time static storage allocation is efficient because no time or space is expended for storage management during execution. The translator can generate the direct l-value address for all data items.

However, it is incompatible with recursive subprogram calls, with data structures whose size is dependent on compacted or input data, and with many other desirable language features.

9.4.2 Stack-based Storage Management

It is also very simple run-time storage management technique.

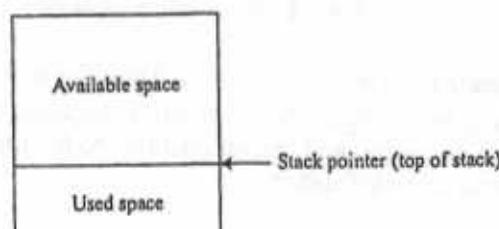


Fig. 9.3 Stack management

Storage allocation begins at one end. Storage must be freed in the reverse order of allocation so block allocated space recently must free the space first. *Only a single stack pointer is all that is needed to control storage management.*

The stack pointer always points to the top of the stack, the next available word of free storage in the stack block. Compaction occurs automatically recovers the freed storage and makes it available for reuse.

We know that the most of the time program and data elements requiring storage are tied to subprogram activations. When a subprogram is called, per call a new activation record is created on the top of stack and termination causes its deletion from the stack. *Algol-like languages including pascal and C are the example.* Let us take the example of C-language to understand the concept.

Now consider the following c program:

```
void term( )
{
    int x, y, z; /* level '0' declaration */
    .....
    .....

    {
        int a, b; /* level '1x' declaration */
        .....
        .....

    }
    {
        float z, p /* level '1y' declaration */
        {
            int q; /* level '2' declaration */
            .....
            .....

        }
    }
}
```

Now let us analyse the program.

Initially stack is empty:



Fig. 9.4(a) Empty stack

After the declaration of x, y, z (level 0) the run-time stack could look as in Fig. 9.4(b).

After the declaration of level 1x it could appear as in Fig. 9.4(c).

On leaving level 1x, at execution time the run-time stack may revert to the Fig. 9.4(d) and to entering level 1y it could become as in Fig. 9.4(e).

On entering level2 run-time stack becomes as in Fig. 9.4(f) and on leaving level 2 (at run-time) (Fig. 9.4(g)).

On leaving level 1y, it reverts to Fig. 9.4(h)

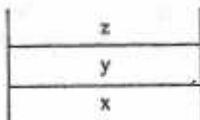


Fig. 9.4(b) Stack at level 0

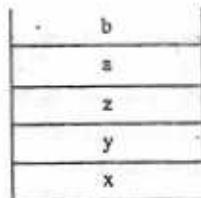


Fig. 9.4(c) Stack at level 1x

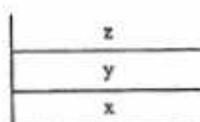


Fig. 9.4(d) Stack after leaving 1x

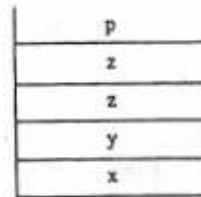


Fig. 9.4(e) Stack at level 1y

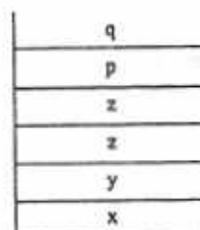


Fig. 9.4(f) Stack at level 2

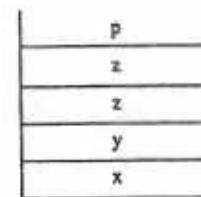


Fig. 9.4(g) Stack after leaving level 2

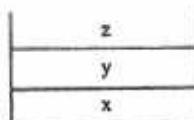


Fig. 9.4(h) Stack after leaving level 1y

Stack again becomes empty on leaving the function scopes. According to the above description on leaving a compound statement is simply removed from the stack. To do this, array of pointers may be maintained pointing to the foot of the stack segments corresponding to the compound statements that are currently being executed.

The calculation of the address of a variable with respect to the foot of the stack frame merely requires acknowledgement of the amount of storage occupied by the value of each of the variables below it on the stack-information that is known at compile time.

9.4.3 Dynamic Memory Management

We cannot use a stack-based discipline for function calls in a functional language because of difficulties in returning functions as values from other functions. As a result, activation records

must be allocated from a heap. Similar difficulties in passing around closures result in most object-oriented language relying on *heap* allocated memory for objects. Because it is often not clear when memory can be safely freed, such languages usually rely on an automatic mechanism for recycling memory.

9.4.3.1 Memory Management in the Heap

A heap is usually maintained as a list or stack of block of memory. Initially all of the free space is maintained as one large block, but requests (whether explicit or implicit) for storage and subsequent recycling of blocks of memory will eventually result in the heap being broken down in to smaller pieces.

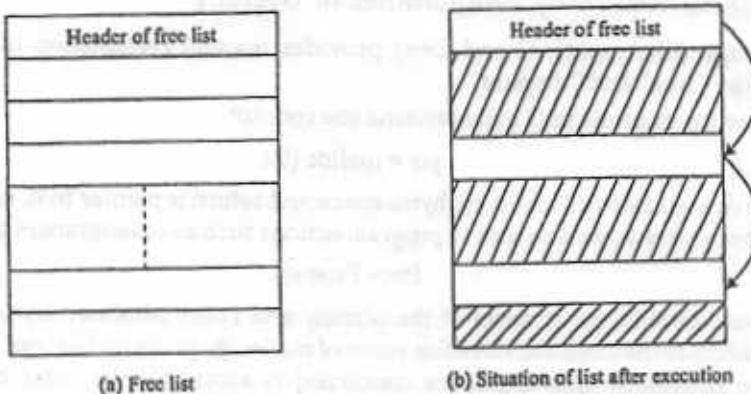


Fig. 9.5 Heap-free list

Figure 9.5(a) represent free list, that is initial situation of heap. In Fig. 9.5(b) shaded regions are represent elements that were allocated and yet not freed.

When a request is made (that is, via a "new" statement) for a block of memory, some strategy will be undertaken to allocate a block of memory of the desired size. For instance one might search on the list of free space for the first block which is at least as large as the block desired or one might look for a "best fit" in the sense of finding a block which is as small as possible, yet large enough to satisfy the need.

Whichever technique is chosen, only enough memory as is needed will be allocated, with the remainder of the block returned to the stack of available space.

Unless action is taken, the heap will eventually be composed of smaller and smaller blocks of memory. In order to prevent this, the operating system will normally attempt to merge or coalesce adjacent block of free memory. Thus whenever a block of memory is ready to be returned to the heap, the adjacent memory locations are examined to determine whether they are already on the heap of available space. If either (or both) are, the they are merged with the new block and put in the heap.

Even with coalescing the heap can still becomes fragmented, with lots of small blocks of memory being used alternating with small blocks in the heap of available space.

This problem can be fixed by occasionally compacting memory by moving all blocks in use to one end of memory and then coalescing all the remaining space in to one large block. This can be very complex since pointers in all data structures in use must be updated.

9.5 RECLAMATION OF FREE STORAGE

Reclamation of free storage is very important issue in memory management. We can divide the reclamation technique in three parts:

1. Explicit Return by programmer or system.
2. Reference Counting
3. Garbage Collection.

Every technique has its own advantages and disadvantages. In this section we will critically analyse these techniques.

9.5.1 Explicit Return by Programmer or System

Some language (like pascal, C and C++) provides manual reclamation of storage using operations like "free" and "dispose".

Let us see the example of C to understand the concept.

```
ptr = malloc (6);
```

The above statement allocates six bytes space and return a pointer to it, now this storage space may become inaccessible due to program actions such as reassignment to pointers; etc.

```
Ptr = Ptnew;
```

Now ptr contains the address of the ptnew and space allocated by above malloc(6) statement becomes inaccessible. Here this space of malloc (6) becomes *Garbage*. However, there remains the possibility that the space concerned is accessible via some other variables. Consider the effect of executing an assignment such as

```
ptr 1 = ptr;
```

This assignment would make the space allocated by malloc accessible via the variable ptr 1.

But we have no method by which we can know that which of the path a program will follow at run-time. This means that code to reclaim heap space cannot be generated at compile time, despite the fact that large areas of space allocated on the heap can actually become inaccessible. Language C provides the facility to free the space as follows:

```
free (ptr);
```

But these kind of facilities put a considerable amount of responsibility on the programmer.

9.5.1.1 Problem of Garbage and Dangling in Explicit Allocation

When all the access path to a data object are destroyed but the data object continues to exist, the data object is *said to be garbage*.

Let us see example of C language;

```
int *p;
P = malloc (10)
int * q;
q = malloc (6)
p = q;
```

In above code segment initially p has the address of malloc (10) but when p = q then p has the address of location of malloc (6), in this space allocated by malloc (6) becomes *garbage*.

In this case object can no longer be accessed from other parts of the program, so it is of no further use. However, the binding of data object to storage location has not been broken, so the storage is not available for reuse.

A *dangling reference* is an access path that continues to exist after the life time of the associated data object. At the end of the life time of the object, this block of storage is recovered for reallocation at some later point to another data object. However, the recovery of the storage block does not necessarily destroy the existing access path block, and thus they may continue to exist as *dangling reference*.

Let us consider code segment of c language:-

```
int *x ;
x = malloc (6);
int *y;
y = x;
free (x);
```

Now y is reference to that location which does not exist so it is dangling reference.

Dangling references are a particular serious problem for storage management because they may compromise the integrity of the entire run-time structure during program execution. Garbage is less serious, but still troublesome problem. A data object that has become garbage ties up storage that might otherwise be reallocated for another purpose.

9.5.2 Reference Counting

Reference counting is conceptually simpler than garbage collection, but often turn out to be less efficient overall. The idea behind reference counting is that each block of memory is required to reserve space to count the number of pointers to it.

Thus an assignment of pointers of the form p: = q is executed then the block of memory that p originally points to has its reference count decreased by one, while that pointed to by q would have its count increased by one.

If the count on a block of memory is reduced to zero, it should be returned to the heap of available space. However, if it has pointer to other blocks of memory, those blocks should also have their reference counts reduced accordingly.

One big drawback of this system is that each block of memory allocated must have sufficient space available to maintain its reference count. However a more serious problem is the existence of circular lists. Even if nothing else to the circular list, each item of the list will have another item of the list pointing to it. Thus even if a circular list is inaccessible from the program, the reference count of all of its will still be positive and it will not be reclaimed.

9.5.3 Garbage Collection

Garbage collection is a more common way of handling automatic storage reclamation. The basic idea is that computation continues until there is no storage left to allocate. Then the garbage collector marks all the blocks of memory that are currently in use and gathers the rest (the garbage) into the heap available space.

The procedure of garbage collection can be understood as follows:

When (free-space list is entirely exhausted and more storage is needed)

```

    {
        garbage collection ( )
        {
            mark ( ); /* method of garbage collection contains
            two phases */ sweep ( );
        }
    }

```

Marking phase: In this step each element in the heap which is active (An element is active if there is a pointer to it from outside the heap or from another active heap element) that is, which is part of an accessible data structure, must be marked. A bit map may be used containing sufficient bits to correspond to each cell in the heap. The bit map will not be part of the heap itself, but will be distinct from it. Each bit in the bit-map may take one of the two values.

- (i) meaning the corresponding memory cell is not able to be accessed via a program variable (active element)
- (ii) meaning the corresponding memory cell may be accessed via a program variable

At the start of garbage collection, all the elements of the bit map will be set 0, and the execution of the garbage collection algorithm progresses, various elements of the bit map will be set to 1. Eventually all the bit map elements corresponding to memory cells that are reachable via program variables will have been set to 1.

Sweep: Now after executing the marking algorithm, we can easily distinguish the active elements and non active objects in the program. Through one sequential scan we can identify all those objects, having bit value 0 and now the memory occupied by these objects can be linked to the heap free-space list for further use.

There are two problems with this technique. The first is the space necessary in order to hold the mark (though this can just be one bit). A more serious problem is that this algorithm requires two passes through memory: The first to mark and the second to collect. This can often take a significant amount of time, making this sort of garbage collection unsuitable for real-time systems. This disadvantage has led to this method being abandoned by most practical systems.

9.5.3.1 Improvement In Garbage Collection

There have been several recent improvements in garbage collection algorithms. The first is sometimes known as a copying collector.

In this algorithm the memory is first divided into two halves, the *working half* and the *free half*. When memory is exhausted in the working half, live nodes are copied to the free half memory, and the roles of the two half halves can be switched. Notice that the collector only looks at live cells, rather than all. This kind of concept tends to work well with a virtual memory system.

Another strategy is to use a generational collection in which only bother to garbage collect recently allocated blocks of memory. Older blocks are moved into stable storage and not collected as often. Studies have shown that the most reclaimed garbage comes from more recently allocated blocks of memory.

In highly parallel architectures can have garbage collection take place in background, minimizing or eliminating delays.

9.5.3.2 The Advantages of Garbage Collection

The garbage collection have following advantages:

1. Reference counting is a common solution to solve explicit memory allocation problems. The code to implement the increment and decrement operations wherever assignments are made is one source of slowdown. Hiding it behind smart pointer classes doesn't help the speed.
2. Destructors are used to deallocate resources acquired by an object. For most classes, this resources is allocated memory memory. With garbage collection, most destructors then become empty and can be discarded entirely.
3. All those destructor freeing memory can become significant when objects are allocated on the stack. For each one, some mechanism must be established so that if an exception happens, but destructors become irrelevant, then there is no need to setup special stack frames to handle exceptions, and the code runs faster.
4. Garbage collection kicks in only when memory gets tight. When memory is not tight, the program runs at full speed and does not spend any time freeing memory.
5. Modern garbage collectors are far more advanced now than the older, slower ones. Generational, coping collectors eliminates much of the inefficiency of early mark and sweep algorithms.
6. Modern garbage collectors do heap compaction. Heap compaction tends to reduce the number of pages actively referenced by a program, which means that memory accesses are more likely to be cache hits and less swapping.
7. Garbage collected program do not suffer from gradual deterioration due to an accumulation of memory leaks.
8. Garbage collectors reclaim unused memory, therefore they do not suffer from "memory leak" which can cause long running applications to gradually consume more and more memory until they bring down the system. Garbage collection programs have longer term stability.
9. Garbage collection program have fewer hard-to-find pointer bugs. This is because there are no dangling references to freed memory. There is no code to explicitly manage memory, hence no bugs in such code.
10. Garbage collected program are faster to develop and debug, because there is no need for developing debugging testing, or maintaining the explicit deallocation code.
11. Garbage collected program can be significantly smaller, because there is no code to manage deallocation, and there is no need for exception handlers to deallocate memory.

9.5.3.3 Drawbacks of Garbage Collection

Garbage collection is not a panacea. There are some downsides:

1. It is not predictable when a collection gets run, so the program can arbitrarily pause.
2. The time it takes for a collection to run is not bounded while in practice it is very quick, this cannot be guaranteed.

3. All threads other than the collector thread must be idle while the collection is in progress.
4. Garbage collectors can keep around some memory that an explicit deallocator would not. In practice, this is not much of an issue since explicit deallocators usually have memory leaks causing them to eventually use up more memory, and because explicit deallocators do not normally return deallocated memory to the operating system anyway, instead just returning it to its own internal pool.
5. Garbage collection should be implemented as a basic operating system kernel service. But since they are not, garbage collecting programs must carry around with them the garbage collection implementation. While this can be shared DLL, it is still there.

9.5.3.4 Interfacing Garbage Collection Objects with Foreign Code

The garbage collector looks at roots in its static data segment, and the stacks and register contents of each thread. If the only root of an object is held outside of this, then collector will miss it and free the memory. To avoid this from happening,

- Maintain a root to the object in an area the collector does scan for roots.
- Relocate the object using the forcing code's storage allocator or using the C runtime library's malloc/free.

9.5.3.5 Pointers and the Garbage Collector

Broadly, we can divide pointers into two categories: those that point to garbage collected memory, and those that do not. Examples of the latter are pointers created by calls to C's malloc(), Pointers received from C library routines, pointers to static data, pointers to objects on the stack, etc. For those pointers, anything that is legal in C can be done with them. For garbage collected pointers and references, however, there are some restrictions. These restrictions are minor, but they are intended to enable the maximum flexibility in garbage collector design. The undefined behavior:

1. We should not XOR pointers with other values.
2. We should not use the XOR trick to swap two pointer values.
3. We should store pointers into non-pointer variables using casts and other ticks.

```
void * p;
.....
int x = cast (int) p; // error: undefined behaviour
```

4. We should not take advantage of alignment of pointers to store bit flags in the low order bits:


```
p = cast (void *) (cast (int) p|1); // error
```
5. We should not store into pointers values that may point into the garbage collected heap:


```
p = cast (void *) 12345678; // error: undefined behavior
```

 A compacting garbage collector may change this value.
6. We should not store magic values into pointers, other than null.
7. We should not write pointer values out to disk and read them back in again.

8. We should not use pointer values to compute a hash function. A copying garbage collector can arbitrarily move objects around in memory, thus invalidating the computed has value.

9. We should not depend on the ordering of pointers:

```
if (p1 < p2) // error: undefined behavior
```

Since, again, the garbage collector can move objects around the memory.

10. We should not add or subtract an offset to a pointer such that the result points outside of the bounds of the garbage collected object originally allocated.

```
char *p = new char [10];
char *q = p + 6; // this is ok
q = p + 11; // error: undefined behavior
q = p - 1; // error: undefined behavior
```

11. We should not misaligned pointers if those pointers may point into garbage collection heap, such as:

```
align (1) struct kiet
{
    byte b;
    char *p; // misaligned pointer
}
```

Misaligned pointer may be used if the underlying hardware support them and the pointer is never used to point into the garbage collection heap.

12. We should not use byte-by-byte memory copies to copy pointer values. This may result in intermediate conditions where there is not a valid pointer, and if the garbage collection pauses the thread in such a condition, it can corrupt memory.

Now let us see the things that are reliable and can be done:

1. Use a union to share storage with a pointer:

```
Union U { void *ptr; int value}
```

Using such a union, however, as a substitute for a cast (int) will result in undefined behavior.

2. A pointer to the start of a garbage collected object need not be maintained if a pointer to the interior of the object exists.

```
char [ ]P = new char [10];
char [ ]q = p[3 ... 6];
```

Here q is enough to hold on to the object, don't need to keep p as well.

9.5.3.6 Working with Garbage Collector

Garbage collection doesn't solve every memory deallocation problem. For example, if a root to a large data structure is kept, the garbage collector cannot reclaim it, even if it is never referred to again. To eliminate this problem, it is good practice to set a reference or pointer to an object to null when no longer needed.

This advice applies only to static references or references embedded inside other objects. There is not much point for such stored on the stack to be nulled, since the collector doesn't scan for roots past the top of the stack, and because new stack frames are initialized anyway.

9.5.4 Memory Leaks

In any language, application-level memory management problems revolve around the deallocation of memory. These problems fall into two categories:

- (i) Premature deallocation (corrupted pointers)
- (ii) Incomplete deallocation (memory leaks).

In case of incomplete deallocation, there are two subases:

- Coding bugs and
- Design bugs.

Coding bugs are language dependent. In the case of C, this would involve free()ing less than was malloc()ed, while in C++ this might involve using delete in lieu of delete []. Design bugs, on the other hand, do not depend on the language; instead, they involve simple programmer negligence.

In languages like C/C++, all memory management is handled by the programmer, so all of these problems can arise, even after the programmer has expended much effort to ensure the code is free of such defects. In fact, in C/C++ the more try to avoid memory leaks, the more likely. We have to create corrupted pointers, and vice versa. And, by nature the risk of such bugs increases with code size and complexity, so it's difficult to protect large C/C++ applications from these types of bugs.

In Java, on the other hand, the Java language and runtime together entirely eliminate the problems of corrupted pointers and code-level memory leaks. Let us see how Java deals with this problem:

- In Java, memory is allocated only to objects. There is no explicit allocation of memory, there is only the creation of new objects. (Java even treats arrays types as objects)
- The Java runtime employs a garbage collector that reclaims the memory occupied by an object once it determines that object is no longer accessible. This automatic process makes it safe to throw away unneeded object references because the garbage collectors does not collect the object if it is still needed elsewhere. Therefore, in Java the act of letting go of unneeded references never runs the risk of deallocating memory prematurely
- In Java, it's easy to let go of an entire "tree" of objects by setting the reference of the tree's root to null; the garbage collector will then reclaim all the objects (unless some of the objects are needed elsewhere). This is a lot easier than coding each of the objects destructors to let go of its own dependencies (which is a coding-level problem with C++)

But memory leaks may be caused by poor program design too. In such designs, unnecessary object references originating in long-lived part of the system prevent the garbage collector from reclaiming objects that are in fact no longer needed. Such errors typically involve a failure "in the longer" to properly encapsulate object references among various parts of the code. Another design flaw occurs "in the small" at the level of a faulty algorithm; the use of collections objects in such cases will typically magnify the error.

As the technology improves, a suitable implemented JVM (Java Virtual Machine) could help reduce the effect of such designed-in memory leaks by using the garbage collector to track object usage over time. The garbage collector could then rearrange object in memory according to a freshness factor based on when they are last referenced.

Ideally of course, programmers would design applications with objects lifecycles in mind, rather than rely on clever features of state-of-the-art JVM implementations.

9.5.5 Reference Counting vs. Automatic Garbage Collection

In this section we will compare Automatic Garbage Collection with reference counting with the assumption that reference counting is not used for cyclic data:

- The destructor is invoked immediately upon the reference count dropping to zero. This is better than AGC technique. It might even be better than using manual reclamation, since a programmer might not optimally place deletes.
- Some AGC technique pause the program while garbage collection occurs. These pauses can vary with overall system conditions. The suitability of these AGC technique for real-time or interactive application tends to be less than either reference counting or manual reclamation.
- Reference counting is less efficient than manual reclamation. There would be overhead each time a reference is added or removed. Also, the reference count takes up space, either with smart pointer or the object pointed to.

AGC technique can introduce significant overhead into a program even if only used for a single object. This is not so for reference counting, which has more of a per-object overhead (with some objects having more active reference count than others). The per-object nature of reference counted overhead, along with the similarities in behavior noted above between reference counting and manual reclamation, makes it feasible to consider *hybrid arrangements*.

Cyclic data would be the prime candidate to be manually reclaimed, as reference counting does not work correctly with it. But other objects could also be manually reclaimed for better efficiency. A programmer might use reference counting for objects during the initial phases of development, with the option of later investing time to code delete statements for some or all of these objects.

Some programming languages use different approach, except for cyclic data structures, the language automatically manages dynamically created objects with reference counting, and consider it a form of AGC. As explained above, reference counting does not work correctly with cyclic data. The designer sometimes chose to manage cyclic data with mark-and-sweep AGC, instead of manual reclamation.

9.6 FIXED AND VARIABLE-SIZE ELEMENT

To manage the heap storage is more difficult where the programmer controls the allocation and recovery of variable-size elements in spite of that almost all concepts are the same. By analysis of following points we can clearly understand the concept.

1. In case of fixed-size elements heap can be divided into equal size blocks, as per size of element. But in case of variable-size elements complete heap area is treated as one block initially since we can not divide it as we do in fixed-size case.
2. The major difficulties with variable-size elements concern reuse of recovered space. Even if we recover two five-word blocks of space in the heap, it may be impossible to satisfy a later request for a seven-word block. Fixed-size case never encountered with such problems; recovered space could always be immediately reused.

3. Heap storage management system using variable-size elements faces the problem of fragmentation, since some time it is not possible to satisfy a request inspite of that memory is available in fragmented blocks, which are not large enough to satisfy the request without compaction.
4. In case of fixed-size elements allocation is very simple. As the request comes, it can be satisfied with exactly fit block of memory. On the other hand in case of variable-size elements either we should adopt first-fit (The free-space list is scanned for the first block of required or more words, which is then split into an required word block, and remainder, which is returned to the free-space list) or Best-fit: In this case heap free-space list is scanned for the block with minimum number of words greater than or equal to required.

9.7 RECOVERY IN CASE OF VARIABLE-SIZE BLOCKS

Generally recovery schemes are same with some modifications, as we study in case of fixed-size blocks:

- Here also we can use *explicit recovery scheme*, which is the simplest one. Here also this method suffers with problem of garbage and dangling references.
- In case of variable-size block we can use *reference count* as usual.
- Garbage collection is also a feasible technique with little modification. Here during collection procedure (sweep) we have to maintain two things: *One garbage collection bit* to tell that block is active or non active and an *integer length indicator* specifying the length of the block.

9.8 MEMORY MANAGEMENT IN VARIOUS LANGUAGE

Most of the modern high-level language have automatic memory management, with C++ as the notable exception. For object oriented language, like Java, common Lips or Modula-3, it helps in designing systems with a high level of abstraction and low coupling between classes. In this section we will study memory management in various languages.

9.8.1 Memory Management of ALGOL

ALGOL, designed in 1958 for scientific computing, was the first block-structured language.

The block structure of ALGOL 60 induced a *stack allocation discipline*. It had limited dynamic arrays, but no general *heap allocation*. The substantially redesigned ALGOL 68 had both *heap and stack allocation*. It also had like the modern pointer type, and required garbage collection for the heap. The new language was complex and difficult to implement, and it was never as successful as its predecessor.

9.8.2 Memory Management of BASIC

BASIC is a simple and easily-learned programming language developed by T.E Kurtz and J.C Kemeny in 1963–4. The motivation was to make computers easily accessible to under graduate student in all disciplines.

Most BASICs had quite powerful string handling operations that required a simple *garbage collector*. In many implementation, the garbage collector could be forced to run by running the mysterious expression `FRE("")`.

9.8.3 Memory Management in C

C is a systems programming language sometimes known as "a portable assembler" because it was intended to be sufficiently low-level to allow performance comparable to assembler or machine code, but sufficiently high-level to allow programs to be reused other platform with little or no modification.

Memory management is typically manual (the standard library functions for memory management in C, `malloc` and `free`, have become almost synonymous with manual memory management). The language is notorious for fostering large number of memory management bugs, including:

1. Accessing arrays with indexes that are out of bounds;
2. Using *stack-allocated structures beyond their life times*.
3. Using *heap-allocated structures after freeing them*.
4. Neglecting to free heap-allocated objects when they are no longer required.
5. Failing to allocate memory for a pointer before using it;
6. Allocating insufficient memory for the intended contents;
7. Loading from allocated memory before storing in to it.
8. Dereferencing non-pointers as if they were pointers.

