NEHRU COLLEGE OF ENGINEERING AND RESEARCH CENTRE
(NAAC Accredited)
(Approved by AICTE, Affiliated to APJ Abdul Kalam Technological University, Kerala)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

COURSE MATERIALS

CS 403 PROGRAMMING PARADIGMS

VISION OF THE INSTITUTION

To mould true citizens who are millennium leaders and catalysts of change through excellence in education.

MISSION OF THE INSTITUTION

NCERC is committed to transform itself into a center of excellence in Learning and Research in Engineering and Frontier Technology and to impart quality education to mould technically competent citizens with moral integrity, social commitment and ethical values.

We intend to facilitate our students to assimilate the latest technological know-how and to imbibe discipline, culture and spiritually, and to mould them in to technological giants, dedicated research scientists and intellectual leaders of the country who can spread the beams of light and happiness among the poor and the underprivileged.

ABOUT DEPARTMENT

- Established in: 2002
- Course offered : B.Tech in Computer Science and Engineering

  M.Tech in Computer Science and Engineering

  M.Tech in Cyber Security
- Approved by AICTE New Delhi and Accredited by NAAC
- Affiliated to the University of A P J Abdul Kalam Technological University.

DEPARTMENT VISION

Producing Highly Competent, Innovative and Ethical Computer Science and Engineering Professionals to facilitate continuous technological advancement.

DEPARTMENT MISSION

1. To Impart Quality Education by creative Teaching Learning Process
2. To Promote cutting-edge Research and Development Process to solve real world problems with emerging technologies.
3. To Inculcate Entrepreneurship Skills among Students.
4. To cultivate Moral and Ethical Values in their Profession.

PROGRAMME EDUCATIONAL OBJECTIVES

PEO1: Graduates will be able to Work and Contribute in the domains of Computer Science and Engineering through lifelong learning.

PEO2: Graduates will be able to Analyse, design and development of novel Software Packages, Web Services, System Tools and Components as per needs and specifications.

PEO3: Graduates will be able to demonstrate their ability to adapt to a rapidly changing environment by learning and applying new technologies.

PEO4: Graduates will be able to adopt ethical attitudes, exhibit effective communication skills, Teamworkand leadership qualities.

PROGRAM OUTCOMES (POS)

Engineering Graduates will be able to:

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES (PSO)

PSO1: Ability to Formulate and Simulate Innovative Ideas to provide software solutions for Real-time Problems and to investigate for its future scope.

PSO2: Ability to learn and apply various methodologies for facilitating development of high quality

System Software Tools and Efficient Web Design Models with a focus on performance optimization.

PSO3: Ability to inculcate the Knowledge for developing Codes and integrating hardware/software products in the domains of Big Data Analytics, Web Applications and Mobile Apps to create innovative career path and for the socially relevant issues.

COURSE OUTCOMES

CO1	To acquire the knowledge of Scope and binding of names in different programming languages. To analyze control flow structures in different programming languages.
CO2	To appraise data types in different programming languages.
CO3	To acquire the knowledge of Subroutines and Co- routines.
CO4	To appraise constructs in functional, logic and scripting languages.
CO5	To analyze object oriented constructs in different programming languages.
CO6	To Learn different concurrency constructs.

MAPPING OF COURSE OUTCOMES WITH PROGRAM OUTCOMES

	PO 1	**PO 2**	**PO 3**	**PO 4**	**PO 5**	**PO 6**	**PO 7**	**PO 8**	**PO 9**	**PO 10**	**PO 11**	**PO 12**
CO 1	2	2	3	-	-	-	-	-	-	-	-	
CO 2	3	3	3	-	3	-	-	-	-	-	-	2
CO 3	3	-	2	-	-	-	-	-	-	-	-	
CO 4	3	-	3	-	3	-	-	-	-	-	-	3
CO 5	3	-	2	-	-	-	-	-	-	-	-	
CO	2	-	3	-	3	-	-	-	-	-	-	2

6												

Note: H-Highly correlated=3, M-Medium correlated=2, L-Less correlated=1

MAPPING OF COURSE OUTCOMES WITH PROGRAM SPECIFIC OUTCOMES

	PSO 1	**PSO2**	**PSO 3**
CO1	3	3	-
CO2	3	-	-
CO3	3	-	-
CO4	3	-	-
CO5	3	-	-
CO6	3	3	-

SYLLABUS

Course code	Course Name	L-T-P Credits	Year of Introduction
CS403	PROGRAMMING PARADIGMS	3-0-0-3	2016

Course Objectives:

- To introduce the basic constructs that underlie all programming languages
- To introduce the basics of programming language design and implementation
- To introduce the organizational framework for learning new programming languages.

Syllabus:

Names, Scopes, and Bindings - Binding Time, Scope Rules, Storage Management, Overloading, Polymorphism; Control Flow - Expression Evaluation, Structured and Unstructured Flow, Non-determinacy; Data Types - Type Systems, Type Checking, Equality Testing and Assignment; Subroutines and Control Abstraction - Static and Dynamic Links, Calling Sequences, Parameter Passing, Exception Handling, Co-routines; Functional and Logic Languages; Data Abstraction and Object Orientation -Encapsulation, Inheritance, Dynamic Method Binding; Innovative features of Scripting Languages; Concurrency - Threads, Synchronization, Language-Level Mechanisms; Run-time program Management.

Expected Outcome:

The Students will be able to :

i. compare scope and binding of names in different programming languages
ii. analyze control flow structures in different programming languages
iii. appraise data types in different programming languages
iv. analyze different control abstraction mechanisms
v. appraise constructs in functional, logic and scripting languages
vi. analyze object oriented constructs in different programming languages
vii. compare different concurrency constructs
viii. interpret the concepts of run- time program management

Text book:

1. Scott M L, Programming Language Pragmatics, 3rd Edn., Morgan Kaufmann Publishers, 2009.

References:

1. David A Watt, Programming Language Design Concepts, Wiley Dreamtech, 2004
2. Ghezzi C and M. Jazayeri, Programming Language Concepts, 3rd Edn, Wiley.1997
3. Kenneth C Louden, Programming Languages: Principles and Practice, 3rd Edn., Cengage Learning, 2011.
4. Pratt T W, M V Zelkowitz, and T. V. Gopal, Programming Languages: Design and Implementation, 4th Edn., Pearson Education, 2001
5. R W Sebesta, Concepts of Programming Languages, 11th Edn., Pearson Education, 2015
6. Ravi Sethi, Programming Languages: Concepts & Constructs, 2nd Edn., Pearson Education, 2006
7. Tucker A B and R E Noonan, Programming Languages: Principles and Paradigms, 2nd Edn,McGraw Hill, 2006.

Course Plan			
Module	**Contents**	**Hours**	**End Sem. Exam Marks**
I	Names, Scopes and Bindings:- Names and Scopes, Binding Time, Scope Rules, Storage Management, Binding of Referencing Environments. Control Flow: - Expression Evaluation, Structured and Unstructured Flow, Sequencing, Selection, Iteration, Recursion, Non-determinacy.	7	15 %
II	Data Types:-Type Systems, Type Checking, Records and Variants, Arrays, Strings, Sets, Pointers and Recursive Types, Lists, Files and Input/Output, Equality Testing and Assignment.	7	15 %
FIRST INTERNAL EXAM			
III	Subroutines and Control Abstraction: - Static and Dynamic Links, Calling Sequences, Parameter Passing, Generic Subroutines and Modules, Exception Handling, Co-routines.	7	15 %
IV	Functional and Logic Languages:- Lambda Calculus, Overview of Scheme, Strictness and Lazy Evaluation, Streams and Monads, Higher-Order Functions, Logic Programming in Prolog, Limitations of Logic Programming.	7	15 %
SECOND INTERNAL EXAM			
V	Data Abstraction and Object Orientation:-Encapsulation, Inheritance, Constructors and Destructors, Aliasing, Overloading, Polymorphism, Dynamic Method Binding, Multiple Inheritance. Innovative features of Scripting Languages:-Scoping rules, String and Pattern Manipulation, Data Types, Object Orientation.	7	20 %
VI	Concurrency:- Threads, Synchronization. Run-time program Management:- Virtual Machines, Late Binding of Machine Code, Reflection, Symbolic Debugging, Performance Analysis.	7	20 %
END SEMESTER EXAM			

Question Paper Pattern (End semester exam)

1. There will be ***FOUR*** parts in the question paper – **A, B, C, D**
2. **Part A**
   a. **Total marks : 40**
   b. ***TEN*** questions, each have **4 marks**, covering **all the SIX modules** (***THREE*** questions from **modules I & II**; ***THREE*** questions from **modules III & IV**; ***FOUR*** questions from **modules V & VI**).
   All the TEN questions have to be answered.
3. **Part B**
   a. **Total marks : 18**
   b. ***THREE*** questions, each having **9 marks**. One question is from **module I**; one question is from **module II**; one question ***uniformly*** covers **modules I & II**.
   c. ***Any TWO*** questions have to be answered.
   d. Each question can have ***maximum THREE*** subparts.
4. **Part C**
   a. **Total marks : 18**
   b. ***THREE*** questions, each having **9 marks**. One question is from **module III**; one question is from **module IV**; one question ***uniformly*** covers **modules III & IV**.
   c. ***Any TWO*** questions have to be answered.
   d. Each question can have ***maximum THREE*** subparts.
5. **Part D**
   a. **Total marks : 24**
   b. ***THREE*** questions, each having **12 marks**. One question is from **module V**; one question is from **module VI**; one question ***uniformly*** covers **modules V & VI**.
   c. ***Any TWO*** questions have to be answered.
   d. Each question can have ***maximum THREE*** subparts.
6. There will be ***AT LEAST* 60%** analytical/numerical questions in all possible combinations of question choices.

QUESTION BANK

Q:NO:	QUESTIONS	CO	KL
	MODULE I		
1	What do you mean by Boxing and Unboxing?	CO1	K1
2	Explain Lazy evaluation.	CO1	K2
3	Explain Iterators.	CO1	K4
4	Explain about the foll: storage allocation mechanisms used	CO1	K5
5	Discuss about Expression evaluation ordering with examples	CO1	K2
6	Explain tail recursive function	CO1	K5
7	How a Case statement is internally implemented? What are the different search strategies that can be used?	CO1	K2
	MODULE II		
1	Distinguish between strict and loose name equivalence.	CO2	K1
2	Explain bit vector representation of sets	CO2	K2
3	Describe iterative deepening with DFS.	CO2	K4
4	Explain about Type checking.	CO2	K5
5	Differentiate between structural equivalence and named equivalence with examples	CO2	K2
6	Describe the meaning of Type. Why C is not a strongly typed language?	CO2	K4
7	Write notes on Records.	CO2	K5
8	Describe about Variant Records.	CO2	K2
9	Differentiate between enumeration and subrange data types.	CO2	K2
	MODULE III		
1	Explain abstraction.	CO3	K2
2	Explain the subroutine calling sequence.	CO3	K2

3	Describe the various tasks performed by prolog and epilog.	CO3	K2
4	Differentiate static and dynamic link with suitable examples.	CO3	K4
5	Describe how to maintain the static chain during a subroutine call.	CO3	K5
6	Describe the purpose of stack pointer and frame pointer registers.	CO3	K2
7	Distinguish between formal and actual parameters.	CO3	K2
8	Draw and explain the layout of stack frame.	CO3	K5
9	Explain the parameter passing mechanisms.	CO3	K2
	MODULE IV		
1	Explain the applications of functional programming languages.	CO4	K2
2	Describe the features of functional programming languages.	CO4	K2
3	Explain the concept of functional programming.	CO4	K4
4	Write short note on Lambda calculus.	CO4	K5
5	Describe the behavior of LISP/Scheme read-eval-print loop.	CO4	K1
6	Sketch the internal representation of LISP List: (A B C D) (A(BC)D(E(FG)))	CO4	K2
	MODULE V		
1	Explain the distinction between private, public and protected class members. What are the visibility rules in C++.	CO5	K2
2	Explain the importance of virtual methods for object closures (or),What is the use of Vtable?.	CO5	K4
3	Elaborate the characteristics of scripting languages.	CO5	K2
4	List the principal ways in which scripting	CO5	K5

	languages differ from conventional "systems" languages.		
5	Compare the approaches to object orientation taken by Perl 5, PHP 5 and Javascript	CO5	K2
6	Write notes on dynamic method binding in object oriented programming	CO5	K5
7	Explain data types supported by scripting languages	CO5	K2
8	Explain the features of any one scripting language.	CO5	K2
9	Explain how pattern matching is done in scripting languages..	CO5	K2
10	Compare the numeric types of popular scripting languages..	CO5	K2
	MODULE VI		
1	Explain the language support/constructs needed for thread implementation. Also explain thread architecture	CO6	K2
2	Explain the methods used for creating a new thread of control in concurrent programming?	CO6	K2
3	Explain about 'Late binding of machine code'?	CO6	K5
4	With a state diagram, explain states of a thread	CO6	K4
5	Explain about symbolic debugging	CO6	K3
6	How do binary and general semaphores differ?	CO6	K5
7	Explain about symbolic debugging.	CO6	K3

APPENDIX 1		
CONTENT BEYOND THE SYLLABUS		
S:NO;	**TOPIC**	**PAGE NO:**
1	Python programming paradigms	148

MODULE NOTES

MODULE - 1

Introduction to Programming paradigms:

Programming paradigms are a way to classify programming languages based on their features.

Languages can be classified into multiple paradigms.

Common programming paradigms include:

- **Imperative** which allows side effects,
  - **Procedural** which groups code into functions(eg: **C, FORTRAN, Pascal, COBOL,BASIC, Ada etc**)
  - **Object-oriented** which groups code together with the state the code modifies.(eg:**C++,C#, Java, Simula,Smalltalk**)
- **Declarative** which does not state the order in which operations execute,
  - **Functional** which disallows side effects(eg: **Lisp, Scheme, Haskell, Erlang, ML,APL,Miranda,etc)**
  - **Logic** which has a particular style of execution model coupled to a particular style of syntax and grammar(eg:**Prolog**)
- **Symbolic** programming which has a particular style of syntax and grammar

1. Names

- A name is a mnemonic character string used to represent something else. Names in most languages are identifiers (alphanumeric tokens), though certain other symbols, such as + or :=, can also be names.
- Names allow us to refer to variables, constants, operations, types, and so on using symbolic identifiers rather than low-level concepts like addresses.
- Names are also essential in the context of a second meaning of the word abstraction.

- In this second meaning, abstraction is a process by which the programmer associates a name with a potentially complicated program fragment, which can then be thought of in terms of its purpose or function, rather than in terms of how that function is achieved.
- By hiding irrelevant details, abstraction reduces conceptual complexity, making it possible for the programmer to focus on a manageable subset of the program text at any particular time.
- Names are control abstractions and data abstractions for program
- Subroutines are control abstractions: they allow the programmer to hide arbitrarily complicated code behind a simple interface.
- Classes are data abstractions: they allow the programmer to hide data representation details behind a (comparatively) simple set of operations.

2. Scope

- A scope is the context within a computer program in which a variable name or other identifier is valid and can be used, or within whicha declaration has effect.
- The scope of a binding is also known as the **visibility** of an entity.
- **Static scope**: Scoping follows the structure of the program. C is said to be *statically scoped.*
- **Dynamic scope**, where scoping follows the execution path.the languages, including APL, Snobol, and early versions of Lisp, are dynamically scoped: their bindings depend on the flow of execution at run time.

Example:

```
int i = 1;          //global variable

void printdata()
{
cout << i << endl;
}

int main ( )
{
int i = 2;
printdata();
return 0;
}
```

If static scoping used, the result will be 1

If dynamic scoping used, this would print out 2.

- The lexical scope of a variable's definition is resolved by searching its containing block or function, then if that fails searching the outer containing block, and so on.
- Whereas with dynamic scope the calling function is searched, then the function which called that calling functions, and so on, progressing up the call stack. Of course, in both rules, we first look for a local definition of a variable.

- Most modern languages use lexical scoping for variables and functions, though dynamic scoping is used in some languages, notably some dialects of Lisp, some "scripting" languages, and some template languages.Perl 5 offers both lexical and dynamic scoping.

3. Binding Time

- A **binding** is an association between two things, such as a name and the thing it names.
- **Binding time** is the time at which a binding is created or, more generally, the time at which any implementation decision is made to create a name ←→ entity binding.
- There are many **different times at which decisions may be bound:**
  - ✓ **Language design time**: the design of specific program constructs (syntax), primitive types, and meaning (semantics).
  - ✓ **Language implementation time**: fixation of implementation constants such as numeric precision, run-time memory sizes, max identifier name length, number and types of built-in exceptions, etc.
  - ✓ **Program writing time**: Programmers, of course, choose algorithms, data structures,and names.
  - ✓ **Compile time**: Compilers choose the mapping of high-level constructs to machinecode, including the layout of statically defined data in memory.
  - ✓ **Link time**: the time at which multiple object codes (machine code files) and libraries are combined into one executable
  - ✓ **Load time**: when the operating system loads the executable in memory
  - ✓ **Run time**: when a program executes

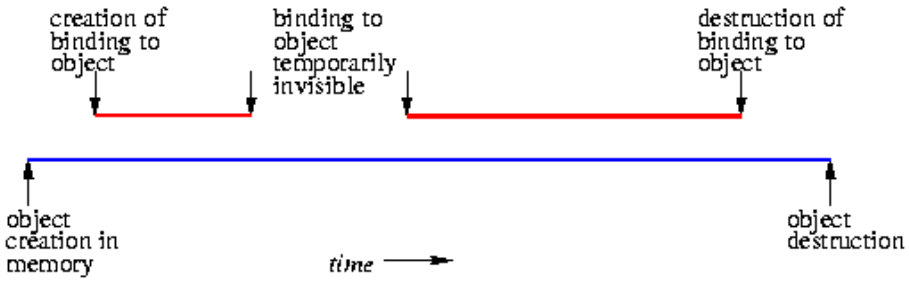
Language feature	*Binding time*
Syntax, e.g. `if (a>0) b:=a;` in C or `if a>0 then b:=a end if` in Ada	Language design
Keywords, e.g. `class` in C++ and Java	Language design
Reserved words, e.g. `main` in C and `writeln` in Pascal	Language design
Meaning of operators, e.g. + (add)	Language design
Primitive types, e.g. `float` and `struct` in C	Language design
Internal representation of literals, e.g. `3.1` and `"foo bar"`	Language implementation
The specific type of a variable in a C or Pascal declaration	Compile time
Storage allocation method for a variable	Language design, language implementation, and/or compile time
Linking calls to static library routines, e.g. `printf` in C	Linker
Merging multiple object codes into one executable	Linker
Loading executable in memory and adjusting absolute addresses	Loader (OS)
Nonstatic allocation of space for variable	Run time

The Effect of Binding Time

- The compiler performs a process called binding when an object is assigned to an object variable. The early binding (static binding) refers to compile time binding and late binding (dynamic

binding) refers to runtime binding. In general, early binding times are associated with greater efficiency, while later binding times are associated with greater flexibility.

- Binding lifetime: time between creation and destruction of binding to object.
    - Example: A pointer variable is set to the address of an object
    - Example: A formal argument is bound to an actual argument
- Object lifetime: time between creation and destruction ofan object.



- Key events in object lifetime
    - Object creation
    - Creation of bindings
    - References to variables, subroutines, types are made using bindings
    - Deactivation and reactivation of temporarily unusable bindings
    - Destruction of bindings
    - Destruction of objects

- Bindings are temporarily invisible when code is executed where thebinding (name ←→ object) is out of scope
- Memory leak: object never destroyed (binding to object may havebeen destroyed, rendering access impossible)
- Dangling reference: object destroyed before binding is destroyed
- Garbage collection prevents these allocation/deallocation problems

4. Scope Rules

The textual region of the program in which a binding is active is its scope. In most modern languages, the scope of a binding is determined statically, that is at compile time. A scope is the body of a module, class, subroutine, or structured control flow statement, sometimes called a block. In C family languages it would be delimited with {...} braces.

Statically scoped language: the scope of bindings is determined at compile time.
Dynamically scoped language: the scope of bindings is determined at run time.

➔ *Static Scoping*

The bindings between names and objects can be determined by examination of the program text.
Scope rules of a program language define the scope of variables and subroutines, which is the region of program text in which a name-to-object binding is usable.

- **EarlyBasic**:

- all variables are **global** and visible everywhere.

- **Fortran77**:
  - the scope of a local variable is limited to a subroutine;
  - the scope of a **global**variable is the whole program text unless it is hidden by a **local** variable declaration with thesame variable name.
- **Algol60**, **Pascal**, and **Ada**:
  - these languages allow nested subroutines definitions and adopt the **closest nested scope rule** with slight variations in implementation.
- **C**: one declares the variable **static**; A**lgol:** one declares it **own**.
  - A save-ed (static, own) variable has a lifetime that encompasses the entire execution of the program. Instead of a logically separate object for every invocation of the subroutine, the compiler creates a single object that retains its value from one invocation of the subroutine to the next.

➔ ***Nested Subroutines***

- ✓ The ability to nest subroutines inside each other, introduced in Algol 60, is a feature of many modern languages, including Pascal, Ada, ML, Python, Scheme, Common Lisp, and (to a limited extent) Fortran 90.
- ✓ Other languages, including C and its descendants, allow classes or other scopes to nest.
- ✓ But C based languages has no nested subroutines
- ✓ In Algol-family languages. Algol-style nesting gives rise to the closest nested scope rule for bindings from names to objects:
  - ✓ A name that is introduced in a declaration is known in the scope in which it is declared, and in each internally nested scope, unless it is hidden by another declaration of the same name in one or more nested scopes.
  - ✓ To find the object corresponding to a given use of aname, we look for a declaration with that name in the current, innermost scope.
  - ✓ If there is one, it defines the active binding for the name.
  - ✓ Otherwise, we look for a declaration in the immediately surrounding scope.
  - ✓ We continue outward, examining successively surrounding scopes, until we reach the outer nesting level of the program, where global objects are declared.
  - ✓ If no declaration is found at any level, then the program is in error.
- ✓ In Pascal, a procedure is defined using the **procedure** keyword. The general form of a procedure

  In this below example, procedure Nested scopes P2 is called only by P1, and need not be visible outside. It is therefore declared inside P1, limiting its scope (its region of visibility) to the portion of the programshown here.
- In a similar fashion, P4 is visible only within P1
- P3 is visible only within P2
- and F1 is visible only within P4.
- Under the standard rules for nested scopes, F1 could call P2 .,

- and P4 could call F1.,
- but P2 could not call F1.
- Though they are hidden from the rest of the program, nested subroutines are able to access the parameters and local variables (and other local objects) of the surrounding scope(s). In our example, P3 can name (and modify) A1, X, and A2, in addition to A3.
- Because P1 and F1 both declare local variables named X, the inner declaration hides the outer one within a portion of its scope.
- Uses of X in F1 refer to the inner X; uses of X in other regions of the code refer to the outer X.

```
procedure P1(A1:T1)
var X:real;
...
  procedure P2(A2:T2);
  ...
    procedure P3(A3:T3);
    ...
    begin
    (* body of P3: P3,A3,P2,A2,X of P1,P1,A1 are visible *)
    end;
  ...
  begin
  (* body of P2: P3,P2,A2,X of P1,P1,A1 are visible *)
  end;
  procedure P4(A4:T4);
  ...
    function F1(A5:T5):T6;
    var X:integer;
    ...
    begin
    (* body of F1: X of F1,F1,A5,P4,A4,P2,P1,A1 are visible *)
    end;
  ...
  begin
  (* body of P4: F1,P4,A4,P2,X of P1,P1,A1 are visible *)
  end;
...
begin
(* body of P1: X of P1,P1,A1,P2,P4 are visible *)
end
```

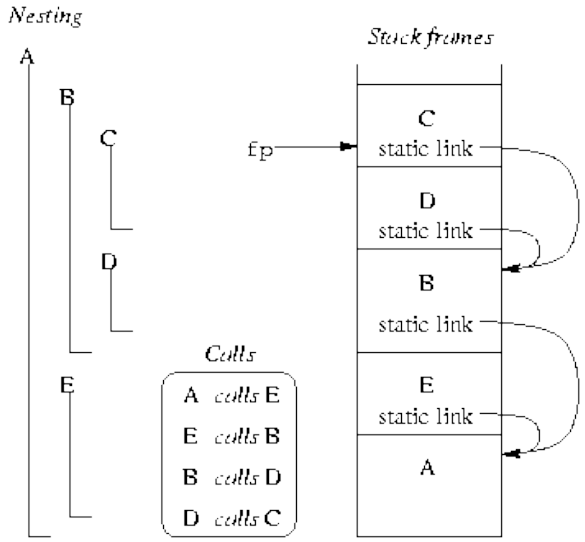
- To find the object referenced by a given name:
  - Look for a declaration in the current innermost scope
  - If there is none, look for a declaration in the immediately surrounding scope, etc.

Static Scope Implementation with static links

- Scope rules are designed so that we can only refer to variables that are alive: the variable must have been stored in the frame of a subroutine

- If a variable is not in the local scope, we are sure there is a frame for the surrounding scope somewhere below on the stack:
  - The current subroutine can only be called when it was visible
  - The current subroutine is visible only when the surrounding scope is active
- The simplest way in which to find the frames of surrounding scopes is to maintain a static link in each frame that points to the "parent frame".Each frame on the stack contains a static link pointing tothe frame of the static parent.

- Example static links.



- Subroutines C and D are declared nested in B.
- B is static parent of C and D.
- B and E are nested in A.
- A is static parent of B and E.
- The fp(frame pointer) points to the frame at the top of the stack.
- The static link in the frame points to the frame of the static parent.
- If a subroutine is declared at the outermost nesting level of the program, then its frame will have a null static link at run time. That is, A.

Static Chains

- How do we access non-local (global) objects?
- The static links form a static chain, which is a linked listof static parent frames
- When a subroutine at nesting level j has a reference to an object declared in a static parent at the surrounding scope nested at level k, then j-k static links forms a static chain that is traversed to get to the frame containing theobject.
- The compiler generates code to make these traversals over frames to reach non-local objects

- Subroutine A is at nesting level1 and C at nesting level 3
- When C accesses an object of A, 2 static links are traversed to get to A's frame thatcontains that object

Out of Scope

- Non-local objects can be hidden by local name-to-object bindings and the scope is said to have a "***hole***" in which the non-local binding is temporarily inactive but notdestroyed.
- Some languages, notably Ada and C++ use qualifiers or scope resolution operators to access non-local objectsthat are hidden

```
procedure P1;
var X:real;
  procedure P2;
  var X:integer
  begin
    ... (* X of P1 is hidden *)
  end;
begin
  ...
end
```

- P2 is nested in P1
- P1 has a local variable X
- P2 has a local variable X that hides X in P1
- When P2 is called, no extra code is executed to inactivate the binding of X to P1

➔ ***Dynamic Scoping***

- Scope rule: the "current" binding for a given name is the one encountered most recently during execution.
- Typically adopted in (early) functional languages that are interpreted
- Languages with Dynamic scoping:
  - APL, Snobol, TEX, early versions of Lisp, Perl
- Perl v5 allows you to choose scope method for each variableseparately
- With dynamic scope:
  - Name-to-object bindings cannot be determined by a compiler in general
  - Easy for interpreter to look up name-to-object binding in a stack ofdeclarations
- Sometimes useful:
  - Unix environment variables have dynamic scope
  - It makes it very easy for an interpreter to look up the meaning of a name
  - All that is required is a stack of declarations.
- Generally considered to be "a bad programming language feature"
  - Hard to keep track of active bindings when reading a program text
  - Most languages are now compiled, or a compiler/interpreter mix
  - Unfortunately, this simple implementation has a very high run-time cost, and experience indicates that dynamic scoping makes programs harder to understand.
  - The modern consensus seems to be that dynamic scoping is usually a bad idea.

```
1.  n : integer                  -- global declaration

2.  procedure first
3.      n := 1

4.  procedure second
5.      n : integer              -- local declaration
6.      first()

7.  n := 2
8.  if read_integer() > 0
9.      second()
10. else
11.     first()
12. write_integer(n)
```

Static versus dynamic scoping.

Program output depends on both scope rules and, in the case of dynamic scoping, a value read at run time.

- If **static scoping** is in effect, the above program Static vs. dynamic scoping
  - Prints a 1.
- If **dynamic scoping** is in effect, the output depends on the value read at line 8 at run time:
  - if **the input is positive**, the program **prints a 2**;
  - if **the input is negative or 0.,,** the program **prints a 1.**
- Why the difference? At issue is whether the assignment to the variablen at line 3 refers to the global variable declared at line 1 or to the local variabledeclared at line 5.
- **Static scope rulesrequire** that the reference resolve to the **closestlexically enclosing declaration**, namely the global n. Procedure first changes n to1, and line 12 prints this value.
- **Dynamic scope rules**, on the other hand, **require**that we choose the **most recent, active binding for n at run time.**

PROBLEMS WITH DYNAMIC SCOPING

- Run-time errors with dynamic scoping:With dynamic scoping, errors associated with the referencing environment maynot be detected until run time.
- For example, the declaration of local variable max_score in procedure foo accidentally redefines a global variable used by function scaled_score, which is then called from foo. Since the global max_score is an integer, while the local max_score is a floating-point number, dynamic semantic checks in at least some languages will result in a type clash message at run time.
- If the local max_score had been an integer, no error would have been detected, but the program would almost certainly have produced incorrect results.
- This sort of error can be very hard to find.
- In this example, function scaled_score probably does not do what the programmer intended: with dynamic scoping, max_score in scaled_score is bound to foo's local variable max_score after foo calls scaled_score, which was the most recent bindingduring execution.

```
max_score : integer        – – maximum possible score

function scaled_score(raw_score : integer) : real
    return raw_score / max_score * 100
...
procedure foo
    max_score : real := 0        – – highest percentage seen so far
    ...
    foreach student in class
        student.percent := scaled_score(student.points)
        if student.percent > max_score
            max_score := student.percent
```

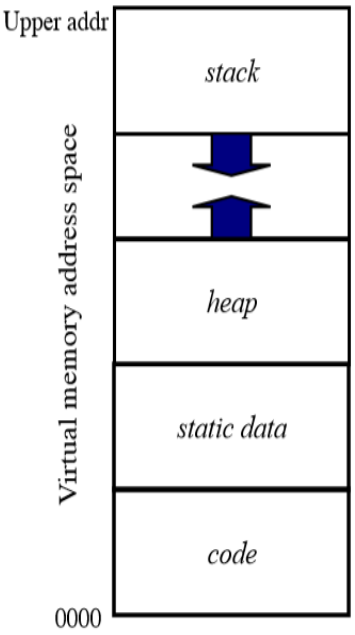
The problem with dynamic scoping.
Procedure scaled_score probably does not do what the programmer intended when dynamic scope rules allow procedure foo to change the meaning of max_score.

➔ ***Implementing Scope***

- To keep track of the names in a **statically scoped program**, a compiler relies on a data abstraction called a **symbol table**. In essence, the symbol table is a dictionary: it maps names to the information the compiler knows about them.
- In a language with **dynamic scoping**, an **interpreter** (or the output of a compiler) must perform operations analogous to symbol table insert and lookup at runtime. In principle, any organization used for a symbol table in a compiler could be used to track name-to-object bindings in an interpreter, and vice versa. In practice, implementations of dynamic scoping tend to adopt one of two specific organizations: **an association list or a central reference table.**

5. Storage Management

Objects (program data and code) have to be stored in memory during their lifetime. Object lifetimes generally correspond to one of three principal storage allocation mechanisms, used to manage the object's space. They are static allocation, stack allocation, and heap allocation.



- Program code is at the bottom of the memory region (code section)
  - The code section is protected from run-time modification by the OS
- Static data objects are stored in the static region
- Stack grows downward
- Heap grows upward

FIGURE: TYPICAL PROGRAM AND DATA LAYOUT IN MEMORY

(1) STATIC ALLOCATION

Static Objects have an absolute storage addressthat is retained throughout the execution of the program. (**Absolute addresses are also called real addresses and machine addresses. A fixed address in memory).**They are often allocated in protected, read-only memory. So that any attempt to write to them will cause a processor interrupt, allowing the operating system to announce a run-time error.**Advantage** of statically allocated object is the **fast access** due to absolute addressing of the object.

Examples of static objects are:

- ✓ Global variables are static objects
- ✓ Program code is statically allocated in most implementations of imperative languages.
- ✓ Variables that are local to single subroutine, but retain their values from one invocation to the next; their space is statically allocated.
- ✓ Numeric and string valued constants can be allocated statically.
- ✓ Run time tables produced by compilers (these tables are used for support of debugging, type checking, garbage collection, exception handling and other purposes) are statically allocated.
- ✓ static local variables in C
- ✓ In earlier versions of FORTRAN, recursion was not supported.As a result, there can never be more than one invocation of subroutine active. Thus the compiler may choose to use static allocation for local variables in that language. In languages with recursion, we can't use static allocation of local variables.

(2) STACK ALLOCATION

Static allocation does not work for local variables in potentially recursive subroutines. Every (recursive) subroutine call must haveseparate instantiations of local variables. So we use stack allocation.**Stack Objects**are allocated in last-in first-out(LIFO) order, usually in conjunction with subroutine calls and returns**.**

- ✓ Each instance of a subroutine at run time has a frame on the run-time stack (also called activation record).
- ✓ Compiler generates subroutine calling sequence to setup frame, call the routine, and to destroy the frame afterwards.
- ✓ Frame layouts vary between languages and implementations
- ✓ A frame pointer (fp) points to the frame of the currently active subroutine at run time (always topmost frame on stack)
- ✓ Subroutine arguments, local variables, and return values are accessed by constant address offsets from fp
- ✓ The stack pointer (sp) points to free space on the stack.
- ✓ Even in a language without recursion, it can be advantageous to use a stack for local variables, rather than allocate them statically.

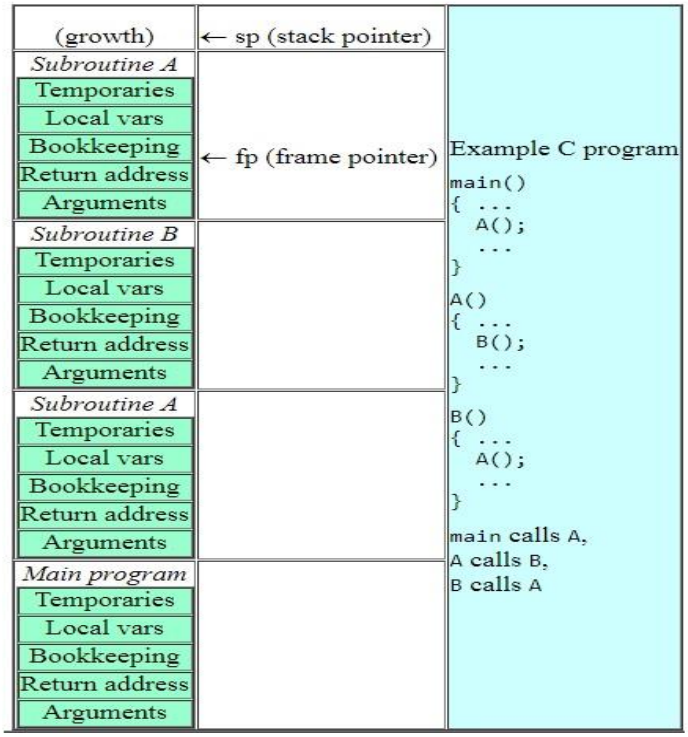


FIGURE: STACK BASED ALLOCATION OF SPACE FOR SUBROUTINES

(3) HEAP ALLOCATION

A heap is a region of storage in which sub-blocks can be allocated and de-allocated at arbitrary times. Heaps are required for the dynamically allocated pieces of linked data structures, and for objects like fully general character strings, lists, and sets, whose size may change as a result of an assignment

statement or other update operation.

Strategies to manage space in a heap

- The principal concerns are **speed and space**, and as usual there are tradeoffs between them.
- Space concerns can be further subdivided into issues of **internal and external fragmentation**

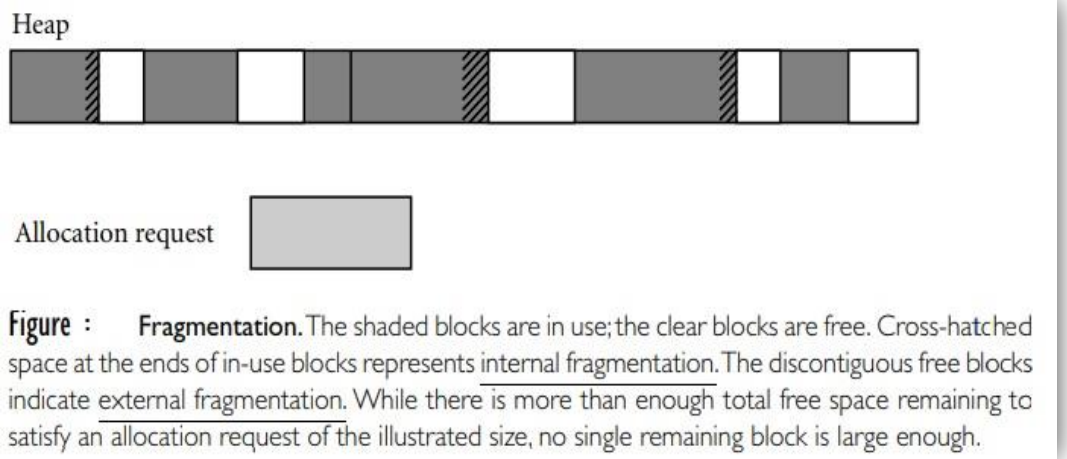


Figure : **Fragmentation.** The shaded blocks are in use; the clear blocks are free. Cross-hatched space at the ends of in-use blocks represents internal fragmentation. The discontiguous free blocks indicate external fragmentation. While there is more than enough total free space remaining to satisfy an allocation request of the illustrated size, no single remaining block is large enough.

Internal fragmentation:

It occurs when a storage-management algorithm allocates a block that is larger than required to hold a given object.The extra space is then unused.

External fragmentation:

It occurs when the blocks that have been assigned to active objects are scattered through the heap in such a way that the remaining, unused space is composed of multiple blocks: there may be quite a lot of free space, but no one piece of it may be large enough to satisfy some future request.

Many **storage-management algorithms** maintain a **single linked list—the free list**—of heap blocks not currently in use.

- Initially the list consists of a single block comprising the entire heap.
- At each allocation request the algorithm searches the list for a block of appropriate size.
- With a ***first fit algorithm*** we select the first block on the list that is large enough to satisfy the request.
- With a ***best fit algorithm*** we search the entire list to find the smallest block that is large enough to satisfy the request.
- In either case, if the chosen block is significantly larger than required, then we divide it in two and return the unneeded portion to the free list as a smaller block. (If the unneeded portion is below some minimum threshold in size, we may leave it in the allocated block as internal fragmentation.)
- When a block is de-allocated and returned to the free list, we check to see whether either or both of the physically adjacent blocks are free; if so, we coalesce them.
- Intuitively, one would expect a ***best fit algorithm to do a better job*** of reserving large

blocks for large requests. At the same time, ***it has higher allocation cost than a first fit algorithm***, because it must always search the entire list, and it tends to result in a larger number of very small "left-over" blocks.

- Which approach—first fit or best fit—results in lower external fragmentation depends on the distribution of size requests.

In any algorithm that maintains a single free list, the cost of allocation is linear in the number of free blocks. **To reduce this cost to a constant**, some storage management algorithms ***maintain separate free lists for blocks of different sizes***.

- ✓ Each request is rounded up to the next standard size (at the cost of internal fragmentation) and allocated from the appropriate list.
- ✓ In effect, the ***heap is divided into*** "***pools***," ***one for each standard size***.
- ✓ The division may be static or dynamic.

Two common mechanisms for dynamic pool adjustment are known as the **buddy system** and the **Fibonacci heap**.

- ✓ In **the buddy system**, the standard block sizes are powers of two. If a block of size $\mathbf{2^k}$ is needed, but none is available, a block of size $\mathbf{2^{k+1}}$ is split in two. One of the halves is used to satisfy the request; the other is placed on the $\boldsymbol{k^{th}}$ free list. When a block is deallocated, it is coalesced with its "buddy"—the other half of the split that created it—if that buddy is free.
- ✓ **Fibonacci heaps** are similar, but use Fibonacci numbers for the standard sizes, instead of powers of two. The algorithm is slightly more complex, but leads to slightly lower internal fragmentation, because the Fibonacci sequence grows more slowly than $\mathbf{2^k}$**.**
- ✓ The problem with external fragmentation is that the ability of the heap to satisfy requests may degrade over time. To eliminate external fragmentation, we must be prepared to ***compactthe heap***, by moving already-allocated blocks. This task is complicated by the need to find and update all outstanding references to a block that is being moved.

What happens when the user no longer needs the heap-allocated space?

- ✓ Manual de-allocation: The user issues a command like free or delete to return the space (C, Pascal). When a block (including any internal wasted space) is returned, it is coalesced, if possible, with any adjacent free blocks.
- ✓ Automatic de-allocation via garbage collection (Java, C#, Scheme, ML, Perl).
- ✓ Semi-automatic de-allocation using destructors (C++, Ada). The destructor is called automatically by the system, but the programmer writes the destructor code.

Poorly done manual de-allocation is a common programming error.

- If an object is de-allocated and subsequently used, we get a dangling reference.
- If an object is never de-allocated, we get a memory leak.

We can run out of heap space for at least three different reasons.

- What if there is not enough free space in total for the new request and all the currently allocated space is needed? Solution: Abort.

- What if we have several, non-contiguous free blocks, none of which are big enough for the new request, but in total they are big enough? This is called external fragmentation since the wasted space is outside (external to) all allocated blocks.
  Solution: Compactify.
- What if some of the allocated blocks are no longer accessible by the user, but haven't been returned?
  Solution: Garbage Collection

→GARBAGE COLLECTION

- A *garbage collection* algorithm is one that automatically de-allocates heap storage when it is no longer needed.
- There are two aspects to garbage collection: first, determining automatically what portions of heap allocated storage will (definitely) not be used in the future, and second making this unneeded storage available for reuse.

Disadvantages of Garbage Collection:

- Extra burden for the language implementer.
- When the garbage collector is running, machine resources are being consumed.
- For some programs the garbage collection overhead can be a significant portion of the total execution time.
- The programmer can easily tell when the storage is no longer needed, it is normally much quicker for the programmer to free it manually than to have a garbage collector do it.

6. Binding of Referencing Environments

What is Referencing Environment?

- It is the collection of all names that are visible in the statement.
- In a static-scoped language, referencing Environments is the local variables plus all of the visible variables in all of the enclosing scopes.
- The referencing environment of a statement is needed while that statement is being compiled, so code and data structures can be created to allow references to non-local variables in both static and dynamic scoped languages.
- A subprogram is active if its execution has begun but has not yet terminated.
- In a dynamic-scoped language, the referencing environment is the local variables plus all visible variables in all active subprograms.

Ex, Ada, **static-scoped language**

```
procedure Example is
   A, B : Integer;
   …
   procedure Sub1 is
      X, Y : Integer;
      begin      -- of Sub1
      …                          ← 1
      end        -- of Sub1
   procedure Sub2 is
      X : Integer;
      …
      procedure Sub3 is
         X : Integer;
         begin   -- of Sub3
         …                       ← 2
         end;    -- of Sub3
    begin  -- of Sub2
    …                            ← 3
    end;   { Sub2}
begin
   …                             ← 4
end;       {Example}
```

- The referencing environments of the indicated program points are as follows:

Point	***Referencing Environment***
1	X and Y of Sub1, A & B of Example
2	X of Sub3, (X of Sub2 is hidden), A and B of Example
3	X of Sub2, A and B of Example
4	A and B of Example

Ex, **dynamic-scoped language**

```
void sub1( )
{
  int a, b;
   …                    ← 1
}       /* end of sub1 */
void sub2( )
{
  int b, c;
   …                    ← 2
  sub1;
}       /* end of sub2 */
void main ( )
{
  int c, d;
   …                    ← 3
  sub2( );
}       /* end of main */
```

The referencing environments of the indicated program points are as follows:

Point	***Referencing Environment***
1	a and b of sub1, c of sub2, d of main
2	b and c of sub2, d of main
3	c and d of main

Binding of Referencing Environment:

There are mainly two types of binding of referencing environment

- Deep Binding
- Shallow Binding

Deep/shallow binding makes sense only when a procedure can be passed as an argument to a function.

- **Deep binding**
  - Binds the environment at the *time a procedure is passed as an argument*.
  - In the Example program: f2(**f3**);
  - It is the **earlybinding** (**static or lexical**) of the referencing environment of a subroutine.
  - The need for deep binding is sometimes referred to as ***the funarg problem in Lisp***

- **Shallow binding**
  - Binds the environment at the *time a procedure is actually called*.
  - In the Example program:f3 ( ); function call.
  - It is the **late binding (dynamic**) of referencing environment of a subroutine.

Deep binding (static)	**Example program**	**Shallow binding (dynamic)**
Point (1) **x of f1()→10** ---------- Point (2) x of f2()→6 (x of f1()hidden) ---------- Point (3) x of f1()→10 **Print x=10** ----------	**function f1()** **{** var x = 10; ---------- ***function f2(fx)*** *{* *var x;* *x = 6;* ***fx();*** *};* ---------- ***function f3()*** *{* *print x;* *};* ---------- **f2(f3);** **};**	Point (1) x of f1()→10 ---------- Point (2) **x of f2()→6** (x of f1()hidden) ---------- Point (3) x of f2()→6 (x of f1()hidden) **Print x=6** ----------

- In case of deep binding,f3() gets the environment of f1() and prints the value of x as 10 which is local variable of f1( ).In case of shallow binding, f3() is called in f2( ) and hence gets the environment of f2( )and prints the value of x as 6 which is local to f2( ).
- These Binding rules are irrelevant in languages like C (which has no nested subroutines), Modula-2 (which allow only outermost subroutines to be passed as parameters), PL/I and Ada (which do not permit subroutines to be passed as parameters at all.

7. Expression Evaluation

An expression consists of

- A simple object, e.g. number or variable
- An operator (eg:+)applied to a collection of operands(a,b,..) which is expression.
  a + b →an expression
- Or function(eg:addition( )) applied to a collection of arguments(a,b) is an expression.
  addition(a,b)→an expression.

Common syntactic **forms for operators**:

- **Function call notation**, e.g. addnum(A, B, C), where A, B, and C are expressions
- **Infix** notation[**a op b**] for binary operators, e.g. **A + B**
- **Prefix** notation[**op a**] for unary operators, e.g. **–A**

- **Postfix** notation [**a op**] for unary operators, e.g. **i++**

- **CambridgePolish(prefix)notation**, **(op a b)**.,
  e.g. **(* (+ 1 3) 2)** in **Lisp**, meaning is =(1+3)*2=8.
  It places the function name inside the parentheses.

- **Mixfix notation(**multiword infix notation) **in smalltalk**
  In Smalltalk to send a message to myBox graphical object:
  myBox displayOn: myScreen at: 100@50 ,
  where displayOn: and at:are written infix with arguments mybox, myScreen, and 100@50(a pixel location)

- **Conditional expressions**: In Algol
  a:= if b <> 0 then a/b else 0;
  In algol (if…..then….else) is a **three operand infix operator**.
  The equivalent operator in C is :
  a=b != 0 ? a/b : 0;

Most imperative languages (procedural,object oriented) use infix notation for binary operators and prefix notation for unary operators. Lisp(functional) uses prefix notation for all functions. Smalltalk use infix notation for all functions.

Expression Evaluation Ordering: Precedence and Associativity

- The use of infix, prefix, and postfix notation leads to ambiguity as to what is an operand of what, e.g. a+b*c**d**e/f in Fortran, should this grouped as
  ((((a + b) * c)**d)**e) / f or
  a + (((b * c)**d)**(e/f)) or
  a + ((b *(c**(d**e)))/f) or some other option?

In Fortran, the answer is the last of the options shown.

- The **choice** among alternative evaluation orders **depends***on**precedence and associativity of operators.*
  - **Operator precedence**: higher operator precedence means that a collection of) operator(s) group more tightly in an expression than operators of lower precedence.
  - **Operator associativity**: determines evaluation order of operators of the same precedence
    - **left associative**: operators are evaluated left-to-right (common)
    - **right associative**: operators are evaluated right-to-left
      (Fortran power operator **, C assignment operator = and unary minus)
    - **non-associative**: requires parenthesis when composed
      (Ada power operator **)
- **Pascal's flat precedence levels is a design mistake**
  - if A<B and C<D then is read as if A<(B and C)<D then results in static

semantic error. Most languages avoid this problem by giving arithmetic operators higher precedence than relational (comparison) operators, which in turn have higherprecedencethanthelogicaloperators.NotableexceptionsincludeAPL and Smalltalk, in which all operators are of equal precedence; parentheses must be used to specify grouping. Below table show precedence and associativity of operators in C

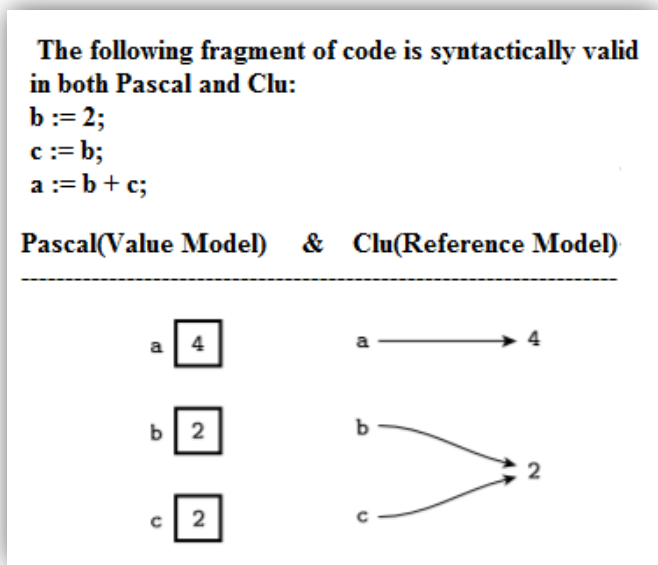
Precedence and Associativity of Operators

Precdence Group (Highest to Lowest)	Operators	Associativity
(param) subscript etc.,	( ) [ ] –>.	L → R
Unary operators	- + ! ~ ++ – – (type) * & sizeof	R → L
Multiplicative	* / %	L → R
Additive	+ –	L → R
Bitwise shift	<< >>	L → R
Relational	< <= > >=	L → R
Equality	= = !=	L → R
Bitwise AND	&	L → R
Bitwise exclusive OR	^	L → R
Bitwise OR	|	L → R
Logical AND	&&	L → R
Logical OR	||	L → R
Conditional	?:	R → L
Assignment	= += –= *= /= %= &= ^= |= <<= >>=	R → L
Comma	,	L → R

Assignments

- Fundamental difference between imperative and functional languages
  - Imperative Languages: "computing by means of side effects" i.e. computation is an ordered series of changes to values of variables in memory and statement ordering is influenced by run-time testing values of variables
  - Expressions in functional language are referentially transparent:All values used and produced depend on the referencing environment of the expression and not on the evaluation time of the expression
  - A function is idempotent in a functional language:Always returns the same value given the same arguments because of the absence of side-effects
- Assignment a:=b
  - Left-hand side ‘a’ of the assignment is a location, called **l-value** which is an expression that should denote a **location**.Right-hand side ‘b’ of the assignment is a **value**, called **r-value** which is an expression
  - Languages that adopt **value model** of variables copy values: Ada, Pascal, C, C++ copy the value of 'b’ into the location of ‘a’

- Languages that adopt **reference model** of variables copy references: Algol 68,Clu,Lisp/Scheme,ML,Haskell, and smalltalk.Clu copies the reference of b into a and both aand b refer to the same object. Here variable is not a named container for a value; rather it is named reference to a value.
- Java uses value model for built-in types and reference model for classes



- Variable initialization
  - Implicit: e.g. 0 or NaN (not a number) is assigned by default
  - Explicit: by programmer (more efficient than using an explicit assignment, e.g. int i=1; declares i and initializes it to 1 in C)
- Multiway assignments in Clu, ML, and Perl
  - a,b := c,d assigns c to a and d to b simultaneously,
    e.g.a,b := b,a swaps a with b
  - a,b := 1 assigns 1 to both a and b

Evaluation Ordering within Expressions

- Precedence and associativity define rules for structuring expressions
- But do not define operand evaluation order.
  Example:
  - Expression a-f(b)-c*d is structured as (a-f(b))-(c*d) by compiler,
  - but either (a-f(b)) or (c*d) can be evaluated first at run-time
  - What is the evaluation order of arguments in subroutine calls with multiple arguments?
- **Two main reasons why the order can be important:**
  - **Side effects:** e.g. if f(b) above modifies d (i.e. f(b) has a side effect) the expression value will depend on the operand evaluation order
  - **Code improvement**: compilers rearrange expressions to maximize efficiency
    - **Improve memory loads:**

a:=B[i]; load a from memory
c:=2*a+3*d; compute 3*d first, because loads are slow.
because waiting for a to arrive in processor

- **Common sub-expression elimination:**

a:=b+c;
d:=c+e+b; rearranged as d:=b+c+e, it can be rewritten into d:=a+e

- **Register allocation**: rearranging operand evaluation can decrease the number of processor registers used for temporary values

Expression Reordering Problems

- Rearranging expressions may lead to arithmetic overflow or different floating point results
  - Assume b, d, and c are very large positive integers, then if b-c+d is rearranged into (b+d)-c arithmetic overflow occurs
  - Floating point value of b-c+d may differ from b+d-c
  - Most programming languages will not rearrange expressions when parenthesis are used, e.g. write (b-c)+d to avoid problems
- Java: expressions evaluation is always left to right and overflow is always detected
- Pascal: expression evaluation is unspecified and overflows are always detected
- C anc C++: expression evaluation is unspecified and overflow detection is implementation dependent.

Short-Circuit Evaluation

- Boolean expressions provide a special and important opportunity for code improvement and increased readability.
- Short-circuit evaluation of Boolean expressions means that computations are skipped when logical result of a Boolean operator can be determined from the evaluation of one operand.
- Example :
  - ✓ Consider the expression (a<b) and (b<c);If a > b, there is really no point in checking to see whether b < c. we know that the overall expression must be false.
  - ✓ In the expression (a>b) or (b>c); If a > b, there is no point in checking to see whether b >c. we know that the overall expression must be true.
- C, C++, and Java use conditional and/or operators: && and ||
  - If a in a&&b evaluates to false, b is not evaluated
  - If a in a||b evaluates to true, b is not evaluated
  - Useful to increase program efficiency(save time), e.g.if (unlikely_condition&& expensive_condition()) ...
- **Pascal does not use short-circuit evaluation.**
- **In Ada,"and/or" uses "then" keyword,**
  e.g.: (a<b) and then (b<c);
  (a>b) or else (b>c);
- Ada, C, and C++ also have regular Boolean operators

- Short circuiting is not necessarily as attractive for situations in which a Boolean sub expression can cause a side effect.
- **Delayed or Lazy Evaluation:**Short circuiting can be considered as a delayed or lazy evaluation because the operands are passed unevaluated. Internally, the operator evaluates the first operand in any case, the second only when needed.

8. Structured and Unstructured Flow

- Unstructured flow: the use of goto statements and statement labels to obtain control flow
  - Merit or evil? Generally considered bad, but sometimes useful for jumping out of nested loops and for programming errors and exceptions
  - Java has no goto statement
  - Example:In early FORTRAN,
    if(A.lt.B) goto
    ……
    10 …….
- Structured flow:
  - Sequencing: the subsequent execution of a list of statements in that order
  - Selection: if-then-else statements and switch or case-statements
  - Iteration: for and while loop statements
  - Subroutine calls and recursion

All of which promotes structured programming

9. Sequencing

- A list of statements in a program text is executed in top-down order
- A compound statement is a delimited list of statements
  - A compound statement is a block when it includes variable declarations
  - C, C++, and Java use { and } to delimit a block
  - Pascal and Modula use begin ... end
  - Ada uses declare ... begin ... end
- C, C++, and Java: expressions can be used where statements can appear
- In pure functional languages, sequencing is impossible (and not desired!)

10. Selection

- Forms of if-then-else selection statements:
  - C and C++ EBNF syntax:
    if (<expr>) <stmt> [else <stmt>]
    Condition is integer-valued expression. When it evaluates to 0, the else-clause statement is executed otherwise the then-clause statement is executed. If more than one statement is used in a clause, grouping with { and } is required
  - Java syntax is like C/C++, but condition is Boolean type
  - Ada syntax allows use of multiple elsif's to define nested conditions:
    if <cond> **then**
    <statements>

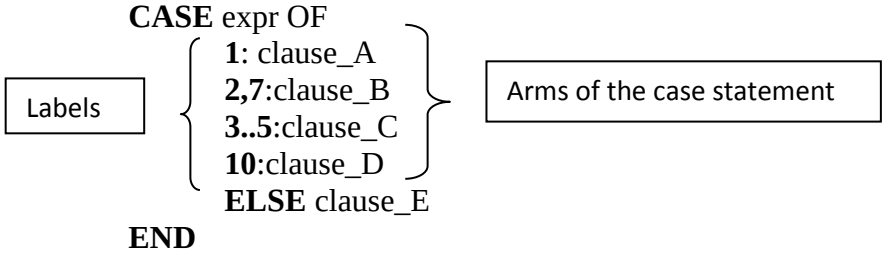
elsif <cond> **then**
<statements>
elsif <cond> **then**
<statements>
...
else
<statements>
end if

- In Modula-2,
  IF a=b **THEN**….
  ELSIF a=c **THEN**….
  ELSIF a=d **THEN**….
  ELSE….
  END
- In Lisp,(equivalent to Modula-2)
  (cond
  ((= a b)
  ( ))
  ((= a c)
  ( ))
  ((= a d)
  ( ))
  (**T**
  (…)))
- In algol 60,
  if condition **then** statements
  elseif condition **then** statements
  elseif condition **then** statements
  ……
  else statements

- Case/switch statements are different from if-then-else statements in that an expression can be tested against multiple constants to select statement(s) in one of the arms of the case statement:
  - C, C++, and Java syntax:

    switch (<expr>)
    {
    case <const>: <statements> break;
    case <const>: <statements> break;
    ...
    default: <statements>
    }

    break is necessary to transfer control at the end of an arm to the end of the switch statement
  - In modula-2,



- The use of a switch statement is much more efficient compared to nested if-then-else statements

11. Iteration

- ***Enumeration-controlledloops*** repeat a collection of statements a number of times, where in each iteration a loop index variable takes the next value of a set of values specified at the beginning of the loop
- ***Logically-controlledloops*** repeat a collection of statements until some Boolean condition changes value in the loop
  i. Pre-test loops test condition at the begin of each iteration
  ii. Post-test loops test condition at the end of each iteration
  iii. Mid-test loops allow structured exits from within loop with exit conditions

Enumeration-Controlled Loops

- Enumeration controlled iteration originated with the do loop of FORTRAN I.
- Similar mechanisms have been adopted in some form by almost every subsequent language, but syntax and semantics vary widely.

Fortran-90:

```
do i=1,10,2
  …….
enddo
```

- Variable i is called index of the loop.
- The expression that follow the equals sign are ***i's initial value, its bound, and the step size.***
- The body of the loop executes 5 times,with I set to 1,3,5,7,9 in successive iterations.

Algol-60 combines logical conditions:

- Difficult to understand and too many forms that behave the same:

```
for i := 1, 3, 5, 7, 9 do ...
for i := 1 step 2 until 10 do ...
for i := 1, i+2 while i < 10 do ...
```

Pascal has simple design:

- for <id> := <expr> to <expr> do <stmt>
  for <id> := <expr> downto <expr> do <stmt>

- Can iterate over any discrete type, e.g. integers, chars, elements of a set
- Index variable cannot be assigned and its terminal value is undefined

Ada for loop is much like Pascal's:

- for <id> in <expr>..<expr> loop
  <statements>
  end loop

  for <id> in reverse <expr>..<expr> loop
  <statements>
  end loop
- Index variable has a local scope in loop body, cannot be assigned, and is not accessible outside of the loop

C, C++, and Java:

- *do not have enumeration-controlled loops although the logically-controlled for statement can be used to create an enumeration-controlled loop:*
- for (i = 1; i <= n; i++)
  {
  ...
  ...}
  Iterates i from 1 to n by testing i <= n before each iteration and updating i by 1 after each iteration
- Programmer's responsibility to modify, or not to modify, i and n in loop body
- C++ and Java also allow local scope for index variable, for example

  for (int i = 1; i <= n; i++)
  {
  ...
  ...}

Problems With Enumeration-Controlled Loops

C/C++:

- This C program never terminates:
  #include <limits.h>
  main()
  { int i;
  for (i = 0; i <= INT_MAX; i++)
  ...
  }
  because the value of i overflows (INT_MAX is the maximum positive value int can hold) after the iteration with i==INT_MAX and i becomes a large negative integer
- In C/C++ it is easy to make a mistake by placing a ; at the end of a while or for statement, e.g. the following loop never terminates:

  i = 0;
  while (i < 10);

{
i++;
}

The C/C++ overflow problem is avoided by calculating the number of iterations in advance

- However, for REAL typed index variables an exception is raised when overflow occurs

Pascal and Ada:

- Can only specify step size 1 and -1
- Pascal and Ada can avoid overflow problem

Logically-Controlled Pretest Loops

- Logically-controlled pretest loops test an exit condition before each loop iteration
- Not available Fortran-77 (!)
- Pascal:

  while <cond>
  do <stmt>

  where the condition is a Boolean expression and the loop will terminate when the condition is false. Multiple statements need to be enclosed in begin and end
- C, C++:

  while (<expr>)
  <stmt>
  where the loop will terminate when expression evaluates to 0 and multiple statements need to be enclosed in { and }
- Java is like C++, but condition is Boolean expression

Logically-Controlled Posttest Loops

- Logically-controlled posttest loops test an exit condition after each loop iteration
- Not available in Fortran-77 (!)
- Pascal:

  repeat <stmt> [; <stmt>]* until <cond>

  where the condition is a Boolean expression and the loop will terminate when the condition is true
- C, C++:

  do <stmt> while (<expr>)
  where the loop will terminate when the expression evaluates to 0 and multiple statements need to be enclosed in {and }
- Java is like C++, but condition is a Boolean expression

Logically-Controlled Midtest Loops

- Logically-controlled midtest loops test exit conditions within the loop
- Ada:
  - loop

    <statements>
    exit when <cond>;
    <statements>
    exit when <cond>;
    <statements>

    ...
    end loop
  - Also allows exit of outer loops using labels:
    outer: loop

    ...
    for i in 1..n loop
    ...
    exit outer when cond;
    ...
    end loop;
    end outer loop;
- C, C++:
  - Use break statement to exit loops
  - Use continue to jump to beginning of loop to start next iteration
- Java is like C++, but combines Ada's loop label idea to allow jumps to outer loops

12. Recursion

- Iteration and recursion are equally powerful: iteration can be expressed by recursion and vice versa
- Recursion can be less efficient, but most compilers for functional languages will optimize recursion and are often able to replace it with iterations
- Recursion can be more elegant to use to solve a problem that is recursively defined
  - The GCD function is mathematically defined by

$$\gcd(a,b) = \begin{cases} a & \text{if } a = b \\ \gcd(a-b,b) & \text{if } a > b \\ \gcd(a,b-a) & \text{if } b > a \end{cases}$$

```
int gcd(int a, int b)
{
if (a==b) return a;
  else if (a>b) return gcd(a-b,                b);
  else return gcd(a,                           b-a);
```

}

Tail Recursive Functions

- Tail recursive functions are functions in which no computations follow a recursive call in the function
- gcd example recursively calls gcd without any additional computations after the calls
- A recursive call could in principle reuse the subroutine's frame on the run-time stack and avoid deallocation of old frame and allocation of new frame
- This observation is the key idea to tail-recursion optimization in which a compiler replaces recursive calls by jumps to the beginning of the function
  - For the gcd example, a good compiler will optimize the function into:

```
int gcd(int a, int b)
{                                                                    start:
   if (a==b) return a;
   else if (a>b) { a = a-b; goto start; }
   else { b = b-a; goto start; }
}
```

which is just as efficient as the iterative implementation of gcd:

```
int gcd(int a, int b)
{ while (a!=b)
   if (a>b)  a = a-b;
   else b = b-a;
  return a;
}
```

Continuation-Passing-Style: Even functions that are not tail-recursive can be optimized by compilers for functional languages by using continuation-passing style:With each recursive call an argument is included in the call that is a reference (continuation function) to the remaining work.The remaining work will be done by the recursively called function, not after the call, so the function appears to be tail-recursive

13. Non-determinacy

- Our final category of control flow is non-determinacy.
- A nondeterministic constructison is in which the choice between alternatives (i.e.,between control paths) is deliberately unspecified.
- We have already seen examples of non-determinacy in the evaluation of expressions in most languages, operator or subroutine arguments may be evaluated in any order.
- Some languages, notably Algol68 and various concurrent languages,provide more extensive non-deterministic mechanisms,which cover statements as well.