9.8.4 Memory management in COBOL

COBOL was designed by the CODASYL committee in 1959-60 to be a business programming language, and has been extended many times since.

COBOL has no *heap allocation*, and done quite well in its domain without it. The next version of the ISO standard, stated in 2002, is having *pointers* and *heap allocation* through `ALLOCATE` and `FREE`, mainly in order to be able to use C-style interfaces. It also in order to be supports a high level of abstraction through object-oriented programming and *garbage collection*.

9.8.5 C++ Memory Management

C++ is a (weakly) object orient language, extending the systems programming language C with a multiple-inheritance class mechanism and simple method dispatch.

The standard level of C++ makes the bookkeeping required for *manual memory management* even harder. Although the standard library provides only manual memory management, with the Boehm-weiser collector. It is now possible to use garbage collection. Smart pointers are another popular solution.

The language is notorious for fostering large number of memory management bugs, including:

1. Using *stack-allocated structures beyond their life times*.
2. Using *heap-allocated structures after freeing them*.

3. Neglecting to free heap-allocated objects when they are no longer required.
4. Excessive copying by copy constructor.
5. Unexpected sharing due to insufficient copying to copy constructors.
6. Allocating insufficient memory for the intended contents.
7. Accessing arrays with indexes that are out of bounds.

9.8.6 Fortran Memory Management

Fortran, created in 1957, was one of the first language qualifying as a high-level language. It is popular among scientist and has substantial support in the form of numerical libraries. For a long time, it had *static allocation* only. The Fortran 90 standard added recursion with *stack allocation* (automatic arrays). It also added *dynamic allocation* using ALLOCATE with manual deallocation using DEALLOCATE.

9.8.7 Java Memory Management

Java is a modern object oriented language with a rich collection of useful features. The Java language started as an attempt by the Java group at sun to overcome software engineering problems introduced by C++. Key reasons for the language's success were the security model and the portable execution environment the Java virtual machines (JVM), which created a lot of interest for it as a platform for distributed computing on open network.

Java is *Garbage-collected*, as this facilitates object-oriented programming and is essential for security. It had *finalization*.

9.8.8 Java Script Memory Management

Javascript is a scripting language used by some web browsers. Note that Javascript is not related to Java in any way except name. There is a standard by ECMA known as ECMS script.

String management resembles BASIC: there is no way to *manually* deallocate strings created by string concatenation and string methods.

9.8.9 Lisp Memory Management

Lisp is a family of computer languages combining functional and procedural features with *automatic memory management*.

Lisp was invented by John McCarthy around 1958 for the manipulation of symbolic expression. As part of the original implementation of Lisp, he invented *garbage collection*. Modern Lisp used advanced *garbage collectors*.

9.8.10 ML Memory Management

Like other functional language, ML provides *automatic memory management*. Modern ML implementation usually have advanced *garbage collectors*.

9.8.11 Modula-3 Memory Management

This language is object-oriented descendant of pascal. Modula-3 is mostly *garbage-collected*, although it is possible to use *manual memory management* in certain modules.

9.8.12 Pascal Memory Management

It is an imperative language characterized by block structure and a relatively strong static type system. Pascal was designed by Niklaus Wirth around 1970. Pascal uses *manual memory management* (with operators NEW and DISPOSE). The descendants mentioned all after *automatic memory management*.

9.9.13 Perl Memory Management

Perl is a complex but powerful language that is an electric mixture of scripting languages and programming languages.

Perl programmers can work with strings, arrays and associative arrays without having to worry about manual memory management. Perl is well-suited to complex text file manipulation, such as report generation, file format conversion, and web server CGI scripts. It is also useful for rapid prototyping, but large perl scripts are often unmaintainable. Perl's *memory management* is *well-hidden*, but is based of *reference count* and *garbage collection*. It also has mortal variables, whose *lifetimes* are limited to the current context. It is possible to free the memory assigned to variables (including arrays) explicitly, by undefining the only reference to them.

9.8.14 Prolog Memory Management

Prolog is logic programming language invented by Alain Colmerauer around 1970. Prolog is popular in the AI and symbolic computation community. It is special because it deals directly with relationships and inference rather than functions or commands.

Storage is usually managed using a *garbage collector*, but the complex control flow places special requirements on the collector.

9.8.15 Scheme Memory Management

Scheme is a small functional language blending influences from *LISP* and *ALGOL*. Key features of Scheme include symbol and list operations, *heap allocation* and *garbage collection*, lexical scoping with first class function objects (implying closures), reliable fail-call elimination (allowing iterative procedures to be described fail-recursively), the ability to dynamically obtain the current *continuation* as a first-class object, and a language description that includes a formal semantics.

9.8.16 Simula Memory Management

Simula was designed as a language for simulation, but it expanded into a full general-purpose programming language and the first object-oriented language.

Simul I, designed in 1962 - 67 by Kristen Nygaard and Ole-Johan Dahl, was based on *ALGOL 60*, but the *stack allocation* discipline was replaced by a two-dimensional *free-list*.

It was Simula 67 that pioneered classes and inheritance to express behaviour. The domain-oriented design was supported by garbage collection.

9.8.17 Smalltalk Memory Management

Smalltalk is an object-oriented language with single inheritance and message-passing.

Automatic memory management is an essential part of the smalltalk philosophy.

EXERCISE

1. Differentiate between memory allocation technique of C++ and Java.
2. Discuss the following storage-allocation strategies
 - (i) Stack allocation
 - (ii) Heap allocation
3. If a language has features in it which require heap allocation, then perhaps there is no need for a stack at all. Is a stack truly necessary? If not, provide an implementation of run time storage allocation which uses only a heap.
4. What are various phases in storage management? Describe.
5. Striking feature of garbage collection is that its cost is inversely proportional to storage recovered. Why is so? Explain.
6. If a language has features in it which require heap allocation, then perhaps there is no need for a stack at all. Is a stack truly necessary? If not, provide an implementation of run time storage allocation which uses only a heap.
7. What is heap? Explain how heap can be used to implement storage management. In this context explain the most suitable data structure to implement heap such that garbage collection and removal of dangling references is carried out efficiently.
8. An alternative garbage-collection strategy is to have two storage areas. During the first-pass marking phase, copy each referenced object to the head of the other storage area. In that way, once all active nodes are marked, garbage collection is complete. Now allocate from the second storage area. When this fills, compact the storage back in to the first area. Compare this algorithm to the two-pass mark-sweep algorithm.
9. It was discussed in the chapter that "reference counts are the preferred in parallel processing applications, for mark-sweep algorithm be ineffective in that environment."
10. Discuss the advantages and disadvantages of:
 - (i) Explicit memory Allocation
 - (ii) Implicit memory Allocation
11. What are the memory leaks? Discuss the memory leaks in C++.
12. "Automatic garbage collection is found less suitable for the real time system", justify this statement.
13. "Reference counting technique does not work on cyclic data", Explain this statement.
14. How allocation differs in case of Fixed and variable-size elements?

15. Discuss the memory management in following languages:

- (i) ALGOL
- (ii) Pascal
- (iii) BASIC
- (iv) C
- (v) COBOL
- (vi) C++
- (vii) Fortran
- (viii) Java
- (ix) LISP
- (x) ML

CHAPTER 10

Object Oriented Languages

10.1 INTRODUCTION

In general we can have following type of programming methodologies:

- (i) Unstructured programming
- (ii) Procedural programming
- (iii) Modular programming and
- (iv) Object-oriented programming.

10.1.1 Unstructured Programming

Usually people start learning programming by writing small and simple programs consisting only of one main program. Here "main program" stand for a sequence of commands or statements which modify data which is global throughout the whole program. It can be seen by following figure.

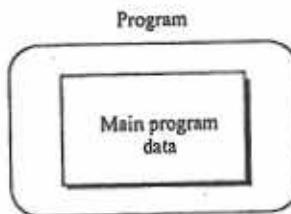


Fig. 10.1 Unstructured programming

We know that, this programming techniques provide tremendous disadvantages once the program gets sufficiently large.

10.1.2 Procedural Programming

With procedural programming we are able to combine returning sequences of statements into one single place. A procedure call is used to invoke the procedure. After the sequence is processed, flow of control proceeds right after the position where the call was made (Fig. 10.2).

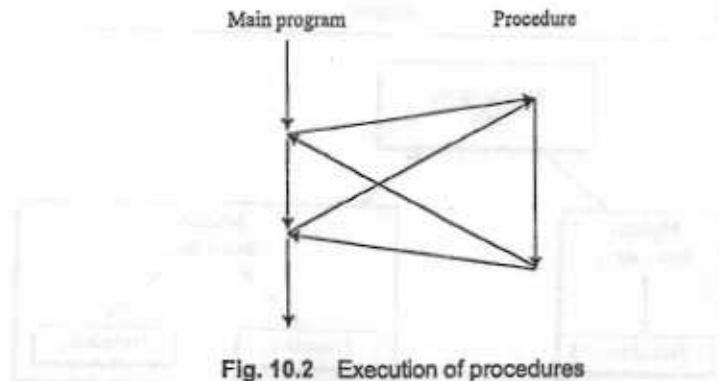


Fig. 10.2 Execution of procedures

It is clear from the above figure that, after processing flow of controls proceed where the call was made. The main program is responsible to pass data to the individual calls, the data is processed by the procedures and, once the program has finished, the resulting data is presented. Thus, the flow of data can be illustrated as a hierarchical graph, a tree, as shown in Fig. 10.3.

The main program coordinates calls to procedures and hands are appropriate data as parameters.

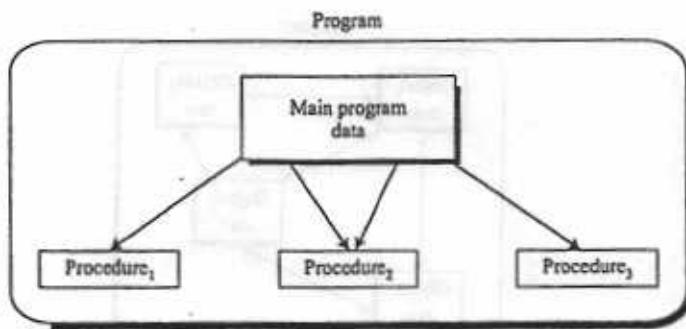


Fig. 10.3 Procedural programming

10.1.3 Modular Programming

With modular programming procedures of a common functionality are grouped together in to separate *modules*. A program therefore no longer consists of only one single part. It is now divided into several smaller parts which interact through procedure calls and which form the whole program (Fig. 10.4).

In modular programming the main program coordinates calls to procedures in separate modules and hands over appropriate data as parameters. Each module can have its own data. This allows each module to manage an internal *state* which is modified by calls to procedures of this module. However, there is only one state per module and each module exists at most once in the whole program.

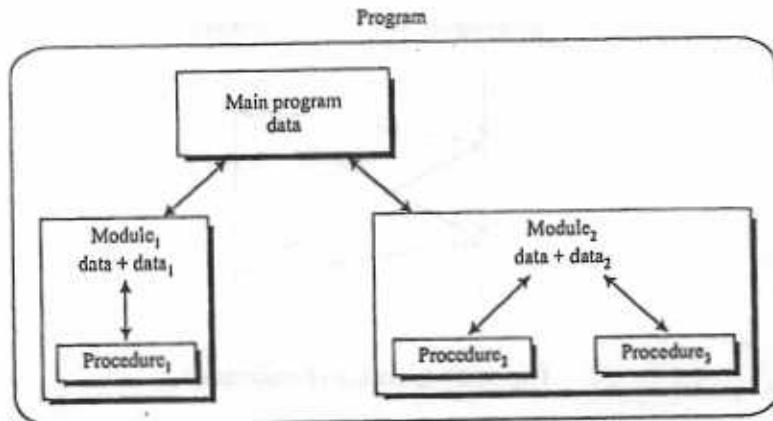


Fig. 10.4 Modular programming

10.1.4 Object-Oriented Programming

Object-oriented programming solves the problem of previous programming methodologies. In contrast to the other techniques, we now have a web of interacting objects, each house-keeping its own state as shown in Fig. 10.5.

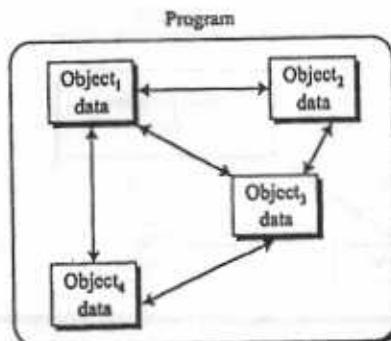


Fig. 10.5 Object-oriented programming

Object of the program interact by sending messages to each other. The problem with modular programming is, that we must explicitly create and destroy our memory. In contrast to that, in object-oriented programming we would have as many objects as needed. Each object is responsible to initialize and destroy itself correctly. Consequently. There is no longer the need to explicitly call a creation or termination procedure. Object is one feature of object-oriented programming, but there are some other features which makes object-oriented programming to a new programming technique.

10.2 OBJECT-ORIENTED LANGUAGES

The programming language which follows the concept of object-oriented programming are said to be object-oriented programming. A language is object-oriented if:

1. It supports objects that are data abstractions.
2. All objects have an associated object type (often called classes).
3. Classes may inherit attributes from superclasses.
4. Computation proceed by sending messages to objects
5. Routines may be applied to object which are variants of those they are designed to be applied (subtype polymorphism).
6. Supports dynamic method invocation.

All the object-oriented languages built around the concepts of object message, class, Instance, method, subtype (polymorphism) and subclass (inheritance).

Simula 67 is first object-oriented language, which was designed for discrete simulations. C++ is object-oriented extension to pascal, C, LISP etc.

Use of the nicest is Eiffel and sather. Java is now becoming the most popular object-oriented languages.

10.3 CONCEPT OF ABSTRACTION

Some times we describe object-oriented programming *abstract data types* and their relationship. Let us consider an example to understand the concept of abstraction. Suppose we are facing a problem in our "real-life" and we want to solve that problem through the software. So very first task is to understand the problem to separate necessary from unnecessary detail, that we have to try to obtain our own abstract view, or model of the problem. This process of modeling is called abstraction as shown in Fig. 10.6.

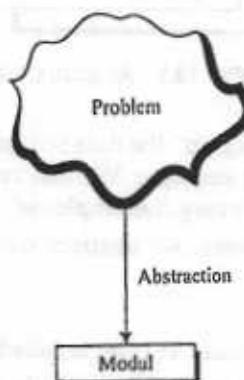


Fig.10.6 Creation of module from a problem with abstraction

The model defines an abstract view to the problem. This implies that the model focuses only on problem related stuff and that we try to define properties of the problem. These properties include

1. The data which are affected and
2. The operations which are identified

Finally we can say that:

"Abstraction is the structuring of a nebulous problem in to well-defined entities by defining their data and operations. Consequently, these entities combine data and operations. They are not decoupled from each other."

10.3.1 Properties of Abstract Data Types

With the abstraction we created well-defined entity which can be properly handled. These entities define the data structure of a set of items. For example, each administered employee has a name, date of birth and social number.

The data structure can only be accessed with defined operations. This set of operations is called interface and is exported by the entity. An entity with the properties just described is called as abstract data type (ADT).

Following figure show an ADT which consists of an abstract data structure and operations. Only the operations are viewable form the outside and define the interface.

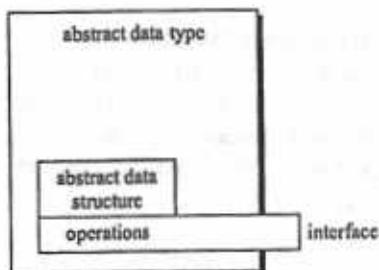


Fig. 10.7 An abstract data type

Once a new employee is “created” the data structure is filled with actual values: We now have an instance of an *abstract employee*. We can create as many instances of an abstract employee as needed to describe every real employed person.

Definition (Abstract Data Type): An abstract data type (ADT) is characterized by the following properties:

1. It exports a *type*.
2. It exports a set of operations. This set is called *interface*.
3. Operations of the interface are the one and only access mechanism to the type's data structure.

Each ADT description consists of two parts:

- *Data*: This part describes the structure of the data used in the ADT in an informal way.

- **Operations:** This part describes valid operations for this ADT, hence, it describes its interface. We use special operation *constructor* to describe the actions which are to be performed once an entity of this ADT is created and *destructor* to describe the actions which are to be performed once an entity destroyed.

10.4 CONCEPT OF OBJECT-ORIENTED PROGRAMMING

Object-oriented language built around following concepts

10.4.1 Class

A *class* is the implementation of an abstract data type (ADT). It defines *attributes* and *methods* which implement the data structure and operations of the ADT, respectively. Instances of classes are called *objects*. Consequently, classes define properties and behaviour of set of objects.

Let us see following example to understand the concept of class:

In real world there may be thousands of other bicycles in existence, all of the same make and model. Each bicycle was built from the same set of blueprints and therefore contains the same components. In object-oriented terms, we say that our bicycle is an instance of the class of objects known as bicycles. A *class* is the blueprint from which individual objects are created.

The following Bicycle class is one possible implementation of bicycle in Java language:

```
Class Bicycle {
    int speed = 0;
    int gear = 1;
    void changeGear (int newValue) {
        gear = newValue;
    }
    void speedUp (int increment) {
        speed = speed + increment;
    }
    void applyBrakes (int decrement) {
        speed = speed - decrement;
    }
}
```

In above Java code segment fields speed and gear represent the object's state, and the methods define interaction with outside world.

10.4.2 Object

Objects are key to understanding object-oriented technology.

"An object is an instance of a class. It can be uniquely identified by its *name* and it defines a *state* which is represented by the values of its attributes at particular time."

The *behaviour* of an object is defined by the set of methods which can be applied on it.

Real-world objects also share two characteristics: They all have state and behaviour. Bicycles have state (current gear, current pedal cadence, current speed) and behaviour (changing gear, changing pedal cadence, applying breaks). Identifying the state and behaviour for real-world objects is a great way to begin thinking in terms of object-oriented programming.

Software objects are conceptually similar to real-world objects: they too consist of state and related behaviour. An object stores its state in fields (variable in some programming languages) and exposes its behaviour through methods (functions in some programming languages). Methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication.

Hiding internal state and requiring all interaction to be performed through an object's method is known as data encapsulation—a fundamental principle of object-oriented programming.

Bundling code into individual software objects provides a number of benefits, including:

1. *Modularity*: The source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system.
2. *Information-hiding*: By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.
3. *Code re-use*: If an object already exists (perhaps written by another software developer), you can use that object in our program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.
4. *Pluggability and debugging ease*: In a particular object turns out to be problematic, you can simply remove it from our application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a bolt break, we replace it, not the entire machine.

10.4.3 Message

A running program is a pool of objects where objects are created, destroyed and interacting. This interacting is based on messages which are sent from one object to another asking the recipient to apply a method on itself.

Sending a message asking an object to apply a method is similar to a procedure call in traditional programming languages. However, in object-orientation there is a view of autonomous objects which communicate with each other by exchanging messages. Objects react when they receive messages by applying methods on themselves. They also may deny the execution of a method, for example if the calling object is not allowed to execute the requested method.

"A message is a request to an object to invoke one of its methods."

A message therefore contains

- The *name* of the method and
- The *arguments* of the method.

Consequently, invocation of a method is just a reaction caused by receipt of a message. This is only possible, if the method is actually known to the object.

A *method* is associated with a class. An object invokes a method as a reaction to receipt of a message.

10.4.4 Encapsulation

"In object-oriented programming, the technique of keeping together *data structure* and the *methods (procedures)* which act on them, is known as **encapsulation**."

Well encapsulated objects act as a "black box" for other parts of the program which interact with it. They provide a service, but the calling object do not need to know the details how the service is accomplished.

Encapsulation is critical to building large complex software which can be maintained and extended. Many studies have shown that the greatest cost in software is not the initial development, but the thousand of hours spent in maintaining the software. Well encapsulated components are far easier to maintain.

Once software is in place, another great expense is extending its functionality. As you add new features, there is risk that you will break existing parts of the application. Again, encapsulation helps to minimize the risk.

In a well designed program, each object should have a single area of responsibility. *That object presents an interface which defines the services the object provides.*

Note that a single object can offer more than one interface C++ does not support the concept of multiple interfaces (though Java does).

So, encapsulation hides the implementation details of the object and the only thing that remains extremely visible is the interface of the object (that is the set of all messages the object can respond to).

10.4.5 Implementation

Implementation is the set of class methods which provide the service promised by the interface. In a well designed object, the implementation is entirely hidden from your clients.

"Implementation is how we perform the service in our interface".

This is the essence of *information hiding*. This is the principle that your class should not expose its data, but rather should offer an interface allowing clients to access the data indirectly. This allows us to modify how we represent that data in the class without breaking the interface. Doing so decouples the implementation of our class from its clients, which helps with making our code reusable.

10.4.6 Polymorphism

Polymorphism allows an entity (for example, variable, function or object) to take a variety of representations. There we have to distinguish different types of polymorphism which will be outlined here:

The *first type* is similar to the concept of dynamic binding. Here the type of a variable depends on its content. Thus, its type depends on the content at a specific time:

```
v: = 123      /* v is integer */
.....        /* use v as integer */
```

```
v: = 'abc' /* v "switches" to string */
..... /* use v as string */
```

Definition (polymorphism (1)) The concept of dynamic binding allows a variable to take different types dependent on the content at a particular time. This ability of a variable is called polymorphism.

Another type of polymorphism can be defined for functions. For example, suppose we want to define a function `is null()` which returns TRUE if its argument is 0 (zero) and FALSE otherwise. For integer number this is easy:

```
boolean is NULL (int i) {
    if (i == 0) then
        return TRUE
    else
        return FALSE
    end if
}
```

10.4.7 Inheritance

Different kind of objects often have a certain amount in common with each other. Mountain bikes, road bikes, and tandem bikes, for example, all share the characteristics, of bicycles (current speed, current pedal cadence, current gear). Yet each also defines additional features that make them different: tandem bicycles have two seats and two sets of handlebars; road bikes have drop handlebars; some mountain bikes have an additional chain ring, giving them a lower gear ratio.

Object-oriented programming allow classes to inherit commonly used state and behaviour from other classes.

"Inheritance is the mechanism which allow a class A to inherit properties of a class B. We say "*A inherits from B*". Object of class A thus have access to attributes and methods of class B without the need to redefine them."

If class A inherits from class B, then B is called *superclass* A. A is called *subclass* of B. Objects of a subclass can be used where objects of the corresponding superclass are expected. This is due to the fact that objects of the subclass share the same behaviour as objects of the superclass. Superclasses are also called *parent classes*. Subclasses may also be called *child classes* or just *derived classes*.

10.4.7.1 Multiple Inheritance

One important object-oriented mechanism is multiple inheritance. Multiple inheritance does not mean that multiple subclasses share the same superclass. It also does not mean that a subclass can inherit from a class which itself is a subclass of another class.

Multiple inheritance means that one subclass can have more than one superclass. This enables the subclass to inherit properties of more than one superclass and to "merge" their properties.

"If class A inherits from more than one class, that is A inherits from B₁, B₂, ..., B_n we speak of *multiple inheritance*. This may introduce *naming conflicts* in A if at least two of its superclasses define properties with the same name".

The above definition introduce naming conflicts which occur if more than one superclass of a subclass use the same name for either attributes or methods.

These conflicts can be solved in at least two ways:

- (i) The order in which the superclasses are provided define which property will be accessible by the conflict causing name. Other will be "*hidden*".
- (ii) The subclass must resolve the conflict by providing a property with the name and by defining how to use the ones from its superclasses.

The first solution is not very convenient as it introduce implicit consequences depending on the order in which classes inherit from each other. For the second case, subclasses must explicitly redefine properties which are involved in a naming conflict. Another kind of naming conflict introduced by a shared superclass of superclasses used with multiple inheritance. Let us see the following figure:

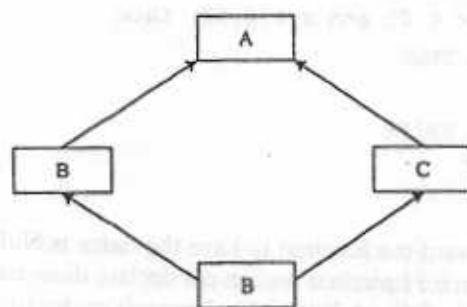


Fig. 10.8 A name conflict

Here a special type of naming conflicts is introduced if a class D multiply inherits from superclasses B and C which themselves are derived from one superclass A. This leads to an inheritance graph as shown in Fig. 10.8

The question arises what properties class D actually inherits from its superclasses B and C. Some existing programming languages solve this special inheritance graph by deriving D with

- The properties of A plus
- The properties of B and C without the properties they have inherited from A.

Consequently, D cannot introduce naming conflicts with names of class A. However, if B and C add properties with the same name, D runs into a naming conflict.

Another possible solution is, that D inherits from both inheritance paths. In this solution, D owns two copies of the properties of A: one is inherited by B and one by C.

10.4.8 Abstract Classes

A class 'A' is called *abstract class* if it is only used as a superclass for other classes. Class A only specifies the properties. It is not used to create objects. Derived classes must define the properties of A.

Let us see an example, in which we are forcing a draw able object to include such method (print ()), we define a class Drawable object from which every other class in our example inherits general properties of drawable objects:

```
abstract class DrawableObject {
    attributes:
    methods:
        print ( )
}
```

We introduce the new keyword *abstract* here. It is used to express the fact that derived classes must "redefine" the properties to fulfill the desired functionality. Thus from the abstract class point of view, the properties are only specified but not fully defined. The full definition including the semantics of the properties must be provided by derived classes.

However, if we want to check this for real numbers, we should use another comparison due to precision problem:

```
boolean is NULL (real r) {
    if (r < 0.01 and r > 0.99) then
        return TRUE
    else
        return FALSE
    end if
}
```

In both cases we want the function to have the name is Null. In programming languages without polymorphism for functions we can not declare these two functions because the name in Null would be doubly defined. Without polymorphism for functions, doubly defined names would be ambiguous. However, if the language would take the parameters of the function into account it would work. Thus, function (or methods) are uniquely identified by:

- The name of the function (or method) and
- The types of its parameter list.