MODULE – 2

.

Data Types

Most programming languages include a notion of *type* for expressionsand/or objects.

Two principal purposes of types.

1.Operations that leveragetype information- Types provide implicit context for many operations, so that the programmerdoes not have to specify that context explicitly.
In C, for instance, the expression a + b will use integer addition if a andb are of integer type; it will usefloating-point addition if a and b are of double (floating-point) type.
2. Errors captured by typeinformation - Types limit the set of operations that may be performed in a semanticallyvalid program.
They prevent the programmer from adding a character and arecord, for example, or from taking the arctangent of a set, or passing a fileas a parameter to a subroutine that expects an integer.

2.1 TYPE SYSTEMS

Computer hardware can interpret bits in memory in several different ways: asinstructions, addresses, characters, and integer and floating-point numbers ofvarious lengths.
1. Assemblylanguages reflect this lack of typing.
2. High-level languages, on the other hand, almostalways associate types with values, to provide the contextual information anderror checking.

A *type system* consists of
(1) a mechanism to define types andassociate them with certain language constructs,and
(2) a set of rules for *typeequivalence*, *type compatibility*, and *type inference*.
i. The constructs that must havetypes are precisely those that have values, or that can refer to objects that have
values. These constructs include:

a)named constants,literal constants
b) variables,
c) record fields,
d) parameters,and
e) subroutines.

ii. Type equivalence rules determinewhen the types of two values are the same.
iii. Type compatibility rules determinewhen a value of a given type can be used in a given context.

iv. Type inference rulesdefine the type of an expression based on the types of its constituent parts or
(sometimes) the surrounding context.

2.1.1 Type Checking

Type checking is the process of ensuring that a program obeys the language's typecompatibility rules.

1. A violation of the rules is known as a *type clash.*
2. A language issaid to be *strongly typed* if it prohibits, in a way that the language implementation
can enforce, the application of any operation to any object that is not intendedto support thatoperation.
3. A language is said to be *statically typed* if it is stronglytyped and type checking can be performed at compile time.

Examples:

1. Ada is strongly typed, and for the most part statically typed(certain type constraints must be checked at run time).
2. A Pascal implementationcan also do most of its type checking at compile time, though the language is not
quite strongly typed: untagged variant records are its only loophole.
3. C89 is significantly more strongly typed than its predecessordialects, but still significantly less strongly typed than Pascal.
4. Implementations of Crarely check anything at run time.
5. Dynamic (run-time) type checking is a form of late binding, and tends to befound in languages that delay other issues until run time as well. Lisp and Smalltalkare dynamically (though strongly) typed. Most scripting languages are alsodynamically typed; some (e.g., Python and Ruby) are strongly typed. Languageswith dynamic scoping are generally dynamically typed (or not typed at all): if thecompiler can't identify the object to which a name refers, it usually can't determinethe type of the object either.

2.1.2 Polymorphism

Polymorphism allows a single body of code to work with objects ofmultiple types.

1. It may or may not imply the need for run-time type checking.
2. Only at run time does the language implementation check to see that the objectsactually implement the requested operations.
3. Types of objects can bethought of as implied (unspecified) parameters, dynamic typing is said to support
implicit parametric polymorphism.
4. *type inference*-forevery object and expression a (possibly unique) type that captures precisely those

properties that the object or expression must have to be used in the context(s) inwhich it appears.
5. *unification*: With rare exceptions, the programmer need not specify the typesof objects explicitly. The task of the compiler is to determine whether there existsa consistent assignment of types to expressions that guarantees, statically, that nooperation will ever be applied to a value of an inappropriate type at run time.
This job can be formalized as the problem of *unification.*
6. *subtype polymorphism*- In object-oriented languages, *subtype polymorphism* allows a variable *X* of
type*T* to refer to an object of any type derived from *T*. Since derived typesare required to support all of the operations of the base type, the compiler canbe sure that any operation acceptable for an object of type *T* will be acceptablefor any object referred to by *X*.
7. *explicit parametric polymorphism (generics)*- allow the programmer to define classes with type parameters. Genericsare particularly useful for *container* (*collection*) classes: "list of T" (List<T>),"stack of T" (Stack<T>), and so on, where T is left unspecified. Like subtypepolymorphism, generics can usually be type checked at compile time, thoughJava sometimes performs redundant checks at run time for the sake of interoperabilitywith preexisting nongeneric code. Smalltalk,Objective-C, Python, and Ruby
use a single mechanism for both parametric and subtype polymorphism, withchecking delayed until run time.

2.1.4 CLASSIFICATION OF TYPES

Most languages providebuilt-in types similar to those supported in hardware by most processors: integers,
characters, Booleans, and real (floating-point) numbers.
1. *NumericTypes*
2. EnumerationTypes
3. SubrangeTypes
4. CompositeTypes

1. NumericTypes

a) A few languages (e.g., C and Fortran) distinguish between different lengths ofintegers and real numbers; most do not, and leave the choice of precision to theimplementation.
b) A few languages, including C, C++, C# andModula-2, provide both signed andunsigned integers (Modula-2 calls unsigned integers *cardinal*s).
c) Fortran, C99, Common Lisp, and Schemeprovide a built-in complex type,usually implemented as a pair of floating-point numbers that represent the realand imaginary Cartesian coordinates; other languages support these as a standardlibrary class.

d) Scheme and Common Lisp-provide a built-inrational type, usually implemented as a pair of integers that represent the numeratorand denominator.
e) Ada supports *fixed-point* types, which are represented internally by integers,but have an implied decimal point at a programmer-specified position amongthe digits.
f) Integers, Booleans, and characters are all examples of *discrete* types (also called*ordinal* types): the domains to which they correspond are countable (they havea one-to-one correspondence with some subset of the integers), and have a welldefinednotion of predecessor and successor for each element other than the first
and the last.

3. Subrange Types

1. A subrange is a type whose values compose a contiguoussubset of the values of some discrete *base* type (also called the *parent*type)
2. Subranges were first introduced in Pascal.
3. Subranges in Pascal- In Pascal, subrangeslook like this:
typetest_score = 0..100;
workday = mon..fri;
4. Subranges in Ada-In Ada one would write
typetest_score is new integer range 0..100;
subtype workday is weekday range mon..fri;
5. The range... portion of the definition in Ada is called a type *constraint*. In thisexample test_score is a *derived* type, incompatible with integers. The workdaytype, on the other hand, is a *constrained subtype*; workdays and weekdays can bemore or less freely intermixed. The distinction between derived types and subtypes
is a valuable feature of Ada.

4. Composite Types

Nonscalar types are usually called *composite*, or *constructed* types.
1. They are generallycreated by applying a *type constructor* to one or more simpler types.
2. Commoncomposite types include records (structures), variant records (unions), arrays, sets, pointers, lists, and files.
i. *Records (structures)* were introduced by Cobol, and have been supported by mostlanguages since the 1960s.
A record consists of collection of fields, each ofwhich belongs to a (potentially different) simpler type.
A record type corresponds to the Cartesian product ofthe types of the fields
ii. *Variant records (unions)*only one of a variantrecord's fields (or collections of fields) is valid at any given time.
iii. *Arrays*An array can be thought ofas a function that maps members of an index type to members of a componenttype. Arrays of characters are often referred to as strings, and are oftensupported by special-purpose operations not available for other arrays.

iv. *Sets*A set type isthe mathematical powerset of its base type, which must often be discrete.A variable of a set type contains a collection of distinct elements of the basetype.
v. *Pointers*are l-values. A pointer value is a *reference* to an object of the pointer'sbase type. Pointers are often but not always implemented as addresses. Theyare most often used to implement *recursive* data types. A type *T* is recursiveif an object of type *T* may contain one or more references to other objects oftype *T.*
vi. *Lists*- A list is defined recursively as either an empty list or apair consisting of a head element and a reference to a sublist. Lists are always of variable length.To find a given element of alist, a program must examine all previous elements, recursively or iteratively,starting at the head. Because of their recursive definition, lists are fundamentalto programming in most functional languages.
vii. *Files* are intended to represent data on mass-storage devices, outside the memoryin which other program objects reside.Files usually havea notion of *current position*, which allows the index to be implied implicitly
in consecutive operations.

2.2 TYPE CHECKING

Type Checking can be done by using
1. *type equivalence,*
2. type compatibility,
3. type inference.

2.2.1 Type Equivalence

In a language in which the user can define new types, there are two principal waysof defining type equivalence.
a) *Structural equivalence*is based on the content of type definitions, two types are the same if they consist of the samecomponents. Used in Algol-68, Modula-3, and C and ML.

```
type R2 = record
a, b : integer
end;
```

should probably be considered the same as

```
type R3 = record
a : integer;
b : integer
end;
```

But what about

```
type R4 = record
b : integer;
a : integer
end;
```

Should the reversal of the order of the fields change the type? ML says no; most languages say yes.

b) *Name equivalence*is based on the lexicaloccurrence of type definitions, each definition introduces a new type. Used in Java, C#, standard Pascal, and most Pascal descendants,including Ada.

Name equivalence is based on the assumption that if the programmer goesto the effort of writing two type definitions, then those definitions are probablymeant to represent different types.

Variants of Name Equivalence

One subtlety in the use of name equivalence arises in the simplest of type declarations: Alias types

TYPE new_type = old_type; (* Modula-2 *)

Here new_type is said to be an *alias* for old_type. Should we treat them as twonames for the same type, or as names for two different types that happen to havethe same internal structure? The "right" approach may vary from one program toanother.

1. *strictname equivalence*- A language in which aliased types are considered distinct is said to have *strict name equivalence.*

Most Pascal-family languages (includingModula-2) use loose name equivalence.

2. *loose name equivalence* - A language in which aliased types are considered equivalentis said to have *loose name equivalence.*

Derived types andsubtypes in Ada- Ada achieves the best of both worlds byallowing the programmer to indicate whether an alias represents a *derived* typeor a *subtype.*

One way to think about the difference between strict and loose name equivalenceis to remember the distinction between *declarations and definitions*. Under strict name equivalence, a declaration type A = B is considered a definition. Under loose name equivalence it is merely a declaration; Ashares the definition of B.

Type Conversion and Casts

Dependingon the types involved, the conversion may or may not require code to be executedat run time. There are three principal cases:

1. The types would be considered structurally equivalent, but the language usesname equivalence. In this case the types employ the same low-level representation,and have the same set of values. The conversion is therefore a purelyconceptual operation; no code will need to be executed at run time.

2. The types have different sets of values, but the intersecting values are representedin the same way. One type may be a subrange of the other, for example,or one may consist of two's complement signed integers, while the other isunsigned.

3. The types have different low-level representations, but we can nonethelessdefine some sort of correspondence among their values. A 32-bit integer, forexample, can be converted to a double-precision IEEE floating-point numberwith no loss of precision. Most processors provide a machine instruction toeffect this conversion.

NonconvertingType Casts

Occasionally, particularly in systems programs, oneneeds to change the type of a value *without* changing the underlying implementation;in other words, to interpret the bits of a value of one type as if they wereanother type.

A change of type that does not alter the underlying bits is called a *nonconvertingtype cast*, or sometimes a *type pun*. It should not be confused with use of the term*cast* for conversions in languages like C.

1. Conversions andnonconverting casts inC++

C++ inherits the casting mechanism of C, but also provides a family of semantically cleaner alternatives.

2. Specifically, static_cast performs a type conversion,reinterpret_cast performs a nonconverting type cast, and dynamic_cast allows programs that manipulate pointers of polymorphic types to performassignments whose validity cannot be guaranteed statically, but can be checked atrun time.

3. There is also a const_cast that can be used to remove read-only qualification.C-style type casts in C++ are defined in terms of const_cast, static_cast,and reinterpret_cast; the precise behavior depends on the source and targettypes.

2.2.2Type Compatibility

Most languages do not require equivalence of types in every context. A value's type must be *compatible* with that of the contextin which it appears.

1. In an assignment statement, the type of the right-hand sidemust be compatible with that of the left-hand side.

2. The types of the operandsof + must both be compatible with some common type that supports addition

(integers, real numbers, or perhaps strings or sets).

3. In a subroutine call, the typesof any arguments passed into the subroutine must be compatible with the types

of the corresponding formal parameters, and the types of any formal parameterspassed back to the caller must be compatible with the types of the correspondingarguments.

Coercion

Whenever a language allows a value of one type to be used in a context that expectsanother, the language implementation must perform an automatic, implicit conversionto the expected type. This conversion is called a *type coercion.*

1. A coercion may require run-time code toperform a dynamic semantic check, or to convert between low-level representations.

Coercion in Ada

Ada coercions sometimes need the former, though never the latter.

```
d : weekday; -- as in
k : workday; -- as in
typecalendar_column is new weekday;
c :calendar_column;
...
k := d; -- run-time check required
d := k; -- no check required; every workday is a weekday
c := d; -- static semantic error;
```

-- weekdays and calendar_columns are not compatible

To perform this third assignment in Ada we would have to use an explicit conversion:

c := calendar_column(d);

Coercion in C

1. Coercions are a controversial subject in languagedesign. Because they allow types to be mixed without an explicit indication ofintent on the part of the programmer, they represent a significant weakeningof type security.

2. C, which has a relatively weak type system, performs quite a bit ofCoercion in C coercion. It allows values of most numeric types to be intermixed in expressions,and will coerce types back and forth “as necessary.”

```
shortint s;
unsigned long int l;
char c; /* may be signed or unsigned -- implementation-dependent */
float f; /* usually IEEE single-precision */
double d; /* usually IEEE double-precision */
...
s = l; /* l’s low-order bits are interpreted as a signed number. */
l = s; /* s is sign-extended to the longer length, then
its bits are interpreted as an unsigned number. */
s = c; /* c is either sign-extended or zero-extended to s’s length;
the result is then interpreted as a signed number. */
f = l; /* l is converted to floating-point. Since f has fewer
significant bits, some precision may be lost. */
d = f; /* f is converted to the longer format; no precision lost. */
f = d; /* d is converted to the shorter format; precision may be lost.
```

If d’s value cannot be represented in single-precision, the

result is undefined, but NOT a dynamic semantic error. */
3. Fortran 90 allows arrays and records to be intermixed if their types have thesame *shape.*
a) Two arrays have the same shape if they have the same number ofdimensions, each dimension has the same size (i.e., the same number of elements),and the individual elements have the same shape. (In some other languages, theactual bounds of each dimensionmust be the same for the shapes to be consideredthe same.)
b) Two records have the same shape if they have the same number of fields,and corresponding fields, in order, have the same shape.
4. Ada's compatibility rules for arrays are roughly equivalent to those of Fortran90.
5. C provides no operations that take an entire array as an operand. C does,however, allow arrays and *pointers* to be intermixed in many cases.
6. Neither Adanor C allows records (structures) to be intermixed unless their types are nameequivalent.

Overloading and Coercion
1. An overloaded name can refer to morethan one object; the ambiguitymust be resolved by context.
a) In the expression a + b, + may refer to either the integeror the floating-point addition operation.
b) In a language without coercion, a and bmust either both be integer or both be real; the compiler chooses the appropriateinterpretation of + depending on their type.
c) In a language with coercion, + refersto the floating-point addition operation if either a or b is real; otherwise it refers tothe integer addition operation.

Universal ReferenceTypes
i. For systems programming,or to facilitate thewriting of general-purpose *container*(*collection*) objects (lists, stacks, queues, sets, etc.) that hold references to otherobjects, several languages provide a *universal* reference type.
1. In C and C++, thistype is called void *.
2. In Clu it is called any;
3. InModula-2, address;
4. InModula-3,refany;
5. InJava, Object;
6. In C#, object.
ii. Arbitrary l-values can be assigned into anobject of universal reference type, with no concern about type safety: because thetype of the object referred to by a universal reference is unknown, the compilerwill
not allow any operations to be performed on that object.
iii. In Java and C#, a universal to specificassignment requires a type cast, and will generate an exception if the universalreference does not refer to an object of the casted type.

iv. in C++ it uses adynamic_cast operation.

2.2.3 Type Inference

1. Theresult of an arithmetic operator usually has the same type as the operands.
2. Theresult of a comparison is usually Boolean.
3. The result of a function call has the typedeclared in the function's header.
4. The result of an assignment (in languages inwhich assignments are expressions) has the same type as the left-hand side.

Subranges

Inference of subrange types

```
typeAtype = 0..20;
Btype = 10..20;
var a : Atype;
b :Btype;
```

what is the type of a + b?Certainly it is neither Atype nor Btype, since the possiblevalues range from10 to 40.

1. The usual answer in Pascal and its descendants isto say that the result of any arithmetic operation on a subrange has the subrange'sbase type, in this case integer.
2. To avoid the expenseof some unnecessary checks, a compiler may keep track at compile time of the
largest and smallest possible values of each expression, in essence computing theanonymous 10. . . 40 type.
3. In languages like Ada, the type of an arithmetic expression assumes specialsignificance in the header of a for loop, because it determines thetype of the index variable.

CompositeTypes

1. Most built-in operators in most languages take operands of built-in types.
2. Type inference becomes an issue when an operation on composites yields aresult of a different type than the operands.

Type inference on stringoperations:

Character strings provide a simple example. In Pascal, the literal string 'abc'has type array [1..3] of char.

1. In Ada, the analogous string (denoted "abc")is considered to have an incompletely specified type that is compatible with anythree-element array of characters.
2. In the Ada expression "abc" & "defg", "abc"is a three-character array, "defg" is a four-character array, and the result is aseven-character array formed by concatenating the two.
3. The seven-character result of the concatenation could beassigned into an array of type array (1..7) of character or into an array oftype array (weekday) of character, or into any other seven-element character
array.

Type inference for sets:
1. Operations on composite values also occur when manipulating sets. Pascal andModula, for example, support union (+), intersection (*), and difference (-) onsets of discrete values.
2. Set operands are said to have compatible types if theirelements have the same base type T.

2.3RECORDS (STRUCTURES) AND VARIANTS (UNIONS)

Record types allow related data of heterogeneous types to be stored and manipulatedtogether.
1. Some languages (notably Algol 68, C, C++, and Common Lisp)use the term *structure* (declared with the keyword struct) instead of *record.*
2. Fortran 90 simply calls its records "types": they are the only formof programmerdefinedtype other than arrays, which have their own special syntax.
3. Java has no distinguished notion of struct; its programmersuse classes in all cases.
4. C# uses a reference model for variables of class types,and a value model for variables of struct types.

2.3.1Syntax and Operations

A C struct

In C, a simple record might be defined as follows.

```
struct element {
char name[2];
intatomic_number;
doubleatomic_weight;
_Bool metallic;
};
```

A Pascal record

In Pascal, the corresponding declarations would be

```
typetwo_chars = packed array [1..2] of char;
type element = record
name :two_chars;
atomic_number : integer;
atomic_weight : real;
metallic : Boolean
end;
```

Accessing record fields

Each of the record components is known as a *field*. To refer to a given field of a record, most languages use "dot" notation. In C:

```
element copper;
const double AN = 6.022e23; /* Avogadro's number */
...
```

```
copper.name[0] = 'C'; copper.name[1] = 'u';
double atoms = mass / copper.atomic_weight * AN;
```

Nested records

Most languages allow record definitions to be nested. Again in C:

```
Nested records
struct ore {
        char name[30];
        struct {
                char name[2];
                intatomic_number;
                doubleatomic_weight;
                _Bool metallic;
                } element_yielded;
}
```

ML records and tuples

1. ML differs from most languages in specifying thatthe order of record fields is insignificant. The ML record value {name = "Cu",atomic_number = 29, atomic_weight = 63.546, metallic = true} is thesame as the
value {atomic_number = 29, name = "Cu", atomic_weight =63.546, metallic = true} (they will test true for equality).
2. ML tuples aredefined as abbreviations for records whose field names are small integers. The values ("Cu", 29), {1 = "Cu", 2 = 29}, and {2 = 29, 1 = "Cu"} will all testtrue for equality.

2.3.2Memory Layout and Its Impact

1. The fields of a record are usually stored in adjacent locations in memory. In itssymbol table, the compiler keeps track of the offset of each field within each recordtype.
2. When it needs to access a field, the compiler typically generates a load orstore instruction with displacement addressing.
3. For a local object, the base registeris the frame pointer; the displacement is the sum of the record's offset from theregister and the field's offsetwithin the record.
4. On a RISC machine, a global recordis accessed in a similar way, using a dedicated *globals pointer* register as base.
5. On aCISC machine, the compiler may access the field directly at its absolute address or,if many fields are to be accessed in a short period of time, it may load a temporaryregister with the (absolute) address of the record and then use the field's offset asdisplacement.

Memory layout for a record type

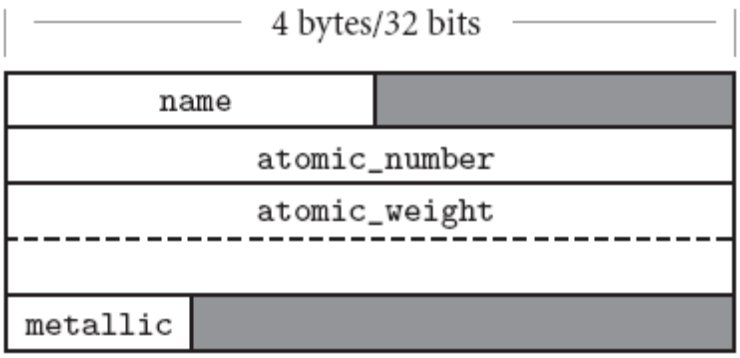


Figure 2.1.Likely layout in memory for objects of type element on a 32-bit machine. Alignment restrictions lead to the shaded "holes."

A layout for our element type on a 32-bit machine appears inFigure 2.1.

1. Because the name field is only two characters long, it occupies two bytesin memory.
2. Since atomic_number is an integer, and must (on most machines)be word-aligned, there is a two-byte "hole" between the end of name and thebeginning of atomic_number.
3. Similarly, since Boolean variables (in most languageimplementations) occupy a single byte, there are three bytes of empty spacebetween the end of the metallic field and the next aligned location. In an arrayof elements, most compilers would devote 20 bytes to every member of thearray.
4. In a language with a value model of variables, nested records are naturallyembedded in the parent record, where they function as large fields with word ordouble-word alignment.
5. In a language with a reference model of variables, fieldsof record type are typically references to data in another location. The difference isa matter not only of memory layout, but also of semantics.

Nested records as values

In Pascal, the followingprogram prints a 0.

```
type
        T = record
                j : integer;
        end;
        S = record
                i : integer;
                n : T;
        end;
var s1, s2 : S;
...
s1.n.j := 0;
s2 := s1;
s2.n.j := 7;
```

writeln(s1.n.j); (* prints 0 *)
The assignment of s1 into s2 copies the embedded T.

Nested records as references
By contrast, the following Java program prints a 7. (Simple classes in Java playthe role of structs.)

```
class T {
        publicint j;
}
class S {
        publicinti;
        public T n;
}
...
S s1 = new S();
s1.n = new T();                 // fields initialized to 0
S s2 = s1;
s2.n.j = 7;
System.out.println(s1.n.j);     // prints 7
```

Here the assignment of s1 into s2 has copied only the reference, so s2.n.j is analias for s1.n.j.

Layout of packed types
A few languages—notably Pascal—allow the programmer to specify that arecord type (or an array, set, or file type) should be *packed*:

```
type element = packed record
        name :two_chars;
        atomic_number : integer;
        atomic_weight : real;
        metallic : Boolean
end;
```

1. The keyword packed indicates that the compiler should optimize for space insteadof speed. In most implementations a compiler will implement a packed recordwithout holes, by simply“pushing the fields together.”
2. To access a nonaligned field,however, it will have to issue amulti-instruction sequence that retrieves the piecesof the field from memory and then reassembles them in a register.

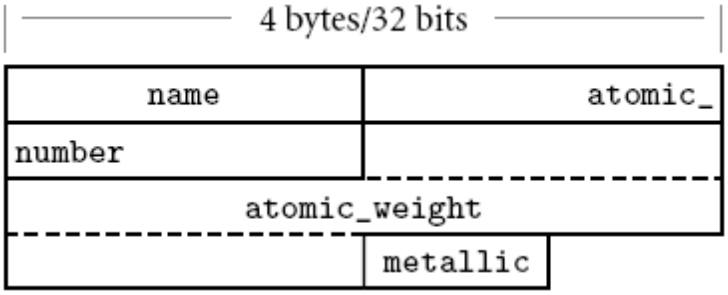


Figure 2.2 Likely memory layout for packed element records. The atomic_number and atomic_weight fields are nonaligned, and can only be read or written (on most machines) via multi-instruction sequences.

3. Ada, Modula-3, and C provide more elaboratepacking mechanisms, which allow the programmer to specify precisely how manybits are to be devoted to each field.

Assignment and comparison of records
Most languages allow a value to be assigned to an entire record in a singleoperation:
my_element := copper;

Minimizing holes by sorting fields
1. holes in records waste space.
2. Packingeliminates holes, but at potentially heavy cost in access time.
3. A compromise,adopted by some compilers, is to sort a record's fields according to the size of their alignment constraints.
4. All byte-aligned fields might come first, followed byany half-word aligned fields, word-aligned fields, and (if the hardware requires)double-word–aligned fields.
5. For element type, the resulting rearrangementis shown in Figure 2.3.

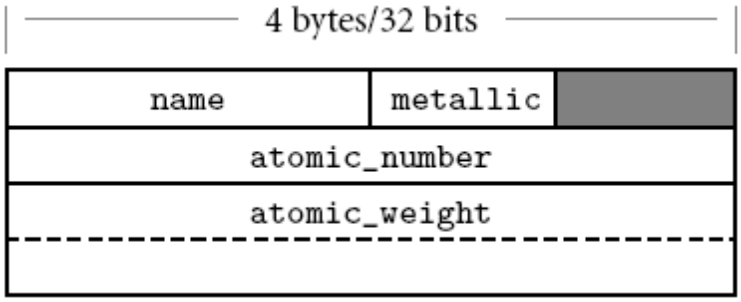


Figure 2.3 Rearranging record fields to minimize holes. By sorting fields according to the sizeof their alignment constraint, a compiler can minimize the space devoted to holes, while keepingthe fields aligned.

2.3.3 With **Statements**

In programs with complicated data structures, manipulating the fields of a deeply nested record can be awkward:

```
ruby.chemical_composition.elements[1].name := 'Al';
ruby.chemical_composition.elements[1].atomic_number := 13;
ruby.chemical_composition.elements[1].atomic_weight := 26.98154;
ruby.chemical_composition.elements[1].metallic := true;
```

Pascal provides a with statement to simplify such constructions:

```
withruby.chemical_composition.elements[1] do begin
        name := 'Al';
        atomic_number := 13;
        atomic_weight := 26.98154;
        metallic := true
end;
```

2.3.4 Variant Records (Unions)

Programming languages of the 1960s and 1970s were designed in an era of severememory constraints.

1. Many languages allowed the programmer to specify that certain variables(presumably ones that would never be used at the same time) should beallocated "on top of" one another, sharing the same bytes in memory.

2. C's syntax,

heavily influenced by Algol 68, looks very much like a struct:

```
union {
        inti;
        double d;
        _Bool b;
};
```

3. Unions have been used for two main purposes.

i. Insystems programs, where unions allow the same set of bytes to be interpretedin different ways at different times.

ii. The second common purpose for unions is to represent alternative sets offields within a record.

2.4 ARRAYS

1. Composite data types.
2. Beginning with Fortran I.
3. A mapping from an *indextype* to a *component* or *element type.*

i. Fortran require that the index type be integer;

ii. Fortran 77 require that the element type of an array be scalar.

iii. Fortran 90 allow any element type

iv. scripting languages allow non discrete index types.

2.4.1 Syntax and Operations

Most languages refer to an element of an array by appending a subscript—delimited by parentheses or square brackets—to the name of the array.

i. In FortranandAda, one says A(3);

ii. In Pascal and C, one says A[3]

Declarations

1. In C:

```
char upper[26];
```

In C, the lower bound of an index range is always zero: the indices of an *n*-elementarray are 0 . . . $n-1$.

2. In Fortran:

```
character, dimension (1:26) :: upper
character (26) upper ! shorthand notation
```

In Fortran, the lower bound of the index range is one bydefault.

3. In Pascal:

```
var upper : array ['a'..'z'] of char;
```

arrays are declared with an array constructor

4. In Ada:

```
upper : array (character range 'a'..'z') of character;
```

Multidimensional arrays

1. Ada

```
mat : array (1..10, 1..10) of real;
```

2. Fortran

```
real, dimension (10,10) :: mat
```

3. In Modula-3

```
VAR mat : ARRAY [1..10], [1..10] OF REAL;
```

Array constructor.

Multidimensional vs built-up arrays

In Ada

```
mat1 : array (1..10, 1..10) of real;
```

is not the same as

```
type vector is array (integer range <>) of real;
type matrix is array (integer range <>) of vector (1..10);
mat2 : matrix (1..10);
```

i. Variable mat1 is a two-dimensional array;can access individual real numbers as mat1(3, 4);

ii. mat2 is an array of one-dimensionalarrays.with the latter we must say mat2(3)(4)

Arrays of arrays in C

In C, one must also declare an array of arrays, and use two-subscript notation

double mat[10][10];

1. Given this definition, mat[3][4] denotes an individual element of the array
2. mat[3] denotes a *reference*

either to the third row of the array or to the first element of that row, depending on context.