Since the parameter list of both in Null functions differ, the compiler is able to figure out the correct function call by using the actual types of the arguments:

```
var i : integer
var r : real
i = 0
r = 0 - 0
.....
if (is Null (i)) then .../* Use is Null (int) */
.....
if (is Null (r)) then .../* Use is Null (real) */
```

Definition (polymorphism (2)): If a function (or method) is defined by the combination of

- its name and
- the list of types of its parameters

We speak of *polymorphism*. This type of polymorphism allows us to reuse the same name for function (or methods) as long as the parameter list differs. Sometimes this type of polymorphism is called overloading.

The last type of polymorphism allows an object to choose

Definition (polymorphism (3)): Objects of superclasses can be filled with objects of their subclasses. Operators and methods of subclasses can be defined to be evaluated in two contexts:

1. Based on object type, leading to an evaluation within the scope of the superclass.
2. Based on object content, leading to an evaluation within the scope of the contained subclass.

The second type is called *polymorphism*.

Let us see some of the object-oriented languages

10.5 C++

"C with classes" was released in 1980 as an enhanced version of C (implemented using C compilers with a preprocessor) which included classes for data abstraction. C with classes was designed so that a preprocessor could make direct conversions from classes to structs, by making the argument member functions global, renaming them to include the class name and modifying the argument list to include a struct equivalent of the method's class as the first argument.

In 1983 the first version of C++ was released and more advanced object oriented features were rapidly until 1995 when the first commercial version was released.

10.5.1 Features and Design

C++ is a class based language, designed to allow the programmer very low level control over object structure and access. Object oriented features of C++ include: Virtual classes, public/private/protected access control over individual member functions and attributes, friend classes, nested classes, multiple inheritance with method redefinition, and templated (generic) classes and functions.

Other general features include: user controlled memory (the heap), direct memory references, static type checking, method overloading, exceptions, threads, and explicitly constructed namespaces.

10.5.2 Inheritance in C++

Lets us write a *building* class; it will serve as the base for two derived classes.

```
class building {
    int rooms;
    int floors;
    int area;
```

```

public:
    void set_rooms (int num);
    int get_rooms( );
    void set_floors (int num);
    int get_floors( );
}
// Flat is derived from building class flat: public building{
class flat : public building {
    int bedroom;
    int baths;
public:
    void set_bedrooms (int num);
    int get_bedrooms ( );
}
In general the syntax is:
class derived_class : access base_class
{
// body of new class
}.

```

10.5.3 Constructor and Destructors in C++

"A constructor is a special function that is a member of a class and has the same name as that class."

In general, a prior to the execution of the particular constructor body, constructors of every superclasses are called to initialize their part of the created object.

For example:

```

class stack{
    int stck [SIZE];
    int top;
public:
    stack( ); // constructor
    void push (int i);
    int pop ( )
};

```

If an object is destroyed, for example by leaving its definition scope, the *destructor* of the corresponding class invoked. If this class is derived from other classes their destructor are also called, leading to a recursive call chain.

In above class we can write destructor as:

```
stack ( ); // destructor
```

Parameterized constructors: It is possible to pass arguments to constructor functions. These arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way we do to any other function.

For example:

```
class demo {
    int a, b;
public:
    demo (int i, int j) // parameterized constructor
    {
        a = i;
        b = j; }
```

10.5.4 Multiple Inheritance

C++ allows a class to be derived from more than one superclass, as was already briefly mentioned in previous sections. We can easily derive from more than one class by specifying the superclasses in a comma separated list:

```
class X: public A, public B{
public:
    X();
    X(); }
```

Here X derive from class A and B.

10.5.5 Polymorphism in C++

A *virtual function* is a member function that is declared within a base class and redefined by a derived class. To create a virtual function, we must use keyword *virtual*.

```
class base {
public:
    virtual void vfun () ; // virtual function.
}
class sub1: public base {
public:
    void vfun () {
        cout << "This is sub1's vfun \n";
    }
}
class sub2: public base {
    void vfun () {
        cout << "This is sub2's vfun \n";
    }
}
```

Here function with same name *v fun()* is used in two ways so it is *polymorphism* in C++.

10.5.6 Abstract Class in C++

Abstract classes are defined just as ordinary classes. However, some of their methods are designated to be necessarily defined by subclasses. We just mention their signature including their return type, name and parameters but not a definition. One could say, we omit the method body or, in other words, specify "nothing". This is expressed by appending "= 0" after the method signature:

```
class abc {
    .....
public:
    .....
    virtual void print () = 0;
```

This class definition would force every derived class from which objects should be created to define a method `print()`. These method declaration are also called *pure methods*.

Pure methods must also be declared virtual, because we only want to use objects from derived classes. Classes which define pure methods are called *abstract classes*.

10.6 SIMULA

Simula language is developed by Kristen Nygaard and Ole-johan Dahl at the Norwegian Computing Center. Nygaard and Dahl proposed their idea for a simulation language in 1963, and although their ideas did not receive much enthusiasm, political issues at the NCC resulted in a contract between the NCC and UNIVAC to provide a simula implementation and compiler by 1965-NCC did so, and the result is known as "simula-I". Refinement were made to simula I and in 1967 "simula-67" was released. The most recent standard is simula-87.

10.6.1 Features and Design

Simula started out as an activity/process based programming language, in which different types (and behaviour) of activities are declared, and then multiple processes can be created to carry out different activities. The key power in this original design, was that in addition to having lists of actions to be performed, processes were also data structures, and activities has associated methods. These activities and processes had so much use besides just that of simulation that simula-67 was released, they had been renamed "classes" and "objects" (thus the birth of "object oriented" programming).

Its interesting to note that as the further of object oriented languages, simula-67 does not support dynamic dispatch something most people consider necessary for "true" object oriented programming. Instead, an object must be down cast (error checked at run time) and the appropriate attribute/method can then be accessed.

10.7 SMALL TALK

In 1961, Alan Kay was a programmer for the airforce and developed the language. Smalltalk was redefined from scratch in 1972 on a bet, (that Kay could define the "most powerful

language in the world" in a page of code") and smalltalk-72 has since been considered the first "real small talk". Smalltalk was redesigned again in 1976 by Dan Ingalls, and then again in 1980 just before it was released to the public.

10.7.1 Features and Design

Smalltalk is an untyped, class based language. To better model the difference between performing action on types of objects, as a distinct object, Smalltalk classifies attributes and methods as belonging to either the class or to instances. Class methods are located in the class meta-object and can only refer to class attributes (also located in the meta-object to provide shared state for all instances). Instance methods are kept locally to each object and can refer to both the class variables and the instance variables (which provide local state). All methods are public, while all attributes are private. Single inheritance is provided, along with (partially) abstract classes, and method overriding (including signature modification). Although Smalltalk is untyped, the main purpose of inheritance is not just code sharing. The underlying principle is similar to subtyping in that programmers should use inheritance to provide specialization of objects, and can subsequently feel secure that it will be safe to use an instance of a subclass as if it were an instance of the superclass-without having to worry about run time error.

10.8 MODULE-3

Modula is designed by Luca Cardelli, Jim Donahue, Mick Jordahl, Bill Kalsow, Greg Nelson. The specification was written by Lucille Glassman and Greg Nelson. The initial language definition was published in August of 1988, and then revised (based on the recommendations of implementors) in January of 1989.

10.8.1 Features and Design

Modula-3 is a class based language in which class names act as type names-there are no explicit type definition or declarations.

Class definitions are "partial opaque"- meaning that methods and attributes may or may not be visible to other classes. Single inheritance is provided as a means of type specialization, (allowing sub-classes to override the methods of their super-class) which also provides code reuse, and Abstract classes are allowed as a means to specify types (which contain only method declarations and signatures) without implementations. In addition to inheritance as a method of class specification, Modula-3 provides Generic modules which are templates parameterized by types. Generics are not polymorphic, and provide only source code reuse (not target code reuse).

Modula-3 is strongly typed, with no automatic conversion, or type inference. In addition, Modula-3 defines typed equality based on type/class structure, and not type/class name.

10.9 EIFFEL

The OO aspects of Eiffel were directly influenced by Simula 67, while the "Design by contract" aspects of the language were heavily influenced by Meyer's earlier academic work in software

verification. The initial ideas for Eiffel were conceived in September of 1985, and subsequently released to the public (as ISE Eiffel 1) in October of 1986. Eiffel is still evolving and the latest version is ISE Eiffel 4.

10.9.1 Features and Design

Eiffel is a class based language, in which the definition of "Type" and "class" are identical. Type equivalence is based on class name equivalence.

Classes may contain (multiple) Features clauses which can in turn contain multiple attributes/values and Routines/procedures. The classification of a given feature (Routine or Attribute) is known to other classes that is an Attribute of type T has the same "appearance" as a Routine which takes no argument and returns an item of type T.

Eiffel supports multiple inheritance (including code reuse) with compiler enforced renaming of conflicting features. In addition, Eiffel allows the programmer to not only Redefine (or Undefine) the implementation of particular features, but also modify the client list of inherited features.

10.10 JAVA

The premise for Java arose from James Gosling in 1991, because of the frustrations he had using C++ to program embedded systems software for "smart" electronics devices at Sun Microsystems. Gosling began developing the language "Oak" to be safe, object oriented systems language. By 1993 Oak had been renamed "Java" and several prototype electronic devices that had been programmed with Java were available—but the market didn't see interest. Around the time, the www was drastically increasing in use, and sun began to see uses for Java's small, safe, platform independent byte code in the online community. During 1994 the language was refined and eventually released in as version 1.0 in 1995. An update (1.1) was released in 1996.

10.10.1 Features and Design

Java is a class based language, that was originally designed for programming embedded systems. Because of this, the ideas of speed, platform independence, and run time safety are crucial in its design. As the motivation behind the language shifted to the www, the issues of speed, platform independence and safety remained, but the idea of distributed programs became extremely important. One of the main features Java provides is the remote method invocation (RMI) system, which allows semi-transparent method invocation and exchange of objects between virtual machines (even across the network).

In addition to RMI, other general programming features supported are exceptions, garbage collection (which is not only supported—but considered crucial), byte code verification (which validates the safety of a given program), threads, method overloading, and package (for creating name spaces).

As an object oriented language, Java supports multiple levels of implementation hiding, partially abstract classes, final classes (which can not be inherited from) and static (class) variable. Single inheritance of classes is provided for subtypeing and code sharing, in addition to multiple inheritance of "Interfaces"—which act as Type declarations, or completely abstract classes. As of version 1.1, Java now also supports nested and anonymous classes.

EXERCISE

1. What are various access specifiers used in object-oriented programming languages? Discuss the main role of constructor and destructors.
2. Using C++, write a program segment that uses on stack to evaluate postfix expressions.
3. How object oriented languages support information hiding and data abstraction. Explain in detail giving suitable example.
4. Write short notes on:
 - (i) polymorphism
 - (ii) Inheritance
 - (iii) Operator overloading
5. Compare the object description of C++ and Java.
6. How are Java objects be allocated?
7. How are Java objects deallocated?
8. What are the various types programming methodologies?
9. What is data abstraction concept? also discuss the properties of abstract data types.
10. What are the abstract classes?
11. What are uses of constructor and deconstructor in C++?
12. Write short notes on following object oriented languages.
 - (i) Smalltalk
 - (ii) Modula -3
 - (iii) Eiffel
 - (iv) Java.

Functional Programming Languages

11.1 INTRODUCTION TO DECLARATIVE LANGUAGES

In earlier chapters we have seen imperative languages in detail, now we are able to think about a very high level language which will be looking at here are based far more closely on expression, and come under the various headings of applicative/ declarative/ descriptive language. The design of these language is not so much influenced by particular machine details but rather by a clear mathematical understanding of descriptions. A programming language which supports the high-level view of programming rather than the machine controlling view, must have following:

1. The language should be expressive so that descriptions of problems, situations, methods and solutions are not too difficult to write and understand.
2. The language should also protect users from making too many errors as far as possible.
3. It should be possible to write programs with the language which run efficiently on currently available machines.
4. Major programming activities must be supported by mathematical principles.

There are two specific classes of declarative language in current use namely *functional languages* and *logic programming languages*. Here in this chapter we will concentrate on functional languages only.

11.2 FUNCTIONAL PROGRAMMING

The functional programming paradigm is one of the major paradigm, emerged in the early 1960s. Its creation was motivated by needs of researchers in artificial intelligence and its sub fields like symbolic computation, theorem proving, rule-based system, and natural language processing.

In functional programming paradigm concentration is on efficiency, rather than the suitability of the language for software development. The design of the functional languages is based on mathematical functions. This paradigm is based on sound theoretical frame works (that is lambda calculus), which is closer to the user, but relatively unconcerned with the architecture of the machines or which programs will run

When using functional languages we do away with notions such as variables and reassignment. This allows us to define programs which may be subjected to analysis much more easily. When a value is assigned it does not change during the execution of the program. This property is *referential transparency*. There is no state corresponding to the global variables of a traditional language or the instances of objects in an object oriented language. This paradigm is also known as applicative programming and value-oriented programming. The important language of this paradigm are Lisp, ML, Haskell, and hope

11.2.1 Characteristics of Pure FPLS

If the design of a functional language completely based on the mathematical function then functional language is called pure functional programming language and have following characteristics:

- Have no side-effects
- Have no assignment statement
- Often have no variables
- Built on a small, concise frame work
- Have a simple, uniform syntax
- Be implemented via interpreters rather than compilers
- Be mathematically easier to handle

11.2.3 Importance of FP

The importance of functional paradigm can be understood by the following points:

1. *In their pure form FPLs dispense with notion of assignment.* It is easier to write programs by using FPLS.
2. *FPLs encourage thinking at higher level of abstraction.* FPLs supports modifying and combining existing programs, thus, FPLs encourage programmers to work in units larger than statements of conventional languages: "Programming in the large".
3. *FPLs Provide a paradigm for parallel computing.* Absence of assignment (or single statement), independence of evaluation order and ability to operate on entire data structures provides basis for parallel functional programming.
4. *FPLs are valuable in developing executable specifications and proto type implementation.* FPLs have rigorous mathematical foundations, ability to operate on entire data structures. They are the ideal vehicle for capturing specifications.
5. *FPLs are very useful for AI and other applications which require extensive symbol manipulation.*
6. Functional language such as Lisp, ML, and Hope allow us to develop programs which will submit logical analysis relatively easily.
7. A functional language like ML offers all of the features that we have come to expect from a modern programming language. Objects may be packaged with details hidden.
8. Input and output tend to be rather more primitive then we might expect, however there are packages which allow ML to interface ends such as x-windows.

11.2.4 Comparison of Imperative and Functional Paradigms

We can compare formal methods and functional programming with same traditional imperative programming and traditional software engineering.

Comparison Parameters	Imperative programming and traditional software Engineering	Functional programming and formal methods
The Development Cycle	Using in formal language a specification may be open to interpretation. Using appropriate testing strategies we can improve confidence but not in any measurable way.	Using logic we can state the specification exactly. Using mathematics we may be able to prove useful properties of our programs.
The Development Language	Using structured programming or object oriented techniques we can reuse code. Using structured programming or object orientation we can partition the problem in to more manageable chunks.	Using structured programming or object oriented technique the problem into easy to use chunks plus there are often "higher-level" abstraction, which would be difficult or impossible in a traditional language.
The Run-time system	The compiler can produce fast compact code taking a fixed amount of memory. Parallel processing is not possible (in general) Fancy GUI's may be added.	The memory requirements are large and unpredictable. Parallel processing is possible. Fancy GUI's may be added, with difficulty.

11.3 MATHEMATICAL FUNCTIONS

In mathematics, the concept of function has become fundamental. In formally, a function is seen as a correspondence between argument values and result values.

In another way, a mathematical is a *mapping* of member of one set, called the *domain set*, to another set, called range set. Function definition are often written as a function name, followed by a list of parameters in parentheses, followed by the mapping expression. For example

$$\text{Squre}(x) = x \times x \text{ where } x \text{ is a real number}$$

In above function definition both domain and range sets are the real numbers. The parameter, x , can represent any member of the domain set, but it is fixed to represent one specific element during evaluation of the function expression. This is how the parameters of mathematical functions differ from the variables in imperative languages.

A function is called *total* if it associated exactly one element in the target with each element in its source (domain). A *partial* function may or may not associate an element in the target with each element in the source and is said to be undefined for these arguments which it does not map to a result. Usually, when mathematicians speak about functions they mean total function, but here we will use function to mean partial function, which includes the possibility of being total.

11.4 REFERENTIAL TRANSPARENT

The essence of functional programming is that all values produced in sub calculation are communicated to other parts of a program only by the use of function application (e.i. as

arguments and result of functions). This is in contrast to the communication mechanism used in imperative languages where by global storage variables are updated and parts of programs make use of the storage location of such values to share the effect of an update.

The phrase *referential transparent* is used to describe notations where only the meaning (value) of immediate component expression is significant in determining the meaning of a compound expression (sentence/ phrase). If expressions are equal of and only if they have the same meaning, referential transparency means that substitutivity of equality holds (i.e. equal sub expressions can be interchanged in the context of a large expression to give equal results).

The concept will be more clear by following example: let us consider a function either even which is defined as a composition of the function odd and the Boolean disjunction (or), such that for any integers x and y :

$$\begin{aligned}\text{eithereven}(x, y) &= (\text{even } x) \text{ or } (\text{even } y) \\ &= \text{even}(x \times y) \\ &= \text{evenprod}(x, y)\end{aligned}$$

Here eithereven and evenprod describe the same function since they have the same external behaviour above concept can be viewed by following figure:

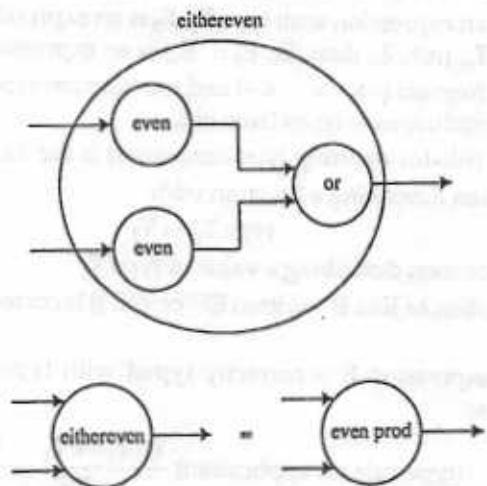


Fig. 11.1 The function eithereven = evenprod

We write $\text{eithereven} = \text{evenprod}$ to indicate that they describe the same value (function) even though the descriptions are quite different.

11.5 TYPE SYSTEM OF FUNCTION PROGRAMMING LANGUAGES

"The classification technique of data values according to their use is known as typing of data".

In a typed programming language, each value that deals with in a program has an associated type which determines how the value can be used. A programming language which performs type checking and thus ensures that type errors do not lead to erroneous computations is said to be *strongly typed*.

If the types of program components can be determined before evaluation, the time and space penalty introduced by type checking during evaluation can be removed. This is known as *static type checking* and it also enables many programming mistakes to be caught much earlier during program development. Many modern programming languages are strongly typed with static type checking.

In 1978 Milner has shown an extremely general type system, can be provided for functional languages without the programmer having to specify the types of newly defined values. Milner's polymorphic type system has been extended and used in several recent programming languages (including standard ML). Like other paradigm, functional programming languages also support many basic data types. Further types can be built from these primitive types using the following type operators:

If T_1 and T_2 are types, then $(T_1 \rightarrow T_2)$ is the type of function whose source consists of values of type T_1 and whose target consists of values of type T_2 . So, for example, even would have type $\text{int} \rightarrow \text{bool}$. A type will have a (Cartesian) product type formed from the type of the components of the type. For example, the pair $(3, \text{true})$ has type $\text{int} \times \text{bool}$ since 3 has type int and true has type bool . Similarly $((3, \text{"ab"}), \text{false}, 6)$ has type $(\text{int} \times \text{string}) \times \text{bool} \times \text{int}$ since it is a triple whose last two components are an integer and Boolean, respectively, and whose first component is a pair consisting of an integer and a string.

In general, if E_1 is an expression with type T_1 , E_2 is an expression of type T_2 , ... and E_n is an expression with type T_n ($n > 2$), then (E_1, E_2, \dots, E_n) is an expression of type $T_1 \times T_2 \times \dots \times T_n$.

The product type formers ($- \times - \times \dots \times -$) and the function type formers (\rightarrow) are called type operators since they produce new types from old.

The fundamental rule for ensuring type correctness is the following:

IF E is an expression describing a function with

type $T_1 \rightarrow T_2$

AND E' is an expression describing a value of type T_1

THEN the application of E to E' (written EE' or $E(E')$) is correctly typed and will have type T_2 .

$E:T$ means that expression E is correctly typed with type T so that rule can be more succinctly expressed as:

$$\text{(type rule for application)} \quad \frac{E:T_1 \rightarrow T_2 \quad E':T_1}{EE':T_2}$$

We will frequently use this way of representing a rule, where the statement below the line (in this case EE' has type T_2) follows from the statement above the line. As another example, we express the rule which determines the types of types of tuples from the types of the components:

$$\text{(type rule for tuples)} \quad n \geq 2 \quad \frac{E_1:T_1 \quad E_2:T_2 \quad \dots \quad E_n:T_n}{(E_1, E_2, \dots, E_n):T_1 \times T_2 \times \dots \times T_n}$$

On the basis of above discussion we can have following table to makes the concept more clear:

Function	Type
+	int × int → int
-	int × int → int
×	int × int → int
div	int × int → int
mod	int × int → int
<	int × int → bool
≤	int × int → bool
>	int × int → bool
≥	int × int → bool
~	int → int
abs	int → int
not	bool → bool
^	string × string → string
size	string → int
chr	int → string
ord	string → int
&	bool × bool → bool
or	bool × bool → bool
even	int → bool
double	int → int
even prod	int × int → bool
either even	int × int → bool
sum between	int × int → int

11.6 NAMES, BINDING, ENVIRONMENTS AND SCOPE OF FPLS

We know that functional programming language programs plays around values. By naming them we can identify them and it allows us to associate a name with a value. The name is said to be *bound* to the value by the definition. For example in ML:

```
val the pair = (size "abc", size "abcdef")
```

Here we given a name to pair of integers. *Val* is just the standard *ML* convention for introducing definitions (of value names).

A collection of bindings (of names to values) is called an *environment*. Definition or modify environment, and any expression we write down can only be evaluated in the context of environment where all the names in the expression are bound to values.

The textual context of the definitions is decided by the *scope of the definition*, that is the expressions and other definitions which are to be regarded as within the context of the given definition. For example, the scope of the most of the definitions introduced in the text can be taken to be the rest of the book (beyond the definition) or, in case we re-use a name to stand for introduce some thing else, up to the pointer where the name is redefined.

11.7 SOME FPLS

Now at this stage it is necessary that, we should discuss the some of the examples of FPLs. Lisp, scheme, ML, and Haskell are the examples of the functional programming languages.

- *Lisp* is first FPL, defined by John Mc Carthy in 1958 as a language for AI. Originally, LISP was a typeless language with only two data types: atom and list. LISP's lists are stored internally as single-linked lists. LISP uses lambda notation to specify the functions. Functions, Function definitions, function applications and data all have the same form:

If the list (A,B,C) is interpreted as data it is simple list of three atoms, A, B and C but if interpreted as a function application, it means that the function named A is applied to the two parameters, B and C.

Common Lisp is the ANSI standard Lisp specification.

- *Scheme* was defined by sussman and steele (MIT) in mid 70's as a new LISP-like language. Goal was to move lisp back toward it's simpler roots and incorporate ideas which had been developed in the PL community since 1960. Scheme uses only static scoping and more uniform in treating functions as first-class objects which can be values of expressions and elements of lists, assigned to variables and passed as parameters.

Scheme includes the ability to create and manipulate closures and continuations. A closure is a data structure that holds an expression and an environment of variable bindings in which it's to be evaluated. A continuation in scheme is a data structure which represents "the rest of a computations".

Scheme has mostly been used as a language for teaching computer programming concepts where as common lisp is widely used as a practical language.

- *ML* (meta Language) is a strict, static-scoped functional language with a pascal-like syntax that was defined by Robin Milner in 1973. It was the first language to include statically checked polymorphic typing. ML uses type declarations, but also does type inferencing to determine the types of undeclared variables. ML is strongly typed (whereas scheme is essentially typeless) and has no type coercions. It includes exception handling and a module facility for implementing abstract data types, garbage collection and a formal semantics. Most common dialect is standard ML (SML), for example: fun cube (x: int) = $x * x * x$.
 - *Haskell* is very similar to ML (syntax, static, scoped, strongly typed, type inferencing). Haskell is purely functional (different from ML and most other FPLs) language - no variables, no assignment statements, and no side effects of any kind. Haskell uses lazy evaluation (evaluate no subexpression until the value is needed). It has "list comprehensions", which allow it to deal with infinite lists.
- For example: fib 0 = 1, fib 1 = 1, fib (n + 1) = fib (n + 1) + fib n

11.8 CONCEPTS OF FPLS

During the evaluation of FPLs, a number of interesting programming languages concepts have arisen, including

1. Polymorphism
2. Type inferencing
3. Higher-Order functions
4. Curried functions
5. Functional abstractions.
6. Lazy evaluation

11.8.1 Polymorphism

Let us consider example of identity function `I`. The definition of `I` is simply `fun 1 x = x` and the function returns its argument as result. This suggest that the function is quite general and could be applied to objects of any type (int, bool, string, int × string, etc). So what is the type of `I`? When applied to an integer, it returns an integer (that is $16 = 6$) and when applied to an int × string it returns an int × string (that is $|(6, "xyz")$). In general, if T is a type then `I` should be applicable to objects of type T returning object of type T , which suggests `I` should have type $T \rightarrow T$ for all types T .

Here `I` is a *polymorphic* with a “single definition and a more general type or type schema.”