Slices and Array Operations

A *slice* or *section* is a rectangular portion of an array.

Fortran 90 and SingleAssignment C provide extensive facilities for slicing, as do many scripting languages,

including Perl, Python, Ruby, and R.

real, dimension (10,10) :: mat

1. Ada provides more limited support: a slice is simply a contiguous range of elementsin a one-dimensional array.

operations

1.selection ofan element

2. assignment

3. compared for equality (Ada and Fortran 90)

Ada allows one-dimensional arrays whose elements arediscrete to be compared for *lexicographic ordering* : A < B if the first element of Athat is not equal to the corresponding element of B is less than that correspondingelement.

4. Ada allows the built-in logical operators (or, and, xor) to be appliedto Boolean arrays.
5. Fortran 90

i. A + B is an arrayeach of whose elements is the sum of the corresponding elements of A and B

ii. Fortran 90 provides a huge collection of *intrinsic*, or built-in functions.Morethan 60 of these (including logic and bit manipulation, trigonometry, logs andexponents, type conversion, and string manipulation) are defined on scalars, butwill also perform their operation element-wise if passed arrays as arguments.

6. APL, an array manipulation language developed by Iverson and others in the earlyto mid-1960s.

2.4.2 Dimensions, Bounds, and Allocation

1. static shape arrays(the shape of the array (includingbounds) was specified in the declaration)

storage canbe managed in the usual way:

static allocation for arrays whose lifetime is theentire program;

2.stack allocation for arrays whose lifetime is an invocation of asubroutine;

3. heap allocation for dynamically allocated arrays with more generallifetime.

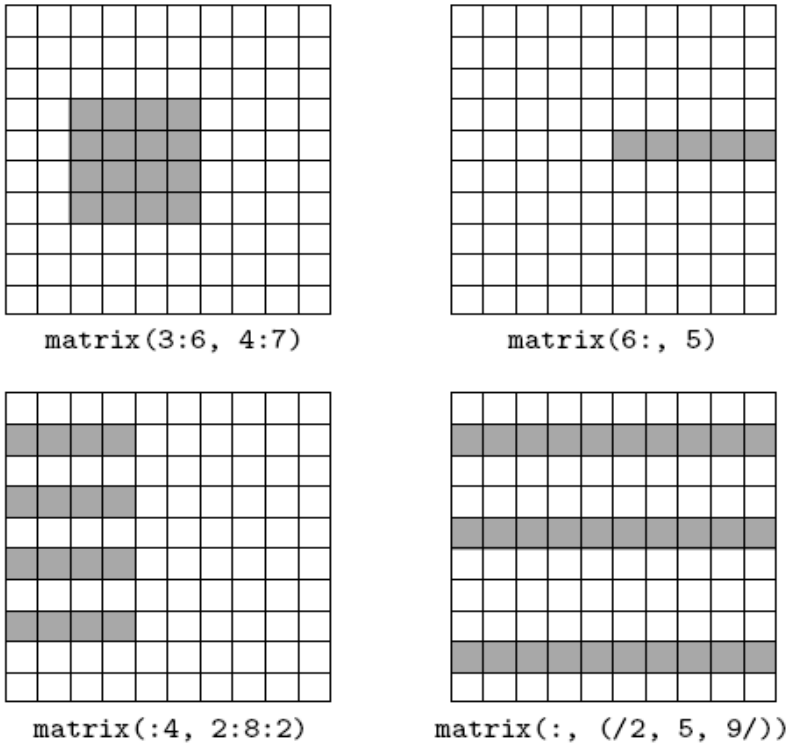


Figure 2.4 Array slices (sections) in Fortran 90. Much like the values in the header of an enumeration-controlled loop , a : b : c in a subscript indicates positions a, a + c, a +2c, . . . through b. If a orb is omitted, the corresponding bound of the array is assumed. If c is omitted, 1 is assumed. It is even possible to use negative values of c in order to select positions inreverse order. The slashes in the second subscript of the lower right example delimit an explicitlist of positions.

Descriptors or Dope Vectors

During compilation, the symbol table maintains dimension and bounds informationfor every array in the program.When the number and bounds of array dimensions are not statically known, the compiler must generate code to look them up in a dope vector at run time.

1. a dope vector will contain the lower bound of each dimension andthe size of each dimension other than the last.
2. If the language implementationperforms dynamic semantic checks for out-of-bounds subscripts in array

references, then the dope vector may contain upper bounds as well.

3. The contents of the dope vector are initialized at elaboration time, or wheneverthe number or bounds of dimensions change.

Fortran 90- notion of shape includes dimension sizes but not lower bounds, an assignmentstatementmay need to copy not only the data of an array, but dope vector contentsas well.

Stack Allocation

Subroutine parameters are the simplest example of dynamic shape arrays.
Pascal: allow array parameters to have bounds thatare symbolic names rather than constants. It calls these parameters *conformantarrays*:

```
function DotProduct(A, B:array [lower..upper:integer] of real):real;
var i : integer;
    rtn : real;
begin
    rtn := 0;
    for i := lower to upper do rtn := rtn + A[i] * B[i];
    DotProduct := rtn
end;
```

Here lower and upper are initialized at the time of call, providing DotProductwith the information it needs to understand the shape of A and B. In effect, lowerand upper are extra parameters of DotProduct.
Conformant arrays are highly useful in scientific applications, many of whichrely on numerical libraries for linear algebra and the manipulation of systems ofequations. Since different programs use arrays of different shapes, the subroutinesin these libraries need to be able to take arguments whose size is not known at
compile time.
Pascal allows conformant arrays to be passed by reference or by value.
Stack allocation of elaborated arrays

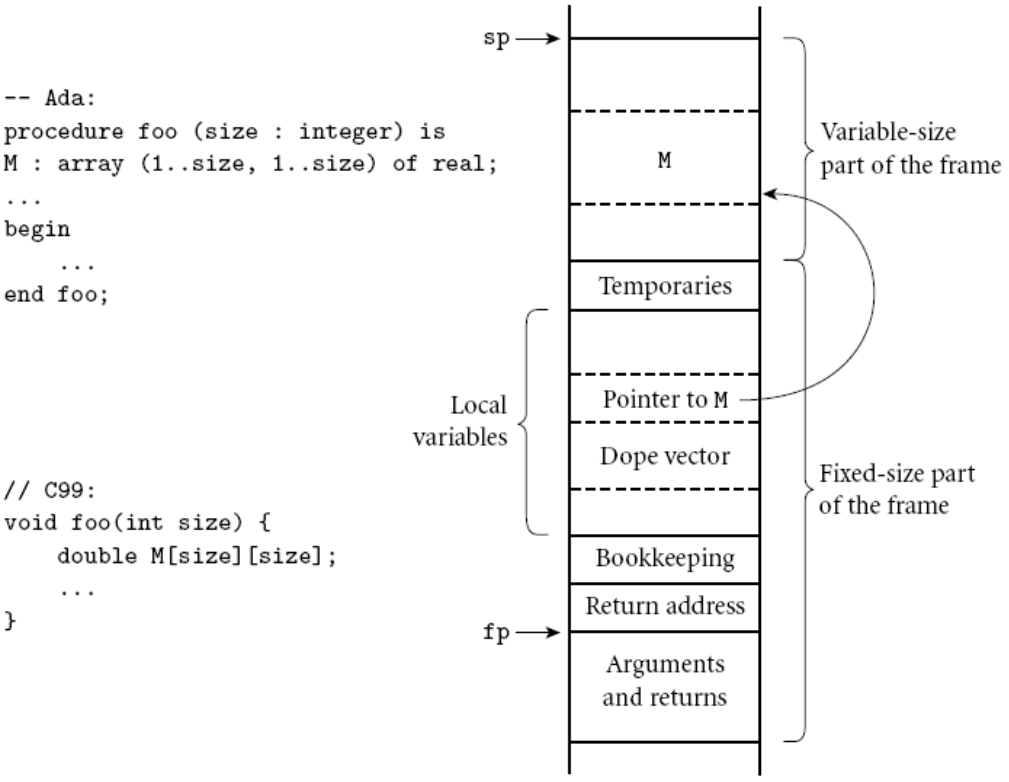


Figure 2.5 Elaboration-time allocation of arrays in Ada or C99. Here M is a square twodimensional array whose bounds are determined by a parameter passed to foo at run time. The compiler arranges for a pointer to M and a dope vector to reside at static offsets from the frame pointer. *M cannot be placed among the other local variables because it would prevent those higher in the frame from having static offsets. Additional variable-size arrays or records are easily accommodated*.

Every local object can be found using a known offset fromthe frame pointer.

1. divide the stack frame into a *fixedsize-part* and a *variabl-size part*
2. An object whose size is statically known goes inthe fixed-size part.
3. An object whose size is not known until elaboration time goesin the variable-size part, and a pointer to it, together with a dope vector, goes inthe fixed-size part.

Heap Allocation

1. Arrays that can change shape at arbitrary times are sometimes said to be *fullydynamic*
2. fully dynamic arrays must be allocated in the heap

Dynamic strings in Java and C#

String variables in these languages are references toimmutable string objects:

```
String s = "short";    // This is Java; use lowercase 'string' in C#
...
s = s + " but sweet";  // + is the concatenation operator
```

Here the declaration String s introduces a string variable, which we initializewith a reference to the constant string "short". In the subsequent assignment, +creates a new string containing the concatenation of the old s and the constant" but sweet"; s is then set to refer to this new string, rather than the old.

3. If the number of dimensions of a fully dynamic array is statically known, thedope vector can be kept, together with a pointer to the data, in the stack frameof the subroutine in which the array was declared. If the number of dimensionscan change, the dope vectormust generally be placed at the beginning of the heapblock instead.
4. In the absence of garbage collection, the compiler must arrange to reclaim thespace occupied by fully dynamic arrays when control returns from the subroutinein which theywere declared. Space for stack-allocated arrays is of course reclaimedautomatically by popping the stack.

2.4.3 Memory Layout

1. one-dimensional array
    i. the second element of the array is storedimmediately after the first (subject to alignment constraints); the third is storedimmediately after the second, and so forth.
    ii. For arrays of records
    it is commonfor each subsequent element to be aligned at an address appropriate for any type
2. multidimensional arrays

it still makes sense to put the first element ofthe array in the array's first memory location. But which element comes next?

i. *row-major*

consecutive locations in memory hold elements that differ byone in the final subscript (except at the ends of rows)

e.g., A[2, 4]is followed by A[2, 5].

The advantage ofrow-major order is that it makes it easy to define a multidimensional array as anarray of subarrays

ii. *column-major*

consecutive locations hold elementsthat differ by one in the *initial* subscript

e.g., A[2, 4] is followed by A[3, 4]

theelements of the subarray would not be contiguous in memory

Array layout and cache performance

a) When code traverses a small array, all or mostof its elements are likely to remain in the cache through the end of the nestedloops, and the orientation of cache lines will not matter.

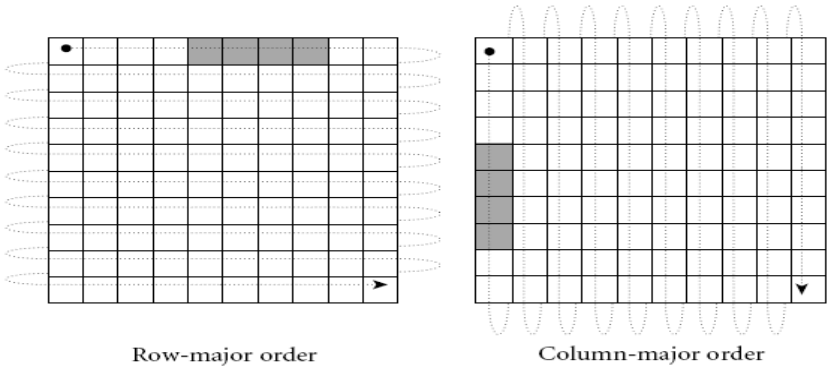


Figure 2.6 Row- and column-major memory layout for two-dimensional arrays.

In row-majororder, the elements of a row are contiguous in memory; in column-major order, the elementsof a column are contiguous. The second cache line of each array is shaded, on the assumption

that each element is an eight-byte floating-point number, that cache lines are 32 bytes long (acommon size), and that the array begins at a cache line boundary. If the array is indexed fromA[0,0] to A[9,9], then in the row-major case elements A[0,4] through A[0,7] share a cache line;in the column-major case elements A[4,0] through A[7,0] share a cache line.

b) For a large array, however,lines that are accessed early in the traversal are likely to be evicted to make

room for lines accessed later in the traversal.

c) If array elements are accessed inorder of consecutive addresses, then each miss will bring into the cache not onlythe desired element, but the next several elements as well.

d) If elements are accessed*across* cache lines instead then there is a good chance that almost everyaccess will result in a cache miss, dramatically reducing the performance of thecode.
In C,

```
for (i = 0; i < N; i++) {        /* rows */
    for (j = 0; j < N; j++) {    /* columns */
        ... A[i][j] ...
    }
}
```

In Fortran:

```
do j = 1, N                      ! columns
    do i = 1, N                  ! rows
        ... A(i, j) ...
    end do
end do
```

Row-Pointer Layout

Rather than require the rows of an array to be adjacent, they allow them tolie anywhere in memory, and create an auxiliary array of pointers to the rows.
1. If the array has more than two dimensions, it may be allocated as an array ofpointers toarraysofpointers to. . . .This *row-pointer* memory layout requires morespace in most cases.
2. Advantages of Row-Pointer Layout
i. First, it sometimes allowsindividual elements of the array to be accessed more quickly, especially on CISC
machines with slow multiplication instructions.
ii.Second, it allows the rows to have different lengths, withoutdevoting space to holes at the ends of the rows. This representation is sometimescalled a *ragged array*. The lack of holes may sometimes offset the increased spacefor pointers.
iii. Third, it allows a program to construct an array from preexistingrows without copying.
3. C, C++, and C#provide both contiguous and row-pointer organizations for multidimensionalarrays.
Java uses therow-pointer layout for all arrays.
4. The contiguous layout is a true multidimensionalarray.
5. The row-pointer layout is an array of pointers to arrays.

```
char days[][10] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday", "Saturday"
};
...
days[2][3] == 's';  /* in Tuesday */
```

```
char *days[] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday", "Saturday"
};
...
days[2][3] == 's';  /* in Tuesday */
```

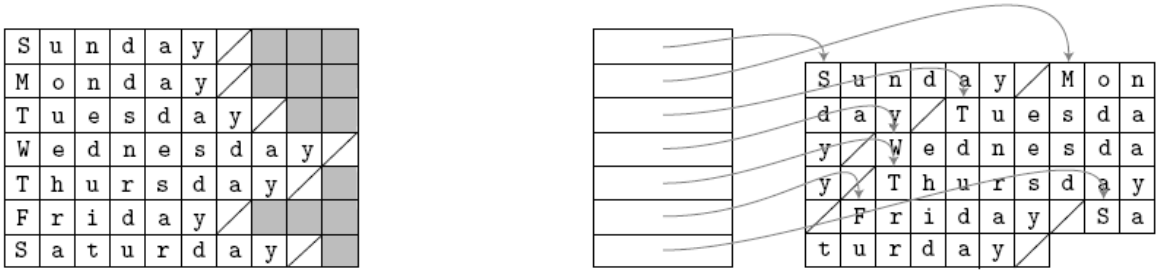


Figure 2.7 Contiguous array allocation vs row pointers in C

The declaration on the left is a true two-dimensional array.Theslashed boxes are NUL bytes; the shaded areas are holes. The declaration on the right is a ragged array of pointers to arraysof characters. In both cases, we have omitted bounds in the declaration that can be deduced from the size of the initializer(aggregate). Both data structures permit individual characters to be accessed using double subscripts, but the memory layout(and corresponding address arithmetic) is quite different.

Contiguous vs row-pointerarray layout

By far the most commonuse of the row-pointer layout inCis to represent arraysof strings. A typical example appears in Figure 2.7. In this example (representingthe days of the week), the row-pointer memory layout consumes 57 bytes forthe characters themselves (including a NUL byte at the end of each string), plus28 bytes for pointers (assuming a 32-bit architecture), for a total of 85 bytes.The contiguous layout alternative devotes 10 bytes to each day (room enoughfor Wednesday and its NUL byte), for a total of 70 bytes. The additional spacerequired for the row-pointer organization comes to 21%.

Address Calculations

Indexing a contiguous array

For the usual contiguous layout of arrays, calculating the address of a particularelement is somewhat

complicated, but straightforward. Suppose a compiler isgiven the following declaration for a three-dimensional array:

A : array [$L1$ . . $U1$] of array [$L2$ . . $U2$] of array [$L3$ . . $U3$] of elem_type;

Let us define constants for the sizes of the three dimensions:

$S3$ = size of elem_type

$$S2 = (U3 - L3 + 1) \times S3$$
$$S1 = (U2 - L2 + 1) \times S2$$

Here the size of a row ($S2$) is the size of an individual element ($S3$) times thenumber of elements in a row (assuming row-major layout). The size of a plane($S1$) is the size of a row ($S2$) times the number of rows in a plane. The address ofA[i, j, k] is then

$$\begin{aligned} &\text{address of A} \\ &+ (i - L1) \times S1 \\ &+ (j - L2) \times S2 \\ &+ (k - L3) \times S3 \end{aligned}$$

if the bounds of the array are known at compiletime, then $S1$, $S2$, and $S3$ are compile-time constants, and the subtractions of lowerbounds can be distributed out of the parentheses:

$$\begin{aligned} &(i \times S1) + (j \times S2) + (k \times S3) + \text{address of A} \\ &-[(L1 \times S1) + (L2 \times S2) + (L3 \times S3)] \end{aligned}$$

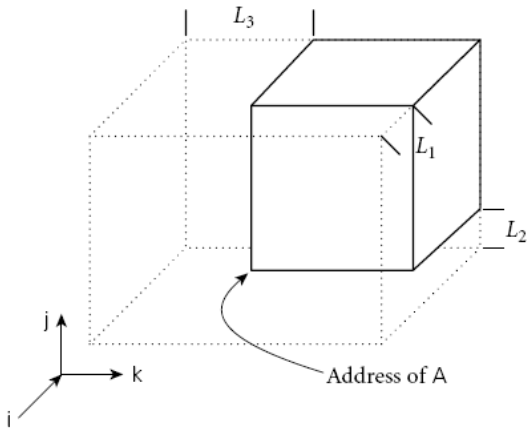


Figure 2.8 Virtual location of an array with nonzero lower bounds.

By computing the constantportions of an array index at compile time, we effectively index into an array whose startingaddress is offset in memory, but whose lower bounds are all zero.

The bracketed expression in this formula is a compile-time constant (assumingthe bounds of A are statically known). If A is a global variable, then the address ofA is statically known as well, and can be incorporated in the bracketed expression.If A is a local variable of a subroutine (with static shape), then the address of Acan be decomposed into a static offset (included in the bracketed expression) plusthe contents of the frame pointer at run time. We can think of the address of Aplus the bracketed expression as calculating the location of an imaginary arraywhose [i, j, k]th element coincides with that of A, but whose lower bound in eachdimension is zero. This imaginary array is illustrated in Figure 2.8.

Static and dynamic portions of an array index

If i, j, and/or k is known at compile time, then additional portions of thecalculation of the address of A[i, j, k] will move from the dynamic to the staticpart of the formula shown above. If all of the subscripts are known, then theentire address can be calculated statically. Conversely, if any of the bounds ofthe array are not known at compile time, then portions of the calculation willmove from the static to the dynamic part of the formula. For example, if $L1$ is notknown until run time, but k is known to be 3 at compile time, then the calculation becomes$(i \times S1) + (j \times S2) - (L1 \times S1)$ + address of A $-$ $[(L2 \times S2) + (L3 \times S3) - (3 \times S3)]$Again, the bracketed part can be computed at compile time. If lower bounds arealways restricted to zero, as they are in C, then they never contribute to run-timecost.

2.5 STRINGS

A string is simply an array of characters.

1. Manylanguages, including C and its descendants, distinguish between literal characters(usually delimited with single quotes) and literal strings (usually delimited withdouble quotes)
2. Other languages (e.g., Pascal) make no distinction: a character isjust a string of length one.
3. Most languages also provide *escape sequences* that allownonprinting characters and quote marks to appear inside of strings.
4. The set of operations provided for strings is strongly tied to the implementationenvisioned by the language designer(s)
5. Several languages that do not in generalallow arrays to change size dynamically do provide this flexibility for strings.

i.First, manipulation of variable-length strings is fundamentalto a huge number of computer applications, and in some sense "deserves" specialtreatment.

ii. Second, the fact that strings are one-dimensional, have one-byte elements,and never contain references to anything else makes dynamic-size stringseasier to implement than general dynamic arrays.

6. Some languages require that the length of a string-valued variable be boundno later than elaboration time, allowing the variable to be implemented as acontiguous array of characters in the current stack frame.
7. Pascal and Ada supporta few string operations, including assignment and comparison for lexicographic ordering.
8. Cprovides only the ability to create a pointer to astring literal.
9. Other languages allow the length of a string-valued variable to change over itslifetime, requiring that the variable be implemented as a block or chain of blocksin the heap.
10. ML and Lisp provide strings as a built-in type.
11. C++, Java, and C#provide them as predefined classes of object, in the formal, object-oriented sense.

12. In all these languages a string variable is a *reference* to a string.
13. Concatenation andother string operators implicitly create new objects.
14. The space used by objects thatare no longer reachable from any variable is reclaimed automatically.

2.6 **Sets**

A programming language set is an unordered collection of an arbitrary number ofdistinct values of a common type.
1. Sets were introduced by Pascal, and are foundin many more recent languages as well.
2. The type from which elements of a set aredrawn is known as the *base* or *universe* type.
3. Set types in Pascal

Pascal supports sets of any discretetype, and provides union, intersection, and difference operations:

```
var A, B, C : set of char;
    D, E : set of weekday;
...
A := B + C;  (* union; A := {x | x is in B or x is in C} *)
A := B * C;  (* intersection; A := {x | x is in B and x is in C} *)
A := B - C;  (* difference; A := {x | x is in B and x is not in C} *)
```

4. Sets appear in the standardlibraries of many object-oriented languages, including C++, Java, and C#.
5. There are many ways to implement sets, including arrays, hash tables, andvarious forms of trees.

2.7 POINTERS AND RECURSIVETYPES

In languages that use a valuemodel of variables, recursive types require the notion of a *pointer*: a variable (or
field) whose value is a reference to some object.
1. Pointers were first introducedin PL/I.
2. In some languages (e.g., Pascal, Ada 83, and Modula-3), pointers are restrictedto point only to objects in the heap.

2.7.1 Syntax and Operations

Operations on pointers include
1. allocation and deallocation of objects in the heap,
2. dereferencing of pointers to access the objects towhich they point,
3. and assignmentof one pointer into another.

The behavior of these operations depends heavily onwhether the language is functional or imperative

1. Functional languages generally employ a reference model for names

2. Variables in an imperative language may useeither a value or a reference model, or some combination of the two

Reference Model

1. *Tree type in ML*

In ML, the datatype mechanism can be used to declare recursive types:

datatypechr_tree = empty | node of char * chr_tree * chr_tree;

Here a chr_tree is either an empty leaf or a node consisting of a character andtwo child trees.

It is natural in ML to include a chr_tree within a chr_tree because everyvariable is a reference.

The tree node (#"R", node (#"X", empty, empty),node (#"Y", node (#"Z", empty, empty), node (#"W", empty, empty)))

would most likely be represented in memory as shown in Figure 2.9.

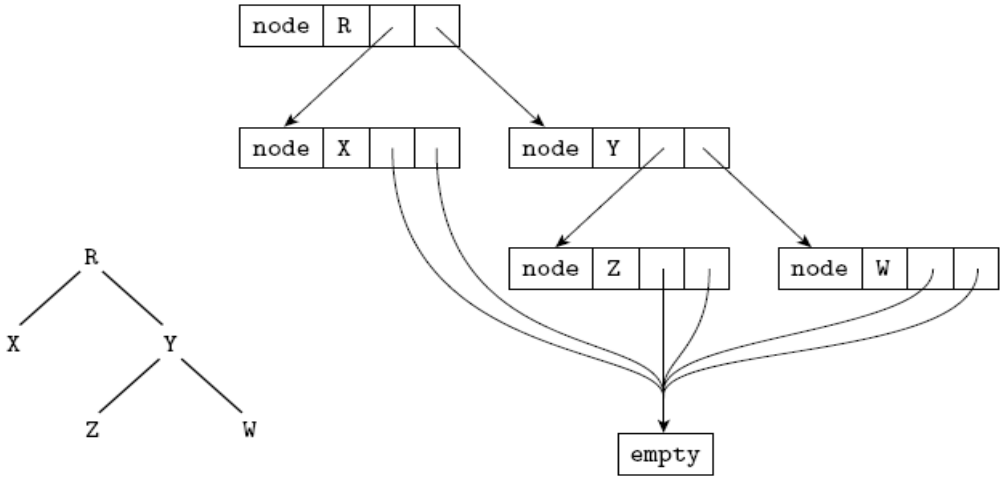


Figure 2.9 Implementation of a tree in ML. The abstract (conceptual) tree is shown at the lower left

2. *Tree type in Lisp*

In Lisp,which uses a reference model of variables but is not statically typed, ourtree could be specified textually as '(#\R (#\X ()()) (#\Y (#\Z ()()) (#\W()()))).

i.Each level of parentheses brackets the elements of a *list*.

ii. the outermost such list contains three elements: the character R and nested liststo represent the left and right subtrees.

iii. each list is a pairof references: one to the head and one to the remainder of the list.

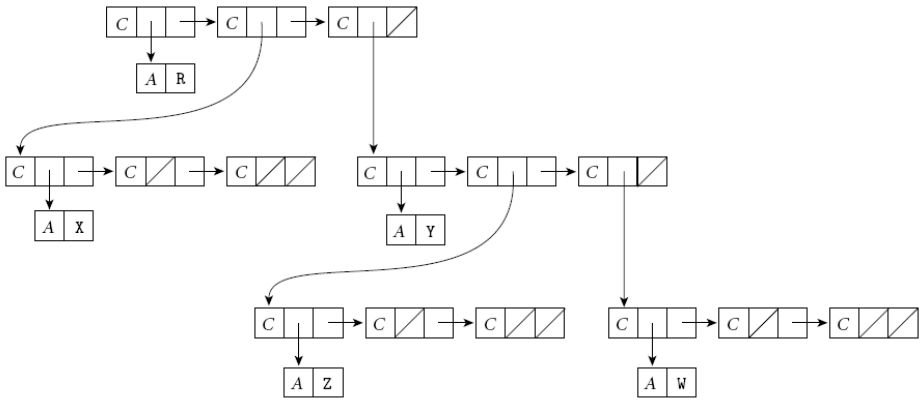


Figure 2.10 Implementation of a tree in Lisp. A diagonal slash through a box indicates a null pointer. The *C* and *A* tags serveto distinguish the two kinds of memory blocks: cons cells and blocks containing atoms.

At the top levelof the figure, the first cons cell points to R; the second and third point to nestedlists representing the left and right subtrees. Each block of memory is tagged toindicate whether it is a cons cell or an *atom*. An atom is anything other than acons cell; that is, an object of a built-in type (integer, real, character, string, etc.), ora user-defined structure (record) or array.

2.7.3 Garbage Collection

Explicit reclamation of heap objects is a serious burden on the programmer and amajor source of bugs (memory leaks and dangling references).

1. An attractive alternative is to have the language implementationnotice when objects are no longer useful and reclaim them automatically (otherwise known as *garbage collection*).
2. Automatic collection is difficult toimplement.
3. Automatic collection also tends tobe slower than manual reclamation.

Reference Counts

When is an object no longer useful?

1. One possible answer is: when no pointers toit exist.
2. The simplest garbage collection technique simply places a counter ineach object that keeps track of the number of pointers that refer to the object.
3. When the object is created, this *reference count* is set to 1, to represent the pointerreturned by the new operation.
4. When one pointer is assigned into another, therun-time system decrements the reference count of the object formerly referred toby the assignment's left-hand side, and increments the count of the object referredto by the right-hand side.

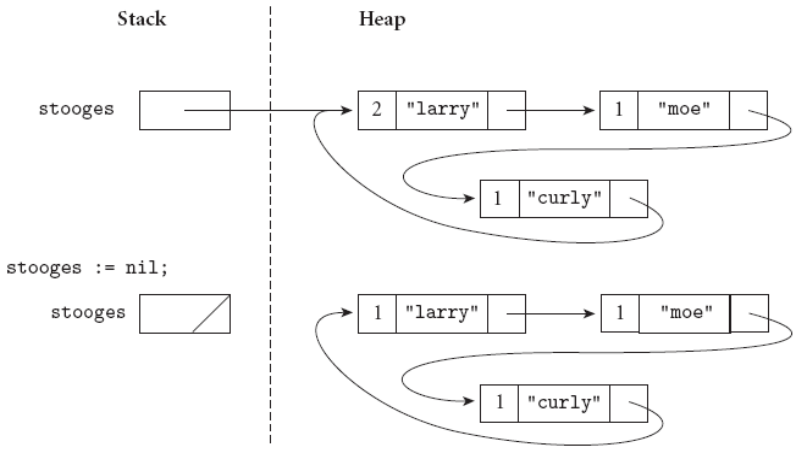


Figure 2.12 Reference counts and circular lists. The list shown here cannot be found via any program variable, but because it is circular, every cell contains a nonzero count.

5. When a reference count reaches zero, its object canbe reclaimed.
6. The standard technique to track the location of every pointer information relies on *typedescriptors* generated by the compiler.
7. There is one descriptor for every distincttype in the program, plus one for the stack frame of each subroutine, and onefor the set of global variables.

Tracing Collection

1. A better definition of a "useful" object is one that can be reached by following achain of valid pointers starting from something that has a name (i.e., somethingoutside the heap).
2. Tracingcollectors work by recursively exploring the heap, starting from external pointers,to determine what is useful.

Mark-and-Sweep

1. The classic mechanism to identify useless blocks, under thismore accurate definition, is known as *mark-and-sweep*
2. It proceeds in three mainsteps, executed by the garbage collector when the amount of free space remaining
in the heap falls below some minimum threshold.

1. The collector walks through the heap, tentatively marking every block as "useless."

2. Beginning with all pointers outside the heap, the collector recursively exploresall linked data structures in the program, marking each newly discovered blockas "useful." (When it encounters a block that is already marked as "useful," thecollector knows it has reached the block over some previous path, and returns
withoutrecursing.)

3. The collector again walks through the heap, moving every block that is stillmarked "useless" to the free list.

Pointer Reversal

As the collector explores the path to a given block, it *reverses* the pointers it follows,so that each points*back* to the previous block instead of forward to the next.

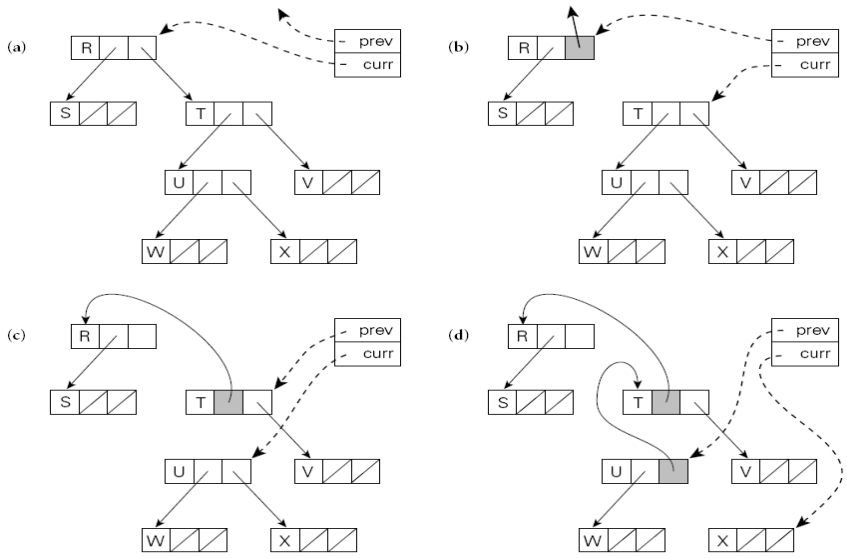


Figure 2.13 Heap exploration via pointer reversal

The block currently under examination is indicated by the curr pointer.The previous block is indicated by the prev pointer. As the garbage collector moves from one block to the next, it changes thepointer it follows to refer back to the previous block. When it returns to a block it restores the pointer. Each reversed pointer
must be marked (indicated with a shaded box), to distinguish it from other, forward pointers in the same block.

1. To return from block X to block U (after part (d) of the figure), the collectorwill use the reversed pointer in U to restore its notion of previous block (T).
2. Itwill then flip the reversed pointer back to X and update its notion of currentblock to U.

Stop-and-Copy

1. In a language with variable-size heap blocks, the garbage collectorcan reduce external fragmentation by performing storage compaction.
2. Manygarbage collectors employ a technique known as *stop-and-copy* that achieves compactionwhile simultaneously eliminating Steps 1 and 3 in the standard mark-andsweepalgorithm.

i. Divide the heap into two regions of equal size.
ii. All allocation happens in the first half.
iii. When this half is (nearly) full, the collectorbegins its exploration of reachable data structures.
iv. Each reachable block is copiedinto the second half of the heap, with no external fragmentation.

Generational Collection

1. exploiting the observation that mostdynamically allocated objects are short-lived.
2. The heap is divided into multipleregions (often two)

3. When space runs low the collector first examines the youngestregion (the “nursery”), which it assumes is likely to have the highest proportionof garbage.
4. Only if it is unable to reclaim sufficient space in this region does thecollector examine the next-older region.

Conservative Collection

1. Everything that seems to point into a heapblock is in fact a valid pointer, then we can proceed with mark-and-sweep collection.
2. When space runs low, the collector (as usual) tentatively marks all blocks inthe heap as useless.
3. It then scans all word-aligned quantities in the stack and inglobal storage.
4. If any of these words appears to contain the address of somethingin the heap, the collector marks the block that contains that address as useful.
5. Recursively, the collector then scans all word-aligned quantities in the block, andmarks as useful any other blocks whose addresses are found therein.
6. Finally, the collector reclaims any blocks that are still marked useless.

2.8 LISTS

A list is defined recursively as either the empty list or a pair consisting of an object(which may be either a list or an atom) and another (shorter) list.

1. Lists are ideallysuited to programming in functional and logic languages.
2. In Lisp, in fact, a program *is* a list, and can extend itself at run time by constructinga list and executing it.
3. Lists can also be used in imperative programs.
4. a list class is easy to write in most object-oriented languages.
5. Most scripting languages provide extensive list support.
6. Lists work best in a languagewith automatic garbage collection.

Lists in ML and Lisp

1. lists in ML are homogeneous:every element of the list must have the same type.

i. An ML list is usually a chain of blocks, each ofwhich contains an element and a pointer to the next block

ii. Clu resembles ML

iii. An ML list is enclosedList notation in square brackets, with elements separated by commas: [a, b, c, d].

2. Lisp lists, by contrast, areheterogeneous: any object may be placed in a list.

i. A Lisp list is a chainof cons cells, each of which contains *two* pointers, one to the element and one to

the next cons cell.

ii. Python and Prolog resemble Lisp.

iii. A Lisplist is enclosed in parentheses, with elements separated by white space: (a b cd).
a*proper* list: one whose innermostpair consists of the final element and the empty list.
an*improper* list: whose final pair contains two elements.
3. Lispsystems provide amore general, but cumbersome *dotted* list notation that capturesboth proper and improper lists.
4. A dotted list is either an atom (possibly null)or a pair consisting of two dotted lists separated by a period and enclosed inparentheses. The dotted list (a . (b . (c . (d . null)))) is the same as (a b cd).
5. The list (a . (b . (c . d))) is improper; its final cons cell contains a pointerto d in the second position, where a pointer to a list is normally required.
Basic list operations in Lisp
(cons 'a '(b)) =⇒(a b)
(car '(a b)) =⇒a
(car nil) =⇒ *??*
(cdr '(a b c)) =⇒(b c)
(cdr '(a)) =⇒nil
(cdr nil) =⇒ *??*
(append '(a b) '(c d)) =⇒(a b c d)
Basic list operations in ML
a :: [b] =⇒[a, b]
hd [a, b] =⇒a
hd [ ] =⇒ *run-time exception*
tl [a, b, c] =⇒[b, c]
tl [a] =⇒nil
tl [ ] =⇒ *run-time exception*
[a, b] @ [c, d] =⇒[a, b, c, d]

list functions
1. test a list to see if it is empty;
2. return the length of a list;
3. return the *n*th elementof a list, or a list consisting of all but the first *n* elements;
4. reverse the order of theelements of a list;
5. search a list for elements matching some predicate;
6. apply afunction to every element of a list, returning the results as a list.

2.9 FILES AND INPUT/OUTPUT

1. Interactive I/O generally implies communicationwith human users or physical devices, which work in parallel with the runningprogram, and whose input to the program may depend on earlier output from

the program (e.g., prompts).
i. I/O is one of the most difficult aspects of a language to design, and one thatdisplays the least commonality from one language to the next.
ii. Some languagesprovide built-in file data types and special syntactic constructs for I/O.
iii. Othersrelegate I/O entirely to library packages, which export a (usually opaque) filetype and a variety of input and output subroutines.
iv. The principal advantage oflanguage integration is the ability to employ non–subroutine-call syntax, and to
performoperations (e.g., type checking on subroutine calls with varying numbersof parameters) that may not otherwise be available to library routines
2. Files generally refer to off-line storage implementedby the operating system.
3. Files may be further categorized into
    i. *temporary* and
    ii. *persistent*.
4. Temporary files exist for the duration ofa single program run; their purpose is to store information that is too large to fitin the memory available to the program.
5. Persistent files allow a program to readdata that existed before the program began running, and to write data that willcontinue to exist after the program has ended.

2.10 EQUALITYTESTING AND ASSIGNMENT

Consider for example the problem of comparing two character strings. Shouldthe expression s = t determine whether s and t.

are aliases for one another?
occupy storage that is bit-wise identical over its full length?
contain the same sequence of characters?
would appear the same if printed?

1. The second of these tests is probably too low-level to be of interest in most programs;
it suggests the possibility that a comparison might fail because of garbagein currently unused portions of the space reserved for a string.
2. The other threealternatives may all be of interest in certain circumstances, and may generatedifferent results.
3. In many cases the definition of equality boils down to the distinction between
l-values and r-values.
4. *shallow*comparison: should expressions be consideredequal only if they refer to the same object,
Under a reference model of variables: a shallow assignment a := bwill make a refer to the object to which b refers.

Under a value model of variables: a shallow assignment will copy the value of b into a
5.*deep*comparison:if the objects to whichthey refer are in some sense equal?
Under a reference model of variables:A deep assignment will create acopy of the object to which b refers, and make a refer to the copy.
6. Most programming languages employ both shallow comparisons and shallowassignment.
7. Scheme, for example, has three general-purposeequality-testing functions:

(eq? a b) ; do a and b refer to the same object?
(eqv? a b) ; are a and b known to be semantically equivalent?
(equal? a b) ; do a and b have the same recursive structure?

Both eq?andeqv? perform a shallow comparison.

8. Deep assignments are relatively rare.
i. They are used primarily in distributedcomputing, and in particular for parameter passing in remote procedure call(RPC) systems.

MODULE – 3

- ➔ *DATA ABSTRACTION*, in which the principal purpose of the abstraction is to represent information
- ➔ ***CONTROL ABSTRACTION***, in which the principal purpose of the abstraction is to perform a well-defined operation
- ➔ **SUBROUTINES** are the principal mechanism for control abstraction in most programming languages.

Subroutines are the principal mechanism for control abstraction in most programming languages.
A subroutine performs its operation on behalf of a ***caller***, who waits for the subroutine to finish before continuing execution.
Most subroutines are parameterized: the caller passes arguments that influence the subroutine's behavior, or provide it with data on which to operate. Arguments are also called *actual parameters*. They are mapped to the subroutine's *formal parameters* at the time a call occurs.
A subroutine that returns a value is usually called a *function*.
A subroutine that does not return a value is usually called a *procedure*.
Most languages require subroutines to be declared before they are used, though a few do not (including Fortran, C, and Lisp).
Declarations allow the compiler 8to verify that every call to a subroutine is consistent with the declaration; for example, that it passes the right number and types of arguments.
Each routine, as it is called, is given a new *stack frame*, or *activation record*, at the top of the stack. This frame may contain arguments and/or return values, bookkeeping information

(including the return address and saved registers), local variables, and/or temporaries. When a subroutine returns, its frame is popped from the stack.
At any given time, the *stack pointer* register contains the address of either the last used location at the top of the stack, or the first unused location, depending on convention. The *frame pointer* register contains an address within the frame.
Objects in the frame are accessed via displacement addressing with respect to the frame pointer. If the size of an object (e.g., a local array) is not known at compile time, then the object is placed in a variable-size area at the top of the frame; its address and dope vector (descriptor) are stored in the fixed-size portion of the frame, at a statically known offset from the frame pointer. If there are no variable-size objects, then every object within the frame has a statically known offset from the stack pointer, and the implementation may dispense with the frame pointer, freeing up a register for other use. If the size of an argument is not known at compile time, then the argument may be placed in a variable-size. portion of the frame *below* the other arguments, with its address and dope vector at known offsets from the frame pointer. Alternatively, the caller may simply pass a temporary address and dopevector, counting on the called routine to copy the argument into the variable-size area at the top of the frame.

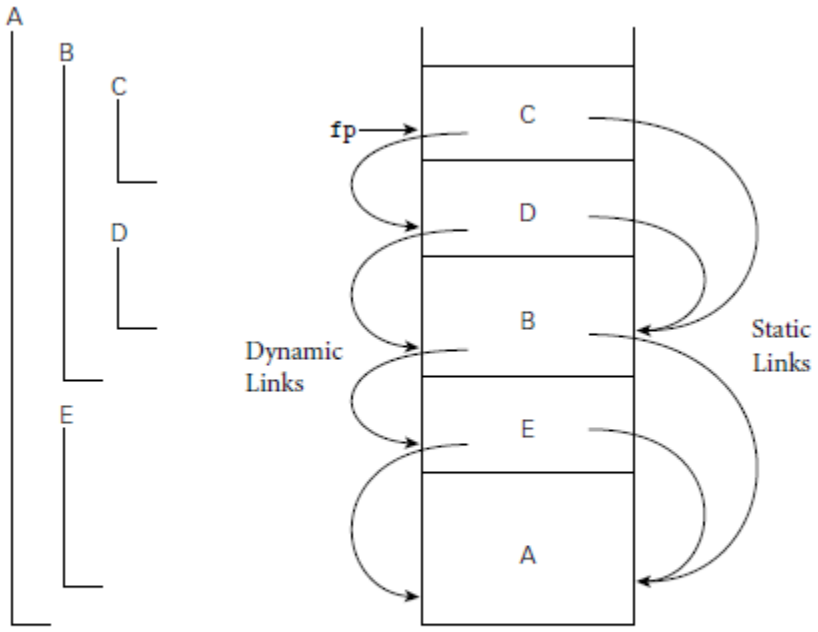


Figure 8.1 Example of subroutine nesting, taken from Figure 3.5. Within B, C, and D, all five routines are visible. Within A and E, routines A, B, and E are visible, but C and D are not. Given the calling sequence A, E, B, D, C, in that order, frames will be allocated on the stack as shown at right, with the indicated static and dynamic links.

In a language with nested subroutines and static scoping (e.g., Pascal, Ada, ML, Common Lisp, or Scheme), objects that lie in surrounding subroutines, and that are thus neither local nor global, can be found by maintaining a *static chain.* Each stack frame contains a reference to the frame of the lexically surrounding subroutine. This reference is called the ***static link.*** By analogy, the saved value of the frame pointer, which will be restored on subroutine return, is called the ***dynamic link***. The static and dynamic links may or may not be the same, depending on whether

the current routine was called by its lexically surrounding routine, or by some other routine nested in that surrounding routine.

Whether or not a subroutine is called directly by the lexically surrounding routine, we can be sure that the surrounding routine is active; there is no other way that the current routine could have been visible, allowing it to be called. If subroutine D is called directly from B, then clearly B's frame will already be on the stack. *How else could D be called?* It is not visible in A or E, because it is nested inside of B. A moment's thought makes clear that it is only when control enters B (placing B's frame on the stack) that D comes into view. It can therefore be called by C, or by any other routine (not shown) that is nested inside C or D, but only because these are also within B.

CALLING SEQUENCES

Maintenance of the subroutine call stack is the responsibility of the *calling sequence*—the code executed by the caller immediately before and after a subroutine call—and of the ***prologue*** (code executed at the beginning) and ***epilogue*** (code executed at the end) of the subroutine itself. Sometimes the term "calling sequence" is used to refer to the combined operations of the caller, the prologue, and the epilogue.

Tasks that must be accomplished on the way into a subroutine include passing parameters, saving the return address, changing the program counter, changing the stack pointer to allocate space, saving registers (including the frame pointer) that contain important values and that may be overwritten by the callee, changing the frame pointer to refer to the new frame, and executing initialization code for any objects in the new frame that require it.

Tasks that must be accomplished on the way out include passing return parameters or function values, executing finalization code for any local objects that require it, de-allocating the stack frame (restoring the stack pointer), restoring other saved registers (including the frame pointer), and restoring the program counter. Some of these tasks (e.g., passing parameters) must be performed by the caller, because they differ from call to call.

Most of the tasks, however, can be performed either by the caller or the callee. In general, we will save space if the callee does as much work as possible: tasks performed in the callee appear only once in the target program, but tasks performed in the caller appear at every call site, and the typical subroutine is called in more than one place.

Maintaining the Static Chain

In languages with nested subroutines, at least part of the work required to maintain the static chain must be performed by the caller, rather than the callee, because this work depends on the lexical nesting depth of the caller. The standard approach is for the caller to compute the callee's static link and to pass it as an extra, hidden parameter.

Two subcases arise:

1. *The callee is nested (directly) inside the caller*.

In this case, the callee's static link should refer to the caller's frame. The caller therefore passes its own frame pointer as the callee's static link.

2. *The callee is k ≥0 scopes "outward"—closer to the outer level of lexical nesting.*
In this case, all scopes that surround the callee also surround the caller (otherwise the callee would not be visible). The caller dereferences its own static link *k* times and passes the result as the callee's static link.

A Typical Calling Sequence
The stack pointer (sp) points to the first unused location on the stack (or the last used location, depending on the compiler and machine). The frame pointer (fp) points to a location near the bottom of the frame. Space for all arguments is reserved in the stack, even if the compiler passes some of them in registers (the callee will need a place to save them if it calls a nested routine).

To maintain this stack layout, the calling sequence might operate as follows.

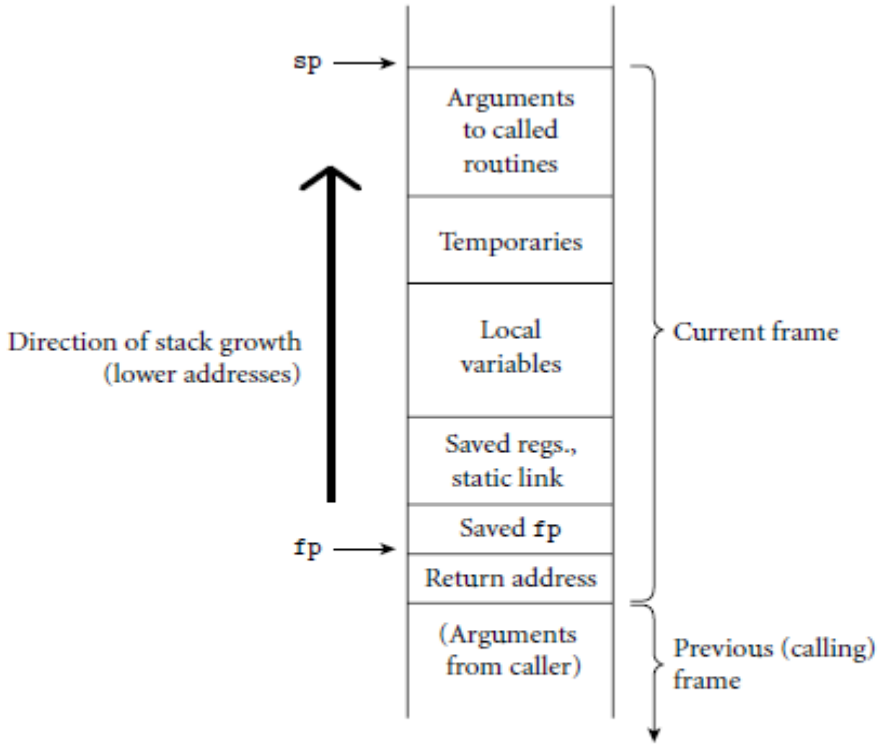


Figure 8.2 **A typical stack frame.** Though we draw it growing upward on the page, the stack actually grows downward toward lower addresses on most machines. Arguments are accessed at positive offsets from the fp. Local variables and temporaries are accessed at negative offsets from the fp. Arguments to be passed to called routines are assembled at the top of the frame, using positive offsets from the sp.

The caller
1. saves any caller-saves registers whose values will be needed after the call
2. computes the values of arguments and moves them into the stack or registers
3. computes the static link (if this is a language with nested subroutines), and passes it as an extra, hidden argument

4. uses a special subroutine call instruction to jump to the subroutine, simultaneously passing the return address on the stack or in a register

In its prologue, the callee
1. allocates a frame by subtracting an appropriate constant from the sp
2. saves the old frame pointer into the stack, and assigns it an appropriate new value
3. saves any callee-saves registers that may be overwritten by the current routine (including the static link and return address, if they were passed in registers)

After the subroutine has completed, the epilogue
1. moves the return value (if any) into a register or a reserved location in the stack
2. restores callee-saves registers if needed
3. restores the fp and the sp
4. jumps back to the return address

Finally, the caller
1. moves the return value to wherever it is needed
2. restores caller-saves registers if needed

Special-Case Optimizations
Many parts of the calling sequence, prologue, and epilogue can be omitted in common cases. If the hardware passes the return address in a register, then a *leaf routine* (a subroutine that makes no additional calls before returning) can simply leave it there; it does not need to save it in the stack. Likewise it need not save the static link or any caller-saves registers.

A subroutine with no local variables and nothing to save or restore may not even need a stack frame on a RISC machine. The simplest subroutines (e.g., library routines to compute the standard mathematical functions) may not touch memory at all, except to fetch instructions: they may take their arguments in registers, compute entirely in (caller-saves) registers, call no other routines, and return their results in registers. As a result they may be extremely fast.

Displays
One disadvantage of static chains is that access to an object in a scope k levels out requires that the static chain be dereferenced k times. If a local object can be loaded into a register with a single (displacement mode) memory access, an object k levels out will require $k + 1$ memory accesses. This number can be reduced to a constant by use of a *display*.

Case Studies: C on the MIPS; Pascal on the x86
Calling sequences differ significantly from machine to machine and even compiler to compiler (though typically a hardware manufacturer publishes a suggested set of conventions for a given architecture, to promote interoperability among program components produced by different compilers). Some of the most significant differences can be found in a comparison of CISC and RISC conventions.

- Compilers for CISC machines tend to pass arguments on the stack; compilers for RISC machines tend to pass arguments in registers.

- Compilers for CISC machines usually dedicate a register to the frame pointer; compilers for RISC machines often do not.

- Compilers for CISC machines often rely on special-purpose instructions to Implement parts of the calling sequence; available instructions on a RISC machines are typically much simpler.

In-Line Expansion

As an alternative to stack-based calling conventions, many language implementations allow certain subroutines to be expanded in-line at the point of call. A copy of the "called" routine becomes a part of the "caller"; no actual subroutine call occurs.

In-line expansion avoids a variety of overheads, including space allocation, branch delays from the call and return, maintaining the static chain or display, and (often) saving and restoring registers.

It also allows the compiler to perform code improvements such as global register allocation, instruction scheduling, and common sub expression elimination across the boundaries between subroutines, something that most compilers can't do otherwise.

QUESTION BANK

1. What is a subroutine *calling sequence*? What does it do? What is meant by the subroutine *prologue* and *epilogue*?
2. How do calling sequences typically differ in CISC and RISC compilers?
3. Describe how to maintain the *static chain* during a subroutine call.
4. What is a *display*? How does it differ from a static chain?
5. What are the purposes of the *stack pointer* and *frame pointer* registers? Why does a subroutine often need both?
6. Why do RISC machines typically pass subroutine parameters in registers rather than on the stack?
7. List the optimizations that can be made to the subroutine calling sequence in important special cases (e.g., *leaf routine*s).
8. How does an *in-line subroutine* differ from a *macro*?
9. Under what circumstances is it desirable to expand a subroutine in-line?

PARAMETER PASSING

Most subroutines are parameterized: they take arguments that control certain aspects of their behavior, or specify the data on which they are to operate.

Parameter names that appear in the declaration of a subroutine are known as *formal parameters*.

Variables and expressions that are passed to a subroutine in a particular call are known as *actual parameters*.

Most languages use a prefix notation for calls to user-defined subroutines, with the subroutine name followed by a parenthesized argument list. Lisp places the function name inside the parentheses, as in (max a b). ML allows the programmer to specify that certain names represent infix operators, which appear between a pair of arguments:

```
infixr 8 tothe;                                    (* exponentiation *)
fun x tothe 0 = 1.0
      |x tothe n = x * (x tothe(n-1));          (* assume n >= 0 *)
```

The *infixr* declaration indicates that *tothe* will be a right-associative binary infix operator, at precedence level 8 (multiplication and division are at level 7, addition and subtraction at level 6).
Fortran 90 also allows the programmer to define new infix operators, but it requires their names to be bracketed with periods (e.g., A .cross. B), and it gives them all the same precedence. Smalltalk uses infix (or "mixfix") notation (without precedence) for all its operations.

Control abstraction in Lisp and Smalltalk
The uniformity of Lisp and Smalltalk syntax makes control abstraction particularly effective: user-defined subroutines (functions in Lisp, "messages" in Smalltalk) use the same style of syntax as built-in operations. As an example, consider

if. . . then . . . else:

```
if a > b then max := a else max := b; (* Pascal *)
(if (> a b) (setf max a) (setf max b)) ; Lisp
(a > b) ifTrue: [max <- a] ifFalse: [max <- b]. "Smalltalk"
```

In Pascal or C it is clear that if. . . then . . . else is a built-in language construct: it does not look like a subroutine call.
In Lisp and Smalltalk, on the other hand, the analogous conditional constructs are syntactically indistinguishable from user-defined operations. They are in fact defined in terms of simpler concepts, rather than being built in, though they require a special mechanism to evaluate their arguments in normal, rather than applicative, order

Parameter Modes

Suppose for the moment that x is a global variable in a language with a value model of variables, and that we wish to pass x as a parameter to subroutine p:

```
p(x);
```

From an implementation point of view, we have two principal alternatives: we may provide p with a copy of x's value, or we may provide it with x's address.
The two most common parameter-passing modes, called *call-by-value* and *callby- reference*, are designed to reflect these implementations.

- With value parameters, each actual parameter is assigned into the corresponding formal parameter when a subroutine is called; from then on, the two are independent.
- With reference parameters, each formal parameter introduces, within the body of the subroutine, a new name for the corresponding actual parameter.

If the actual parameter is also visible within the subroutine under its original name (as will generally be the case if it is declared in a surrounding scope), then the two names are *aliases* for the same object, and changes made through one will be visible through the other. In most languages
an actual parameter that is to be passed by reference must be an l-value; it cannot be the result of an arithmetic operation, or any other value without an address.
As a simple example, consider the following pseudocode:

```
x : integer – – global
procedure foo(y : integer)
y := 3
print x
. . .
x := 2
foo(x)
print x
```

- If y is passed to foo by value, then the assignment inside foo has no visible effect— y is private to the subroutine—and the program prints 2 twice.
- If y is passed to foo by reference, then the assignment inside foo changes x—y is just a local name for x—and the program prints 3 twice.

If the purpose of call-by-reference is to allow the called routine to modify the actual parameter, we can achieve a similar effect using *call-by-value/result*, a mode first introduced in Algol W.

Like call-by-value, call-by-value/result copies the actual parameter into the formal parameter at the beginning of subroutine execution.
Unlike call-by-value, it *also* copies the formal parameter back into the actual parameter when the subroutine returns.

In the psuedocode given above - value/result would copy result x into y at the beginning of foo, and y into x at the end of foo.
Because foo accesses x directly in-between, the program's visible behavior would be different than it was with call-by-reference: the assignment of 3 into y would not affect x until after the inner print statement, so the program would print 2 and then 3.

In Pascal, parameters are passed by value by default; they are passed by reference if preceded by the keyword *var* in their subroutine header's formal parameter list.
Parameters in C are always passed by value, though the effect for arrays is unusual: because of the interoperability of arrays and pointers in C, what is passed by value is a pointer; changes to array elements accessed through this pointer are visible to the caller.

To allow a called routine to modify a variable other than an array in the caller's scope, the C programmer must pass the address of the variable explicitly:

```
void swap(int *a, int *b) { int t = *a; *a = *b; *b = t; }
...
swap(&v1, &v2);
```

Fortran passes all parameters by reference, but does not require that every actual parameter be an l-value. If a built-up expression appears in an argument list, the compiler creates a temporary variable to hold the value, and passes this variable by reference.

A Fortran subroutine that needs to modify the values of its formal parameters without modifying its actual parameters must copy the values
into local variables, and modify those instead.

Call-by-Sharing
Call-by-value and call-by-reference make the most sense in a language with a value model of variables: they determine whether we copy the
variable or pass an alias for it. Neither option really makes sense in a language like Smalltalk, Lisp, ML, or Clu, in which a variable is already a reference.
Here it is most natural simply to pass the reference itself, and let the actual and formal parameters refer to the same object. Clu calls this
mode *call-by-sharing*. It is different from call-by-value because, although we do copy the actual parameter into the formal parameter, both of them are references; if we modify the object to which the formal parameter refers, the program will be able to see those changes through the actual parameter after the subroutine returns.
Call-by-sharing is also different from call-by-reference, because although the called routine can change the value of the object to which the actual parameter refers, it cannot change the *identity* of that object.

The Purpose of Call-by-Reference In a language that provides both value and reference parameters (e.g., Pascal orModula), there are two principal reasons why the programmer might choose one over the other. → First, if the called routine is supposed to change the value of an actual parameter (argument), then the programmer must pass the parameter by reference. Conversely, to ensure that the called routine cannot modify the argument, the programmer can pass the parameter by value.
→Second, the implementation of value parameters requires copying actuals to formals, a potentially time-consuming operation when arguments are large. Reference parameters can be implemented simply by passing an address

EXCEPTION HANDLING.
An exception can be defined as an unexpected—or at least unusual—condition that arises during program execution, and that cannot easily be handled in the local context. It may be detected automatically by the language implementation, or the program may *raise* it explicitly. The most common exceptions are various sorts of run-time errors. In an I/O library, for example, an input routine may encounter the end of its file before it can read a requested value, or it may find

punctuation marks or letters on the input when it is expecting digits. To cope with such errors without an exception-handling mechanism, the programmer has basically three options, none of which is entirely satisfactory:

1. "Invent" a value that can be used by the caller when a real value could not be returned.

2. Return an explicit "status" value to the caller, who must inspect it after every call. The status may be written into an extra, explicit parameter, stored in a global variable, or encoded as otherwise invalid bit patterns of a function's regular return value.

3. Rely on the caller to pass a closure (in languages that support them) for an error-handling routine that the normal routine can call when it runs into trouble. The first of these options is fine in certain cases, but does not work in the general case. Options 2 and 3 tend to clutter up the program, and impose overhead that we should like to avoid in the common case. The tests in option 2 are particularly offensive: they obscure the normal flow of events in the common case. Because they are so tedious and repetitive, they are also a common source of errors; one can easily forget a needed test. Exception-handling mechanisms address these issues by moving error-checking code "out of line," allowing the normal case to be specified simply, and arranging for control to branch to a *handler* when appropriate. In many languages, dynamic semantic errors automatically result in exceptions, which the program can then catch. The programmer can also define additional, application-specific exceptions. Examples of predefined exceptions include arithmetic overflow, division by zero, end-of-file on input, subscript and subrange errors, and null pointer dereference. The rationale for defining these as exceptions (rather than as fatal errors) is that they may arise in certain valid programs. Some other dynamic errors (e.g., return from a subroutine that has not yet designated a return value) are still fatal in most languages. In C++ and Common Lisp, exceptions are all programmer defined. In PHP, the set_error_handler function can be used to turn built-in semantic errors into ordinary exceptions. In Ada, some of the predefined exceptions can be *suppressed* by means of a pragma. If a subroutine raises an exception but does not catch it internally, it may "return" in an unexpected way. This possibility is an important part of the routine's interface to the rest of the program. Consequently, several languages, including Clu, Modula-3, C++, and Java, include in each subroutine header a list of the exceptions that may propagate out of the routine. This list is mandatory in Modula-3: it is a run-time error if an exception arises that does not appear in the header, but is not caught internally. The list is optional in C++: if it appears, the semantics are the same as in Modula-3; if it is omitted, all exceptions are permitted to propagate. Java adopts an intermediate approach: it segregates its exceptions into "checked" and "unchecked" categories. Checked exceptions must be declared in subroutine headers; unchecked exceptions need not. Unchecked exceptions are typically run-time errors that most programs will want to be fatal (e.g., subscript out of bounds)—and that would therefore be a nuisance to declare in every function—but that a highly robust program may want to catch if they occur in library routines.

Exception Propagation

In most languages, a block of code can have a *list* of exception handlers. In C++:

```
try { // try to read from file
...
// potentially complicated sequence of operations
// involving many calls to stream I/O routines
...
```

```
} catch(end_of_file) {
...
} catch(io_error e) {
// handler for any io_error other than end_of_file
...
} catch(...) {
// handler for any exception not previously named
// (in this case, the triple-dot ellipsis is a valid C++ token;
// it does not indicate missing code)
}
```

When an exception arises, the handlers are examined in order; control is transferred to the first one that *matches* the exception. In C++, a handler matches if it names a class from which the exception is derived, or if it is a catch-all (...). In the example here, let us assume that end_of_file is a subclass of io_error. Then an end_of_file exception, if it arises, will be handled by the first of the three catch clauses. All other I/O errors will be caught by the second; all non-I/O errors will be caught by the third. If the last clause were missing, non-I/O errors would continue to propagate up the dynamic chain. _ An exception that is declared in a recursive subroutine will be caught by the innermost handler for that exception at run time. If an exception propagates out of the scope inwhich it was declared, it can no longer be named by a handler, and thus can be caught only by a "catch-all" handler. In a language with concurrency, one must consider what will happen if an exception is not handled at the outermost level of a concurrent thread of control. InModula-3, the entire programterminates abnormally; in Ada and Java, the affected thread terminates quietly; in C# the behavior is implementation defined.