Let us see some more example, as follows:

Consider the function in ML:

```
fun mymax (x, y) = if x > y then x else y;
```

SML infers `mymax` is an integer function: `int -> int`

```
fun mymax (x: real, y) = if x > y then x else y;
```

SML infers `mymax` is `real`

So here `mymax` is polymorphic function

11.8.2 Type Inferencing

It is the ability of the language to infer types without having programmer provide type signatures. For example in SML:

```
fun min (a: real, b) = if a > b then b else a
```

The type of ‘`a`’ has to be given, but then that’s sufficient to figure out the type of `b` and the type of `min`.

In Haskell (as with ML) guarantees type safety (if it compiles, then it’s type safe). For example in Haskell `eq = (a = b)` is a polymorphic function that has return type of `bool`, assumes only that its two arguments are of the same type and can have the equality operator applied to them.

Overuse of type inferencing in both languages is discouraged.

11.8.3 Higher-order Functions

An essential feature of modern functional programming language is a facility to treat functions as data objects which can be used in programs like other data objects. Functional programming goes beyond just the simple view of programs components as mathematical functions. It

involves the use of functions to manipulate, create and generalize other functions. This in turn provides for a higher level approach to program creation.

If functions are to be treated as data objects we should also allow:

- (1) functions to be arguments for other functions,
- (2) functions which produce functions as result,
- (3) data structures (such as tuple) with functions as components.

A language with these possibilities is usually called *higher-order* and we also refer to functions which take functional arguments and/or produce functional results as higher order functions

The type of a higher order function will typically involve an arrow (\rightarrow) nested within the T_1 or T_2 of some type $T_1 \rightarrow T_2$. For example, $(\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \times \text{bool})$ is the type of a higher order function which expects as argument a function of type $\text{int} \rightarrow \text{int}$, and $\text{bool} \rightarrow (\text{int} \rightarrow \text{int})$ is the type of a higher order function which when applied to a value of type bool returns a function of type $\text{int} \rightarrow \text{int}$.

The compositional operator is very good example of higher order function, for example for any argument x ,

$$(f \circ g)(x) = f(g(x)).$$

The symbol (\circ) is being to denote a higher order function-composition operator, and this gives us as direct way of combining functions without having to use parameters.

We can also define *Higher-order Functions* in another way. We know that zero-order functions takes data in traditional sense, first-order functions operates on zero-order functions, and second-order functions operate on first order.

"Higher-order functions are thus that can operate on functions of any order as long as types match."

Applicative programming has often been considered the application of first-order function. Functional programming has been considered to include higher-order functions.

11.8.4 Curried Functions

The logician Frege noted in 1883 that we only need *functions of one argument*.

We can replace a function $f(x, y)$ by a new function $f'(x)$ that when called produces a function of another argument to compute $f(x, y)$

$$\text{that is : } (f'(x))(y) = f(x, y)$$

"Function which are defined to expect arguments one at a time so that they can be supplied at different points in a computation or program are called *curried functions*".

This terminology is derived from the mathematician and logician H-B. Curry who used higher order functions extensively in the study of *combinatory logic*. Any tuple-expecting function f (of type $(\alpha_1 \times \alpha_2 \times \dots \times \alpha_n) \rightarrow \beta$, say) has a corresponding curried version $\text{curry } f$ (say) of type $\alpha_1 \rightarrow (\alpha \rightarrow \dots (\alpha_n \rightarrow \beta) \dots)$ and vice versa. Any expression if the form $f(E_1, E_2, \dots, E_n)$ can thus be replaced by $\text{curry } f(E_1, E_2, \dots, E_n)$ and vice versa.

To curry: $((a, b) \rightarrow c) \rightarrow (a \rightarrow b \rightarrow c)$

To uncurry: $(a \rightarrow b \rightarrow c) \rightarrow ((a, b) \rightarrow c)$

11.8.5 Functional Abstraction

Functional programming allows *functional abstraction* that is not supported in imperative languages, namely the definition and use of functions that take functions as argument and return functions as argument and return functions as values.

Let us see a simple non-recursive function such as:

```
fun double x = x + x
```

It is the name double which is being bound to a value by this definition, and the name (parameter) x serves as an aid in the description of the functional value. We introducing the following alternative way of writing this:

```
val double = fn x => x + x
```

Although this seems less readable, it shows that double is the name being bound to a value which is described on the right-hand side of the definition and that x is part of the value's description. The expression $fn x => x + x$ is called an *abstraction* and can be read as the function which maps x to $x + x$.

11.8.6 Lazy Evaluation

So far, we have not been very explicit about how expression can be evaluated. We have also not discussed how functions should be have when their arguments are undefined. Up to now, we have assumed that expression can be evaluated simply by expanding definitions and calculating the results of primitive function applications.

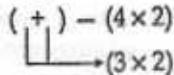
So now it is important that, we should learn, which arguments to a function are evaluated and when.

"*An evaluation strategy in which arguments to a function are evaluated only when needed for computation is called lazy evaluation*". This strategy is supported by many FPLs including Scheme, Haskell and common Lisp. Lazy evaluation is very important for dealing with very large or infinite streams of data. It is usually implemented using closures - data structures containing all the information required to evaluate the expression.

Let us consider an example:

```
let val y = 3 * 2 in (y + y) - (4 * 2) end
```

If we simplify 3×2 first and then substitute the result for y in the main expression we get $(6 + 6) - (4 \times 2)$. The simplification of 4×2 to 8 could have preceded or followed this step or even have been done simultaneously to get $(6 + 6) \times 8$ and eventually 4. A less obvious order might have been to substitute 3×2 for y at the start and then to calculate $(3 \times 2 + 3 \times 2) - (4 \times 2)$. One could draw a diagram for the expression with arrows to indicate the common subexpression so that its evaluation could be delayed but done at most once:



This latter technique forms the basis of what is called *lazy evaluation*.

In opposite to lazy evaluation, *eager evaluation*, is the usual default in a programming language in which argument to a function are always evaluated before the functions is applied.

11.9 APPLICATION OF FUNCTIONAL LANGUAGE

The basic concepts of functional programming originated with LISP. Functional programming language are important and having following application:

- Lisp is used for artificial intelligence applications
 - Knowledge learning
 - Machine learning
 - Natural language processing
 - Modeling of speech and vision.
- Functional programming encourages thinking at higher levels of abstraction by providing higher-order functions - functions that modify and combine existing programs.
- Functional programming has natural implementation in concurrent programming.
- Functional programming is useful for developing executable specifications and photo type implementation.
- Functional programming has a close relationship to computer science theory. Functional programming is based on the lamda-calculus which in turn provides a frame work for studying decidability questions of programming. The essence of denotational semantics is the translation of conventional programs into equivalent programs.
- Pure FPLs like Haskell are useful in contexts requiring some degree of program verification.

11.10 THE LAMBDA CALCULUS

Functional programming languages are based on the lamda-calculus grew out of an attempt by Alonzo church and Stephen kleene in the early 1930s to formalize the notion of computability (also known as constructibility and effective calculability). It is a formalization of the notion of functions as rules. As with mathematical expression, it is characterized by the principle that the value of an expression depends only on the values of its subexpression. The lamda-calculus is a simple language with few constructs and a simple semantics. But it is expressive; it is sufficiently powerful to express all computable functions.

Lambda calculus is used as a bridge between high-level functional languages and their low-level implementations. The reason for introducing the lambda calculus as an intermediate language are:

1. It is a simple language, with only a few syntactic constructs, and simple semantics.
2. It is an *expressive* language, which is sufficiently powerful to express all functional programs. This means that if we have an implementation of the lamda calculus, we can implement any other functional language by translating it into the lamda calculus.

As an informal example of the lambda-calculus, consider the function defined by the polynomial expression

$$x^2 + 3x - 5.$$

The variable x is a parameter. In the lambda-calculus, the notation $\lambda x. M$ is used to denote a function with parameter x and body M . That is, x is mapped to M we rewrite out function in this format

$$\lambda x. (x^2 + 3x - 5)$$

and read it as "the function of x whose value is defined by $x^2 + 3x - 5$ ". The lambda-calculus uses prefix form and so we rewrite the body in prefix form,

$$\lambda x. (- (+ (\times xx) (\times 3x)) 5)$$

The lambda-calculus *curries* it functions of more than one variable that is $(+ xy)$ is written as $((+ x) y)$, the function $(+ x)$ is the function which adds something to x . Rewriting our example in this form we get:

$$\lambda x. ((- ((+ ((\times x) y)) ((\times 3) x))) 5)$$

To denote the application of a function f to an argument 'a' we write.

To apply our example to the value 1 we write

$$\lambda x. ((- ((+ ((\times x) x)) ((\times 3) x))) 5) 1$$

To evaluate the function application, we remove the $-x.$ and replace each remaining occurrence of x with 1 to get

$$((- ((+ ((\times 1) 1)) ((\times 3) 1))) 5)$$

then evaluate the tow multiplication expressions

$$((- ((+ 1) 3)) 5)$$

then the edition

$$((- 4) 5)$$

and finally the subtraction

$$- 1.$$

11.10.1 Use of Brackets

In mathematics it is conventional to omit redundant to avoid cluttering up expressions. For example, we might omit brackets from the expression $(ab) + ((2c)/d)$ to give

$$ab + 2c/d$$

The second expression es easier to read than the first, but there is a danger that it may be ambiguous. It is rendered unambiguous by establishing conventions about the precedence of the various function (for example, multiplications binds more tightly than addition).

Sometimes brackets cannot be omitted, as in the expression:

$$(b + c)/a$$

Similarly conventions are useful when writing down expressions in the lambda calculus. Consider the expression:

$$((+ 3) 2)$$

By establishing the convention that *function application associates* to the left, we can write the expression more simple as:

(+ 32)

or even

+ 32

We performed some such abbreviation in the examples given earlier. As a more complicated example, the expressions:

 $((f ((+ 4) 3)) (gx))$

Is fully bracketed and unambiguous. Following our convention, we may omit redundant brackets to make the expression easier to read, giving:

 $f(+ 43) (gx)$

No further brackets can be omitted. Extra brackets may of course, be inserted freely without changing the meaning of the expression; for example

 $(f (+ 43) (gx))$

is the same expression.

11.10.2 Lambda Abstraction

The lambda calculus provides a construct, called a *lambda abstraction*, to denote new (non-built-in) functions.

Let us consider an example of lambda abstraction:

 $(\lambda x. + x 1)$

The λ says 'here comes a function', and is immediately followed by variable, x in this case; then comes a followed by the body of the function, $(+ x 1)$ in this case. The variable x is called the formal parameter, and we say that the λ binds it. We can think of it like this:

$$\begin{array}{ccccc} (\lambda & & x & * & + x 1) \\ \uparrow & & \uparrow & \uparrow & \uparrow \end{array}$$

That function of x which adds x to 1.

A lambda abstraction always consists of all the four parts mentioned: the λ the formal parameter, the $*$ and the body.

The body of a lambda abstraction extends as far to the right as possible, so that in the expression

 $(\lambda x. + x 1) 4$

The body of the λx abstraction is $(+ x 1)$, not just $+$. As usual, we may add extra brackets to clarity, thus

 $(\lambda x. (+ x 1)) 4$

when a lambda abstraction appears in isolation we may write it without any brackets:

 $\lambda x. + x 1$

11.10.3 Bound and Free Variables

Let us consider a lambda expression

$$(\lambda x + xy) 4$$

In above expression, x is just a formal parameter so to evaluate this expression completely, we do not need to know a "global" value for x . On the other hand we need to know the 'global' value of y .

Here occurrence of x is bound with λx so x is called *bound variable* and 4 has to be replaced with x , during evaluation of expression. y is not bound by any λ , and so occurs free in the expression. Here y is said to be *free variable*. In general, the value of an expression depends only on the values of its free variables.

Consider the following lambda expression:

$$\lambda x. + ((\lambda y. + yz) 7) x$$

Here y is bound and x is also bound since there is an enclosing lambda abstraction which binds it, is free variable.

Some time a variable may have both a bound occurrence and free occurrence in an expression: consider for example:

$$+ x ((\lambda x. + x1) 4)$$

in which x is both free and bound. Clearly the terms *bound* and *free* refer to specific occurrences of the variable in an expression.

11.10.4 Beta-Reduction

A lambda-expression is executed by evaluating it. Evaluation proceeds by repeatedly selecting a reducible expression (or redex) and reducing it. For example, the expression $(+ (\times 56) (\times 83))$ reduce to 54 in the following sequence of reductions.

$$\begin{aligned} (+ (\times 56) (\times 83)) &\rightarrow (+ 30 (\times 83)) \\ &\rightarrow (+ 30 24) \\ &\rightarrow 54 \end{aligned}$$

When the expression in the application of a lambda abstraction to a term, the term is substituted for the bound variable. This substitution is called beta-reduction. In the following sequence of reduction, the first step an example of beta-reduction. The second step is the reduction required by the addition operator.

$$\begin{aligned} (\lambda x. ((+ 3) x)) 4 &\rightarrow ((+ 3) 4) \\ &\rightarrow 7 \end{aligned}$$

Let us see some more examples of simple beta-reduction:

- (i) The formal parameter may occur several times in the body:

$$\begin{aligned} (\lambda x. + xx) 5 &\rightarrow + 55 \\ &\rightarrow 10 \end{aligned}$$

- (ii) There may be no occurrences of the formal parameter in the body:

$$(\lambda x. \cdot 3) 5 \rightarrow 3$$

- (iii) The body of a lambda abstraction may consist of another lambda abstraction:

$$\begin{aligned} (\lambda x. (\lambda y. - yx) 45) &\rightarrow (\lambda y. - y4) 5 \\ &\rightarrow - 54 \\ &\rightarrow 1. \end{aligned}$$

- (iv) Functions can be arguments too:

$$\begin{aligned} (\lambda f. f3) (\lambda x. + x1) &\rightarrow (\lambda x. + x1) 3 \\ &\rightarrow + 31 \\ &\rightarrow 4 \end{aligned}$$

11.10.5 Syntax of the Lambda-calculus

The pure lambda-calculus has just three constructs: Primitive symbols, function application and functional creation.

Syntax:

L in Lambda Expression

x in symbols

L ::= x / (LL) / (λ x.L)

(LL) is function application, and

(λ x.L) is a lambda-abstraction which defines a function with argument x and body L.

We adopt following rotational conventions:

- We extend the lambda-calculus with the usual constants and functions so we allow $(\lambda x. ((+ x) 3))$ to represent the function $x + 3$
- We usually drop the outermost parentheses so we may write:
 $\lambda x. ((+ x) 3)$ instead of $(\lambda x. ((+ x) 3))$ and
 $\lambda x. ((+ x) 3)$ instead of $(\lambda x. ((+ x) 3) 4)$
- Function application associates to the left so we may write:
 $(+ x3)$ instead of $((+ x) 3)$ that is we, may write
 $\lambda x. + x3$ instead pf $\lambda x. ((+ x) 3)$
- Replace the body of a lambda- abstraction with conventional in fix notation so we may write
 $(\lambda x. x + 3) 4$ instead of $(\lambda x. + x3) 4$
- Multiple parameters are written together so we may write.
 $\lambda xy. x + y$ instead $\lambda x. \lambda y. x + y$

11.10.6 Alpha-conversion

Consider the two lambda abstractions

$$(\lambda x. + x1)$$

and

$$(\lambda y. + y1)$$

Clearly they 'ought' to be equivalent, and α -conversion allow us to change the name of the formal parameter of any lambda abstraction:

$$(\lambda x. + x1) \xleftarrow{\alpha} (\lambda y. + y1)$$

where the arrow is decorated with an α to specify an α -conversion.

11.10.7 Eta-conversion

Let us consider two expressions:

$$(\lambda x. + 1x)$$

and

$$(+ 1)$$

These expression be have in exactly the same way when applied to an argument: they add 1 to it. η -conversion is a rule expressing their equivalence:

$$(\lambda x. + 1x) \xrightarrow{\eta} (+ 1)$$

More generally, we can express the η -conversion rule like this:

$$(\lambda x. fx) \xrightarrow{\eta} F$$

Provided x does not occur free in F , and F denoted a function.

The condition that x does not occur free in F prevents false conversions. For example,

$$(\lambda x. + xx)$$

is not η -convertible to

$$(+ x)$$

because x occurs free in $(+ x)$.

11.10.8 Normal Form of Expression

A lambda-expression is said to be in *normal form* if no beta-redex, a subexpression of the form $(\lambda x. PQ)$, occurs in it.

So the evaluation of an expression consists of successively reducing redexes until the expression is in normal form.

Non-terminating computations are example of expressions that do not have normal forms. The lambda expression

$$(\lambda x. xx) (\lambda x. xx)$$

does not have a normal. This situation corresponds directly to an imperative program in to an infinite loop.

An expression may contain more than one redex, so reduction can proceed by alternative routes. For example the expression $(+ (\times 34) (\times 78))$ can be reduced to normal form with the sequence.

$$\begin{aligned} & (+ (\times 34) (\times 78)) \\ & \rightarrow (+ 12) (\times 78)) \end{aligned}$$

$\rightarrow (+ 12 56)$

$\rightarrow 68$

or the sequence

$\rightarrow (+ (\times 34) (\times 78))$

$\rightarrow (+ (\times 34) 56)$

$\rightarrow (+ 12 56)$

$\rightarrow 68$

11.10.9 Substitution

We define substitution, $B[x : M]$, to be replacement of all free occurrences of x in B with M . Let us discuss the formal definition of substitution as follows:

$s[x : M] = \text{if } (s = x) \text{ then } M \text{ else } s$

$(A, B)[x : M] = (A[x : M] B[x : M])$

$(\lambda x. B)[x : M] = (\lambda x. B)$

$(\lambda y. B)[x : M] = \text{if } (z \text{ is a symbol not free in } B \text{ or } M) \text{ then } \lambda z. (B[y : z][x : M])$

where s is a symbol, M , A and B are lambda-expressions.

The syntactic transformation of the lambda-expression can be written as follows:

Interpreter: reduce expression E to normal form.

Reduce in $L \rightarrow L$

Reduces $[s] = s$

Reduces $[\lambda x. BM] = \text{Reduce}[B[x : M]]$

Reduces $[L_1, L_2] = [\text{Reduce}[L_1] \text{ Reduce}[L_2]]$ where

s is a symbol and B , L_1 , L_2 and M are lambda-expression.

11.10.10 Normal Order Reduction

Given a lambda expression, the substitution and beta-reduction rules provide the tools required to reduce a lambda-expression to normal form but do not tell us what order to apply the reductions when more than one redex is available. These complications raise an embarrassing question: can two different reduction sequences lead to different normal forms? Fortunately the answer is 'no'. This is a consequence of a profound and powerful pair of theorems, the Church-Rosser Theorems I and II.

According to Church-Rosser Theorem-I

If $E_1 \rightarrow E_2$, there exists an expression E , such that $E_1 \rightarrow E$ and $E_2 \rightarrow E$

As a consequence from CRT-I we can say that no expression can be converted to two distinct normal forms. According to Church-Rosser Theorem-II:

If $E_1 \rightarrow E_2$, and E_2 is in normal form then there exists a normal order reduction sequence from E_1 to E_2 .

As a consequence from CRT-II, we can hope that there is at most one possible result, and normal order reduction will find it if it exists.

Normal order reduction specifies that the left most outermost redex should be reduced first. The left most outer most reduction (normal order reduction) strategy is called *lazy reduction* because it does not first evaluate the argument but substitutes the arguments directly in to the expression. *Eager reduction* is when the arguments are reduced before substitution.

A function is *strict* if it is sure to need its argument. If a function is *non-strict*, we say that it is *lazy*.

In the reduction strategy, parameter passing may be by value, by name and lazy evaluation.

Call by name reduction strategy choose the leftmost-outermost redex and also referred to as normal-order reduction. It is guaranteed to reach a normal form if one exists. Let us see an example:

$$(\lambda xy. x) z N \xrightarrow{\beta} (\lambda y.z)N \xrightarrow{\beta} (\text{all by name})$$

The call-by-value reduction strategy chooses the leftmost-innermost redex in a sub term N rather than entire term $(\lambda y. z) N$. If reduction starting from N do not terminate, then call-by-value will fail to reach the normal form z (call-by-value can not guarantee to reach the normal form). Such an N is $(\lambda x. xx)$ $(\lambda x. xx)$, which reduces to itself:

$$\begin{aligned} (\lambda y. z) ((\lambda x. xx) (\lambda x. xx)) &\xrightarrow{\beta} (\lambda y. z) ((\lambda x. xx) (\lambda x. xx)) \\ &\xrightarrow{\beta} (\lambda y. z) ((\lambda x. xx) (\lambda x. xx)) \\ &\xrightarrow{\beta} \dots \text{ call-by-value} \end{aligned}$$

Consider the following example:

(PP) where P is $(\lambda x. xx)$. The evaluation of this expression would not terminate since (PP) reduces to (PP) :

$$\begin{aligned} (\lambda x. xx) (\lambda x. xx) &\rightarrow (\lambda x. xx) (\lambda x. xx) \\ &\rightarrow (\lambda x. xx) (\lambda x. xx) \end{aligned}$$

This situation corresponds directly to an imperative program going in to infinite loop.

Further more, some reduction sequence may reach a normal form while others do not. For example:

$$(\lambda x. 3) (PP)$$

But this problem can be addressed by normal order reduction: in above example we should choose the λx -redex first, not the (P, P) . This rule embodies the intuition that arguments to functions may be disregarded so we should apply the function $(\lambda x. 3)$ first, rather than first evaluating the argument (PP) .

11.10.11 Operational Semantics and Denotational Semantics

We have seen that calculation in the lambda-calculus is by rewriting (reducing) a lambda-expression to a normal form. For the pure lambda-calculus, lambda-expressions are reduced by the substitution. That is, occurrences of the parameter in the body are replaced with (copies of) the argument. This kind of calculation is called *operational semantics* of the lambda-calculus. It is called operational because it is 'dynamic' if sees a function as a sequence of operations. A

lambda expression was evaluated by purely *syntactic* transformation without reference to what the expression 'mean'.

The purpose of the *denotational semantic* of a language to assign a value to every expression in the language. We can express the semantics of the lambda-calculus as a mathematical function, Eval, from expression to values. For example,

$$\text{Eval } (+ 34) = 7$$

defines the value of the expression (+ 34) to be 7. Actually something more is required, in the case of variables and functions names, the function Eval requires a second parameter containing environment 'e' which contains association between variables and their values. Some programs go into infinite loops, some abort with a runtime error. To handle these situation we introduce the symbol \perp pronounced 'bottom'.

Let us see following denotational semantics for the lambda-calculus:

Semantic Domains: $S \in D$

Semantic Function: $\text{Eval} : L \rightarrow D$

Semantic Equations:

$$\text{Eval}[s] = s$$

$$\text{Eval}[(\lambda x. B M)] = \text{Eval}[B[x : M]]$$

$$\text{Eval}[(L_1 L_2)] = (\text{Eval}[L_1] \text{Eval}[L_2])$$

$$\text{Eval}[E] = \perp$$

where S is a symbol, B , L_1 , L_2 and M are expression $B[x : M]$ is substitution, E is an expression which does not have a normal form, and \perp is pronounced bottom.

The above denotational semantics describe a mapping of lambda expressions to values in some semantic domain.

11.10.12 Recursive Function

One feature of all functional program is recursion, and this throws the viability of the whole venture in to doubt, because the lambda calculus appears to lack anything corresponding to recursion.

Recursive functions will be implemented by modifying the implementation of lambda expression.

A function that multiplies its parameter x by itself can be written using lambda notation:

$$(\lambda(x)(x \times x))$$

This expression has the form

$$(\lambda(\text{formals}) \text{body})$$

A variant of this notation include a name f that can appear recursively within the body

$$(\text{rec } f (\lambda(\text{formals}) \text{body}))$$

Using this syntax, the factorial function is defined by

$$(\text{rec fact } (\lambda(x)$$

$$(\text{if } (\text{eq? } x 0)$$

1

$$(x \times (\text{fact } (- x 1))))))$$

11.10.13 Lexical Scope Rules

Block with local definitions may be defined in the lambda-calculus. We introduce two kinds of blocks, *let* and *letrec* expressions. Non recursive definitions are introduced with *let* expressions:

let n: E in B is an abbreviation for $(\lambda x. B)E$

Here is the example using *let-extension*

let x: 3 in (x xx)

Lets may be used where ever a lambda-expression is permitted. For example,

$\lambda y. \text{let } x: 3 \text{ in } (x yx)$

is equivalent to

$\lambda y. (x yx)$

Simple recursive definitions are introduced with *letrec* expressions which are defined in terms of *let* expressions and *y* combinator:

letrec n: E in B is an abbreviation for *let*

n: Y (λn. E) in B

let and *letrec* expressions may be nested. The definitions of the *let* and *letrec* expression are given below:

let n: E in B = $(\lambda n. B) E$

letrec n : E in B = let n: y(\n. E) in B

EXERCISE

1. What is functional programming language? Discuss the Importance of Functional Programming.
2. Draw a comparison between Functional and Imperative paradigm.
3. What is referential transparency in FP's? Explain it by taking a example.
4. "Functional programming languages are strongly typed", justify the statement.
5. Write short notes on following:
 - (i) Polymorphism in FP's
 - (ii) Type inferencing
 - (iii) Higher-order functions
 - (iv) Curried functions
 - (v) Functional abstractions.
 - (vi) Lazy evaluation.
6. What are the various Application of Functional language?
7. Explain the exception handling in ML.
8. The "91. function" is defined as
 $\text{fun } f(x) = \text{if } x > 100 \text{ then } x - 10 \text{ else } f(f(x + 11))$

In this context discuss various approaches to expression evaluation and evaluate the above function using innermost evaluation.

9. Reduce the following using Beta-Reduction:

- (i) $(\lambda x. + xx) 6$
- (ii) $(\lambda x. 3) 7$
- (iii) $(\lambda x. (\lambda y. - yx)) 45$
- (iv) $(\lambda f. f3) (\lambda x. + x1)$

CHAPTER 12

Logic Programming Languages

12.1 INTRODUCTION TO LOGIC PROGRAMMING

Logic programming began in the early 1970's as a direct out growth of earlier work in automatic theorem Proving and Artificial intelligence. The concept of logic programming was developed by Kowalski and Colmerauer. The PROLOG is first logic programming language and the first PROLOG interpreter was implemented in the language ALGOL-W by Roussel, at Marseille in 1972.

According to main idea of logic programming language, an algorithm consists of two disjoint components, the *logic* and *control*.

- The logic is the statement of what the problem is that has to be solved.
- The control is the *statement of how* it is to be solved by separating these two components, programmer can concentrate only on logic part and can leave control part on logic programming system itself. In other words, an ideal logic programming language is purely declarative programming.

There are two major classes of logic programming languages currently available:

1. *System languages* emphasize on AND-Parallelism, don't care non-determinism and definite programs (That is, no negation). In these languages shared variables act as communication channels between processes. There are several system languages available like PROLOG, Concurrent PROLOG and GHC.
2. *Application languages* can be regarded as general-purpose programming languages with a wide range of applications. Here the emphasis is on OR-parallelism, don't know non-determinism and (unrestricted) programs (that is, the body of a program statement is an arbitrary formula). Examples include Quintus PROLOG, micro PROLOG and NU-PROLOG.

Application languages are better suited to deductive data base system and expert systems.

The emergence of these two kinds of logic programming languages has complicated the already substantial task of building parallel logic machines. Because of the different hardware requirement of the two classes of languages, it seems that a difficult choice has to be made.

In conclusion, logic provides a single formalism for apparently diverse parts of computer science. It provides us with general-purpose, problem-solving languages, concurrent languages

suitable for operating systems and also a foundation for deductive data base system and expert system. This range of application together with the simplicity, elegance and unifying effect of logic programming assures it of an important and influential future.

12.2 INTRODUCTION TO PREDICATE LOGIC

The propositional logic is not powerful enough to represent all types of assertions that are used in computer science and mathematics, or to express certain types of relationship between propositions such as equivalence.

For example, the assertain "x is greater than 1", where x is a variable, is not a proposition because we can not tell wheather it is true or false unless we kow the value of x. Thus propositional logic can not deal with such sentences.

Thus we need more powerful logic to deal with these and other prolems. The predicate logic is one of such logic it addresses these issues among others.

"A predicate is a verb phrase template that describes a property of objects, on a relationship among objects represented by the variables."

Let us see following example:

The sentences "The car Tom is driving is blue", "The sky is blue", and "The cover of this book is blue" come from the template "is blue" by placing an appropriate noun/phrase in front of it. The phrase "is blue" is predicate and it describe the property of being blue. Predicated are often given a *name*. For example any of "is blue", "Blue" or B can be reprsent the predicate "is blue" among other.

B (x) reads as "x is blue"

Note: Throughout the chapter, we will use the following standard logic symbols: " \rightarrow " (material implication), " \neg " (not), " \vee " (or), " \wedge "(and), " \forall " (for all), and " \exists " (there exists).

Let us see some related term as follows:

1. *Universe of Discourse:* alos universe, is the set of objects of interest. The propostion in the predicate logic are statements ob objects of a universe. The universe is thus the domain of the (individual) variables.
2. *The Universal Quantifier:* The expression: $\forall x P(x)$, denotes the universal quantification of the atomic formula $P(x)$. We can read it like "For all x, $P(x)$ holds", for each x, $P(x)$ holds "or" for every x, $P(x)$ holds". \forall is called the *universal quantifier*, and $\forall x$ means all the objects x in the universe. If this is followed by $P(x)$ then the meaning is that $P(x)$ is true for every object x in the universe. For example, "all cars have wheels" could be transformed into the propositional form, $\forall x P(x)$, where:

- $P(X)$ is the predicate denoting; x has wheels, and
- The universe of discourse is anly populated by cars.

If all the elements in the universe of discourse cab be listed then the universal quantification $\forall x P(x)$) is equivalent to the conjunction: $P(x_1) \vee P(x_2) \vee P(x_3) \vee \dots \vee P(x_n)$. For example if we tall be about only four cars in above example (c_1, c_2, c_3 and c_4):

$$\forall x P(x) \text{ is } P(c_1) \wedge P(c_2) \wedge P(c_3) \wedge P(c_4)$$

"Finally we can say that universal quantifiers allows us to make a statement about a collection of object:

$$\forall x \text{ cat}(X) \Rightarrow \text{mammel}(X)$$

All cats are manels.

$$\forall x \text{ Father(Bill, } x) \dots \text{ Mother(Hillary, } x)$$

All of Bill's kids are also Hillary's kids.

3. Existential Quantification: Allow us to state that an object does exist (with naming it):

$$\exists x \text{ cat}(x) \rightarrow \text{Mean}(x)$$

There exist a mean cat.

$$\exists X \text{ Father(Bill, } x) \wedge \text{Mother(Hillary, } x))$$

There is a kid whose father is Bill and whose mother is Hillary.

If all the elements in the universe of discourse can be listed, then the existential quantification $\exists x P(x)$ is equivalent to the disjunction: $P(x_1) \vee (x_2) P(x_3) \vee \dots \vee P(x_n) = \exists x$

12.3 RULES FOR CONSTRUCTING WELL-FORMED FORMULA

Not all strings can represent proposition of the predicate logic. Those which produce a proposition when their symbols are interpreted must follow the rules given below, and they are called Wffs of the first order predicate logic.

A predicate name followed by a list of variables such as $P(x, y)$, where P is a predicate name, and x and y are variable, is called an *atomic formula*.

Wffs are constructed using the following rules:

1. True and False are Wffs.
2. Each propositional constant (That is specific proposition), and each propositional variable (That is a variable representing propositions) are Wffs.
3. Each atomic formula (that is a specific predicate with variables) is a Wff.
4. If A, B and C are Wff, then so are $\neg A$, $(A \wedge B)$, $(A \vee b)$, $(A \rightarrow b)$, and $(A \leftrightarrow B)$.
5. If x is a variable (representing objects of the universe of discourses), and A is as Wff, then so are $\forall x A$ and $\exists x A$.

Example 12.1 Let $p(x)$ be the statement "x is happy", where the universe of discourse for x is the set of students. Express each of the following quantification in English.

- (i) $\exists x P(x)$
- (ii) $\forall x \neg P(x)$
- (iii) $\exists x \neg P(x)$
- (iv) $\neg \forall x \neg P(x)$

Solution:

- (i) There is a student who is happy.
- (ii) Every student is not happy.
- (iii) There is a student who is not happy.
- (iv) Not all students are unhappy.

Example 12.2 Suppose that the universe of discourse of the atomic formula $P(n, y)$ is {1, 2, 3}. Write out the following propositions using disjunction and conjunctions.

- (i) $\exists x P(x, 2)$
- (ii) $\forall y P(3, 4)$
- (iii) $\forall x, \forall y P(x, 4)$
- (iv) $\exists x \forall x P(x, 4)$
- (v) $\forall y \exists x P(x, 4)$

Solution:

- (i) $P(1, 2) \vee P(2, 2) \vee P(3, 2)$
- (ii) $P(3, 1) \wedge P(3, 2) \wedge P(3, 3)$
- (iii) $P(1, 1) \wedge P(1, 2) \wedge P(1, 3) \wedge P(2, 1) \wedge P(2, 2) \wedge P(2, 3) \wedge P(3, 1) \wedge P(3, 2) \wedge P(3, 3)$
- (iv) $(P(1, 1) \wedge P(1, 2) \wedge P(1, 3)) \vee P(2, 1) \wedge P(2, 2) \wedge P(2, 3)) \vee P(3, 1) \wedge P(3, 2) \wedge P(3, 3))$
- (v) $(P(1, 1) \vee P(2, 1)) \vee P(3, 1) \wedge P(1, 2)) \vee P(2, 2) \vee P(3, 2) \wedge ((P(1, 3) \vee P(2, 3) \vee P(3, 3))$

12.4 TRANSCRIBING ENGLISH TO PREDICATE LOGIC WFFS

English sentences appearing in logical reasoning can be expressed as a Wff. This makes the expressions compact and precise. It thus eliminates possibilities of misinterpretation of sentences. The use of symbolic logic also makes reasoning formal and mechanical, contributing to the simplification reasoning and making it less power to error.

To transcribe a proposition stated in English using a given set of predicate symbols, first restate in English the proposition using the predicates, connectives, and quantifiers. Then replace the English phrases with the corresponding symbols.

Example 12.3 Let $L(x, y)$ be the predicate “ x likes y ”, and let the quantifiers to express of the following statements.

- (i) Every one likes every one.
- (ii) Every one likes someone.
- (iii) Some one does not like anyone.
- (iv) Every one likes Ram.
- (v) There is some one whom every one likes.
- (vi) There is no one whom every one likes.
- (vii) Every one does not like some one.

Solution:

- (i) $\forall x \forall y (x, y)$
- (ii) $\forall x \exists y L(x, y)$
- (iii) $\exists x \forall y \neg L(x, y)$
- (iv) $\forall x L(x, \text{George})$
- (v) $\exists x \forall y L(y, x)$

(vi) $\neg \exists x \forall y L(y, x)$ (vii) $\forall x \exists y \neg L(x, y)$

Example 12.4 Let $S(x)$ be the predicate "x is a student," $B(x)$ the predicate "x is a book," and $H(x, y)$ the predicate "x has y," where the universe of discourse is the universe, That is the set of all object. Use quantifiers to express each of the following statement.

- Every student has a book.
- Some student does not have any book.
- Some student has all the book.
- Not every student has a book.
- There is a book which every student has.

Solution:

- $\forall x (S(x) \rightarrow \exists y (B(y) \wedge H(x, y)))$
- $\exists x (S(x) \wedge \forall y (B(y) \rightarrow \neg H(x, y)))$
- $\exists x (S(x) \wedge \forall y (B(y) \rightarrow H(x, y)))$
- $\neg \exists x (S(x) \rightarrow \exists y (B(y) \wedge H(x, y)))$
- $\exists x (B(x) \wedge \forall y (S(y) \rightarrow H(x, y)))$

Example 12.5 Let us consider the following statements:

- Marcus was a man.
- Marcus was a pompian.
- All pompeians were Romans.
- Caesar was a ruler.
- All Romans were either logical to caesar or hated him
- Every one is logical to someone.
- Men only try to assassinate rulers they are not logical to
- Marcus tried to assassinate caesar.

We can write above facts in Wff's as follows:

- Man (Marcus)
- Pompeian (Marcus)
- $\forall x \text{pompeian}(x) \Rightarrow \text{Roman}(x)$
- Ruler (Caesar)
- $\forall x \text{Romans}(x) \Rightarrow \text{Loyal to}(x, \text{caesar}) \vee \text{Hate}(x, \text{caesar})$
- $\forall x : \exists y : \text{loyal to}(x, y)$
- $\forall x \forall y \text{Man}(x) \wedge \text{Ruler}(y) \wedge \text{Tryassassinate}(x, y) \Rightarrow \neg \text{Loyal to}(x, y)$
- Try assassinate (Marcus, caesar)

12.5 DEFINITE CLAUSES

The idea of logic programming is to use a computer for drawing conclusion from declarative description. Such descriptions- called *logic programs*, consist of finite sets of logic formulas.

Thus, the idea has roots in the research on automatic theorem proving. However, the transition from experimental theorem proving to applied logic programming requires improved efficiency of the system. This is achieved by introducing restrictions on the language of formulas—restriction. That makes it possible to use the relatively simple and powerful inference rule called the SLD-resolution principle. In this section we will introduce a restricted language of *definite logic program*.

To start with, attention will be restricted to a special type of declarative sentences of natural language that describe positive *facts* and *rules*. A sentence of this type either states that a relation holds between individuals (in case of a fact), or that a relation holds between individuals provided that some other relation holds (in case of a rule). For example, consider the sentences:

- (i) "Tom is John's child"
- (ii) "Ann is Tom's child"
- (iii) "John is Mark's child"
- (iv) "Alice is John's child"
- (v) "The grandchild of a person is child of a child of this person".

These sentences can be formalized in two steps. First atomic formulas describing facts are introduced:

Child (Tom, John)	...(1)
Child (Ann, Tom)	...(2)
Child (John, Mark)	...(3)
Child (Alice, John)	...(4)

Applying this notation to the final sentence yields:

"For all x and y , grandchild (x, y) if
there exists a z such that child (x, z) and child (z, y)"

This can be further formalized using quantifiers and the logical connectives " \neg ". In expression the implication is reversed and written " \leftarrow ":

$$\forall x \forall y (\text{grand child } (x, y) \leftarrow \exists z (\text{child } (x, z) \wedge \text{child } (z, y))) \quad ... (6)$$

This formula can be transformed into the following equivalent forms using the logical equivalence:

$$\begin{aligned} & \forall x \forall y (\text{grand child } (x, y) \vee \neg \exists z (\text{child } (x, z) \wedge \text{child } (z, y))) \\ & \forall x \forall y (\text{grand child } (x, y) \vee \forall z \neg (\text{child } (x, z) \wedge \text{child } (z, y))) \\ & \forall x \forall y \forall z (\text{grand child } (x, y) \vee \neg (\text{child } (x, z) \wedge \text{child } (z, y))) \\ & \forall x \forall y \forall z (\text{grand child } (x, y) \leftarrow (\text{child } (x, z) \wedge \text{child } (z, y))) \end{aligned}$$

We now focus attention on the language of formulas exemplified by the example above.

It consists of formulas of the form:

$$A_0 \leftarrow A_1 \wedge \dots \wedge A_n \quad (\text{where } n \geq 0)$$

or equivalently:

$$A_0 \vee \neg A_1 \vee \dots \vee \neg A_n$$

where A_0, \dots, A_n are atomic formulas and all variables occurring in a formula are implicitly universally quantified over the whole formula. The formulas of this form are called *definite clauses*.

Facts are definite clauses where $n = 0$ (Facts are sometimes called unit-clauses). The atomic formula A_0 is called the lead of the clauses whereas $A_1 \wedge \dots \wedge A_n$ is called its body.

The initial example shows that definite clauses wise a restricted form of existential quantification—the variables that occur only in body literals are existentially quantified over the body (Though formally this is equivalent to universal quantification on the level of clauses).

12.5.1 Clause

A clause is a formula $\vee(L_1 \vee \dots \vee L_n)$ where each L_i is an atomic formula (a positive literal) or the negation of an atomic formula (a negative literal).

As we have seen above that definite clause is a clause that contains exactly one positive literal. That is, a formula of the from:

$$\vee(A_0 \vee \neg A_1 \vee \dots \vee \neg A_n)$$

The notational convention is to write such a definite clause thus:

$$A_0 \leftarrow A_1, \dots, A_n \quad (n \geq 0)$$

If the body is empty (that is if $n = 0$) the implication arrow is usually omitted.

12.5.2 Definite Programs

A definite program is a finite set of definite clauses. To explain the use of logic formulas as programs, a general view of logic programming is presented in Fig. 12.1.

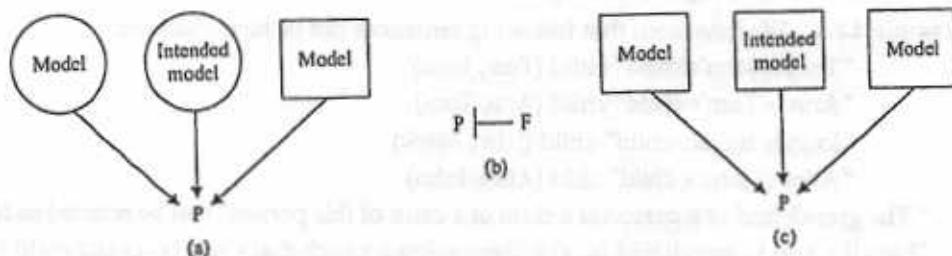


Fig. 12.1 General view of logic programming

The programmer attempts to describe the intended model by means of declarative sentences (That is when writing a program) he has in mind an algebraic structure, usually infinite, whose relations are to interpret the predicate symbols of the program). These sentences are definite clause-facts and rules. The program is a set of logic formulas and it may have many models, including the intended model (Fig. 12.1(a)). The concept of intended model makes it possible to discuss correctness of logical programs—a program P is incorrect iff the intended model is not a model of P . (Notice that in order to prove programs to be correct or to test programs it is necessary to have an alternative description of the intended model, independent of P).

The programmer will be used by the computer to draw conclusions about the intended model (Fig. 12.1(b)). However, the only information available to the computer about the intended model is the program it self. So the conclusions drawn must be true in any model of the program to guarantee that they are true in the intended model (Fig. 12.1(c)).

The set of logical consequences of a program is finite. Therefore the user is expected to query the program selectively for various aspects of the intended model. There is an analogy with relational data base—facts explicitly describes elements of the relations while rules give intensional characterization of some other elements. Since the rules may be recursive, the relation described may be infinite in contrast to the traditional relational database. Another difference is the use of variables and compound terms. This chapter considers only “queries” of the form:

$$\vee (\neg (A_1 \wedge \dots \wedge A_m))$$

Such formulas are called *definite goals* and are usually written as:

$$\leftarrow A_1, \dots, A_m$$

where A_i are atomic formulas called subgoals. The goal where $m = 0$ is denoted \square and called the empty goal. The logical meaning of a goal can be explained by referring to the equivalent universally quantified formula:

$$\forall x_1, \dots \forall x_n \neg (A_1 \wedge \dots \wedge A_m)$$

where x_1, \dots, x_n are all variables that occur in the goal this equivalent to :

$$\neg \exists x_1, \dots \exists x_n (A_1 \wedge \dots \wedge A_m)$$

This, in turn, can be seen as an existential question and the system attempts to deny it by constructing a counter-example. That is, it attempts to find terms t_1, \dots, t_n such that the formula obtained from $A_1 \wedge \dots \wedge A_m$ when replacing the variable x_i by t_i ($1 \leq i \leq n$), is true in any model of the program, that is to construct a logical consequence of the program which is an instance of a conjunction of all subgoals in the goal.

Example 12.6 We have seen that following sentences can be formalized as:

$$\text{"Tom is John's child"} \text{ child (Tom, John)} \quad \dots(1)$$

$$\text{"Ann is Tom's child"} \text{ child (Ann, Tom)} \quad \dots(2)$$

$$\text{"John is Mark's child"} \text{ child (John, Mark)} \quad \dots(3)$$

$$\text{"Alice is John's child"} \text{ child (Alice, John)} \quad \dots(4)$$

“The grandchild of a person is a child of a child of this person” can be notated as follows:

“For all x and y , grandchild (x, y) if there exists a z such that child (x, z) and child (z, y)”
...(5)

We can also write it as follows:

$$\forall x \forall y \forall z \text{ (grandchild (x, y) } \leftarrow \text{ (child (x, z) } \wedge \text{ child (z, y)))$$

Now user may ask the following queries:

Query	Goal
“Is Ann a child of Tom?”	$\leftarrow \text{child (ann, Tom)}$
“Who is a grandchild of Ann?”	$\leftarrow \text{grandchild (X, ann)}$
“Whose grandchild is Tom?”	$\leftarrow \text{grandchild (Tom, X)}$
“Who is a grandchild of whom?”	$\leftarrow \text{grandchild (X, Y)}$

The following answers are obtained:

- Since there are no variables in the first goal the answer is simply “Yes”.

- Since the program contains no information about grandchildren of Ann the answer to the second goal is "no one" (although most Prolog implementations would answer simply "No").
- Since Tom is the grandchild of Mark the answer is $X = \text{mark}$ in reply to the Third goal;
- The final goal yields three answers:

$X = \text{tom}$	$Y = \text{mark}$
$X = \text{alice}$	$Y = \text{mark}$
$X = \text{ann}$	$Y = \text{john}$

It is possible to ask more complicated queries, for example "Is there a person whose grandchild are Tom and Alice?",

$\leftarrow \text{grandchild}(\text{Tom}, x), \text{grandchild}(\text{Alice}, x)$

whose (expected) answer is $x = \text{mark}$.

12.6 SLD-RESOLUTION

In this section we will introduce the inference mechanism which is the basis of the most logic programming system. The idea is a special case of the inference rule called the *resolution-principle* – an idea that was first introduced by A.J. Robinson in the mid-sixties for a richer language than definite programs.

This is also called an SLD-resolution.

The SLD-resolution principle makes it possible to draw correct conclusion from the program, thus providing a foundation for a logically sound operational semantic of definite programs.

"The term resolution in logic refers to a mechanical method for proving statements in first-order logic. It is applied to two clauses in as sentence, and by unification, if eliminates a literal that occur positive in one clause and negative in other".

A literal stated in the antecedent of an implication is negative because an implication $P \rightarrow Q$ is equivalent to $\neg P \vee Q$.

Resolution is very important since it is a procedure with single proof method, carried out by a single operation with statements in predicate logic. In resolution operation takes place on the statement which are converted on very standard form (clause Form).

In Resolution to prove a statement valid, it attempts to show that the regulation of the statement produces a contradiction with. The known statements (that is unsatisfiable). This technique is said to be *refutation*.

For the Resolution it is necessary that all statement must be in clause form. We can follow, following steps for the conversion in to clause form:

1. Eliminate all the implications \rightarrow , using the fact that $a \rightarrow b$ can be written $\neg a \vee b$.
2. Reduce the scope of all \neg to single term, using the facts:

- (i) $\neg(\neg P) = P$
- (ii) $\neg(a \wedge b) = \neg a \vee \neg b$
- (iii) $\neg(a \vee b) = \neg a \wedge \neg b$
- (iv) $\neg \forall x : p(x) = \exists x : \neg P(x)$
- (v) $\neg \exists x : p(x) = \forall x ! \neg P(x)$

3. Make all variable name unique.
4. Move quantifiers left without changing their relative order.
For example:

$$\forall x : \forall y : [\neg P(x) \vee \neg Q(x, y)]$$

5. Eliminate Existential quantifiers:

- (i) Any variable that is existentially quantified means we are saying there is some value for that variable that makes the expression true.
- (ii) To eliminate the quantifier, we can replace the variable with a function.
- (iii) We don't know that the function is, we just know it exists:

For example, \exists chairman (y)

We replace \exists with a new function func:

$$\text{chairman}(\text{func}())$$

Func() is called a *skolen function*.

In general the function must have the same number of arguments as the number of universal quantifiers in the current scope.

Let us see another example:

$$\forall x \exists y \text{ Father}(y, x).$$

here we will create a skolen function fn and replace it with y.

$$\forall x \text{ Father}(\text{fn}(x), x).$$

6. Eliminate the Universal Quantifiers.
7. Converts to conjunction of disjuncts : $[a \vee (b \vee c) = (a \vee b) \vee c, (a \wedge b) \vee c = (a \vee c) \wedge (b \vee c)]$.
8. Create separate clause for each conjunct.

12.6.1 Resolution in Proposition Logic

We can do the Resolution in proposition logic by following steps:

1. First we have to convert all the proposition to clause form.
2. Negate P and convert the result to clause form. We have to add it to the set of clauses obtained in first step.
3. We have to repeat until a contradiction is found or no progress can be made:
 - (i) Select two clauses as parent clauses.
 - (ii) Resolve them and find the resolvent clause. Then use it to further resolution.
 - (iii) If the resolvent is the empty clause, then a contradiction has been found. If it is not, then add it to the set of clauses available to the procedure.

Let us see following example:

Given Axioms are: Converted to clause form

A	A
$(A \wedge B) \rightarrow C$	$\neg A \vee \neg B \vee C$
$(D \vee E) \rightarrow B$	$\neg D \vee B$
E	$\neg E \vee B$

Here we are negating C, as $\neg C$ which is already in clause form. we are assuming that $\neg C$ is true.

Now $\neg A \vee \neg B \vee C$ and $\neg C$ are selected as parent clause.

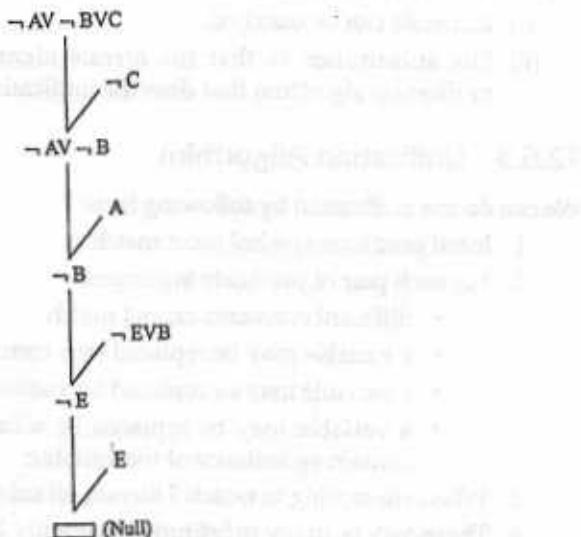


Fig. 12.2 Resolution in preposition Logic

12.6.2 Unification

As demonstrated in the previous section, one of the main ingredients in the inference mechanism is the process of making two atomic formulas syntactically equivalent.

Let S and t be terms. A substitution Q such that $S@_0$ and $t@_0$ are identical (denoted $S@_0 = t@_0$) is called unifier of S and t. The unification can be done by a straight forward recursive procedure called the unification algorithm.

For example: we want to unify the expressions.

$P(a, a)$

$P(b, c)$

The two instances of P. Now we will compare a and b, and decide that if substitute b for a, they could match. We will write that substitution as:

b/a

After substitution expression becomes:

$P(b, b)$

$P(b, c)$

Now we can attempt to unify argument b and c, which succeeds with substitution (c/b) . The entire unification process has now succeeded with a substitution that is the composition of the two substitutions we found. we write the can position as:

$(c/b) (b/a)$

"A unifier θ said to be most general unifier (mgu) of two terms iff θ is more general than any other unifier of the terms.

Definition (Unification): The process of finding substitution for predicate parameter is called **unification**. We need to know that:

- (i) 2 Literals can be matched.
- (ii) The substitution is that the literals identical there is a simple algorithm called unification algorithm that does the unification.

12.6.3 Unification Algorithm

We can do the unification by following facts:

1. Initial predicate symbol must match.
2. For each pair of predicate arguments:
 - different constants cannot match.
 - a variable may be replaced by a constant.
 - a variable may be replaced by another variable.
 - a variable may be replaced by a function as long as the function does not contain an instance of the variable.
3. When attempting to match 2 literals, all substitution must be made to the entire literal.
4. There may be many substitution that unify 2 literals, the most general unifier is always desired.