Handlers on Expressions

In an expression-oriented language such as ML or Common Lisp, an exception handler is attached to an expression, rather than to a statement. Since execution of the handler replaces the unfinished portion of the protected code when an exception occurs, a handler attached to an expression must provide a value for the expression.

Implementation of Exceptions

The most obvious implementation for exceptions maintains a linked-list stack of handlers. When control enters a protected block, the handler for that block is added to the head of the list. When an exception arises, either implicitly or as a result of a raise statement, the language run-time system pops the innermost handler off the list and calls it. The handler begins by checking to see if it matches the exception that occurred; if not, it simply reraises it:

```
if exception matches duplicate in set
. . .
else
reraise exception
```

To implement propagation back down the dynamic chain, each subroutine has an implicit handler that performs the work of the subroutine epilogue code and then reraises the exception. _

If a protected block of code has handlers for several different exceptions, they

Multiple exceptions per
handler
are implemented as a single handler containing a multiarm if statement:
if exception matches end of file
. . .
elsif exception matches io error
. . .
else
. . . – – "catch-all" handler _
The problem with this implementation is that it incurs run-time overhead in the common case. Every protected block and every subroutine begins with code to push a handler onto the handler list, and ends with code to pop it back off the list.We can usually do better.
The only real purpose of the handler list is to determine which handler is active. Since blocks of source code tend to translate into contiguous blocks of machine language instructions, we can capture the correspondence between handlers and protected blocks in the form of a table generated at compile time. Each entry in the table contains two fields: the starting address of a block of code and the address of the corresponding handler. The table is sorted on the first field.When an exception occurs, the language run-time system performs binary search in the table, using the program counter as key, to find the handler for the current block. If that handler reraises the exception, the process repeats: handlers themselves are blocks of code, and can be found in the table. The only subtlety arises in the case of the implicit handlers associated with propagation out of subroutines: such a handler must ensure that the reraise code uses the return address of the subroutine, rather than the current program counter, as the key for table lookup.

Coroutines
Given an understanding of the layout of the run-time stack, we can now consider the implementation of more general control abstractions *coroutines* in particular.
Like a continuation, a coroutine is represented by a closure (a code address and a referencing environment), into which we can jump by means of a nonlocal goto, in this case a special operation known as transfer. The principal difference between the two abstractions is that a continuation is a constant it does not change once created while a coroutine changes every time it runs. When we goto a continuation, our old program counter is lost, unless we explicitly create a new continuation to hold it.When we transfer from one coroutine to another, our old program counter is saved: the coroutine we are leaving is updated to reflect it. Thus, if we performa goto into the same continuationmultiple times, each jump will start at precisely the same location, but if we performa transfer into the same coroutine multiple times, each jump will take up where the previous one left off.

Stack Allocation
Because they are concurrent (i.e., simultaneously started but not completed), coroutines cannot share a single stack: their subroutine calls and returns, taken as a whole, do not occur

in last-in-first-out order. If each coroutine is declared at the outermost level of lexical nesting (as required in Modula-2), then their stacks are entirely disjoint: the only objects they share are global, and thus statically allocated.

Most operating systems make it easy to allocate one stack, and to increase its portion of the virtual address space as necessary during execution. It is usually not easy to allocate an arbitrary number of such stacks; space for coroutines is something of an implementation challenge.
The simplest solution is to give each coroutine a fixed amount of statically allocated stack space.

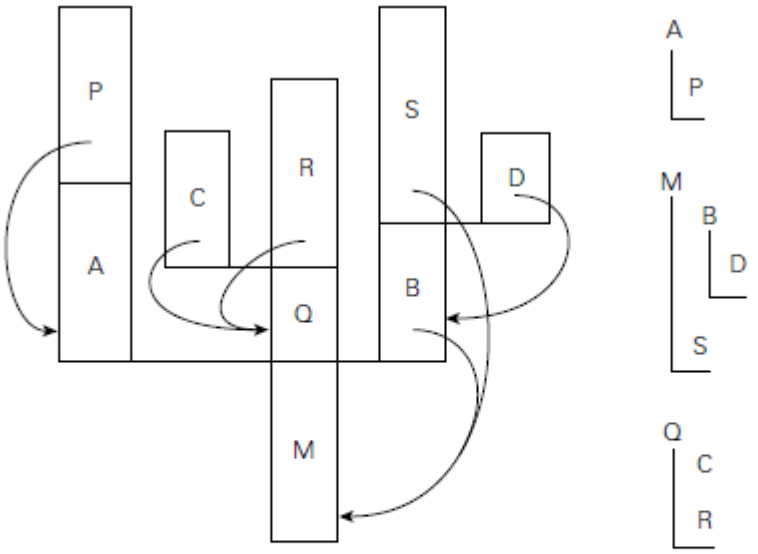


Figure 8.6 **A cactus stack.** Each branch to the side represents the creation of a coroutine (A, B, C, and D). The static nesting of blocks is shown at right. Static links are shown with arrows. Dynamic links are indicated simply by vertical arrangement: each routine has called the one above it. (Coroutine B, for example, was created by the main program, M. B in turn called subroutine S and created coroutine D.)

This approach is adopted inModula-2, which requires the programmer
to specify the size and location of the stack when initializing a coroutine.
It is a run-time error for the coroutine to need additional space. Some Modula-2 implementations catch the overflow and halt with an error message; others display abnormal behavior. If the coroutine uses less space than it is given, the excess is simply wasted.

If stack frames are allocated from the heap, as they are in most Lisp and Scheme implementations, then the problems of overflow and internal fragmentation are avoided. At the same time, the overhead of each subroutine call is significantly increased. An intermediate option is to allocate the stack in large, fixed-size "chunks." At each call, the subroutine calling sequence checks to see whether there is sufficient space in the current chunk to hold the frame of the called routine.

If not, another chunk is allocated and the frame is put there instead. At each subroutine return, the epilogue code checks to see whether the current frame is the last one in its chunk. If so, the chunk is returned to a "free chunk" pool.

MODULE – 4

Functional and Logic Languages

- Functional and logic programming languages are also called ***declarative languages***; programs in these languages are said to describe (declaratively) *what* to do and not (operationally) *how* to do it.
- This is in contrast to imperative languages which are based on models of the underlying machine; programs written in imperative languages can be thus more directly compiled to efficient machine code, but reasoning and program transformations are comparatively difficult.
- Declarative programming languages have been developed since the 1970s, but their roots can be traced to the 1930s when mathematicians and logicians began to study the theory of computability.
- The ***functional programming*** community has focused on the concept of the mathematical function as a value-mapping entity; since such a function is typically defined by a set of equations, this yields a style of "programming with recursive equations".
- The task of the programmer is to construct a wanted result value from the given argument values by some basic constructs with a simple mathematical interpretation;
- Functional languages have also considerably contributed to the theory of type systems by concepts such as polymorphic functions (functions applicable to arguments of different types) and functors (parameterized program modules that take modules as arguments and return modules as results) which yielded the idea of generic programming.
- ***Logic programming*** is an outcome of research in automated theorem proving. In 1965, Robinson published the resolution method as an efficient decision procedure for logic formulas written in a subset of first-order predicate logic called **Horn clause** logic.
- While not every logic formula can be expressed in this language, it is sufficiently rich to serve as the basis of a rule-based programming style where the task of the programmer is to construct a relation between values: those given by the user are considered as input from which the system computes the other ones as output.
- In the early 1970s, Kowalski elaborated the theory of logic programming with Colmerauer producing the first implementation of the programming language Prolog

(Programming in Logic). The resolution mechanism was extended by methods for "constraint solving" which brought mathematics in closer contact to logic programming.

- In the 1990s, new developments have started to blur the distinction between functional programming and logic programming leading to ***functional-logic programming:*** here a logic formula also has a return value or, vice versa, a function call is also a goal which has to be satisfied by functional programming and "resolution" for logic programming and thus enhances the expressiveness of the declarative style of programming.

- **LISP** has traditionally been popular for the manipulation of symbolic data, in the field of Artificial Intelligence.

- Logic languages are widely used for formal specifications and theorem proving. **Prolog** is the most common logic language.

1. Functional Programming

- ***Functional programming*** defines the outputs of a program as a mathematical function of the inputs, with no notion of internal state, and thus no side effects.
- Miranda, Haskell, pH, Sisal, and Single Assignment C are purely functional.
- Functional languages provide a number of features that are often missing in imperative languages, including:
    - First-class function values and higher-order functions
    - Extensive polymorphism
    - List types and operators
    - Structured function returns
    - Constructors (aggregates) for structured objects
    - Garbage collection
- a first-class value as one that can be passed as a parameter, returned from a subroutine, or assigned into a variable.
- A *higher order function* takes a function as an argument, or returns a function as a result.
- Polymorphism is important in functional languages because it allows a function to be used on as general a class of arguments as possible.
- Lists are important in functional languages because they have a natural recursive definition.

Applications of Functional Languages

1. Knowledge representation
2. Machine learning
3. Natural language processing
4. Modeling of speech and vision
5. Scheme is used to teach introductory

2. Lambda calculus

- Mathematically, a function is a single-valued mapping: it associates every element in one set (the *domain*) with (at most) one element in another set (the *range*).
- In conventional notation, we indicate the domain and range of, say, the square root function by writing

$$\text{sqrt} : R \longrightarrow R$$

- We can also define functions using conventional set notation:

$$\text{sqrt} \equiv \{ (x, y) \in R \times R \mid y > 0 \wedge x = y^2 \}$$

- This notation is *non constructive*: it doesn't tell us how to *compute* square roots.
- Church designed the lambda calculus to address this limitation.
- So Lambda calculus is a framework developed by Alonzo Church in 1930s to study computations with functions.
- Lambda calculus is a *constructive* notation for function definitions.
- Any computable function can be written as a lambda expression.
- Computation amounts to macro substitution of arguments into the function definition, followed by reduction to simplest form via simple and mechanical rewrite rules.
- In Lambda calculus
    - **Function creation** − Church introduced the notation **λx.E** to denote a function in which 'x' is a formal argument and 'E' is the functional body. These functions can be of without names and single arguments.
    - **Function application** − Church used the notation $\mathbf{E_1.E_2}$ to denote the application of function $\mathbf{E_1}$ to actual argument $\mathbf{E_2}$. And all the functions are on single argument.

Syntax of Lambda Calculus

- Lamdba calculus includes three different types of expressions, i.e.,

E :: = x(variables)
| E_1 E_2(function application)
| λx.E(function creation)

Where **λx.E** is called Lambda abstraction and E is known as λ-expressions.

Evaluating Lambda Calculus

- Pure lambda calculus has no built-in functions. Let us evaluate the following expression –

```
(+ (* 5 6) (* 8 3))
```

- Here, we can't start with '+' because it only operates on numbers. There are two reducible expressions: (* 5 6) and (* 8 3).
- We can reduce either one first. For example −

```
(+ (* 5 6) (* 8 3))
(+ 30 (* 8 3))
(+ 30 24)
= 54
```

β-reduction Rule

- We need a reduction rule to handle λs

```
(λx . * 2 x) 4
(* 2 4)
= 8
```

This is called β-reduction.

3. Overview of Scheme

- Most Scheme implementations employ an interpreter that runs a "read-eval-print" loop.
- The interpreter repeatedly reads an expression from standard input, evaluates that expression, and prints the resulting value.
- If the user types
  (+ 3 4)
  the interpreter will print 7
- If the user types 7 the interpreter will also print 7
- most Scheme implementations provide a load function that reads (and evaluates) input from a file:
  (load "my_Scheme_program")
- Scheme (like Lisp) uses *Cambridge Polish* notation for expressions.
- Parentheses indicate a function application.
- The first expression inside the left parenthesis indicates the function; the remaining expressions are its arguments.
- Suppose the user types
  ((+ 3 4))
  When it sees the inner set of parentheses, the interpreter will call the function +, passing 3 and 4 as arguments.
- Because of the outer set of parentheses, it will then attempt to call 7 as a zero-argument function—a run-time error:
  eval: 7 is not a procedure
- Unlike all other programming languages, extra parentheses change the semantics of Lisp/Scheme programs.
  (+ 3 4) evaluates to 7

((+ 3 4)) evaluates to *error*

- One can prevent the Scheme interpreter from evaluating a parenthesized expression by ***quoting*** it:

  (quote (+ 3 4)) evaluates to (+ 3 4)

- Here the result is a three-element list. Quoting is specified with a special shorthand notation consisting of a leading single quote mark:

  '(+ 3 4) evaluates to (+ 3 4)

- User-defined functions can implement their own type checks using predefined *type predicate* functions:
  - (boolean? x) ; is x a Boolean?
  - (char? x) ; is x a character?
  - (string? x) ; is x a string?
  - (symbol? x) ; is x a symbol?
  - (number? x) ; is x a number?
  - (pair? x) ; is x a pair?
  - (list? x) ; is x a (proper) list?

- A *symbol* in Scheme is comparable to what other languages call an identifier.
- Identifiers are permitted to contain a wide variety of punctuation marks:

  (symbol? '**x$_%:&=*!**) evaluates to #t
- The symbol #t represents the Boolean value true. False is represented by #f. Note the use of quote ('); the symbol begins with x.
- To create a function in Scheme one evaluates a *lambda expression*:

  (lambda (x) (* x x)) is a *function*
  ((lambda (x) (* x x)) 3) =⇒ 9
- The first "argument" to lambda is a list of formal parameters for the function (in this case the single parameter x). The remaining "arguments" constitute the body of the function.
- A lambda expression does not give its function a name; this can be done using let or define.
- Function evaluation: When a function is called, the language implementation restores the referencing environment that was in effect when the lambda expression was evaluated.
- It then augments this environment with bindings for the formal parameters and evaluates the expressions of the function body in order.
- The value of the last such expression becomes the value returned by the function:

  ((lambda (x) (* x x)) 3) =⇒ 9
- Example

  (define min (lambda (a b) (if (< a b) a b)))
  The expression (min 123 456) will evaluate to 123;

- Simple conditional expressions can be written using if:
  (if (< 2 3) 4 5) =⇒ 4
- The implementation of if checks to see whether the first argument evaluates to #t.
- If so, it returns the value of the second argument , without evaluating the third argument.
- Otherwise it returns the value of the third argument, without evaluating the second.

Bindings

- Names can be bound to values by introducing a nested scope:

```
(let ((a 3)
      (b 4)
   (square (lambda (x) (* x x)))
   (plus +))
  (sqrt (plus (square a) (square b))))    =⇒ 5.0
```

- The special form **let** takes two or more arguments.
- The first of these is a list of pairs. In each pair, the first element is a name and the second is the value that the name is to represent within the remaining arguments to **let**.
- Remaining arguments are then evaluated in order; the value of the construct as a whole is the value of the final argument.

Lists and Numbers

- Like all Lisp dialects, Scheme provides functions to manipulate lists.
- The three most important are
  - **car**, which returns the head of a list,
    (car '(2 3 4)) =⇒ 2
  - **cdr**, which returns the rest of the list (everything after the head),
    (cdr '(2 3 4)) =⇒ (3 4)
  - **cons**, which joins a head to the rest of a list.
    (cons 2 '(3 4)) =⇒ (2 3 4)

Equality Testing and Searching

- Scheme provides several different equality-testing functions.
- For numerical comparisons, = performs type conversions where necessary (e.g., to compare an integer and a floating-point number).
- For general-purpose use,
  - **eqv?** performs a *shallow* comparison, while
  - **equal?** performs a *deep* (recursive) comparison.
- The **eq?** function also performs a shallow comparison, and may be cheaper than eqv? in certain circumstances
- To search for elements in lists, Scheme provides two sets of functions, each of which has variants corresponding to the three general-purpose equality predicates.
- The functions **memq**, **memv**, and **member** take an element and a list as argument, and return the longest suffix of the list beginning with the element:

(memq 'z '(x y z w)) =⇒ (z w)
(memv '(z) '(x y (z) w)) =⇒ #f ; (eq? '(z) '(z)) =⇒ #f
(member '(z) '(x y (z) w)) =⇒ ((z) w) ; (equal? '(z) '(z)) =⇒ #t

- The memq, memv, and member functions perform their comparisons using eq?, eqv?, and equal?, respectively.
- They return #f if the desired element is not found.

Control Flow and Assignment

- if. . . elsif. . . else is represented with **cond**

```
(cond
    ((< 3 2) 1)
    ((< 4 3) 2)
    (else 3))                    =⇒ 3
```

- The arguments to **cond** are pairs.
- The value of the overall expression is the value of the second element of the first pair in which the first element evaluates to #t.
- If none of the first elements evaluates to #t, then the overall value is #f.
- The symbol **else** is permitted only as the first element of the last pair of the construct.

Assignment

- Assignment employs the special form **set!** and the functions **set-car!** and **set-cdr!**:

```
(let ((x 2)                                  ; initialize x to 2
(l '(a b)))                                  ; initialize l to (a b)
(set! x 3)                                   ; assign x the value 3
(set-car! l '(c d))                          ; assign head of l the value (c d)
(set-cdr! l '(e))                            ; assign rest of l the value (e)
 ... x                                       =⇒ 3
... l                                        =⇒ ((c d) e)
```

- The return values of the various varieties of set! are implementation dependent.

Sequencing

- Sequencing uses the special form **begin**:

```
(begin
   (display "hello ")
 (display "world"))
```

Iteration

- Iteration uses the special form **do** and the function **for-each**:

```
(define iter-fib (lambda (n)
   (do ((i 0 (+ i 1))
```

```
      (a 0 b)
      (b 1 (+ a b)))
   ((= i n) b)
 (display b) ; body of loop
 (display " "))))) ; body of loop
```

- (do ((i 0 (+ i 1)) : The first argument to **do** is a list of triples, each of which specifies a new variable, an initial value for that variable, and an expression to be evaluated and placed in a fresh instance of the variable at the end of each iteration.
- (a 0 b) : The second argument to do is a pair that specifies the termination condition and the expression to be returned.

- The function **for-each** takes as argument a function and a sequence of lists.
- There must be as many lists as the function takes arguments, and the lists must all be of the same length.
- **For-each** calls its function argument repeatedly, passing successive sets of arguments from the lists.

```
(for-each (lambda (a b) (display (* a b)) (newline))
'(2 4 6)
'(3 5 7))
```

Programs as Lists

- A program in Scheme takes the form of a list. ie., Lisp and Scheme are *homoiconic*—self-representing.
- A parenthesized string of symbols (in which parentheses are balanced) is called an ***S-expression*** regardless of whether we think of it as a program or as a list.
- An unevaluated program *is* a list, and can be constructed, deconstructed, and manipulated with all the usual list functions.

Evaluating data as code:

- Scheme provides an eval function that can be used to evaluate a list that has been created as a data structure:

```
      (define compose
        (lambda (f g)
          (lambda (x) (f (g x)))))
   ((compose car cdr) '(1 2 3)) =⇒ 2
```

- Compose takes as arguments a pair of functions f and g.
- It returns as result a function that takes as parameter a value x, applies g to it, then applies f, and finally returns the result.

Eval *and* Apply

- The functions eval and apply can be defined as mutually recursive.

- When passed a number or a string, **eval** simply returns that number or string.
- When passed a symbol, it looks that symbol up in the specified environment and returns the value to which it is bound.
- When passed a list it checks to see whether the first element of the list is one of a small number of symbols that name so-called *primitive* special forms, built into the language implementation.
- When passed a function *f* and a list of arguments *l*, apply inspects the internal representation of *f* to see whether it is primitive.
- If so it invokes the built-in implementation.
- Otherwise it retrieves (from the representation of *f*) the referencing environment in which *f* 's lambda expression was originally evaluated. To this environment it adds the names of *f* 's parameters, with values taken from *l*.
- Call this resulting environment *e*.
- Next apply retrieves the list of expressions that make up the body of *f* .
- It passes these expressions, together with *e*, one at a time to eval.
- Finally, apply returns what the eval of the last expression in the body of *f* returned.
- One can choose to evaluate function arguments before passing them to a function, or to pass them unevaluated.
- The former option is called *applicative-order* evaluation; the latter is called *normal-order* evaluation.
- Scheme uses applicative order in most cases.

4. Strictness and Lazy Evaluation

- Evaluation order can have an effect not only on execution speed, but on program correctness as well.
- A function is said to be ***strict* if it is undefined** (fails to terminate, or encounters an error) when **any of its arguments is undefined**. Such a function can safely evaluate all its arguments, so **its result will not depend on evaluation order.**
- A *language* is said to be strict if it is defined in such a way that functions are always strict.
- If a language always evaluates expressions in applicative order, then every function is guaranteed to be strict, because whenever an argument is undefined, its evaluation will fail and so will the function to which it is being passed.
- ML and Scheme are strict.
- A function is said to be ***nonstrict*** if it does not impose this requirement-that is, **if it is sometimes defined even when one of its arguments is not.**
- A language is said to be nonstrict if it permits the definition of nonstrict functions.
- A nonstrict language cannot use applicative order; it must use normal order to avoid evaluating unneeded arguments.
- Miranda and Haskell are nonstrict.
- ***Lazy evaluation*** *g*ives us the advantage of normal-order evaluation (not evaluating un needed sub expressions)
- **Lazy evaluation** does not evaluate an expression until its value is actually needed.

- Every argument is tagged internally with a "memo" that indicates its value, if known. Any attempt to evaluate the argument sets the value in the memo as a side effect, or returns the value (without recalculating it) if it is already set.
- Lazy evaluation is particularly useful for "infinite" data structures.
- Lazy evaluation is used for all arguments in Miranda and Haskell.
- Normal-order evaluation can be thought of as function evaluation using call-by-name parameters, lazy evaluation is sometimes said to employ "call-by-need."
- In addition to Miranda and Haskell, call-by-need can be found in the R scripting language, widely used by statisticians.
- The principal problem with lazy evaluation is its behavior in the presence of side effects.
- If an argument contains a reference to a variable that may be modified by an assignment, then the value of the argument will depend on whether it is evaluated before or after the assignment.

5. I/O: Streams and Monads

Streams

- One way to avoid I/O side effects is to model input and output as ***streams***- unbounded-length lists whose elements are generated lazily.
- If we model input and output as streams, then a program takes the form
  (define output (my_prog input))
- When it needs an input value, function my_prog forces evaluation of the **car** of input, and passes the **cdr** on to the rest of the program.
- To drive execution, the language implementation repeatedly forces evaluation of the **car** of output, prints it, and repeats:

```
(define driver (lambda (s)
  (if (null? s) '()          ; nothing left
 (display (car s))
          (driver (cdr s)))))
(driver output)
```

- Streams formed the basis of the I/O system in early versions of Haskell.
- Streams don't work very well for graphics or random access to files.
- They also make it difficult to accommodate I/O of different kinds

Monads

- Recent versions of Haskell employ a general concept known as *monads*.
- Monads are drawn from a branch of mathematics known as *category theory*.
- In Haskell, monads are essentially a clever use of higher-order functions, coupled with a bit of syntactic sugar, that allow the programmer to chain together a sequence of *actions* (function calls) that have to happen in order.

- The power of the idea comes from the ability to carry a hidden, structured value of arbitrary complexity from one action to the next.
- Monads provide a more general solution to the problem of threading mutable state through a functional program
- The IO monad serves as the central repository for imperative language features-not only I/O and random numbers, but also mutable global variables and shared-memory synchronization.
- Additional monads (with accessible hidden state) support partial functions and various container classes (lists and sets).
- When coupled with lazy evaluation, monadic containers in turn provide a natural foundation for backtracking search, nondeterminism, and the functional equivalent of iterators.

6. Higher-Order Functions

- A function is said to be a *higher-order function* (also called a *functional form*) if it takes a function as an argument, or returns a function as a result.
- **Map function in Scheme:** The Scheme version of map takes as argument a function and a *sequence* of lists. There must be as many lists as the function takes arguments, and the lists must all be of the same length.
- Map calls its function argument on corresponding sets of elements from the lists:
  (map * '(2 4 6) '(3 5 7)) =⇒ (6 20 42)
- **Folding (reduction) in Scheme**: Programmers in Scheme (or in ML, Haskell, or other functional languages) can easily define other higher-order functions.
- If we want to be able to "fold" the elements of a list together, using an associative binary operator:

```
(define fold (lambda (f i l)
        (if (null? l) i      ;  i is commonly the identity element for f
                (f (car l) (fold f i (cdr l))))))
```

- Now (fold + 0 '(1 2 3 4 5)) gives us the sum of the first five natural numbers, and (fold * 1 '(1 2 3 4 5)) gives us their product.
- One of the most common uses of higher-order functions is to build new functions from existing ones

```
(define total (lambda (l) (fold + 0 l)))
        (total '(1 2 3 4 5))                    =⇒ 15

(define total-all (lambda (l)
        (map total l)))
(total-all '((1 2 3 4 5)
             (2 4 6 8 10)
             (3 6 9 12 15)))                    =⇒ (15 30 45)
```

Currying

- A common operation, named Curry, is to replace a multi argument function with a function that takes a single argument and returns a function that expects the remaining arguments:

```
(define curried-plus (lambda (a) (lambda (b) (+ a b))))
((curried-plus 3) 4) =⇒ 7
(define plus-3 (curried-plus 3))
(plus-3 4) =⇒ 7
```

- Among other things, currying gives us the ability to pass a "partially applied" function to a higher-order function.

7. Logic Programming

- Logic programming systems allow the programmer to state a collection of *axioms* from which theorems can be proven.
- The user of a logic program states a theorem, or *goal*, and the language implementation attempts to find a collection of axioms and inference steps (including choices of values for variables) that together imply the goal.
- Of the several existing logic languages, Prolog is by far the most widely used.
- In almost all logic languages, axioms are written in a standard form known as a ***Horn clause*.**
- **A Horn clause consists of a *head*, or *consequent* term *H*, and a *body* consisting of terms *Bi* :**

$$H \leftarrow B1, B2, \ldots, Bn$$

- The semantics of this statement are that when the *Bi* are all true, we can deduce that *H* is true as well.
- When reading aloud, we say "*H*, if *B*1, *B*2, . . . , and *Bn*."
- Horn clauses can be used to capture most, but not all, logical statements.
- In order to derive new statements, a logic programming system combines existing statements, canceling like terms, through a process known as ***resolution*.**
- If we know that *A* and *B* imply *C*, for example, and that *C* implies *D*, we can deduce that *A* and *B* imply *D*:

$$C \leftarrow A, B$$
$$D \leftarrow C$$
$$D \leftarrow A, B$$

- During resolution, free variables may acquire values through ***unification*** with expressions in matching terms

Advantages and Limitations of Logic programming

Advantages

1. It can be used to express knowledge in a way that does not depend on the implementation, making programs more flexible, compressed and understandable
2. It enables knowledge to be separated from use, ie., the machine architecture can be changed without changing programs or their underlying code
3. It can be altered and extended in natural ways to support special forms of knowledge, such as meta-level or higher order knowledge
4. It can be used in non-computational disciplines relying on reasoning and precise means of expression

Limitations

1. Due to insufficient investment in complementary technologies users were poorly served
2. In the beginning, poor facilities for supporting arithmetic, types, etc. had a discouraging effect on the programming community.
3. There is no adequate way of representing computational concepts found in built-in mechanisms of state variables
4. Some programmers always have, and always will prefer operational nature of machine operated programs

8. Prolog logic programming

- Scheme interpreter evaluates functions in the context of a referencing environment in which other functions and constants have been defined
- **Prolog interpreter** runs in the context of a ***database* of *clauses*** (Horn clauses) that are assumed to be true.
- Each clause is composed of *terms*, which may be constants, variables, or *structures*.
- A constant is either an atom or a number. A structure can be thought of as either a logical predicate or a data structure.
- Atoms in Prolog are similar to symbols in Lisp. Lexically, an atom looks like an identifier beginning with a lowercase letter, a sequence of "punctuation" characters, or a quoted character string:

  foo my_Const + 'Hi, Mom'
- Numbers resemble the integers and floating-point constants of other programming languages.
- A variable looks like an identifier beginning with an uppercase letter:

  Foo My_var X
- Variables can be ***instantiated*** to arbitrary values at run time as a result of unification.
- The scope of every variable is limited to the clause in which it appears.
- There are no declarations. As in Lisp, type checking occurs only when a program attempts to use a value in a particular way at run time.
- Structures consist of an atom called the ***functor*** and a list of arguments:

  rainy(rochester)

teaches(scott, cs254)
bin_tree(foo, bin_tree(bar, glarch))

- Prolog requires the opening parenthesis to come immediately after the functor, with no intervening space. Arguments can be arbitrary terms: constants, variables, or (nested) structures.
- Internally, a Prolog implementation can represent a structure using Lisp-like cons cells.
- The **clauses in a Prolog database can be classified as *facts* or *rules***, each of which ends with a period.
- A **fact** is a Horn clause without a right-hand side.

rainy(rochester).

- A **rule** has a right-hand side:

snowy(X) :- rainy(X), cold(X).

The token :- is the implication symbol; the comma indicates "and."Variables that appear in the head of a Horn clause are universally quantified: for all X, X is snowy if X is rainy and X is cold. _

- It is also possible to write a clause with an empty left-hand side. Such a clause is called **a *query*, or a *goal*.** Queries do not appear in Prolog programs.
- Rather, one builds a database of facts and rules and then initiates execution by giving the Prolog interpreter a query to be answered.
- In most implementations of Prolog, queries are entered with a special ?- implication symbol.
- If we were to type the following:
  rainy(seattle).
  rainy(rochester).
  ?- rainy(C).

  the Prolog interpreter would respond with
  C = seattle
- Of course, C = rochester would also be a valid answer, but Prolog will find seattle first, because it comes first in the database.
- If we want to find all possible solutions, we can ask the interpreter to continue by typing a semicolon:
  C = seattle ;
  C = rochester
- If we type another semicolon, the interpreter will indicate that no further solutions are possible:
  C = seattle ;
  C = rochester ;
  No

Resolution and Unification

- The ***resolution principle***, says that if *C*1 and *C*2 are Horn clauses and the head of *C*1 matches one of the terms in the body of *C*2, then we can replace the term in *C*2 with the body of *C*1. Consider the following example:

```
takes(jane_doe, his201).
takes(jane_doe, cs254).
takes(ajit_chandra, art302).
takes(ajit_chandra, cs254).
classmates(X, Y) :- takes(X, Z), takes(Y, Z).
```

- If we let X be jane_doe and Z be cs254, we can replace the first term on the right-hand side of the last clause with the (empty) body of the second clause, yielding the new rule

```
classmates(jane_doe, Y) :- takes(Y, cs254).
```

- In other words, Y is a classmate of jane_doe if Y takes cs254.
- Note that the last rule has a variable (Z) on the right-hand side that does not appear in the head. Such variables are existentially quantified: for all X and Y, X and Y are classmates if there exists a class Z that they both take.
- The pattern-matching process used to associate X with jane_doe and Z with cs254 is known as ***unification*.**
- Variables that are given values as a result of unification are said to be ***instantiated*.**
- The unification rules for Prolog state that
    - A constant unifies only with itself.
    - Two structures unify if and only if they have the same functor and the same arity, and the corresponding arguments unify recursively.
    - A variable unifies with anything. If the other thing has a value, then the variable is instantiated.
- If the other thing is an un instantiated variable, then the two variables are associated in such a way that if either is given a value later, that value will be shared by both.
- Unification of structures in Prolog is very much akin to ML's unification of the types of formal and actual parameters.
- Equality in Prolog is defined in terms of "unifiability."
- The goal =(A, B) succeeds if and only if A and B can be unified.

Lists

- Like equality checking, list manipulation is a sufficiently common operation in Prolog to warrant its own notation.
- The construct [a, b, c] is syntactic sugar for the structure .(a, .(b, .(c, []))), where [] is the empty list and . is a built-in cons-like predicate.
- Prolog adds an extra convenience, however: an optional vertical bar that delimits the "tail" of the list.
- Using this notation, [a, b, c] could be expressed as [a | [b, c]], [a, b | [c]], or [a, b, c | []].
- One of the interesting things about Prolog resolution is that it does not in general distinguish between "input" and "output" arguments

- Thus, given

```
append([], A, A).
append([H | T], A, [H | L]) :- append(T, A, L).
```

we can type

```
?- append([a, b, c], [d, e], L).
L = [a, b, c, d, e]
?- append(X, [d, e], [a, b, c, d, e]).
X = [a, b, c]
?- append([a, b, c], Y, [a, b, c, d, e]).
Y = [d, e]
```

- This example highlights the difference between functions and Prolog predicates.
- The former have a clear notion of inputs (arguments) and outputs (results); the latter do not.

Arithmetic

- The usual arithmetic operators are available in Prolog, but they play the role of **predicates**, not of functions.
- Thus +(2, 3), which may also be written 2 + 3, is a two-argument structure, not a function call. In particular, it will not unify with 5:

```
?- (2 + 3) = 5.
No
```

- To handle arithmetic, Prolog provides a built-in predicate, **is**, that unifies its first argument with the arithmetic value of its second argument:

```
?- is(X, 1+2).
X = 3
?- X is 1+2.
X = 3                    % infix is also ok
?- 1+2 is 4-1.
No                       % first argument (1+2) is already instantiated
?- X is Y.
ERROR                % second argument (Y) must already be instantiated
?- Y is 1+2, X is Y.
Y = 3
X = 3                     % Y is instantiated by the time it is needed
```

Search/Execution Order

- In the realm of formal logic, one can imagine two principal search strategies:
  - Start with existing clauses and work forward, attempting to derive the goal. This strategy is known as ***forward chaining***.

- Start with the goal and work backward, attempting to "unresolve" it into a set of preexisting clauses. This strategy is known as ***backward chaining***.

- If the number of existing rules is very large, but the number of facts is small, it is possible for forward chaining to discover a solution more quickly than backward chaining.
- In most circumstances, however, backward chaining turns out to be more efficient.
- Prolog is defined to use backward chaining. Because resolution is associative and commutative, a backward chaining theorem prover can limit its search to sequences of resolutions in which terms on the right-hand side of a clause are unified with the heads of other clauses one by one in some particular order (e.g., left to right).
- The resulting search can be described in terms of a tree of subgoals.
- The Prolog interpreter (or program) explores this tree depth first, from left to right.
- It starts at the beginning of the database, searching for a rule *R* whose head can be unified with the top-level goal.
- It then considers the terms in the body of *R* as subgoals, and attempts to satisfy them, recursively, left to right.
- If at any point a subgoal fails, the interpreter returns to the previous subgoal and attempts to satisfy it in a different way
- The process of returning to previous goals is known as ***backtracking*.**
- It strongly resembles the control flow of generators in Icon
- Whenever a unification operation is "undone" in order to pursue a different path through the search tree, variables that were given values or associated with one another as

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X).
```

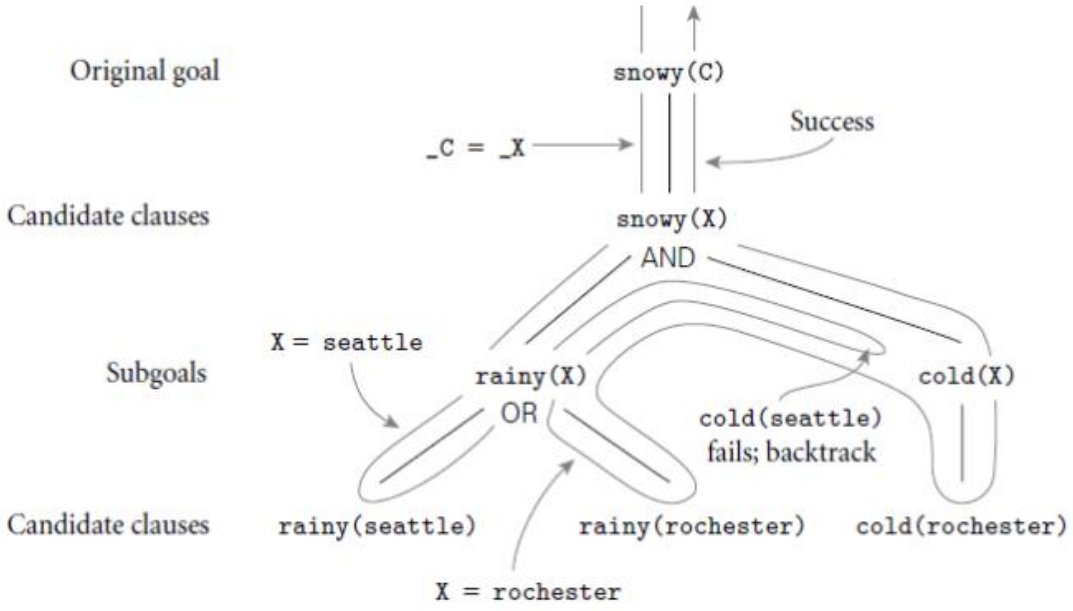


Figure: Backtracking search in Prolog

- a result of that unification are returned to their uninstantiated or unassociated state
- Space management for backtracking search in Prolog usually follows the single stack implementation of iterators.
- The interpreter pushes a frame onto its stack every time it begins to pursue a new subgoal *G*.
- If *G* fails, the frame is popped from the stack and the interpreter begins to backtrack.
- If *G* succeeds, control returns to the "caller", but *G*'s frame remains on the stack.
- At the top level of the interpreter, a semicolon typed by the user is treated the same as failure of the most recently satisfied subgoal.
- The results of a Prolog program are deterministic and predictable.
- Suppose for example that we have a database describing a directed acyclic graph:

```
edge(a, b). edge(b, c). edge(c, d).
edge(d, e). edge(b, e). edge(d, f).
path(X, X).
path(X, Y) :- edge(Z, Y), path(X, Z).
```

- The last two clauses tell us how to determine whether there is a path from node X to node Y.
- If we were to reverse the order of the terms on the right-hand side of the final clause, then the Prolog interpreter would search for a node Z that is reachable from X before checking to see whether there is an edge from Z to Y.

MODULE – 5

Object-Oriented Programming

With the development of complicated computer applications, data abstraction has become essential to software engineering. The abstraction provided by modules and module types has at least **three important benefits:**

1. It reduces *conceptual load* by minimizing the amount of detail that the programmer must think about at one time.

2. It provides *fault containment* by preventing the programmer from using a program component in inappropriate ways, and by limiting the portion of a program's text in which a given component can be used, thereby limiting the portion that must be considered when searching for the cause of a bug.

3. It provides a significant degree of *independence* among program components, making it easier to assign their construction to separate individuals, to modify their internal implementations without changing external code that uses them, or to install them in a library where they can be used by other programs.

ENCAPSULATION AND INHERITANCE

Encapsulation mechanisms enable the programmer to group data and the subroutines that operate on them together in one place, and to hide irrelevant details
from the users of an abstraction.

i) Modules

Scope rules for data hiding were one of the principal innovations of Clu, Modula, Euclid, and other module-based languages of the 1970s. In Clu and Euclid, the declaration and definition (header and body) of a module always appear together.
The header clearly states which of the module's names are to be exported.
If a Euclid module *M* exports a type *T*, by default the remainder of the program can
do nothing with objects of type *T* other than pass them to subroutines exported from *M*. *T* is said to be an *opaque* type.

In Modula-2, programmers have the option of separating the header and body of a module.

In"internal" modules, the two parts appear together. In an "external" module
(meant for separate compilation), the header appears in one source file and the
body in another. Unfortunately, there is no way to divide the header into public and private parts; everything in it is public (i.e., exported). The only concession to data hiding is that a type may be made opaque by listing only its name in the header:

```
TYPE T;
```

In this case variables of type T can only be assigned, compared for equality, and passed to the module's subroutines.
Ada, which also allows the headers and bodies of modules (called packages) to be separated, eliminates the problems of Modula-2 by allowing the header of a package to be divided into public and private parts. A type can be exported opaquely by putting its definition in the private part of the header and simply naming it in the public part:

```
package foo is -- header
...
type T is private;
...
private -- definitions below here are inaccessible to users
...
type T is ... -- full definition
...
end foo;
```

When the header and body of a module appear in separate files, a change to a module body never requires us to recompile any of the module's users. A change to the private part of a module header may require us to recompile the module's users, but never requires us to change their

code. A change to the public part of a header is a change to the module's interface: it will often require us to change the code of users.

Because they affect only the visibility of names, static, manager-style modules introduce no special code generation issues. Storage for variables and other data
inside a module is managed in precisely the same way as storage for data immediately
outside the module. If the module appears in a global scope, then its data can be allocated statically. If the module appears within a subroutine, then its data can be allocated on the stack, at known offsets, when the subroutine is called, and reclaimed when it returns.

Module types, as in Euclid, are somewhat more complicated: they allow a module to have an arbitrary number of *instances*. The obvious implementation then resembles that of a record. If all of the data in the module have a statically known size, then each individual datum can be assigned a static offset within the module's storage. If the size of some of the data is not known until run time, then the module's storage can be divided into fixed-size and variable-size portions, with a dope vector (descriptor) at the beginning of the fixed-size portion. Instances of the module can be allocated statically, on the stack, or in the heap, as appropriate.

ii) The "this***" Parameter***

One additional complication arises for subroutines inside a module.
We could, of course, replicate the code for each subroutine
in each instance of the module, just as we replicate the data. This replication
would be highly wasteful, however, as the copies would vary only in the details
of address computations. A better technique is to create a single instance of each
module subroutine, and to pass that instance, at run time, the address of the storage
of the appropriate module instance. This address takes the form of an extra, hidden first parameter for every module subroutine. A Euclid call of the form

```
my_stack.push(x)
```

is translated as if it were really

```
push(my_stack, x)
```

where my_stack is passed by reference. The same translation occurs in object oriented languages.

iii) Classes

The basic philosophy behind the visibility rules of C++ can be summarized as
follows:

- Any class can limit the visibility of its members. Public members are visible anywhere the class declaration is in scope. Private members are visible only inside the class's methods. Protected members are visible inside methods of the class or its descendants.
- A derived class can restrict the visibility of members of a base class, but can never increase it. Private members of a base class are never visible in a derived class. Protected and public members of a public base class are protected or public, respectively, in a

derived class. Protected and public members of a protected base class are protected members of a derived class. Protected and public members of a private base class are private members of a derived class.

- A derived class that limits the visibility of members of a base class by declaring that base class protected or private can restore the visibility of individual members of the base class by inserting a using declaration in the protected or public portion of the derived class declaration.

Other object-oriented languages take different approaches to visibility. Eiffel is more flexible than C++ in the patterns of visibility it can support, but it does not adhere to the first of the C++ principles above. Derived classes in Eiffel can both restrict *and increase* the visibility of members of base classes.

Every method (called a feature in Eiffel) can specify its own *export status*. If the status is {NONE} then the member is effectively private (called *secret* in Eiffel). If the status is {ANY} then the member is effectively public (called *generally available* in Eiffel).

In the general case the status can be an arbitrary list of class names, in which case the feature is said to be *selectively available* to those classes and their descendants only. Any feature inherited from a base class can be given a new status in a derived class. Java and C# follow C++in the declaration of public, protected, and private members, but do not provide the protected and private designations for base classes; a derived class can neither increase *nor* restrict the visibility of members of a base class. The protected keyword has a slightly different meaning in Java than it does in C++: a protected member of a Java class is visible not only within derived classes, but also within the entire package (namespace) in which the class is declared.

iv) Extending without Inheritance

The desire to extend the functionality to an existing abstraction is one of the principal motivations for object-oriented programming. Inheritance is the standard mechanism that makes such extension possible. There are times, however, when inheritance is not an option, particularly when dealing with pre-existing code. The class one wants to extend may not permit inheritance, for instance: in Java, it may be labeled final; in C#, it may be sealed. Even if inheritance is possible in principle, there may be a large body of existing code that uses the original class name, and it may not be feasible to go back and change all the variable and parameter declarations to use a new derived type.

For situations like these, C# 3.0 provides *extension methods*, which give the appearance of extending an existing class:

static class AddToString {

public static int toInt(this string s) {

return int.Parse(s);

}

}

An extension method must be static, and must be declared in a static class.

Its first parameter must be prefixed with the keyword this. The method can then be invoked as if it were a member of the class of which this is an instance:

int n = myString.toInt();

INITIALIZATION AND FINALIZATION

Most object-oriented languages provide some sort of special mechanism to *initialize* an object automatically at the beginning of its lifetime. When written in the form of a subroutine, this mechanism is known as a *constructor*. Though the name might be thought to imply otherwise, a a constructor does not allocate space; it initializes space that has

already been allocated. A few languages provide a similar *destructor* mechanism to *finalize* an object automatically at the end of its lifetime.

Several important issues arise:

i) Choosing a constructor: An object-oriented language may permit a class to have zero, one, or many distinct constructors. In the latter case, different constructors may have different names, or it may be necessary to distinguish among them by number and types of arguments.

ii) References and values*:* If variables are references, then every object must be created explicitly, and it is easy to ensure that an appropriate constructor is called. If variables are values, then object creation can happen implicitly as a result of elaboration. In this latter case, the language must either permit objects to begin their lifetime uninitialized, or it must provide a way to choose an appropriate constructor for every elaborated object.

iii) Execution order: When an object of a derived class is created in C++, the compiler guarantees that the constructors for any base classes will be executed, outermost first, before the constructor for the derived class. Moreover, if a class has members that are themselves objects of some class, then the constructors for the members will be called before the constructor for the object in which they are contained. These rules are a source of considerable syntactic and semantic complexity: when combined with multiple constructors, elaborated objects, and multiple inheritance, they can sometimes induce a complicated sequence of nested constructor invocations, with overload resolution, before control even enters a given scope. Other languages have simpler rules.

iv) Garbage collection: Most object-oriented languages provide some sort of constructor mechanism. Destructors are comparatively rare. Their principal purpose is to facilitate manual storage reclamation in languages like C++. If the language implementation collects garbage automatically, then the need for destructors is greatly reduced.

4 ISSUES IN DETAIL

i) Choosing a Constructor

Smalltalk, Eiffel, C++, Java, and C# all allow the programmer to specify more than one constructor for a given class. In C++, Java, and C#, the constructors behave like overloaded subroutines: they must be distinguished by their numbers and types of arguments. In Smalltalk and Eiffel, different constructors can have different names; code that creates an objectmust name a constructor explicitly. In Eiffel one might say

```
class COMPLEX
creation
new_cartesian, new_polar
feature {ANY}
x, y : REAL
new_cartesian(x_val, y_val : REAL) is
do
x := x_val; y := y_val
end
new_polar(rho, theta : REAL) is
do
x := rho * cos(theta)
y := rho * sin(theta)
end
-- other public methods
feature {NONE}
-- private methods
end -- class COMPLEX
...
a, b : COMPLEX
...
!!b.new_cartesian(0, 1)
!!a.new_polar(pi/2, 1)
```

The !! operator is Eiffel's equivalent of new. Because class COMPLEX specified constructor ("creator") methods, the compiler will insist that every use of !! specify a constructor name and arguments.

There is no straightforward analog of this code in C++; the fact that both constructors take two real arguments means that they could not be distinguished by overloading.

Smalltalk resembles Eiffel in the use of multiple named constructors, but it distinguishes more sharply between operations that pertain to an individual object and operations that pertain to a class of objects.

Smalltalk also adopts an anthropomorphic programming model in which every operation is seen as being executed by some specific object in response to a request (a "message") from some other object.

Consider, for example, the standard class named Date. Corresponding to Date is a single object (call it *D*) that performs operations on behalf of the class. In particular, it is *D* that creates new objects of class Date. Because only objects execute operations (classes don't), we don't really need a name for *D*; we can simply use the name of the class it represents:

todaysDate <- Date today

This code causes *D* to execute the today constructor of class Date, and assigns a reference to the newly created object into a variable named todays Date. So what is the class of *D* ? It clearly isn't Date, because *D represents* class Date.

Smalltalk says that *D* is an object (in fact the only object) of the *metaclass* Date class.

For technical reasons, it is also necessary for Date class to be represented by an object. To avoid an infinite regression, all objects that represent metaclasses are instances of a single class named Metaclass.

ii) References and Values

Several object-oriented languages, including Simula, Smalltalk, Python, Ruby, and Java, use a programming model in which variables refer to objects.

Other languages, including C++, Modula-3, Ada 95, and Oberon, allow a variable to have a value that *is* an object.

Eiffel uses a reference model by default, but allows the programmer to specify that certain classes should be expanded, in which case variables of those classes will use a value model.

In a similar vein, C# uses struct to define types whose variables are values, and class to define types whose variables are references.

With a reference model for variables every object is created explicitly, and it is easy to

ensure that an appropriate constructor is called.
With a value model for variables object creation can happen implicitly as a result of elaboration.

If a C++ variable of class type foo is declared with no initial value, then the compiler will call foo's zero-argument constructor (if no such constructor exists, but other constructors do, then the declaration is a static semantic error—a call to a nonexistent subroutine):

```
foo b; // calls foo::foo()
```

If the programmer wants to call a different constructor, the declaration must specify constructor arguments to drive overload resolution:

```
foo b(10, 'x'); // calls foo::foo(int, char)
```

The most common argument list consists of a single object, of the same or different class:

```
foo a;
bar b;
...
foo c(a); // calls foo::foo(foo&)
foo d(b); // calls foo::foo(bar&)
```

Usually the programmer's intent is to declare a new object whose initial value is"the same" as that of the existing object. In this case it is more natural to write

```
foo a; // calls foo::foo()
bar b; // calls bar::bar()
...
foo c = a; // calls foo::foo(foo&)
foo d = b; // calls foo::foo(bar&)
```

In recognition of this intent, a single-argument constructor in C++ is called a *copy constructor*. It is important to realize here that the equals sign (=) in these declarations indicates initialization, not assignment.
The effect is *not* the same as that of the similar code fragment

```
foo a, c, d; // calls foo::foo() three times
bar b; // calls bar::bar()
...
c = a; // calls foo::operator=(foo&)
d = b; // calls foo::operator=(bar&)
```

Here c and d are initialized with the zero-argument constructor, and the later use of the equals sign indicates *assignment*, not initialization. The distinction is a common source of confusion in C++ programs. It arises from the combination of a value model of variables and an insistence that every elaborated object be initialized by a constructor.

iii) Execution Order

As we have seen, C++ insists that every object be initialized before it can be used.

Moreover, if the object's class (call it *B*) is derived from some other class (call it *A*), C++ insists on calling an *A* constructor before calling a *B* constructor, so that the derived class is guaranteed never to see its inherited fields in an inconsistent state. When the programmer creates an object of class *B* (either via declaration or with a call to new), the creation operation specifies arguments for a *B* constructor.

These arguments allow the C++ compiler to resolve overloading when multiple constructors exist.

But where does the compiler obtain arguments for the A constructor?

Adding them to the creation syntax (as Simula does) would be a clear violation of abstraction.

The answer adopted in C++ is to allow the header of the constructor of a derived class to specify base class constructor arguments:

foo::foo(*foo params*) : bar(*bar args*)
{
...

Here foo is derived from bar. The list *foo params* consists of formal parameters for this particular foo constructor.

Between the parameter list and the opening brace of the subroutine definition is a "call" to a constructor for the base class bar.

The arguments to the bar constructor can be arbitrarily complicated expressions involving the foo parameters. The compiler will arrange to execute the bar constructor before beginning execution of the foo constructor.

iv) Garbage Collection

When a C++ object is destroyed, the destructor for the derived class is called first, followed by those of the base class(es), in reverse order of derivation. By far the most common use of destructors in C++ is manual storage reclamation.

Suppose, for example, that we were to create a list or queue of character-string names:

```
class name_list_node : public gp_list_node {
char *name; // pointer to the data in a node
public:
name_list_node() {
name = 0; // empty string
```

```
}
name_list_node(char *n) {
name = new char[strlen(n)+1];
strcpy(name, n); // copy argument into member
}
~name_list_node() {
if (name != 0) {
delete[] name; // reclaim space
}
}
};
```

The destructor in this class serves to reclaim space that was allocated in the heap by the constructor.

In languages with automatic garbage collection, there is much less need for destructors. In fact, the entire idea of destruction is suspect in a garbage-collected language, because the programmer has little or no control over when an object is going to be destroyed. Java and C# allow the programmer to declare a finalize method that will be called immediately before the garbage collector reclaims the space for an object, but the feature is not widely used.

DYNAMIC METHOD BINDING

One of the principal consequences of inheritance/type extension is that a derived class *D* has all the members—data and subroutines—of its base class *C*. As long as *D* does not hide any of the publicly visible members of *C* it makes sense to allow an object of class *D* to be used in any context that expects an object of class *C*: anything we might want to do to an object of class *C* we can also do to an object of class *D*. In Ada terminology, a derived class that does not hide any publicly visible members of its base class is a *subtype* of that base class.

The ability to use a derived class in a context that expects its base class is called SUBTYPE POLYMORPHISM.

If we imagine an administrative computing system for a university, we might derive classes student and

professor from class person:

```
class person { ...
class student : public person { ...
class professor : public person { ...
```

Because both student and professor objects have all the properties of a person object, we should be able to use them in a person context:

student s;
professor p;
...
person *x = &s;
person *y = &p;
Moreover a subroutine like

void person::print_mailing_label() { ...

would be polymorphic—capable of accepting arguments of multiple types:

```
s.print_mailing_label();    // i.e., print_mailing_label(s)
p.print_mailing_label();    // i.e., print_mailing_label(p)
```

As with other forms of polymorphism, we depend on the fact that print_mailing_ label uses only those features of its formal parameter that all actual parameters will have in common.

But now suppose that we have redefined print_mailing_label in each of the two derived classes. We might, for example, want to encode certain information (student's year in school, professor's home department) in the corner of the label.
Now we have multiple versions of our subroutine—student:: print_mailing_label and professor::print_mailing_label, rather than the single, polymorphic person::print_mailing_label.
Which version we will get depends on the object:

```
s.print_mailing_label();    // student::print_mailing_label(s)
p.print_mailing_label();    // professor::print_mailing_label(p)
```

But what about

```
x->print_mailing_label();    // ??
y->print_mailing_label();    // ??
```

Does the choice of the method to be called depend on the types of the *variables* x and y, or on the classes of the *objects* s and p to which those variables refer?

The first option (use the type of the reference) is known as *static method binding*.

The second option (use the class of the object) is known as *dynamic method binding*

Dynamic method binding is central to object-oriented programming.
Imagine, for example, that our administrative computing program has created a list of persons who have overdue library books. The list may contain both students and professors. If we traverse the list and print a mailing label for each person, dynamic method binding will ensure that the correct printing routine is called for each individual. In this situation the definitions in the derived classes are said to *override* the definition in the base class.

Virtual and Nonvirtual Methods
In Simula, C++, and C#, which use static method binding by default, the programmer can specify that particular methods should use dynamic binding by labelling them as virtual.
Calls to virtual methods are *dispatched* to the appropriate implementation at run time, based on the class of the object, rather than the type of the reference.

In C++ and C#, the keyword virtual prefixes the subroutine declaration:

```
class person {
public:
virtual void print_mailing_label();
...
```

In Simula, virtual methods are listed at the beginning of the class declaration:

```
CLASS Person;
VIRTUAL: PROCEDURE PrintMailingLabel;
BEGIN
...
PROCEDURE PrintMailingLabel...
COMMENT body of subroutine
...
END Person;
```

Ada 95 adopts a different approach.
Rather than associate dynamic dispatch
with particular methods, the Ada 95 programmer associates it with certain *references*.

Polymorphism
We have already noted that dynamic method binding introduces polymorphism

(specifically, *subtype* polymorphism) into any code that expects a reference to an object of some base class foo. So long as objects of the derived class support the operations of the base class, the code will work equally well with references to objects of any class derived from foo.
By declaring a reference parameter to be of class foo, for example, the programmer asserts that the subroutine uses only the
"foo features" of the parameter, and will work on any object that provides those
features.

MULTIPLE INHERITANCE

At times it can be useful for a derived class to inherit features from more than
one base class. Suppose, for example, that we want our administrative computing system to keep all students of the same year (*freshmen, sophomores, juniors, seniors, nonmatriculated*) on some list. It may then be desirable to derive class student from both person and gp_list_node.

In C++ we can say
class student : public person, public gp_list_node { ...

Now an object of class student will have all the fields and methods of both a person and a gp_list_node.

The declaration in Eiffel is analogous:

```
class student
inherit
person
gp_list_node
feature
...
```

Multiple inheritance also appears in CLOS and Python. Simula, Smalltalk, Objective-C, Modula-3, Ada 95, and Oberon have only single inheritance. Java, C#, and Ruby provide a limited, "mix-in" form of multiple inheritance, in which only one parent class is permitted to have fields.
Multiple inheritance introduces a wealth of semantic and pragmatic issues, which we consider on the PLP CD.
1) Suppose two parent classes provide a method with the same name. Which one
do we use in the child? Can we access both?
2) Suppose two parent classes are both derived from some common"grandparent"
class. Does the "grandchild" have one copy or two of the grandparent's fields?
3) Our implementation of single inheritance relies on the fact that the representation

of an object of the parent class is a prefix of the representation of an object of a derived class.With multiple inheritance, how can *each* parent be a prefix of the child?

Multiple inheritance with a common grandparent" is known as ***REPEATED*** **INHERITANCE.**
Repeated inheritance with separate copies of the grandparent is known as **REPLICATED INHERITANCE;** repeated inheritance with a single copy of the grandparent is known as ***SHARED*** **INHERITANCE**.
Shared inheritance is the default in Eiffel. Replicated inheritance is the default in C++.

Both languages allow the programmer to
obtain the other option when desired.
Much of the complexity disappears if we insist, as Java, C#, or Ada 2005, that
all but one of the parent classes consist of methods only. All three languages call
such a class **AN INTERFACE.**

SCRIPTING LANGUAGES

All scripting languages are programming languages.
The scripting language is basically a language where instructions are written for a run time environment.
They do not require the compilation step and are rather interpreted.
It brings new functions to applications and glue complex system together.
A scripting language is a programming language designed for integrating and communicating with other programming languages.
There are many scripting languages some of them are discussed below:

- **bash:** It is a scripting language to work in the Linux interface. It is a lot easier to use bash to create scripts than other programming languages. It describes the tools to use and code in the command line and create useful reusable scripts and conserve documentation for other people to work with.
- **Node js:** It is a framework to write network applications using **JavaScript** Corporate users of Node.js include IBM, LinkedIn, Microsoft, Netflix, PayPal, Yahoo for real-time web applications.
- **Ruby:** There are a lot of reasons to learn Ruby programming language. Ruby's flexibility has allowed developers to create innovative software. It is a scripting language which is great for web development.
- **Python:** It is easy, free and open source. It supports procedure-oriented programming and object-oriented programming. Python is an interpreted language with dynamic semantics and huge lines of code are scripted and is currently the most hyped language among developers.

- **Perl:** A scripting language with innovative features to make it different and popular. Found on all windows and Linux servers. It helps in text manipulation tasks. High traffic websites that use Perl extensively include priceline.com, IMDB.

Advantages of scripting languages:

- **Easy learning:** The user can learn to code in scripting languages quickly, not much knowledge of web technology is required.
- **Fast editing:** It is highly efficient with the limited number of data structures and variables to use.
- **Interactivity:** It helps in adding visualization interfaces and combinations in web pages. Modern web pages demand the use of scripting languages. To create enhanced web pages, fascinated visual description which includes background and foreground colors and so on.
- **Functionality:** There are different libraries which are part of different scripting languages. They help in creating new applications in web browsers and are different from normal programming languages.

Application of Scripting Languages:

Scripting languages are used in many areas:

- Scripting languages are used in web applications. It is used in server side as well as client side. Server side scripting languages are: JavaScript, PHP, Perl etc. and client side scripting languages are: JavaScript, AJAX, jQuery etc.
- Scripting languages are used in system administration. For example: Shell, Perl, Python scripts etc.
- It is used in Games application and Multimedia.
- It is used to create plugins and extensions for existing applications.

MODULE – 6

CONCURRENCY

Program is said to be *concurrent* if it may have more than one active execution context—more than one "thread of control."

Concurrency has at least three important motivations:

1. *To capture the logical structure of a problem.* Many programs, particularly servers and graphical applications, must keep track of more than one largely independent "task" at the same time. Often the simplest and most logical way to structure such a program is to represent each task with a separate thread of control.

2. *To exploit extra processors, for speed.* Long a staple of high-end servers and super computers ,multiple processors have recently become ubiquitous in desktop and laptop machines. To use them effectively, programs must generally be written (or rewritten) with concurrency in mind.

3. *To cope with separate physical devices*. Applications that run across the Internet or a more local group of machines are inherently concurrent. Likewise, many embedded applications—the control systems of a modern automobile, for example—often have separate processors for each of several devices.

* In general, we use the word *concurrent* to characterize any system in which two or more tasks may be underway at the same time.

* Under this definition, coroutines are not concurrent, because at any given time, all but one of them is stopped at a well-known place.

* A concurrent system is *parallel* if more than one task can be physically *active* at once; this requires more than one processor.

* A parallel system is *distributed* if its processors are associated with people or devices that are physically separated from one another in the real world.

Levels of Parallelism

Parallelism arises at every level of a modern computer system. It is comparatively easy to exploit at the level of circuits and gates, where signals can propagate down thousands of connections at once. As we move up first to processors and then to the many layers of software that run on top of them, the *granularity* of parallelism—the size and complexity of tasks—increases at every level, and it becomes increasingly difficult to figure out what work should be done by each task and how tasks should coordinate.

At the next higher level of granularity, so-called *vector parallelism* is available in programs that perform operations repeatedly on every element of a very large data set.

Unfortunately, vector parallelism arises in only certain kinds of programs.

General-purpose computing has moved to *multicore* processors, which require coarser-grain *thread-level* parallelism.

Levels of Abstraction

With the spread of thread-level parallelism, different kinds of programmers will need to understand concurrency at different levels of detail, and use it in different ways.

The simplest, most abstract case will arise when using "black box" parallel libraries.

At a slightly less abstract level, a programmer may know that certain tasks are mutually independent. Such tasks can safely execute in parallel

If our tasks are *not* independent, it may still be possible to run them in parallel if we explicitly *synchronize* their interactions. Synchronization serves to eliminate ***races*** between threads by controlling the ways in which their actions can interleave in time.

A race condition occurs whenever two or more threads are "racing" toward points in the code at which they touch some common object, and the behavior of the system depends on which thread gets there first.

The most common purpose of synchronization is to make some sequence of instructions, known as a *critical section*, appear to be *atomic*—to happen "all at once" from the point of view of every other thread.

The most common way to make the sequence atomic is with a *mutual exclusion lock*, which we *acquire* before the first instruction of the sequence and *release* after the last.

Example : A simple race condition