Let us see some substitution examples: as follows:

• $P(x)$ and $P(y)$:	substitution = (x/y) (substitute x for y)
$P(x, x)$ and $P(y, z)$:	$(z/y)(y/x)$ (Y for x , then z for y)
$P(x, f(y))$ and $P(\text{Ram}, z)$:	$(\text{Ram}/x, f(y)/z)$
$P(f(x))$ and $P(x)$:	Substitution can't done.
$P(x) \vee Q(\text{jane})$ and $P(\text{Bill}) \vee Q(Y)$:	$(\text{Bill}/x, \text{jane}/y)$

12.6.4 Predicate Logic Resolution

We can perform resolution in predicate logic by following steps:

When no empty clause exists and there are clauses that can be resolved:

1. Select 2 clauses that can be resolved.
2. Resolved the clause (after unification), apply the unification substitution to the result and store in knowledge basis.

Now let us again examine the axioms in clause form:

1. $\text{Man}(\text{marcus})$
2. $\text{Pompeian}(\text{marcus})$
3. $\neg \text{Pompeian}(x_2) \vee \text{Roman}(x_1)$
4. $\text{Ruler}(\text{ceaser})$
5. $\neg \text{Roman}(x_2) \vee \text{loyal to}(x_2, \text{Ceaser}) \vee \text{hate}(x_2, \text{Ceaser})$

6. Loyal to ($x_3, f_1(x_3)$)
7. $\neg \text{man}(x_4) \vee \text{ruler}(y_1) \vee \text{tryassassinate}(x_4, y_1) \vee \text{loyal to}(x_4, y_1)$
8. Tryassassinate (marcus, caesar)

Now goal is to prove:

$\text{hate}(\text{marcus}, \text{caesar})$

We can prove above goal by the help of resolution as follows:

We will start from $\neg \text{hate}(\text{marcus}, \text{caesar})$

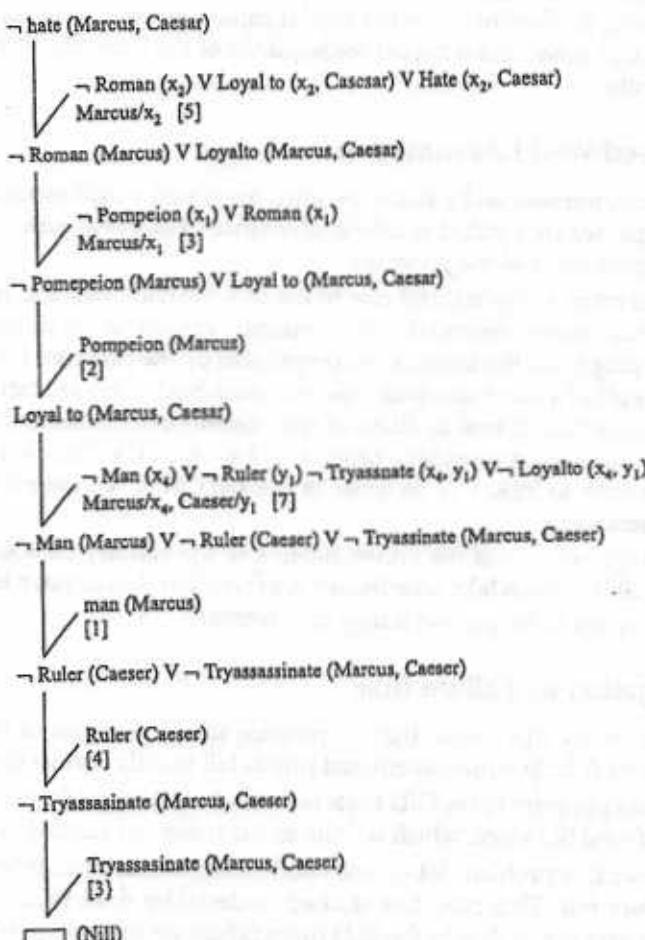


Fig. 12.3 A resolution proof

12.7 NEGATIVE INFORMATION

We know that only a positive information can be a logical consequence of a program, special rules are needed to deduce negative information. The most important of these rules are *closed world assumption (CWA)* and *negative as failure rules*.

The SLD-resolution applies to sets of Horn clauses with exactly one goal clause. Using SLD-resolution, we can never deduce negative information.

Let us consider the program:

```
Student (joe)
Student (bill)
Student (jim)
Teacher (mary)
```

Now suppose we wish to establish that mary is not a student, that is, a student(mary). As we have shown above, student(mary) is not logical consequence of the program. However, note that student(mary) is also not a logical consequence of the program. What we can do now is invoke special rule.

12.7.1 Closed World Assumption

The inference rule, introduced by Reiter, is called the closed world assumption (CWA). Under this inference rule, we are entitled to infer student(mary) on the grounds that student(mary) is not a logical consequence of the program.

The CWA is often a very natural rule to use in a database context. In relational database, this rule is usually applied—information not explicitly present in the database is taken false. Of course, in logic programs, the situation is complicated by the presence of non-unit clauses. The CWA is an example of a non-monotonic inference rule. Such rules are currently of great interest in AI. An inference rule in non-addition of new axioms can decrease the set of theorems that previously held. Now let us consider a program P for which the CWA is applicable. Let A ∈ BP and suppose we wish to infer UA. In order to use the CWA, we have to show that A is not a logical consequence of P.

Unfortunately, because of the undecidability of the validity problem of first order logic, there is no algorithm which takes an arbitrary A as whether A is or not a logical consequence of P. If A is not a logical consequence, it may loop forever.

12.7.2 Negation as Failure Rule

It is clear from above discussion that, in practice the application of the CWA is generally restricted to those A ∈ BP whose attempted proofs fail finitely. Let us make this idea precise.

For a definite program P, the SLD finite failure set of P is the set of all A ∈ BP for which there exist a finitely failed SLD-free, which is finite and contains no success branches.

To rule out this problem let us see another non-monotonic inference rule, called the *negation as failure rule*. This rule, first studied in detail by Drake is also used to infer negative information. It states that if A is in the SLD finite failure set of P, then infer UA. Since the SLD finite failure set is subset of the complement of the success set, we see that the negation as failure is less powerful than the CWA.

EXERCISE

- Convert the following formula of predicate logic into clause form.
 $\forall x [\forall y P(x, y) \rightarrow \forall y (Q(x, y) \rightarrow R(x, y))]$
- Making use of SLD resolution prove that the following argument is valid.
 Whosoever can read is literate.
 Tribals are not literate.
 Some tribals are intelligent.
 Therefore, some who are intelligent can't read.
- Given the relation.
 Father (x, y) x is the father of y
 Mother (x, y) x is the mother of y
 Female (x) x is the female
 Male (x) x is male
 Define the relation of the following.
 - Sibling
 - Sister
 - Grand son
 - First cousin
 - Descendent
- Write short notes on:
 - SLD resolution
 - Negation as failure extension
- Discuss the concept of logic and its use in programming languages:
- Write short notes on:
 - Predicate
 - Universe of discourse
 - The universal quantifier
 - Existential quantifier
 - Clauses
- What is unification and why it is required?
- Write the unification algorithm and also justify each step with an example:
- What do you understand by negative information and how it is useful?
- Write the short notes on:
 - Closed world assumption
 - Negation as failure rule

Data-flow Languages

13.1 INTRODUCTION TO DATA-FLOW LANGUAGES

Data-Flow languages derived from a base paradigm of the data-flow graph—that is, the modeling of a program as a set of operator nodes interconnected by set of data carrying arcs. Under this paradigm there is no current operation, and each operator is free to execute when its data arrives. The interpretation of the language is naturally concurrent, and the concurrency is inherently fine-grained.

The data-flow model has the single-assignment property. Values are data tokens that are transported from their producing node to the node that consumes them; there is no concept of a variable with a state that can be arbitrarily updated at a later time.

In data-flow, identifiers may be used to name these data tokens. Such identifiers are thus either undefined (not yet produced) or carry a single unique value; they can not be updated.

There is strong mutual influence between data-flow and functional language appears. This is probably due to their conterprary development, the sharing of certain fundamental properties that stood in contrast to the popular imperative language of their time, and similar problems of efficient implementation shared by both language classes.

Most modern data-flow languages do not have their semantics defined in terms of program graphs. There are fundamental limitations of such a strict definition of data-flow:

1. Certain programming language facilities that have proved to be useful (such as iteration) are not easily represented in terms of a static data-flow graph.
2. The efficient implementation of certain operations, articulary those associated with creating and manipulatin large data structures, is fundamentally difficult in pure data-flow environment.

13.2 DATA-FLOW COMPUTERS

Data-flow computers are radically different from the majority of parallel computers:

- An instructions is ready for execution when its operands are available.
- All instructions are function (that is there are no side effects).
- There is no concept of control flow, and as a result there is no need for a program counter.

- Information is transmitted rather than stored in memory.
- As there are no global shared resources, there are no bottlenecks caused by resource allocation. (Data-flow computer are not, however, without their own very significant resource allocation problems).

13.3 FEATURES OF DATA-FLOW LANGUAGES

Data-flow languages have following features:

1. Freedom from side effects,
2. Locality of effect.
3. Equivalence of instruction scheduling with data dependencies.
4. Single-assignment semantics.
5. Data-flow languages have unusual notation for iterations because of feature 1 and 4.
6. There is a lack of history sensitivity in procedures. For comparison with conventional languages, a data-flow language generally has the following properties:
 - Declarations bind names to values on a once-only basis.
 - There is no flow of control on a statement-by-statement basis—statements are not executed in textual order, but rather according to data dependencies.
 - The principle unit of modularization is the function, which in common with purely functional language, is textually equivalent to the value it denotes.
 - Concurrency is implicit and therefore not highlighted in the language.
 - While the specific details of I/O vary from language to language, its provision is difficult since the lack of deterministic execution order either prevents sequential I/O or requires an explicit sequencing construct in the language.

Now we will see the brief introduction to some of the data flow languages.

13.4 TDEL (TEXTUAL DATA-FLOW LANGUAGES)

According to Kong-song wen, the conventional languages were not suitable for parallel processing, as they could express only limited amounts of parallelism and provided opportunities where nondeterminacy and deadlock could arise. Therefore he set about designing a textual language with data-flow schemas as its semantic basis and satisfying the following requirements:

- (i) A simple translation into data-flow schemas.
- (ii) Compile time check for deadlocks where possible.
- (iii) Semantics simple enough to be formalized.
- (iv) Features of stream-oriented computation.
- (v) Features for expressing a system of interconnected modules (or processes) that communicate by exchanging data through communication channels.

The result was called Textual Data-flow language (TFL). It was restricted to determinate problems, and its only data types were integers and Booleans. No data structures were provided.

In TDFL, a program consisted of a list of module definitions followed by a list of statements separated by semicolons. A module, if defined recursively, used remodule as its heading, otherwise, module was used. The interface to a module outlined the formal parameters by explicitly defining the input and output identifiers. Each interface was required to have at least one input and output identifiers. The type of identifier was specified when the name was followed by a type: either integer(int) or Boolean (bool). All identifiers in a module interface had to be typed, while those inside the module needed only to be typed when an ambiguity occurred.

TDF statements were an assignment, a module call, or a conditional statement (if statement). An assignment compiled with the single-assignment rule and was different from the normal use in that the identifier being defined was on the right. Iteration was not provided for two regions:

- (i) It did not have corresponding mathematical notation, and.
- (ii) It involved the update of an identifier and this did not conform to single assignment.

Let us see a single recursive module in TDFL to calculate the greatest common divisor using Euclid's algorithm.

```

Euclids:
rmodule (x:integer.y:integer; gcd :integer)
  if x=y
    then y->gcd
  else
    if x>y
      then
        x-y->z;
        Euclids (y,z)->gde
      else
        y-x->z;
        Euclids (x,z)->gcd
      end
    end
  end
mend

```

TDFL was important for later language designers, as it was the first effort to describe data-flow graph formally, and it provided a discussion of suitable translation techniques and implementation detail for streams and communicating modules.

13.5 LAU

During the second half of the 1970s, the computer structures group of OERA-CERT in France was working on a language and computer based on the static data-flow model. The group (comprising Cornte, Durriea, Gelly, Plas, and Syre) successfully completed the first multiprocessor data-flow machine in September 1979.

In a LAU program (shown below) the four main statements that could be used were simple assignment, parallel assignment (expand operator), decision (case operator) and iteration (loop operator).

A data production set (PDS) acted as an assignment statement, encapsulating a number of intermediate assignments necessary for the production of the final set of objects.

Let us see the following program in to LAU:

```

Loop FIBN
  OUT :x,y;
  LOCAL ;N;
  (STAR) : N=1; x=2;
            Y=1;
            FIBN= NEXT
  (NEXT) : CASE OLD N
            (OLD N= 10); FIBN = POWER;
            (ELSE) : X = OLD X+OLDY;
                        Y = OLD X;
                        N = OLD N +1;
            END CASE;
  (POWER) : Y = OLD Y **2;
            X = OLD X **2;
  STOP LOOP FIBN;
END LOOP FIBN;

```

13.6 LAPSE

When the early work on the design of the manchester data-flow machine was being carried out, no thought had been given to the suitability of its instruction set as the target of a language compiler. Therefore, John Glauert undertook to design a high-level language and compiler for his master degree. The result, based on pascal, had single assignment semantics and was called LAPSE.

13.7 REAL-TIME SYSTEM

Timesliness is the single most important aspect of a real-time system. These systems respond to a series of external inputs, which arrive in an unpredictable fashion. The real-time systems process these inputs, take appropriate decisions and also generate output necessary to control the peripherals connected to them. As defined by Donald Gilles "A real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time in which the result is produced."

It is essential that the timing constraints of the system are guaranteed to be met. Guaranteeing timing behaviour requires that the system be predictable.

The design of a real-time system must specify the timing requirements of the system and ensure that the system performance is both correct and timely. There are three types of time constraints:

1. HARD: A late response is incorrect and implies a system failure. An example of such a system is of medical equipment monitoring vital functions of a human body, where a late response would be considered as a failure.
2. Soft: Timeliness requirements are defined by using an average response time. If a single computation is late. It is not usually significant, although repeated late computation can result in system failures. An example of such a system includes airline reservation systems.
3. Firm: This is a combination of both hard and soft timeliness requirements. The computations has a shorter soft requirement and a longer hard requirement. For example, a patient ventilator must mechanically ventilate the patient a certain amount in a given time period. A few seconds' delay in the initiation of breath is allowed, but not more than that.

It is very important to understand the difference between the sequential programming, Concurrent programming and real-time programming. The following table compares some of the key features of real-time software systems with other conventional software systems.

Feature	Sequential Programming	Concurrent Programming	Real-time Programming
Execution	Predetermined order	Multiple sequential programming executing in Parallel	Usually composed of concurrent programs
Numeric results	Independent of programming execution speed	Generally dependent on program execution speed	Dependent on program execution speed
Examples	Accounting, Pay roll	Unix Operating system	An flight controller

13.7.1 Embedded Systems

"Embedded is a digital system, uses a microprocessor (usually) and runs software for some or all of its functions." This kind of system frequently used as controller & having following special characteristics:

- (i) Concurrency (several processes working at some time)
- (ii) Synchronization
- (iii) Generally used sensors and actuators (for inputs and outputs)

Embedded system do not provide standard computing services and normally exist as part of bigger system. A computerized washing machine is an example of an embedded system where the main system provides a non-computing feature(washing cloths) with the help of an embedded computer.

In most of the real-life applications, real time systems often work in an embedded scenario and most of the embedded systems have real-time processing needs. Such software is called Real-time Embedded software system.

Typical examples of embedded application include microwave ovens, washing machines, telecommunication equipment, etc.

13.7.2 Selection of Real-time Operating System

Most moderate-to-complex real-time system use a real-time operating system or RTOS. This is similar to a general purpose operating system and provides functions such as:

- (i) Interface of the underlying hardware.
- (ii) Task scheduling and Preemption.
- (iii) Memory management.
- (iv) I/O services.
- (v) Support for your processor of choice.
- (vi) Portability to new processors.
- (vii) Scalability to match varied application requirements.
- (viii) Multiprocessor support.
- (ix) Extended services such as network support.

Some of the areas where the RTOSs differ from conventional operating systems are:

- Scheduling policies.
- Support for the embedded disk less environment.
- Licensing arrangement and price.
- Development environment.

Some examples of RTOSs are :

1. Vxworks and psos by Wind River.
2. OSE by Enea.
3. Lynxos by Lynuxworks.
4. OS-g by microwave.

13.7.3 Linux and Real-time

Linux is a general-purpose operating system and is designed for average performance. It achieves this performance by a fair sharing of processor among the processes.

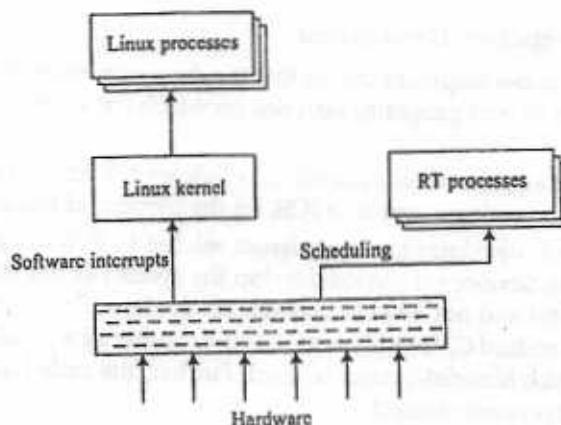


Fig. 13.1 Linux Real-time Environment

The Linux virtual memory implementation can swap out any process. Linux processes are heavy weight and there are significant overheads in process switching. The Linus kernel is non-preemptive, that is a process in the midst of a system call can not be preempted. Linux also disables interrupts in the critical sections of the kernel code. Due to these reasons, Linux in itself is not suitable to meet the timing requirements of a hard real time system. Real Time Linux (RT-Linux) addresses this problem by slipping in a small real-time kernel underneath Linux and running Linux as a process within its control. The control is transferred to the Linux process whenever there is no processor requirement from a real-time process. The above diagram depicts the operating model of RT-Linux.

Communication between the Linux and RT-Linux processes is generally achieved using the FIFO connections or shared memory segments.

13.7.4 Requirement of Real-time Embedded System Development

Real-time system usually have varied requirement:-

- Functional / business requirements of the system.
- Requirement to support and manage the special hardware of the system.
- Requirement to monitor the system, so as to have minimum down time of the system.

Such varied requirements generally necessitate the use of multiple development tools.

The following should be verified before the tools are finalized:

- (i) Integration of various tools with RTOS and also integration of various development tools amongst themselves (compatibility with each other).
- (ii) Some of these tools will run on the host machine and in such cases, their integration with the host operating system must be kept in mind.
- (iii) Run-time requirement of the executables generated by these tools.
- (iv) The memory requirements of the components should be well within the overall resource available to the system.

The list of development tools includes the modeling tools cross-platform development tools, programming language, IDE, configuration management tools, test coverage tools, memory leak detection tools, etc.

13.7.4.1 Cross-platform Development

The cross-platform development means that the development is done on a different platform (called the source or host program) than one on which the system will actually be run (called target platform.)

For example, a system is developed on windows NT and it is then downloaded onto a custom hardware running a separate RTOS, for the purpose of testing.

Cross-platform development brings issues related to differences in the source and target environment. The developers should develop the system as per the facilities available on the target environment and not what is available on the host. For example, the target RTOS may not provide all standard C/C++ libraries, which are otherwise available on a typical windows/Unix setup, so such libraries cannot be used. Further, the code has to be built (compiled and linked) for the target environment.

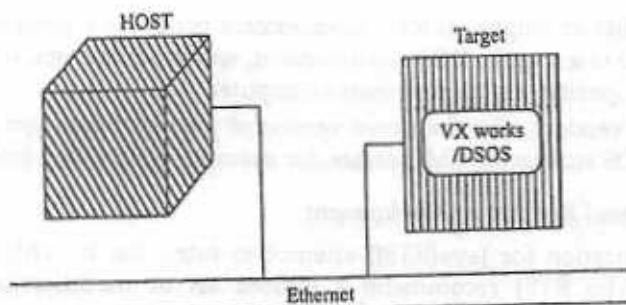


Fig. 13.2 Cross-platform development

13.7.4.2 Integrated Development Environment (IDE)

IDE includes language compilers, debuggers, editor, and a source code control system. Green Hills, for example, offers its MULTI environment for a number of RTOS platforms, including PSOS and Vxworks. MULTI is a multi-language embedded development environment featuring source-level debugging, execution Profiling, memory leak detection, graphical class browser, program builder, editor and source code control. MULTI can be hosted on PCs, Unix workstations and VAX/VMS systems and supports a number of target processors, including the 68k, 1960 and MSPS.

13.7.4.3 Choice of Programming Language

The choice of programming language is very important for real-time embedded software. The following factors influence the choice of language:

- (i) A language compiler should be available for the chosen RTOS and hardware architecture of the embedded system.
- (ii) Compilers should be available on multiple operating systems and microprocessors. This is particularly important if the processor or the RTOS needs to be changed in future.
- (iii) The language should allow direct hardware control without sacrificing the advantages of a high-level language.
- (iv) The language should provide memory management control such as dynamic and static memory allocation.
- (v) Real-time systems are increasingly being designed using object-oriented (oo) methodology and using a language that supports oo concepts is definitely helpful.

The languages that are typically used for embedded systems are Assembly Language, C, C++, Ada and Java.

Choosing to write code in Assembler should be done on a case-by-case basis. While code written in Assembler can be much faster, it is usually very processor-specific and less portable than a high-level language.

C is by far the most popular language and the language that maximizes application portability. C++ is used when real-time applications are developed using object-oriented methodology.

The availability of languages for a development project is limited to the languages that have been ported to a specific RTOS environment, and in some cases, to languages that have been ported to a specific target single-board computer.

Some RTOS vendors offer their own version of popular languages as they optimize the compiler for RTOS architecture. Microware, for example, has its own (compiler called Ultra).

13.7.4.4 Java and Real-time Development

Real-time specification for Java(RTSJ) attempt to bring the benefits of java to real-time programming. The RTSJ recommend a limited set of modifications to the language specifications and the Java virtual machine specifications. Implementation of RTSJ will be implemented on traditional real-time operating systems and will focus on embedded device target. Enterprise-level implementations and implementations on platforms such as Real-time Linux are also likely. The RTSJ specify enhancement to the Java language in the following seven areas:

- (i) Scheduling
- (ii) Memory management
- (iii) Synchronization
- (iv) Asynchronous Event Handling
- (v) Asynchronous Transfer of Control
- (vi) Asynchronous Thread Termination
- (vii) Physical Memory Access

IBM's Visual Age Micro Edition provides the environment, tools and runtime support for building complete, end-to-end embedded systems based on Java technology.

EXERCISE

1. Why data flow language are required discuss their main features.
2. What are data-flow computers.
3. Write short notes on:
 - (i) TDFL
 - (ii) LAU
 - (iii) LAPSE
4. What is real-time environment? Discuss the special characteristics of Embedded system environment?
5. "*Linux has good support for real time environment*" justify the statement.
6. Discuss the requirement of real-time embedded system development.
7. How can we select the choice of programming language?

CHAPTER 14

Programming Languages from Different Paradigms

14.1 JAVA

Java is an extension of C and C++. Originally known as Oak, it was designed to be a platform independent language. Java gained popularity when it provided support for "Applets", programs that could be embedded in web pages.

Three factors worked toward the sudden popularity of Java:

1. Web Applications.
2. Use of C syntax.
3. Growing frustration with C++.

The reason that should have contributed to the rise of Java:

1. Clean and simple language design.
2. Use of garbage collection.
3. Built-in support for event-driven programming and concurrency.

Java is not, by means, a perfect language. The main problems include:

1. Lack of support for parametric polymorphism.
2. Weak support for modules.
3. Overly complex.
4. Use of C syntax.

14.1.1 The Standard Java Packages

The predefined Java classes are grouped into a small number of standard packages. Let us see the overview of the major packages in the Java API:

- Java.applet: It is a small superclass of all applets, which are programs that run inside web browsers.
- Java.awt: Abstract windowing toolkit: contains classes for graphics and GUI interfaces.
- Java.io: Stream and file I/o classes.
- Java.awt: Abstract windowing toolkit; contains classes for graphics and GUI interfaces.
- Java.lang: Core language classes, like object, math, string, Thread, and Throwable (for defining and handling exceptions).

- Java.net: Classes for networking, including URL, Socket, Data gram socket, and Inet Address.
- Java.util: Some useful general classes, like Data, Random, vector, and stack.

14.1.2 Running Java Programs

A Java programs is written as a public class, which may be compiled and run as a standalone application, or else compiled and run and "applet".

Let us see following example:

```
Public Class Hello {
    Public static void main (string [ ] args) {
        System.out-Println ("Hello world");
    }
}
```

We have to store this program as the file Hello.java and following command should be issued:

```
Javac Hello.java
```

This step produces the executable file Hello.class. To run this program, the following command should be issued:

```
Java Hello
```

This step should produce the Hello world on the screen.

14.1.3 Java Design Goals

Java has following design goals:

- (i) *Portability*: Java supports internet-wide distribution, PC Unix and Mac.
- (ii) *Reliability*: Avoid program crashes and error messages
- (iii) *Safety*: Programmer may be malicious
- (iv) *Simplicity and familiarity*: Appeal to average programmer; less complex than C++
- (v) *Efficiency*: Important but secondary.

14.1.4 General Design Decision

Java has three basic design decision:

- (i) *Simplicity*: In Java everything is an object and all objects allocated memory from heap, accessed through pointers. There is no go to, no operator overloading and no automatic coercions.
- (ii) *Portability and network transfer*: Java supports byte code interpreter for portability across the network.
- (iii) *Reliability and safety*: Java is byte code language, having run-time type and bounds checks. Java supports automatic garbage collection for memory management. Java programmer compiles the code and only compiled code is transmitted on network. Receiver checks the code for safety and executes on interpreter (JVM).

14.1.5 Java Virtual Machine Architecture

The JVM is intended to provide a set of specifications that the Java language, compiler, and interpreter adhere to in order to ensure secure, portable programs and runtime environments. The JVM provides a strict set of rules that can be used by a developer to create an original implementation of an interpreter that runs Java code on any machine it is installed on. These rules require that the runtime interpreter include all of the following pieces:

- A set of byte code instructions similar to that CPU, which contains opcodes and operands, and their values and alignments.
- A set of registers that tracks the state of the program at a given time.
- A Java stack, which stores memory that is to be allocated to objects
- Memory areas for storage, which store constants and methods.

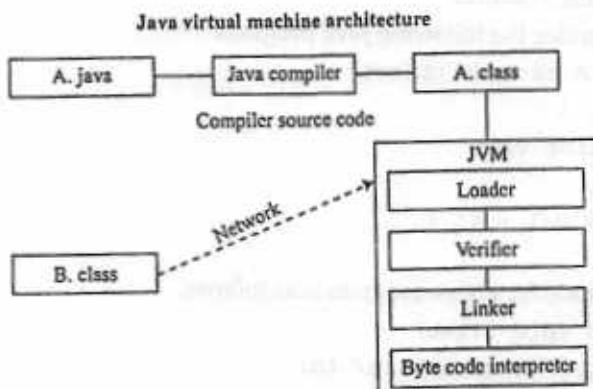


Fig. 14.1 JVM Architecture

Let us see various parts of JVM

1. Java Compiler:

The Java compiler not only checks that our syntax is correct in our source code that was created in the Java language, but it ensures that the code doesn't violate the language's safety rules. The compiler ensures that you have not made any errors, such as casting objects that are incompatible or using incorrect parameters.

Java compilers work similarly to compilers in C-type languages in that it takes intelligible source code converts it to code for machine to interpret. The difference is that the machine that the Java compiler compiles for is the Java virtual machine, and the code is not native machine code for our CPU, it is byte code for the JVM.

2. Java Interpreter:

Standard virtual machine interprets instructions. It performs run-time checks such as array bounds and it is possible to compile byte code class file to native code. Java programs can call native methods (typically functions written in C). Java interpreter performs many functions, some of which are performed solely for the purpose of the security of the system, and others that are performed as a part of the execution of the Java application, but require that security is enforced at each step.

3. *JVM memory Areas:*

In Java program more than one threads are allowed and each thread has own stack. All the threads share same heap.

We can view the memory area as follows.

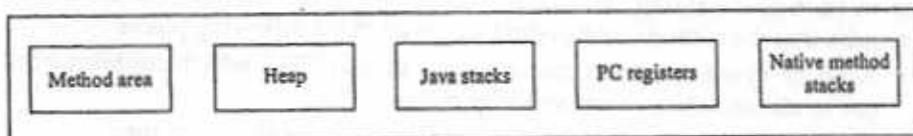


Fig. 14.2 JVM memory areas

4. *JVM uses stack machine:*

Let us consider the following Java program:

```
class A extends object{
    int i
    void (int val)
    {
        i = val + i;
    }
}
```

The byte code for above program is as follows:

```
Method void f(int)
    aload 0; object ref this
    iload 1; int val
    iconst 1
    iadd; add val + 1
    putfield # 4 <Field int i>
    return ;
    refers to cons pool
```

Now finally we will see the stack situations for above byte code as in Fig. 14.3.

5. *The Class Loader:*

The class loader is responsible for loading classes that are called while a Java program is executing and laying them out in memory in such a way that they are not able to interface with each other without explicit measures set forth in the language. It loads both local classes and foreign classes that have been determined clean by the byte code verifier.

6. *The Byte code Verifier:*

The Java Interpreter passes all incoming code to a byte code verifier. The responsibility of the bytecode verifier is to subject every piece of code that the interpreter passes it to a rigorous series of integrity tests. It performs a variety of tests that run from simple verification that the format of the line of code fragment is consistent with the language

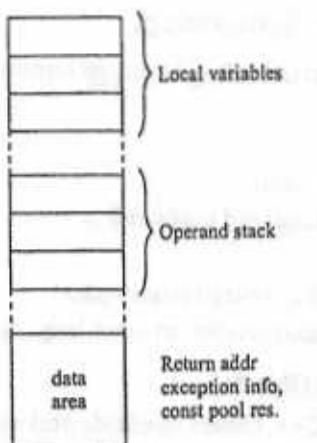


Fig. 14.3 JVM activation record

specification, to passing each line of code through a theorem prover to trap the following types of problems:

- Forged pointers
- Access restriction violations (private, public, or protected)
- Mismatching of object types
- Operand stack overflows and underflows
- Incorrect byte code parameters
- Illegal data conversion.

7. The Execution of Code:

Once the code has been loaded, a piece at a time by the interpreter. The interpreter can execute byte codes that have been coded for the Java virtual machine specification directly. It also provides a just-in-time compiler that compiles intermediate bytecode to native machine code at runtime for cases that you are willing to sacrifice portability to allow the byte code to run at full speed. Security can be implemented at runtime by coding traps and exception handless into our program.

14.1.6 Java Security

Basically there are three general security risks

1. Denial of service
2. Steal private information
3. Compromise our system (Erase files, introduce virus, ..)

Java can encounter these threats by two methods:

- (a) *Sand boxing*: Java runs programs in restricted environment. Java has a Java security manager, a special object that acts as access control "gate keeper".
- (b) *Code signing*: Java can use cryptography to establish origin of class file. This information can be used by security manager.

14.1.7 Java Languages Terminology

The Java Language is based on following language terminology:

1. Class, object
2. Field (data member)
3. Method (member function)
4. Static member (class field and methods)
5. This (self)
6. Package (set of classes in shared name space)
7. Native method (methods written in other language, typically in C).

14.1.7.1 Java Classes and Objects

The syntax is very similar to C++. Object has fields and methods and allocated on the heap. In case of Java objects are accessed through references and garbage collected.

Let us see following class syntax:

```
Class point{
    private int x;
    protected void set x (int y) { x = y; }
    public int get x( ) {return x;}
    Point (int xval) {x = xval;}//constructor
};
```

Java guarantees constructor call for each object, memory allocated through the constructor call only. In Java there is no explicit free function to deallocate the memory from the objects but objects are garbage collected automatically.

14.1.7.2 Encapsulation and Package

In Java every field, method belongs to the classes. Every class is part of some package and to use any class we have to import packaged in our program.

In general it may be clear from the following figure:

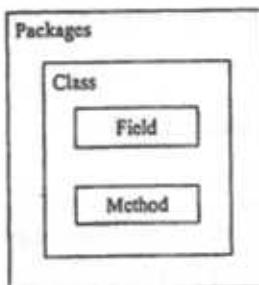
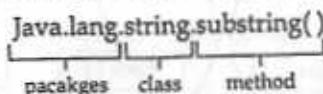


Fig. 14.4 Java package

There are four visibility distinction in Java like public, private, protected and package. Method can refer to:

- Private member of class it belongs to
- Non-private members of all classes in same package.
- Protected members of superclass (in different package)
- Public members of classes in visible packages.

Following example gives the clear distinction if package, class and method.



14.1.7.3 Java Inheritance

Java supports inheritance like small talk and C++. Subclass inherits from superclass but Java supports only single inheritance only.

Let us see an example, in which we inherit the properties of point class:

```

Class colorpoint extends point{
    // Additional fields and methods
    private Color c;
    protected void Setc (color d) {c = d;}
    // Define constructor
    Color point (int xval, Colore cval){
        super (xval); // call point constructor
        c = cval;} // intialize color point field
}
  
```

14.1.7.4 Java Data Types

Java supports primitive (non objects like integers, Booleans etc) and Reference types (classes, interfaces, arrays)

It is very clear from figure 14.5 that Reference types further can be divided into three subtypes:

- (i) user defined
- (ii) arrays
- (iii) Exception

14.1.7.5 Sequence Control in Java

Java support C like sequence control. In Java we can use iterative loops like for loop, do-while and while loop. We can use if-else clause as we do in C and also can use the break and continue statements.

Reference types:

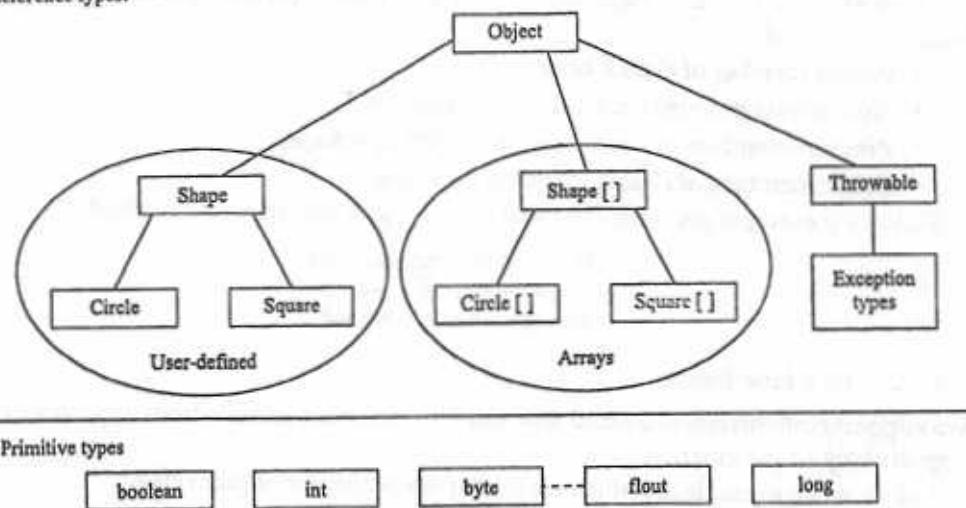


Fig.14.5 Classification of Java types

14.2 ML

ML is developed in Edinburgh in late 1970's as meta-language for automated theorem proving system. Designed by Robin Milnor, mike Gordon and chris wadsworth.

The important attributes of ML are:

- Primarily applicative
- Functions are first class values
- Statically scoped
- Static typing via type inference
- Polymorphic types
- Rich type system including support for ADT's.
- Support for exception handling.
- Automatic storage management via garbage collection.
- Incremental compiler supporting interactive program development

Let us see how we can run ML program:

To Launch ML type: sml

System responds with message saying in ML, and then “-” prompt and can load definitions from UNIX file by typing:

use "my file.sml,"

where my file sml is the name of our file. It should be in the same directly we were in when we typed sml. We can terminate the session by control-D.

14.2.1 Built-in data type in ML

ML supports unit, bool, int real, strings, characters

- Unit has only one value: ()
- bool includes true, false and operators: not, and also, or else
- int includes positive and negative: ..., -2, -1, -2, 0, 1, 2 ... supports +, -, *, /, div, mod, =, <, <=, >, > = operators.
- real of form 3.17, 2.4E 17 with +, -, *, /, <, <=, >, > =, log, exp, sin, arctan. Real no longer supports “=” because of dangers of round-off error. Instead, test the absolute value of the difference of the members to be less than some small tolerance.
- string of form "my string" -\t = tab, \n = newline. ML supports ^ (concatenation), length, substring where substring ("hello", 1, 3) -> "ell".
- char type is new in sm197. Write as # followed by string of length one. Thus # "b" is the character b, while "b" is the string of length one containing only the character b.

14.2.2 Type Declarations

In ML it is must to put types if want to be other than int functions if there are no other clues to type inference. In ML we can include type into if like.

- fun succ (x: real) = x + 1.0;

or

- fun succ x: real = x + 1.0

14.2.3 Compund Types

Three types of main compund types are allowed in ML.

- (i) *Tuples*: (17, "abc", true) : int * string * bool
- (ii) *Records*: {name = "ram", salary = 500000.99, rank = 1}: {name: string, salary: real, rand : int}
- (iii) *Lists*: ML supports many types of list, for example:
 - int list : [1, 2, 3]
 - string list: ["ab", "cd", "ef"]

Following operations are allowed on the lists:

- (i) length
- (ii) @- append, for example: [1, 2, 3] @ [4, 5, 6] = [1, 2, 3, 4, 5, 6,]
- (iii) :: - prefix (that is 1::x = [1, 2, 3, 4, 5, 6])

14.2.4 Sequence Control

ML provides sequence control by the following statements:

1. ML supports *Arithmetic expressions* like ~ (unary minus), *, /, div, mod and +, - additive operators. ML also provide the =, <, >, =, <, and > relationships.

2. ML supports Boolean and string expressions.

For example:

"xyz" ^ "abc" = xyzabc

\wedge is used for concatenation.

3. ML support *If expression*, but ML does not support any conditional statement. ML syntax is as follows:

```
if expression then
  true part
  else
    false part
```

Here else part is must with if expression and it is not optional as if is in C language.

4. ML supports *while expression* with following syntax:

```
while expression1 do expression2
when expression1 is true then expression2 is evaluated.
```

14.2.5 Function in ML

ML supports function with following syntax:

```
fun identifier-function (parameter) = expression;
```

Here fun is keyword, identifier-function is name of the function and expression is the expression which is to be evaluated.

```
- fun add (x: int, y: int) : int = x + y;
```

add is the function which takes two argument of integer type and adds them.

14.2.6 Exceptions in ML

ML also supports exceptions:

```
exception exception-identifier
```

and exception can be invoked as:

```
raise exception-identifier
```

We can handle the exception by using *handle* clause.

14.3 LISP

LISP was developed by John McCarthy in 1960. LISP is functional programming language which works on the basis of symbolic computation (math, logic). LISP provides a chance to think differently than C language. LISP uses the simple machine models and given attention to theoretical considerations like recursive function theory, and Lambda calculus. LISP supports various good ideas like program as data and garbage collection.

LISP supports *Atoms and pairs*.

- Atoms include numbers, indivisible "strings"

```
<atom> ::= <smb1> / <number>
<smb1> ::= <char> / <smb1> <char> / <smb1> <digit>
<num> ::= <digit> / <num> <digit>
```

"LISP is a scheme for representing the partial recursive functions of a certain of symbolic expression"

Basically lisp uses:

- Concept of computable (partial recursive) functions.
- Function expression: based on lambda calculus (lambda calculus is equivalent to Turing machine, but provide useful syntax and computation rules).

LISP provides following innovation in its Design:

- Expression-oriented*: like function expressions, conditional expressions and recursive functions.
- Abstract view of memory*: LISP support cells instead of array of numbered locations. LISP supports garbage collection.
- LISP supports program as data*
- Higher-order functions*.

14.3.1 LISP Memory Model

LISP supports cons cells:

Atoms and lists are represented by cell as follows:

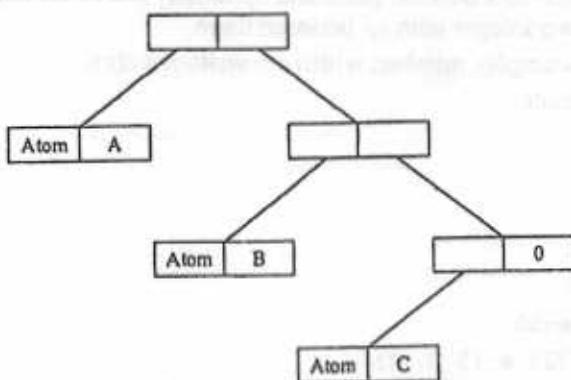


Fig. 14.6 LISP memory model

LISP do the memory management by using automatic garbage collection. At a given point in the execution of a program P, a memory location m is garbage if no continued execution of P from this point can access location m. LISP garbage collection program detects garbage during program execution and decision is made by run-time system, not program.

14.3.2 Symbols

A symbol is just a string of characters. There are restriction on what we can include in a symbol and what the first character can be, but as long as we stick to letters, digits, and hyphens, we will be safe. Let us see some examples:

```
a
b
c
foo
```

Let us see one sample program of LISP

```
> (set a 5) ; store a number as the value of a symbol
      5
> a           ; take the value of a symbol
```

In above program, things in *bold* after a > prompt are what we type to the LISP inter preter while other things are what the LISP interpreter prints back to us. The ; is LISP's comment character.

There are two special symbols, t and nil. The value of t is defined always to be t, and the value of nil is defined always to be nil.

14.3.3 Numbers

An integer is a string of digits optionally preceded by + or -. A real number looks like an integer, except that it has a decimal point and optionally can be written in scientific notation. A rational look like two integer with a / between them.

LISP supports complex number, which are written # c(r i).

Let us see example:

```
5
17
-34
+6
3.1415
1.722 e-15
# c(1.722 e-15 0.75)
```

The standard arithmetic functions are all available: +, -, *, /, floor, ceiling, mod, sin, tan, sqrt, exp, expt, and so forth.

14.3.4 Conses

A cons is just a two-field record. The fields are called "car" and "cdr", for historical reasons. Conses are easy to use

```
> (cons 45) ; Allocate a cons. Set the car to 4 and cdr to 5
      (4.5)
```

```
> (cons (cons 45) 6)
((4.5) .6)
> (car (cons 45))
4
> (cdr (cons 45))
5
```

14.3.5 LISTS

We can build many structures out of conses. Perhaps the simplest is a linked list: The car of each cons points to one of the elements of the list, and the cdr points either to another cons or to nil. We can create such a linked list with the list function:

```
> (list 4 5 6)
(4 5 6)
```

14.3.6 Function in LISP

We saw one function called list above. Here are some more:

```
> (+ 3 4 5 6) ; This function takes any number of arguments
18
> defun foo (xy) (+ x 4 5)); definig a function.
Foo ; calling a function.
> (foo 5 0)
10
> (defun fact (x); a recursive function
(if(> x 0)
(*x (fact (-x 1)))
1))
FACT
> (fact 5)
120
```

14.3.7 Printing

Some functions can cause output. The simplest one is print, which prints its argument and then returns it.

```
> (Print 3)
3
3
```

The first 3 above was Printed, The second was returned.

14.3.8 Binding

Binding is lexically scoped assignment. It happens to the variables in a function's parameter list whenever the function is called: the formal parameters are bound to the actual parameters for

the duration of the function call. You can bind variables anywhere in a program with the let special form, which looks like:

```
(let ( (var 1 val 1)
        (var 2 val 2)
        ...
      )
  body
)
```

Let binds var 1 to val 1, val 2 to val 2 and so forth) Then executes the statement in its body.

14.3.9 Dynamic Scoping

In LISP, dynamically scoped variables are called special variables. We can declare a special variable with the *defvar* special form. Here will see an example of lexically and dynamically scoped variables. In this example, The function check-regular references a regular variable. Since check-regular is lexically outside of the let which binds regular, check-regular returns the variable's global value.

```
> (setq regular 5)
5
> (defun check-regular () regular)
CHECK-REGULAR
> (Check- regular)
5
> (let (regular 6)) (check regular))
5
```

In this example the function check-special references a special (that is dynamically scoped) variable.

14.3.10 Arrays

The function *make-array* makes an array. The are function accesses its elements. All elements of an array are initially set to nil. For example.

```
> (make-array' (3 3))
2a ((NIL NIL NIL) ( NIL NIL NIL ) (NIL NIL NIL))
> (aref * 11)
NIL
> (make-array 4) ; 1 D array
# (NIL NIL NIL NIL)
array indices always start at 0.
```

14.3.11 Strings

A string is a sequence of characters between double quotes. LISP represents a string as a variable-length of characters. We can write a string which contains a double quote by preceding the quote with a backslash; a double backslash stands for a single backslash. For example:

```
"adcd" has 4 character
"\\"" has 1 characters
"\\\" has 1 characters.
```

Here are some functions for dealing with strings:

```
> (concatenate 'string "abcd" "efg")
"abcdefg"
> ((char "abc" 1)
#\\b ; LISP writes character preceded by #\
>(aref "abc" 1)
#\\ b ; remember, strings are really arrays.
```

14.3.12 Structures

LISP structures are analogous to C structs or pascal records. Here is example:

```
> (defstruct demo
  bar
  bazz
  quux)
DEMO
```

This example defines a data type called *demo* which is a structure containing 3 fields. It also defines 4 functions which operate on this data type : *make-demo* (makes a new object of type *demo*), *demo-bazz* (access the field), *demo-quux*.

14.3.13 Setf

Certain forms in LISP naturally define a memory location. For example, if the value of *x* is a structure of type *demo* then (*demo-bar x*) defines the *bar* field to the value of *x*. Or, if the value of *y* is a one-dimensional array, (*aref y 2*) defines the third element of *y*. The *setf* special form uses its first argument to define a place in memory, evaluates its second argument, and stores the resulting value in the resulting memory location.

For example:

```
(setf a 1)
(setf b 2)
(setf c 3)
```

14.3.14 Booleans and Conditionals

LISP provides a standard set of logical functions, for example and, or, and not. The *and* and *or* connectives are short-circuiting and will not evaluate any argument s to the right of the first one which evaluates to nil, while or will not evaluate.

LISP also provides several special forms for conditional execution. The simplest of these is if. The first argument of if determines whether the second or third argument will be executed.

```
> (if t 5 6 )
5
> (if nil 5 6 )
6
> (if 4 5 6 )
5
```

If you need to put more than one statement in the then or else clause of an if statement, we can use the *progn* special form.

14.3.15 Iteration

When we want to do something repeatedly, it is sometimes more natural to use iteration than recursion. This function uses do to print out the squares of the integers from start to end:

```
(defun show-squares (start end)
  (do ((i start (+ i 1)))
    ((> i end) (done))
    (format t "#~A ~A ~% " i (* i i))))
```

The simplest iteration construct in LISP is *loop*: a loop construct repeatedly its body until it hits a return special form.

For example:

```
> (setq a 4)
4
> (loop
  (setq a (+ a 1))
  (when (> a 7) (return)))
8
> (loop
  (setq a (- a 1))
  (when (< a 3) (return)))
NIL
```

The next simplest is dolist : dolist binds a variable to the elements of a list in order and stops when it hit the end of list.

```
> (do list (z' (a bc)) (print x))
A
B
C
NIL
```

Dotimes is like dolist except that it iterates over integers.

```
> ( dotimes (i 4) (print) (* i i ))
0
1
4
9
16
NIL
```

The most complicated iteration primitive is called do. A do statement looks like this:

```
> (do ((x 1 (+ x 1))
(y 1 ((* y 2)))
(> x 5) Y)
(print y)
(print 'working'))
1 WORKING
2 WORKING
4 WORKING
8 WORKING
16 WORKING
32
```

The first part of a do specifies what variables to bind, what their initial values are, and how to update them. The second part specifies a termination condition and a return value. The last part is body.

14.4 ADA

Now, in this section we will see some of the fundamental of the ada language. First let see how "Hello world" displayed on the screen in Ada language.

```
... program : Hello ... (1)
with Ada. Text- IO; ... (2)
procedure Hello is ... (3)
begin
    Ada. Text-IO.put ("Hello world"); ... (4)
    Ada. Text - IO. Now-line; ... (5)
end Hello; ... (6)
```

The output of this program should look like this:

```
Hello world!
```

A comment begins with a pair of hyphens; when compiler sees a pair of hyphens it simply ignores the rest of that line. An Ada program consists of a *procedure* defining a sequence of actions to be carried out. In this case there are two actions which are defined by the *statements* on line numbered 4 and 5; line 4 will display the message Hello world! and line 5 will start a new line on the screen.

14.4.1 Ada Data Types

Ada programs represent different types for real-world objects using different *data types*. The built-in data type are defined in a package called standard which is always available automatically in every Ada program.

Let us see some of the built-in data type in Ada!

1. *Integers*: The basic built-in data type in Ada is called integer. The exact range of numbers that type integer can cope with is implementation defined; the only guarantee we have is that it will at least be able to hold values in the range +32767. Integer types come with a full set of arithmetic operations, some of which we have already seen:

```
+ Addition
* Multiplication
rem Remainder
** Exponentiation
- Subtraction
/ Division
mod Modulus
abs Absolute value
```

2. *Subtypes*: Ada allows us to define *subtypes* of existing types which behave just like original type except that they have a restricted range of values:

```
Subtype Natural is Integer range 0.. Integer 'Last;
Subtype positive is Integer range 1.. Integer 'Last;
```

These declarations define two subtypes of type integer; Natural is an integer which cannot be less than zero, and positive is an integer which cannot be less than 1. In fact, both these subtypes are useful enough that they are already provided as built-in types declared in the package standard, and the exponentiation operator is defined so that it requires a Natural value or its right hand side.

3. *Derived types*: Sometimes it is useful to define a new type in terms of an existing type, like this:

```
type whole_Number is new Integer;
```

This defines a new type called whole_Number which has exactly the same properties as Integer. Whole-Integer is said to be derived from Integer.

4. *Real types*: Ada provides two ways of defining real types. *Flouting point* types have a, more or less unlimited range of values but are only accurate to a specified number of decimal places; *fixed point* types have a more limited range but are accurate to within a specified amount.
5. *Enumeration*: In many cases numbers are unsuitable for representing the types of data required by a program. Consider the days of the week as an example. We could use the numbers 0 to 6 or 1 to 7 to represent the days of the week, but they don't lend themselves to this naturally. The natural way to represent days of the week is the using their names (Monday, Tuesday, Wednesday and Sooon).

```
type Day-of-week is range 0..6; - or "mod 7" perhaps
Sun: Constant Day-of-week: =0;
Mon: Constant Day-of-week: =1;
Tue: Constant Day-of-week: =2;
Wed: Constant Day-of-week: =3;
Thu: Constant Day-of-week: =4;
Fri: Constant Day-of-week: =5;
Sat: Constant Day-of-week: =6;
```

6. *The Boolean type*: The type Boolean is let another one of Ada's standard data types, and is an enumeration type declared in type package standard as follows:

```
type Boolean is (False, True);
```

7. *The Character type*: The other standard type is character. This is an enumeration type whose are the list of characters defined by the International standardization organization (ISO) in standard ISO-8859. There are 256 possible characters in this character set which includes letters, digits, punctuation marks, mathematical symbols and characters like à and ß which are required in some European languages.

14.4.2 Composite Data Type

All the types described in the last section are used to represent individual values; They are known collectively as scalar types. However, in most real-life situations the data we deal can't be represented by simple numbers or enumerations of possible values. Most data is composite in nature, consisting of a collection of simpler data items. Let us see following composite data types in Ada:

1. *Record types*: Ada gather the component of the data type together into a single type known as a *record type*. Here is how we could define a record type to represent a date:

```
Type Date-Type is
  record
    Day: Day-Type;
    Month: Month-Type;
    Year: Year-Type;
  end record;
```

2. *Strings:* String is a predefined array type; an array is a collection of items of the same type.

Details: string (1..100); – a 100-character string

3. *Array types:*

Type Appt-Array is array (1..100) of Appointment-Type;

14.4.3 Statements

Let us see some of the statements in Ada for the sequence control:

1. *If statement:* Here we are discussing a example, in which one ask user whether it's morning or afternoon and then replies 'Good morning' or 'Good afternoon' as appropriate. Here's what it looks like;

```
with Ada.Text-IO; use Ada.Text-IO;
Procedure Greetings is
    Answer : character;
begin
    put("Is it morning (m) or afternoon (a)?");
    Get (Answer);
    If Answer='m' then
        Put-Line ("Good morning!");
    else
        Put-Line ("Good Afternoon!:");
    end if;
end Greeting;
```

2. *Assignment statements:* The assignment can do can be done by using operator:=, as can do in above example

```
if Answer = M then
    Answer := 'm' ;
```

3. *Compound Conditions:* We can use compound conditional statements:

```
if condition then
    .....
    .....
else if condition then
    .....
    .....
    .....
else
    .....
    .....
end if;
```

4. *The case statement:* When there are lot of alternative values of the some variable to be dealt with, it's generally more convenient ot use a *case* statement the an if staement. Here how the previous could be rewritten using case statement:

```
with Ada.Text-IO; use Ada.Text-IO;
Procedure Greetings is
    Answer: character;
begin
    Put ("Is it morning (m) or afternoon (a)?");
    Get (Answer);
    Case Answer is
        when 'M' \ 'M'-
            put-Line ("Good morning!");
        when 'A' \ 'a'-
            put-Line ("Good Afternoon!");
        when other =>
            put-Line ("Please type 'm' or 'a'!");
    end case;
end Greetings;
```

5. *Range tests:* In situations where we want any one of a consecutive range of values to select a particular choice, we can specify a range of values in a *case* statement using '...' to indicate the extent of the range.
6. *The null statement:* Case statement must cover all the possible values of the expression between *case* and *is*. This means that there has usually to be a *when others* clause, but sometimes we don't want to do anything if the value doesn't match any of the other selections. The solution is to use the *null statement*:

```
when other =>
    null;           ... do noting
```

8. *Loops:* At the moment we only get one chance to answer the question that program asks. It would be nicer if we were given more then are attempt. It can be done by using *loop* statement.

```
with Ada.Text-IO; use.Text-IO;
Procedure Greetings is
    Answer: character;
begin
    loop
        .....
        .....
        .....
    end loop;
end Greeting;
```

Quite a lot of the time we want to exit from a loop when a particular condition becomes true. We could do this using an if statement:

```
if This-IS-True then
    exit;
end if;
```

14.4.4 Exception Handling in Ada

Ada allows us to provide *exception handlers* to specify what happens if an exception occurs.

We can put an exception handler into any block of statement enclosed by *begin* and *end*, that a procedure body:

```
Procedure x is
begin
    .....
    .....
exception
    when some-Exception =>
        Do_This;
end x;
```

All allows us to define our own exceptions in Addition to the standard exception, like this;

```
Some thing_Wrong:exception;
```

We can raise an exception as follows!

```
raise something_wrong;
```

14.4.5 Memory Allocation in Ada

Ada supports *dynamic memory allocation explicitly*

```
x := new Apointment_type;
      .. create a new Appointment_Type record
```

Here *new* takes a free block of memory from a *storage pool* of available memory (often referred to as a heap)

Deallocation of memory is required when we don't need it any more. If we don't have deallocate it, it will be there ever if we don't have any way of accessing it any more and we will gradually run out of memory. The memory is only quarteed to be reclaimed when the access type goes out of scope. If the access type is declared in library package this won't happen until after the main program has ferminated, which is probably too late to be of any use.

One way to deal with this problem is to keep a *free list*; that is , a list of items which have been deleted and are free to be used again.

The problem with approach is that our memory usage will only ever increase. By using standard Ada procedure, *Ada.Unchecked_Deallocation*, we can tell ther system to deallocate the memory so that it can be reused by anything that needs it.

14.5 PROLOG

Prolog is logical programming language and it all about programming in logic. Complete prolog is works on facts, rules and queries. For example:

```

Fact: Socrates is a man
      man(socrates)

Rule: All men are mortal.
      mortal(x) :- man(x)

query: Is socrates mortal?
      mortal (socrates).
  
```

To work on the prolog, we should create our own "database" (Program) in any editor and save it as text only, with a pl extension. Here's the complete program:

```

man (socrates)
mortal(x) :- man(x)
  
```

Prolog is completely interactive and its interpreter is invoked by typing sicstus. Then sicstus. Then we load our program-consult ('mortal.pl'). Then finally we ask our question at the prompt:

```
-mortal (socrates).
```

Progol responds: - Yes.

14.5.1 A Simple Prolog Model

Prolog program is a system which has a database composed to two components:

- (i) FACTS: statement about true relations which hold between particular objects in the world. For example:

```
Parent ((adam, able): adam is a parent of able.
```

```
Parent (eve, able): eve is a parent of a able.
```

```
Male (adom): adom is male.
```

- (ii) RULES: Statements about true relations which hold between objects in the world which contain generalization, expressed through the use of variables. For example, the rule:

```
Father (x, y) :- parent (x, y), male (x).
```

might express:

for any x and any y, x is the father of y if x is a parent of y and x is male.

A prolog rule is called a *clause*. A clause has a head, a neck and a body:

<i>father (x, y)</i> :	<i>Parent (x, y)</i> ,	<i>male (x)</i>
<i>head</i>	<i>neck</i>	<i>body</i>

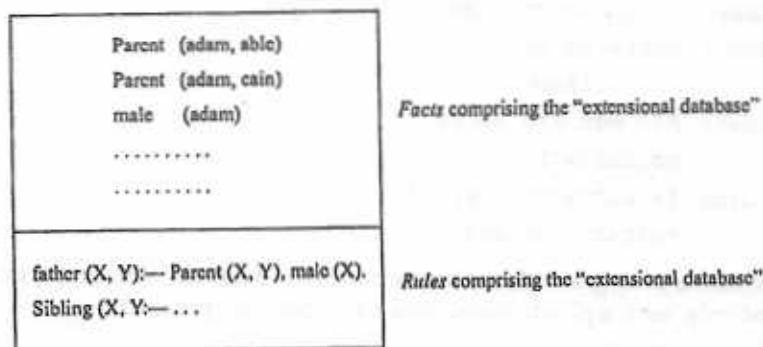
Here head is a rule's conclusion and the body is a rule's premise or condition. In prolog:

```

:- (as if)
, (as AND)
. (mark the end of input).
  
```

14.5.2 Prolog Database

Let us see an example of Prolog database:



The term extensional and intensional are borrowed from the language philosophy. Extension refers to whatever extends, that is "is quantifiable in space as well as in time".

Intension is an antonym of extension referring to "that class of existence which may be quantifiable in time but not in space but not in space".

Not intensional with a "t", which has to do with "will, volition, desire, plan, ..."

If we see the prolog session then it looks like:

```

?- assert (parent( adam, abel)).
Yes
?- assert (Parent ((eve, abel))).
Yes
/?- assert (male(adam)) .
Yes
/?- parent (adam, x)
X = abel
Yes
/?- parent (x, abel)
X = adam;
X = eve;
no
/?= adom;
no

```

14.5.3 Goal Satisfy

Here is an informal description of how prolog satisfies a goal (like `father(adam,x)`). Suppose the goal is G:

- If $G = P, Q$ then first P , carry variable bindings forward to Q , and then satisfy the Q .
- If $G = P; Q$ then satisfy P . If that fails, then try to satisfy Q .
- If $G = \text{not}(P)$ then try to satisfy P . If this, then try to satisfy Q .
- If G is a simple goal then look for a fact in the database that unifies with G look for a rule whose conclusion unifies with G and try to satisfy its body.

Goals can usually be posed with any of several combination of variables and constants:

- Parent (chain, able) - is chain Able's parent?
- Parent (chain, x) - who is a child of chain?
- Parent (x, chain) - who is chain a child of?
- Parent (x, y) - what two people have a parent/child relationship?

14.5.4 Terms in Prolog

The term is the basic data structure in Prolog. The term is to Prolog what the expression is to LISP. A term is either

- A constant, that is : john, 13, 3.1415, +, 'a constant'
- A variable, that is : x, var, -, -foo

A compound term can be thought of as a relation between one or more terms:

part_of (finger, hand)

and is written as:

- The relation name (called the principle functor) which must be a constant.
- An argument- An open parenthesis.
- The argument- one or more separated by commas.
- A closing Parenthesis.

The number of arguments of a compound term is called its arity.

Term	arity
f	0
f(a)	1
f(1, b)	2
f(g(a), b)	2

14.5.5 Unification in Prolog

Let us see some facts about Prolog unification:

- Unification is when two things "become one"
- Unification is kind of like assignment.
- Unification is kind of like equality in algebra.
- Unification is mostly like pattern matching. For example:
- loves (Ram, x) can unify with loves (Ram, sita) and in this process, x gets unified with sita.
- Any value can be unified with itself. For example:
Weather (sunny) = weather (sunny)

- (vi) A variable can be unified with another variable.
 $-x = y$
- (vii) A variable can be unified with ("instantiated to") any Prolog value.
 $-Topic = \text{weather}(\text{sunny})$
- (viii) Two different structures can be unified if their constituents can be unified.
 $-\text{mother}(\text{mary}, X) = \text{mother}(Y, \text{father}(Z))$
- (ix) A variable can be unified with a structure containing that same variable. This is usually a Bad Idea.
 $-x = f(x)$
- (x) The explicit unification operator is =
- (xi) Unification is symmetric:
 $\text{Mohan} = \text{father}(\text{Anu})$
means the same as
 $\text{Father}(\text{Anu}) = \text{Mohan}$.
- (xii) Most unification happens implicitly, as a result of parameter transmission.

14.5.6 Clauses as Cases

A predicate consists of multiple clauses, each of which represents a "case" for example:

```
grandson (x, y) :- son (x, z), son(z, y)
grandson (x, y) :- son (x, z), daughter(z, y)
abs (x, y) :- x < 0, y is -x
abs (x, x) :- x >= 0
```

14.5.7 Ordering of Clauses

- In Prolog clauses are always tried in order. For example:

```
buy(x) :- good(x).
buy(x) :- cheap(x).
cheap('Java 2 complete').
good ('Thinking in Java').
```

Now suppose goal is buy(x) then what will buy(x) choose first?

- In Prolog we always try to handle more specific cases (those having more variables instantiated) first. For example:

```
dislikes (john, bill)
dislikes(john, x) :- rish (x)
dislikes(x, y) :- loves (x, z), loves (z, 4)
```

- Some "actions" cannot be undone by backtracking over them:
 - Write, nl, assert, retract, consult.

- Do tests before we do undoable actions:

```
-take (A) :-
at (A, in-hand)
write ('you/re celready holding it!').
nl.
```

14.5.8 Back Tracking

We can do the backtracking through three predicates.

- (i) *Fail*: The fail/1 predicate is provided by prolog. When it is called, it causes the failure of the rule. And this will be for ever, nothing can change the statement of this predicate. A typical use of fail is a negation of a predicate. We can resume the fail with the shame:

```
goal (x) :- failure(x), !, fail.
```

```
goal(x)
```

failure (x) are the conditions that make goal (x) fail.

- (ii) *Cut*: Sometime it is desirable to selectively turn off back tracking. Prolog provides a predicate that performs this function. It is called the cut/1, represented by an exclamation (!).

The cut/1, effectively tells prolog to freeze all the decisions made so far in this predicate. That is, if required to backtrack, it will automatically fail without trying other.

We will write some simple predicates that illustrate the behaviour of the cut/1, first adding some data to back track over.

```
data (pentium iii).
```

```
data (athlon).
```

Here is the first test case. It has no cut/1 and will be used for comparison purposes.

```
Compare_cut_1(x) :-
```

```
data(x)
```

```
compare_cut_1('last chip').
```

This is the control case, which exhibits the normal behavior.

```
?- compare_cut_1(x), write(X), nl, fail.
```

```
Pentium III
```

```
athlon
```

```
last chip
```

```
no
```

- (iii) *Negation as failure*: In prolog, we can write a premise of the form not (R(X, Y, Z)). This becomes true if prolog is not able to justify R(x, y, z). In other words, for each relation prolog uses a "closed world" assumption — the rules and facts exhaustively determine all the numbers of the relation and anything that is not computable as being part of the negation is out of the relation. This approach is called "negation as failure".

- (iv) *Recursion In Prolog*: The recursion in any language is a function that can call itself until the goal has been succeeded. In prolog, recursion appears when a predicate contains a goal that refers to itself. We know that when a rule is called prolog creates a new query with new variables. So it makes no difference whether a rule calls another rule or calls itself.

14.5.9 Lists in Prolog

Lists are powerful data structures for holding and manipulating groups of things. In prolog, a list is simply a collection of terms. The terms can be any prolog data types, including structures

and other lists. Syntactically, a list is denoted by square brackets with the terms separated by commas. For example, a list of cities is represented as:

```
[Agra, Bareilly, Delhi, Nagpur]
```

14.5.10 Prolog Operators

For the arithmetic operators, prolog has already a list of predefined predicates. These are:

=, is, <, >, = <, > =, ==, =;=, /, *, +, -, mod, div

In prolog, the calculations are rewritten as in binary tree. That is to say That:

$y \times 5 + 10 \times x$ are written in prolog :+ (× (y, 5), × (10, x))

14.5.11 Prolog Arithmetic

Prolog use the infix operator 'is' to give the result of an arithmetic operation to a variable is

x is 3 + 4

Prolog responds

x = 7

Yes

When the goal operator 'is' is used the variable on the right must have been unified to number before. Prolog is not oriented calculations so' after a certain point, he approximates the calculations and he don't answer the exact number but his approximation, for example:

```
?- x is 1000 + 0001
```

```
x = 1000
```

Yes

In prolog and in any language, a recursive definition always has at least two parts. A first fact that out like a stopping has at least two a rule that call simplified. At each level condition and a rule that call itself simplified. At each level the first fact is checked. If the fact is true then the recursion ends. If not the recursion continue.

A recursive rule must never call itself with the same arguments! If that happens then the program will never end.

Let us discuss a example of prolog program to calculate the factorial recursively.

```
factorial (0).
factorial (x, y) :-
    x1 is x-1,
    factorial (x1, z),
    Y is z * x, !.
```

Now if we enter:

```
?- factorial (5, x).
```

```
X=120
```

Yes.

Index

A

Abstract Classes 257
Abstract data Types 154, 251, 252
Abstraction 8, 11, 153, 251
Access function of multiple-dimensional array 137
Access to Non-Local Names 216
Access value 120
Activation of the Procedure 206
Activation Record 206
Actual Parameters 157
Ada 6, 327
Ada Scope Rules 161
Aillasing 113
Algol 6
Alpha-Conversion 280
Ambiguous Grammar 38
Ambiguity 77, 78
Analysis 60
Anonymous Data ITEMS 112
Arithmetic operations 123
Arrays 127
Arrays of Arrays 138
Arrays of records 146
Arrays of structures 146
Assembler 19
Assembly Languages 6
Assignment 107
Associativity 172
Automatic Garbage Collection 241
Automation 8
Axiomatic Semantics 27

B

Back Ends 61
Backtracking 76, 337
Bags 140
Basic 6
Basic Types 106
Beta-Reduction 279
Binding of FPLS 271
Block structure 217
Block structure languages 156
Blocks 155
BNF Notation 30
Boolean Types 120
Bottom up parsing 72, 95
Bound variables 279
British standard for EBNF 38
Building Regular Expression 48

C

C 6
C++ 259
Call by refrence 223
Call by value 223
Call Sequence 210
Call-by constant value 225
Call-by Result 225
Call-by value-Result 225
Call-by-Name 224
Case statement 175, 177
Casting 115
Catching the Exceptions 201

- Categorization of Types 106
 C-Dynamic arrays 132
 Character strings 142
 Character Types 119
 Characteristics of programming Language 10
 Checked Exception 200
 Chronological classification 7
 Church-Rosser Theorem-I 282
 Class 253
 Classification of programming Languages 5
 Clause 293
 Code generation 61, 67
 Code optimization 61, 66
 Coercion 115
 Comments 17
 Compaction and Reuse 230
 Compiler passes 69,
 Compilers 19
 Compilers Design 58, 59
 Components of a Compiler 59
 Compound Types 106
 Computer Architecture 13
 Concept of Binding 23
 Concepts of FPLS 272
 Concurrency 15
 Concurrent programming 186
 Constants 109
 Constrained arrays 132
 Constructor in C++ 260
 Context Sensitivity 41
 Context-Free Grammars 31, 42, 77
 Context-sensitive Grammar 42
 Contextual programming Level 16
 Contextual structure 17
 Control variable 180
 Conversion 115
 Copy Restore 224
 Cost of Use 12
 Cross platform Development 308
 C-Scope Rules 159
 Current Environment pointer (CEP) 213
 Current Instruction pointer (CIP) 213
 Curried Functions 274
D
 Dangling pointers and Garbage 123, 234
 Data base query Languages 2, 5
 Data Declaration 107
 Data Types 105
 Data-Flow Computers 302
 Data-Flow Graph 302
 Data-Flow Languages 302
 Decimal 119
 Declarative Languages 267
 Declarative programming 8
 Deep Access 220, 221
 Defence in Depth 8
 Definite Clauses 291
 Definite Programs 293
 Denotational Semantics 27
 Denotational Semantics of Program 47
 Derivations 30, 45, 47
 Design Issues 11
 Destructors in C++ 260
 Deterministic Finite Automata 52
 Discrete types 106
 Documentation 11
 Dynamic memory management 232
 Dynamic Scoping 219
E
 EIFFEL 263
 Elementary Data Types 104
 Elimination of left Recursion 79
 Embedded system 306
 Encapsulation 153, 255
 Encapsulation By Subprograms 167
 Enumerated Data Types 140
 Error handler 60, 68
 Eta-Conversion 281
 Evaluation of Programming Languages 13
 Exception Handling 196

- Exception Handling in java 199
 Exception Throwing 202
 Explicit Return 234
 Expression Sequence Control 170
 Expressiveness 10
 Extended Transition Function 55
 Extensions to BNF 36
- F**
 Features of Imperative languages 23
 Finite Automata 51
 First and Follow 87
 Fixed count loop 179
 Flexible arrys 133
 Floating point Types 118
 Formal parameters 157
 Fortran 6
 Fortran Memory Management 244
 Free Variables 279
 Front Ends 61
 Functional Abstraction 275
 Functional Programming 267
 Functional Programming Languages 2, 3
- G**
 Garbage Collection 234, 235, 237, 238, 239
 Generality 10
 Global and Local Data 110
 Global refrencing environment 214
 Growth of Imperative-Programming Languages
 Grammars 28
- H**
 Handle 95
 Hardware Computer 15
 High Level Languages 6
 Higher Order Functions 273
 HOPE 3.
- I**
 Identifier 16
 If statement 175
 Imperative Language 2
 Implementation of character string Types 143
- Importance of FP 267
 Importance of Imperative Languages 21
 In heritance 256
 In heritance in C++ 259
 Indirect Left Recursion 77
 Information Hiding 7, 8, 154
 Initialisation 108
 Insecurities of pointers 123
 Integer 117
 Intermediate Code generation 61, 65
 Interpreter 20
 Interrupts 187
 Intial allocation 229
 Introduction to Programming Languages 1
 Introduction to Regular Expression 48
 Iterative statements 179
- J**
 JAVA 28, 264, 311
 Java class and objects 316
 Java data Types 317
 Java Inheritance 317
 Java Languages Terminology 316
 Java Memory Management 244
 Java Script Memory Management 244
 Java Security 315
 Java Threads 187, 194
 Java virtual Machine 28, 313, 314
 JVM memory Areas 314
- K**
 Keywords 16
- L**
 Labeling 9
 Lambda Abstraction 278
 Lambda Calculus 276
 Language design time 23
 Language Generation 8
 Language Implementation time 23
 Language of DFA 55
 Language processing 15
 LAPSE 305
 LAU 304

- Lazy Evaluation 275
- Left factoring 77, 81
- Left Most Derivations 32
- Left Recursion 77
- Left-Liner 43
- Lexical Analysis 60, 62
- Lexical Analyzer 62
- Lexical Scope Rules 285
- Lexical structure Level 16
- Link Time 23
- LISP 3, 320
 - Lisp memory management 244
 - LISP memory model 321
- List 138
- LISTS 323
- Literals 17
- LL(1) Grammar 91
- Load Time 23
- Local Environment 214
- Localized cost 9
- Logical Programming Languages 2
- Logical Programming Languages 287
- LOOPS 3
- LR parsers 99
- LR Parsing Algorithm 100
- LR(x) parsers 99

- M**
- Machine Independence 12
- Machine Languages 6
- Manifest Interface 9
- Mathematical Function 268
- Memory allocation strategies 230
- Memory Leaks 240
- Memory Management 227
- Memory management in C 243
- Memory management in C++ 243
- Memory Management in COBOL 243
- Memory management in the heap 233
- Memory management of ALGOL 242
- Memory management of BASTC 242
- Memory management phases 229
- Message 254
- ML 3
- ML 318
- ML memory management 244
- Modifiability and maintenance 11
- Modul-3 Memory Management 245
- Modular Programming 7, 14, 248, 249
- Module-3 263
- Modules 165
- Monitors 191, 187
- Monitors in java 193
- M-Table 89
- Multi Dimensional arrays 134
- Multipass Compilers 69, 70
- Multiple Assignments 108
- Multiple Inheritance 256
- Mutual exclusion 187, 190

- N**
- Narrowing Conversion 173
- Natural Languages 1
- Negative as Failure Rule 300
- Negative Information 299
- Nested "If-else" statements 177
- Non-Declarative programming 8
- Non-Discrete Types 106
- Non-Local Environment 214
- Normal Form of Expression 281
- Normal order Reduction 282
- Numeric Data Types 117

- O**
- Object 253
- Object Oriented Languages 248, 249, 251
- Object-Oriented Languages 2, 3
- Object-Oriented methodology 15
- One Dimensional arrays 130
- One pass Compiler 69, 70
- Operational Semantics 27
- Operations on pointers 122
- Operator over loading 173
- Operators 17
- Ordering of Declaration 108
- Organization of Code Area 229

- Orthogonality 9, 10
 Overloading 113
- P**
 Parameter Transmission 222
 Parse Tree 32
 Parsers 63, 72
 Pascal 6
 Pascal Memory Management 245
 Pattern matching 175
 Peephole Optimization 67
 Perl Memory Management 245
 Pointers 120
 Pointers and Garbage Collector 238
 Polymorphism 255
 Polymorphism in C++ 261
 Polymorphism in EPLS 273
 Portability 9, 11
 Post-test loop 179, 185
 Precedence Rules 170
 Predefined referencing environment 215
 Pre-defined types 106
 Predicate logic 288, 298
 Predictive Parsing Table 89
 Preservation of Information 9
 Pre-test loop 179
 Primitive Data Types 116
 Principles of Programming Languages 8
 Procedural Programming 248
 Procedure calls 210
 Programmer-defined Types 106
 Programming by Demonstration 2, 5
 Programming Methodologies 14
 Programming Paradigm 1
 PROLOG 4, 288, 333
 Prolog Database 334
 Prolog Memory Management 245
 Protocol base variables 187
- R**
 Readability 11
 Real machine Architecture 20
 Real-Time operating system 307
- Real-Time System 305
 Records and structures 144
 Recovery 230
 Recursive Descent Parsing 82
 Recursive Function 284
 Reference Counting 234, 235, 241
 Referential Transparency 268
 Referencing Environment 214
 Regular Expression 48, 51
 Regular grammar 42, 43
 Regular Sets 51
 Reliability 11
 Renaming 113
 Resolution in Propositional Logic 296
 Right Sentential Form 95
 Right-Liner Grammar 43
 Routines 156
 Row major Order 135
 Rule-based Programming Languages 2, 4
 Run Time 24
 Runtime Exception 200, 201
- S**
 Scalar types 106
 Scanner 68
 SCHEME 3,
 Scheme Memory Management 245
 SCOPE of FPLS 271
 Scope Rules 159
 Scripts programming languages 2, 5
 Security 10
 Selection 175
 Semantic Analysis 60
 Semantic Structure Level 16
 Semantics of the program 26
 Semaphore 190
 Separators 17
 Sequence Control 169
 Sets 139
 Shallow Access 220, 222
 Shift Reduce Parsing 97, 99
 Short-circuit 174
 Simplicity 10

- Simplicity of Language 10, 11
Simula 262
Simula Memory Management 245
Single Pass Compiler 70
SLD-Resolution 295
Slices 139
Smalltalk Memory Management 246
Software Simulated Computers 16
Source Language 19
Stack and Sub Program Calls 206
Stack-based storage management 230
Static Allocation 230
Static Chain 217
Static Storage 66
Static Vs. Dynamic Binding 25
Storage allocation 61, 66
Structure of the Program 16
Structured Data Types 126
Structured Programming 7, 14
Sub-division of run-time memory 228
Sub-Programs 156
Symbol table manager 60, 64
Synchronization 306
Syntactic and Semantic Description 11
Syntactic Consistency 10
Syntactic Structure Level 16, 17
Syntax Analysis 60
Syntax Analyzer 63, 64
Syntax of Imperative Language 45
Syntax of the Lambda-Calculus 280
Syntax Vs. Semantics of Languages 26
Synthesis 60
- T**
Target Architecture of Compiler 20
Target Language 19
Terminating A Loop 185
- Textual Data-Flow Languages 303
Token 62
Top down parsing 72, 73
Top down predictive parsing 86
Transition Diagram 53
Transition Table 53
Translation Time 24
Translators 18
Two way selection 175
Type Checking 114, 123
Type Conversion 173
Type Inference 273
Types of Sequence Control 169
- U**
Unconstrained array 132
Unification 297
Unification Algorithm 298
Unification in Prolog 335
Union Types 150
Unrestricted Grammar 42
Unstructured programming 148
- V**
Variable Count loops 183
Variables 109
Variant Records 149
Vector 130
Virtual Machine 20
Virtual Machine Architecture 20
Visual Programming Languages 2, 5
- W**
Well-Formed Formula 289
Widening Conversion 173
Wirth's EBNF Notation 37
Writability 11