```
int zero_count;

public static int foo(int n) {

int rtn = n - 1;

if (rtn == 0) zero_count++;

return rtn;

}
```

Consider now what may happen when two or more instances of this code run concurrently.

Thread 1	Thread 2
. . .	. . .
r1 := zero_count	r1 := zero_count
r1 := r1 + 1	r1 := r1 + 1
zero_count := r1	zero_count := r1
. . .	. . .

If the instructions interleave roughly as shown, both threads may load the same value of zero count, both may increment it by 1, and both may store the (only 1 greater) value back into zero count. The result may be 1 less than what we expect.

In general, a *race condition* occurs whenever two or more threads are "racing" toward points in the code at which they touch some common object, and the behavior of the system depends on which thread gets there first.

The Case for Multithreaded Programs

The use of many threads ensures that comparatively fast operations (e.g., display of text) do not wait for slow operations (e.g., display of large images).Whenever one thread *blocks* (waits for a message or I/O), the implementation automatically switches to a different thread. In a *preemptive* thread package, the implementation switches among threads at other times as well (i.e., it performs a *context switch*), to prevent any one thread from hogging the CPU. Any reader who remembers the early, more sequential browsers will appreciate the difference that multithreading makes in perceived performance and responsiveness.

Multiprocessor Architecture

Single-site (nondistributed) parallel computers can be grouped into two broad categories: * * 1)those in which processors share access to common memory, and

2)those in which they must communicate with messages.

Shared-memory machines are typically referred to as *multiprocessors*, though occasionally one hears that term applied to message-based machines as well. A multiprocessor typically occupies a single chassis, in which the processors share not only memory, but also disks, power supplies, and a single copy of the operating system in which they must communicate with messages.

From the point of view of language or library implementation, the principal distinction between shared-memory and message-passing hardware is that messages typically require the active participation of processors at both ends of the connection: one to send, the other to receive. On a shared-memory machine, a processor can read and write remote memory without the assistance of a remote processor. In most cases remote reads and writes use the same interface (i.e., load and store instructions) as local reads and writes.

Memory Coherence

On a message-passing machine, each processor caches its own memory independently.

On a shared-memory machine, however, caches introduce a serious problem: unless we do something special, a processor that has cached a particular memory location will not see changes that are made to that location by other processors This problem—how to keep cached copies of a memory location consistent with one another—is known as the *coherence* problem. On bus-based symmetric machines, the problem is relatively easy to solve: the broadcast nature of the communication medium allows cache controllers to eavesdrop (*snoop*) on the memory traffic of other processors. When a processor needs to write a cache line, it requests an *exclusive* copy, and

waits for other processors to *invalidate* their copies. On a bus the waiting is trivial, and the natural ordering of messages determines who wins in the event of near-simultaneous requests. Processors that try to access a line in the wake of invalidation must go back to memory (or to another processor's cache) to obtain an up-to-date copy.

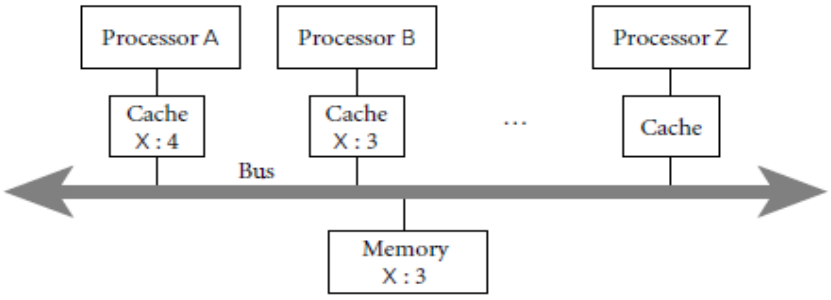


Figure 12.3 The cache coherence problem for shared-memory multiprocessors. Here processors A and B have both read variable X from memory. As a side effect, a copy of X has been created in the cache of each processor. If A now changes X to 4 and B reads X again, how do we ensure that the result is a 4 and not the still-cached 3? Similarly, if Z reads X into its cache, how do we ensure that it obtains the 4 from A's cache instead of the stale 3 from memory?

Supercomputers

Because of the complexity of cache coherence, it is difficult to build large shared memory machines. Today's fastest machines are constructed from special high-density, low-power multicore chips. From a programming language perspective, the special challenge of supercomputing is to accommodate nonuniform access times and (in most cases) the lack of hardware support for shared memory across the full machine. Today's supercomputers are programmed mostly with message-passing libraries (MPI in particular) and with languages and libraries in which there is a clear syntactic distinction between local and remote memory access.

CONCURRENT PROGRAMMING FUNDAMENTALS

Within a concurrent program, we will use the term *thread* to refer to the active entity that the programmer thinks of as running concurrently with other threads. In most systems, the threads of a given programare implemented on top of one or more *processes* provided by the operating system. OS designers often distinguish between a *heavyweight* process, which has its own address space, and a collection of *lightweight* processes, which may share an address space

We will sometimes use the word *task* to refer to a well-defined unit of work that must be performed by some thread. In one common programming idiom, a collection of threads shares a common "bag of tasks"—a list of work to be done. Each thread repeatedly removes a task from the bag, performs it, and goes back for another. Sometimes the work of a task entails adding new tasks to the bag.

1) Communication and Synchronization

In any concurrent programming model, two of the most crucial issues to be addressed are *communication* and *synchronization*.

- ➔ **Communication** refers to any mechanism that allows one thread to obtain information produced by another. Communication mechanisms for imperative programs are generally based on either *shared memory* or *message passing*. In a shared-memory programming model, some or all of a program's variables are accessible to multiple threads. For a pair of threads to communicate, one of them writes a value to a variable and the other simply reads it. In a message-passing programming model, threads have no common state. For a pair of threads to communicate, one of them must perform an explicit send operation to transmit data to another.
- ➔ **Synchronization** refers to any mechanism that allows the programmer to control the relative order in which operations occur in different threads. Synchronization is generally implicit in message-passing models: a message must be sent before it can be received. If a thread attempts to receive a message that has not yet been sent, it will wait for the sender to catch up. Synchronization is generally not implicit in shared-memory models: unless we do something special, a "receiving" thread could read the "old" value of a variable, before it has been written by the"sender."

In both shared-memory and message-based programs, synchronization can be implemented either by ***spinning*** (also called *busy-waiting*) or by ***blocking.***

In busy-wait synchronization (SPINNING**), a thread runs a loop in which it keeps reevaluating some condition until that condition becomes true.*

In **BLOCKING synchronization (also called scheduler-based synchronization), the waiting thread voluntarily relinquishes its processor to some other thread. Before doing so, it leaves a note in some data structure associated with the synchronization condition. A thread that makes the condition true at some point in the future will find the note and take action to make the blocked thread run again.*

2) Languages and Libraries

Thread-level concurrency can be provided to the programmer in the form of explicitly concurrent languages, compiler-supported extensions to traditional sequential languages, or library packages outside the language proper. All three options are widely used, though shared-memory languages are more common at the "low end" (for multicore and small multiprocessor machines), and message passing libraries are more common at the "high end"(for massively parallel supercomputers).

3. THREAD CREATION SYNTAX

Almost every concurrent system allows threads to be created (and destroyed) dynamically. Syntactic and semantic details vary considerably from one language or library to another, but most conform to one of ***six principal options***:

- ➔ co-begin,
- ➔ parallel loops,
- ➔ launch-at-elaboration,
- ➔ fork (with optional join),
- ➔ implicit receipt, and
- ➔ early reply.

The first two options delimit threads with special control-flow constructs.

The others use syntax resembling (or identical to) subroutines.

Most libraries use fork/join, as do Java and C#. Ada uses both launch-at-elaboration and fork. OpenMP uses co-begin and parallel loops. RPC systems are typically based on implicit receipt.

Co-begin

The usual semantics of a compound statement (sometimes delimited with begin. . . end) call for sequential execution of the constituent statements. A co-begin construct calls instead for concurrent execution:

co-begin – – all *n* statements run concurrently
stmt 1
stmt 2
. . .
stmt n
end
Each statement can itself be a sequential or parallel compound, or (commonly) a subroutine call.
Parallel Loops

Many concurrent systems, including OpenMP, several dialects of Fortran, and the recently announced Parallel FX Library for .NET, provide a loop whose iterations are to be executed concurrently. In OpenMP for C, we might say

```
#pragma omp parallel for

for (int i = 0; i < 3; i++) {

printf("thread %d here\n", i);

}
```

Launch-at-Elaboration

In several languages, Ada among them, the code for a thread may be declared with syntax resembling that of a subroutine with no parameters. When the declaration is elaborated, a thread is created to execute the code. In Ada (which calls its threads tasks) we may write

procedure P is
task T is
...
end T;
begin -- P
...
end P;

Task T has its own begin. . . end block, which it begins to execute as soon as control enters procedure P. If P is recursive, there may be many instances of T at the same time, all of which execute concurrently with each other and with whatever task is executing (the current instance of) P. The main program behaves like an initial default task.

When control reaches the end of procedure P, it will wait for the appropriate instance of T (the one that was created at the beginning of this instance of P) to complete before returning. This rule ensures that the local variables of P (which are visible to T under the usual static scope rules) are never deallocated before T is done with them.

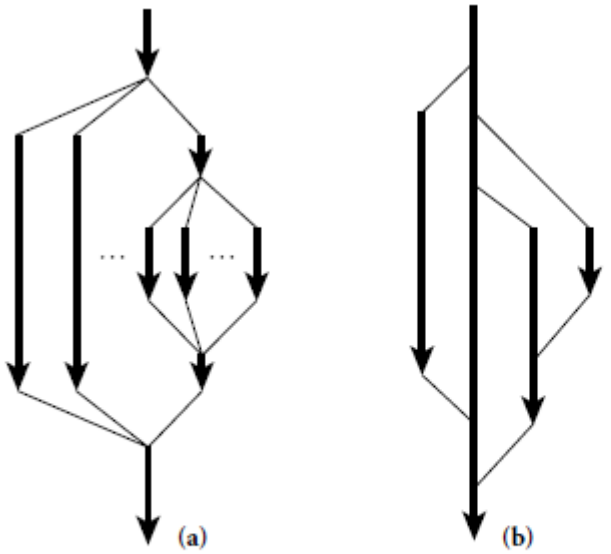


Figure 12.5 **Lifetime of concurrent threads.** With co-begin, parallel loops, or launch-at-elaboration (a), threads are always properly nested. With fork/join (b), more general patterns are possible.

Fork/Join

Co-begin, parallel loops, and launch-at-elaboration all lead to a concurrent control-flow pattern in which thread executions are properly nested (see Figure 12.5a). The fork operation is more general: it makes the creation of threads an explicit, executable operation. The companion join operation, when provided, allows a thread to wait for the completion of a previously forked thread. Because fork and join are not tied to nested constructs, they can lead to arbitrary patterns of concurrent control flow (Figure 12.5b).

In addition to providing launch-at-elaboration tasks, Ada allows the programmer to define task *types*:

task type T is
...
begin
...
end T;

The programmer may then declare variables of type access T (pointer to T), and may create new tasks via dynamic allocation:

pt : access T := new T;

The new operation is a fork; it creates a new thread and starts it executing. There is no explicit join operation in Ada, though parent and child tasks can always synchronize with one another explicitly if desired.

Implicit Receipt

We have assumed in all our examples so far that newly created threads will run in the address space of the creator. In RPC systems it is often desirable to create a new thread automatically in response to an incoming request from some *other* address space. Rather than have an existing thread execute a receive operation, a server can *bind* a communication channel to a local thread body or subroutine. When a request comes in, a new thread springs into existence to handle it. In effect, the bind operation grants remote clients the ability to perform a fork within the server's address space, though the process is often less than fully automatic.

Early Reply

We normally think of sequential subroutines in terms of a single thread, which saves its current context (its program counter and registers), executes the subroutine, and returns to what it was doing before.

The effect is the same, however, if we have two threads—one that executes the caller and another that executes the callee. In this case, the call is essentially a fork/join pair. The caller waits for the callee to terminate before continuing execution.

Nothing dictates, however, that the callee has to terminate in order to release the caller; all it really has to do is complete the portion of its work on which result parameters depend the callee can execute a reply operation that returns results to the caller *without* terminating. After an early reply, the two threads continue concurrently.

IMPLEMENTATION OF THREADS

The threads of a concurrent program are usually implemented on top of one or more *processes* provided by the operating system. At one extreme, we could use a separate OS process for every thread; at the other extreme we could multiplex all of a program's threads on top of a single process. On a supercomputer with a separate processor for every concurrent activity, or in a language in which threads are relatively heavyweight abstractions the one-process-per-thread extreme is often acceptable. Many language implementations adopt an intermediate approach, with a potentially very large number of threads running on top of some smaller number of processes (see Figure 12.6).

The problem with putting every thread on a separate process is that processes are simply too expensive in many operating systems. Because they are implemented in the kernel, performing any operation on them requires a system call. Because they are general purpose, they provide features that most languages do not need, but have to pay for anyway.

In the common two-level organization of concurrency similar code appears at both levels of the system: the language run-time system implements threads on top of one or more processes in much the same way that the operating system implements processes on top of one or more physical processors.

To turn coroutines into threads, we proceed in a series of three steps.

--First, we hide the argument to transfer by implementing a *scheduler* that chooses which thread to run next when the current thread yields the processor.

--Second, we implement a *preemption* mechanism that suspends the current thread automatically on a regular basis, giving other threads a chance to run.

--Third, we allow the data structures that describe our collection of threads to be shared by more than one OS process, possibly on separate processors, so that threads can run on any of the processes.

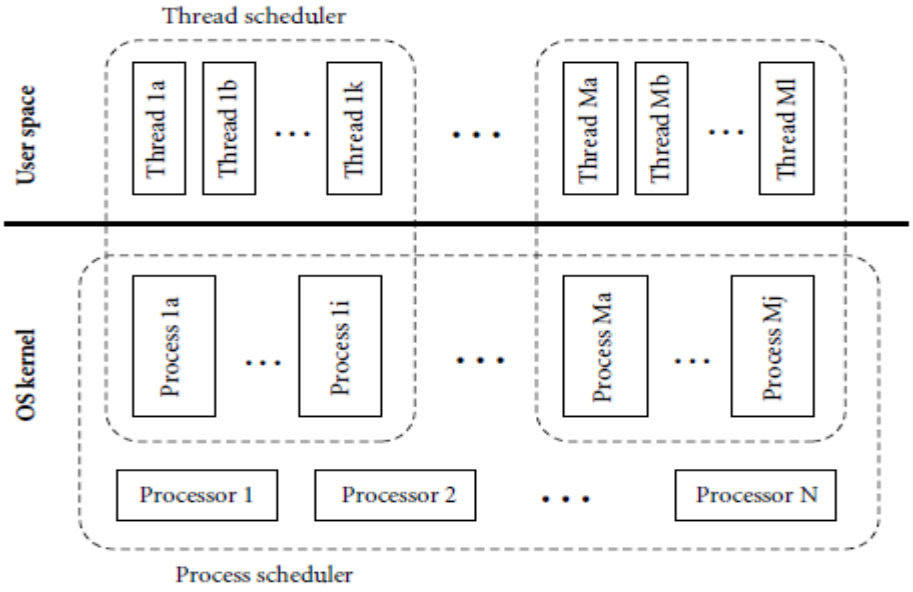


Figure 12.6 **Two-level implementation of threads.** A thread scheduler, implemented in a library or language run-time package, multiplexes threads on top of one or more kernel-level processes, just as the process scheduler, implemented in the operating system kernel, multiplexes processes on top of one or more physical processors.

Uniprocessor Scheduling

At any particular time, a thread is either *blocked* (i.e., for synchronization) or *runnable*. A runnable thread may actually be running on some process or it may be awaiting its chance to do so. Context blocks for threads that are runnable but not currently running reside on a queue called the ***ready list***. Context blocks for threads that are blocked for scheduler-based synchronization reside in data structures associated with the conditions for which they are waiting. To yield the processor to another thread, a running thread calls the scheduler:

procedure reschedule
t : thread := dequeue(ready list)
transfer(t)

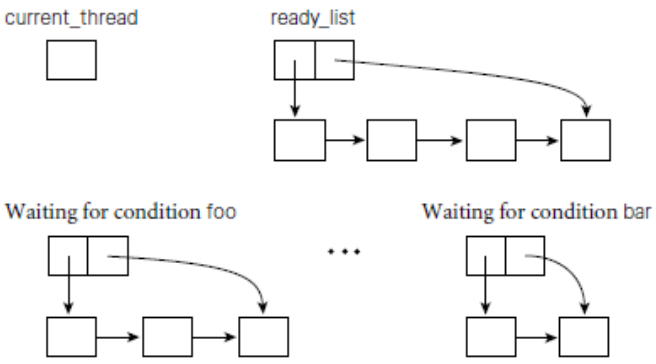


Figure 12.7 **Data structures of a simple scheduler.** A designated current_thread is running. Threads on the ready list are runnable. Other threads are blocked, waiting for various conditions to become true. If threads run on top of more than one OS-level process, each such process will have its own current_thread variable. If a thread makes a call into the operating system, its process may block in the kernel.

SYNCHRONIZATION

Synchronization is the principal semantic challenge for shared-memory concurrent programs. Typically, synchronization serves either to make some operation *atomic* or to delay that operation until some necessary precondition holds. Mutual exclusion ensures that only one thread is executing some *critical section* of code at a given point in time. Critical sections typically transform a shared data structure from one consistent state to another.

Condition synchronization allows a thread to wait for a precondition, often expressed as a predicate on the value(s) in one or more shared variables. It is tempting to think of mutual exclusion as a form of condition synchronization but this sort of condition would require *consensus* among all extant threads, something that condition synchronization doesn't generally provide.

- ➔ **Busy-Wait Synchronization**
- ➔ **Nonblocking Algorithms**

Busy-Wait Synchronization

Busy-wait condition synchronization is easy if we can cast a condition in the form of "location *X* contains value *Y* ": a thread that needs to wait for the condition can simply read *X* in a loop, waiting for *Y* to appear. To wait for a condition involving more than one location, one needs atomicity to read the locations together, but given that, the implementation is again a simple loop.

- SPIN LOCKS
- BARRIERS

Spin Locks

practical spin lock needs to run in constant time and space, and for this one needs an atomic instruction that does more than load or store. Beginning in the 1960s, hardware designers began to equip their processors with instructions that read, modify, and write a memory location as a single atomic operation. The simplest such instruction is known as test_and_set. It sets a Boolean variable to true and returns an indication of whether the variable was previously false. Given test_and_set, acquiring a spin lock is almost trivial:

while not test and set(L)
– – nothing – – spin

In practice, embedding test_and_set in a loop tends to result in unacceptable amounts of communication on a multiprocessor, as the cache coherence mechanism attempts to reconcile writes by multiple processors attempting to acquire the lock.
This over demand for hardware resources is known as *contention*, and is a major obstacle to good performance on large machines. To reduce contention, the writers of synchronization libraries often employ a test-and-test_and_set lock, which spins with ordinary reads.

Barriers

Data-parallel algorithms are often structured as a series of high-level steps, or *phases*, typically expressed as iterations of some outermost loop. Correctness often depends on making sure that every thread completes the previous step before any moves on to the next. A *barrier* serves to provide this synchronization. The simplest way to implement a busy-wait barrier is to use a globally shared counter, modified by an atomic fetch_and_decrement instruction.

Nonblocking Algorithms

When a lock is acquired at the beginning of a critical section, and released at the end, no other thread can execute a similarly protected piece of code at the same time. As long as every thread follows the same conventions, code within the critical section is atomic—it appears to happen all at once. But this is not the only possible way to achieve atomicity.

Semaphores

Semaphores are the oldest of the scheduler-based synchronization mechanisms.
A semaphore is basically a counter with two associated operations, P and V. A thread that calls P atomically decrements the counter and then waits until it is non-negative. A thread that calls V atomically increments the counter and wakes up a waiting thread, if any. It is generally assumed that semaphores are fair, in the sense that threads complete P operations in the same order they start them. A semaphore whose counter is initialized to 1 and for which P and V operations always occur in matched pairs is known as a *binary semaphore*. It serves as a scheduler-based mutual exclusion lock: the P operation acquires the lock; V releases it.

Implicit Synchronization

In several shared-memory languages, the operations that threads can perform on shared data are restricted in such a way that synchronization can be implicit in the operations themselves, rather than appearing as separate, explicit operations.

RUN-TIME PROGRAM MANAGEMENT

Runtime is when a program is running (or being executable). That is, when you start a program running in a computer, it is runtime for that program. In some programming languages, certain reusable programs or "routines" are built and packaged as a "runtime library." These routines can be linked to and used by any program when it is running.

Programmers sometimes distinguish between what gets embedded in a program when it is compiler and what gets embedded or used at runtime. The former is sometimes called "compile time."

For a number of years, technical writers resisted "runtime" as a term, insisting that something like "when a program is run" would obviate the need for a special term. Gradually, the term crept into general usage.

VIRTUAL MACHINES

A *virtual machine* (VM)provides a complete programming environment: its application programming interface (API) includes everything required for correct execution of the programs that run above it. We typically reserve use of the term“ VM to environments whose level of abstraction is comparable to that of a computer implemented in hardware. Every virtual machine API includes an instruction set architecture (ISA) in which to express programs. This may be the same as the instruction set of some existing physical machine, or it may be an artificial instruction set designed to be easier to implement in software and to generate with a compiler. In practice, virtual machines tend to be characterized as either *system* VMs or *process* VMs. A system VM faithfully emulates all the hardware facilities needed to run a standard OS, including both privileged and unprivileged instructions,
memory-mapped I/O, virtual memory, and interrupt facilities. By contrast, a process
VM provides the environment needed by a single user-level process: the unprivileged subset of the instruction set and a library-level interface to I/O and other services.

System VMs are sometimes called ***virtual machine monitors*** (VMMs), because they multiplex a single physical machine among a collection of “guest” operating systems—that is, they monitor the execution of multiple virtual machines, each of which runs a separate guest OS.

Process VMs were originally conceived as a way to increase program portability and to quickly “bootstrap” languages on new hardware.

The Java Virtual Machine

Development of the language that eventually became Java began in 1990–1991, when Patrick Naughton, James Gosling, and Mike Sheridan of Sun Microsystem began work on a programming system for embedded devices. An early version of this system was up and running in 1992, at which time the language was known as Oak. In 1994, after unsuccessful attempts to break into the market for cable TV set-top boxes, the project was retargeted to web browsers, and the name was changed to Java. The first public release of Java occurred in 1995. At that time

code in the JVM was entirely interpreted. A JIT compiler was added in 1998 with the release of Java 2.

Architecture Summary

The interface provided by the JVM was designed to be an attractive target for a Java compiler. It provides direct support for all (and only) the built-in and reference types defined by the Java language. It also enforces both definite assignment and type safety. Finally, it includes built-in support for many of Java's language features and standard library packages, including exceptions, threads, garbage collection, reflection, dynamic loading, and security. Of course, nothing requires that Java byte code (JBC) be produced from Java source.

Compilers targeting the JVM exist for many other languages, including Ruby, JavaScript, Python, and Scheme (all of which are traditionally interpreted), as well as C, Ada, Cobol, and others, which are traditionally compiled. There are even assemblers that allow programmers to write JBC directly. The principal requirement, for both compilers and assemblers, is that they generate correct *class files*. These have a special format understood by the JVM, and must satisfy a variety of structural and semantic constraints.

At start-up time, a JVM is typically given the name of a class file containing the static method main. It loads this class into memory, verifies that it satisfies a variety of required constraints, allocates any static fields, links it to any preloaded library routines, and invokes any initialization code provided by the programmer for classes or static fields. Finally, it calls main in a single thread. Additional classes (needed by the initial class) may be loaded either immediately or lazily on demand. Additional threads may be created via calls to the (built-in) methods of class Thread. The three following subsections provide additional details on JVM
storage management, the format of class files, and the JBC instruction set.

Storage Management

Storage allocation mechanisms in the JVM mirror those of the Java language. There is a global *constant pool*, a set of registers and a stack for each thread, a *method area* to hold executable byte code, and a heap for dynamically allocated objects.

Per-thread data A program running on the JVM begins with a single thread.

Additional threads are created by allocating and initializing a new object of the build-in class Thread, and then calling its start method. Each thread has a small set of base registers, a stack of method call frames, and an optional traditional stack on which to call native (non-Java) methods. Each frame on the method call stack contains an array of local variables, an
operand stack for evaluation of the method's expressions, and a reference into the constant pool that identifies information needed for dynamic linking of called methods. Space for formal parameters is included among the local variables. Variables that are not live at the same time can

share a slot in the array; this means that the same slot may be used at different times for data of different types.

Heap In keeping with the type system of the Java language, a datum in the local variable array or the operand stack is always either a reference or a value of a built-in scalar type. Structured data (objects and arrays) must always lie in the heap. They are allocated, dynamically, using the new and newarray instructions. They are reclaimed automatically via garbage collection. The choice of collection algorithm is left to the implementor of the JVM.

Class Files

Physically, a JVM class file is stored as a stream of bytes. Typically these occupy some real file provided by the operating system, but they could just as easily be a record in a database. On many systems, multiple class files may be combined into a Java archive (.jar) file.

Logically, a class file has a well-defined hierarchical structure. It begins with a "magic number" (0x_cafe_babe), as described in the sidebar on page 662. This is followed by

Major and minor version numbers of the JVM for which the file was created The constant pool Indices into the constant pool for the current class and its superclass Tables describing the class's superinterfaces, fields, and methods Because the JVM is both cleaner and more abstract than a real machine, the Java class file structure is both cleaner and more abstract than a typical object file.

Byte Code

The byte code for a method (or for a constructor or a class initializer) appears in an entry in the class file's method table. It is accompanied by: An indication of the number of local variables, including parameters The maximum depth required in the operand stack

A table of exception handler information, each entry of which indicates
– The byte code range covered by this handler
– The address (index in the code) of the handler itself
– The type of exception caught (an index into the constant pool)

Optional information for debuggers: specifically, a table mapping byte code addresses to line numbers in the original source code and/or a table indicating which source code variable(s) occupy which JVM local variables at which points in the byte code.

Instruction Set Java byte code was designed to be both simple and compact. Orthogonality was a strictly secondary concern. Every instruction begins with a single-byte *opcode*. Arguments, if any, occupy subsequent (unaligned) bytes, with values given in big-endian order. Most instructions actually don't need an argument. Five of these serve special purposes (unused, nop,

debugger breakpoints, implementation dependent). The remainder can be organized into the following categories.

Load/store: Move values back and forth between the operand stack and the local variable array.
Arithmetic: Performinteger or floating point operations on values in the operand stack.
Type conversion: "Widen" or "narrow" values among the built-in types (byte, char, short, int, long, float, and double). Narrowing may result in a loss of precision but never an exception.
Object management: Create or query the properties of objects and arrays; access fields and array elements.
Operand stack management: Push and pop; duplicate; swap.
Control transfer: Perform conditional, unconditional, or multiway branches (switch).
Method calls: Call and return from ordinary and static methods (including constructors and initializers) of classes and interfaces.
Exceptions: throw (no instructions required for catch).
Monitors: Enter and exit (wait, notify, and notifyAll are invoked viamethod calls).
Verification Safety was one of the principal concerns in the definition of the

Java language and virtual machine. Many of the things that can "go wrong" while executing machine code compiled from a more traditional language cannot go wrong when executing byte code compiled from Java. Some aspects of safety are obtained by limiting the expressiveness of the byte-code instruction set or by checking properties at load time. One cannot jump to a nonexistent address, for example, because method calls specify their targets symbolically by name, and branch targets are specified as indices within the code attribute of the current method. Similarly,where hardware allows displacement addressing fromthe frame pointer to access memory outside the current stack frame, the JVM checks at load time to make sure that references to local variables (specified by constant indices
into the local variable array) are within the bounds declared.

The Common Language Infrastructure

Work on the system that became the Common Language Infrastructure (CLI) began at Microsoft Corporation in the late 1990s, and was able to benefit from experience with Java and the JVM, which were already well established. The roots of the CLI, however, go back much further than the advent of Java, and it is these deep roots that account for the most significant differences between the virtual machines.

As early as the mid-1980s, Microsoft recognized the need for interoperability among programming languages running on Windows platforms. In a series of product offerings spanning a decade and a half (see sidebar at bottom of page), the company developed increasingly sophisticated versions of its Component Object Model (COM), first to communicate with, then to call, and finally to share data across program components written in multiple languages.

With the success of Java, it became clear by the mid to late 1990s that a system combining a JVM-style run-time system with the language interoperability of COM could have enormous technical and commercial potential. The .NET project set out to realize this potential. It is in some sense a successor to COM, not based on prior code, nor constrained by backward compatibility, but providing a superset of COM's functionality, and equipped with libraries that allow it to interoperate with older programs. It includes not only a virtual machine, but extensive libraries, servers, and tools for user interface management, database access, security services.

LATE BINDING OF MACHINE CODE

Compilation is a one-time activity, sharply distinguished from program execution. The compiler produces a target program, typically in machine language, which can subsequently be executed many times for many different inputs.

1. Just-in-Time and Dynamic Compilation

To promote the Java language and virtual machine, Sun Microsystems coined the slogan"write once, run anywhere"—the idea being that programs distributed as Java byte code (JBC) could run on a very wide range of platforms. Source code, of course, is also portable, but byte code is much more compact, and can be interpreted without additional pre-processing. Unfortunately, interpretation tends to be expensive. Programs running on early Java implementations could be as much as an order of magnitude slower than compiled code in other languages. Just-in-time compilation is, to first approximation, a technique to retain the portability of byte code while improving execution speed. Like both interpretation and dynamic linking Because a JIT system compiles programs immediately prior to execution, it can add significant delay to program start-up time. Implementors face a difficult tradeoff: to maximize benefits with respect to interpretation, the compiler should

produce good code; to minimize start-up time, it should produce that code very quickly. In general, JIT compilers tend to focus on the simpler forms of target code improvement. Specifically, they often limit themselves to so-called *local* improvements, which operate within individual control flow constructs. Improvements at the *global* (whole method) and *interprocedural* (whole program) level are usually too expensive to consider.

All these factors allows JIT compiler to be faster—and to produce better code— than one might initially expect. In addition, since we are already committed to invoking the JIT compiler at run

time, we can minimize its impact on program start-up latency by running it a bit at a time, rather than all at once:

- ➔ Like a lazy linker JIT compilermay performitswork incrementally. It begins by compiling only the class file that contains the program entry point (i.e., main), leaving *hooks* in the code that call into the run-time system wherever the program is supposed to call a method in another class file. After this small amount of preparation, the program begins execution.

When execution falls into the runtime through an unresolved hook, the runtime invokes the compiler to load the new class file and to link it into the program.

- ➔ To eliminate the latency of compiling even the original class file, the language implementation may incorporate both an interpreter and a JIT compiler. Execution

begins in the interpreter. In parallel, the compiler translates portions of the program into machine code.When the interpreter needs to call a method, it checks to see whether a compiled version is available yet, and if so calls that version instead of interpreting the byte code.We will return to this technique below, in the context of the HotSpot Java compiler and JVM.

- ➔ When a class file is JIT compiled, the language implementation can cache the resulting machine code for later use. This amounts to guessing, speculatively, that the versions of library routines employed in the current run of the program will still be current when the program is run again. Because languages like Java and C# require the appearance of late binding of library routines, this guess must be checked in each subsequent run. If the check succeeds, using a cached copy saves almost the entire cost of JIT compilation.

Finally, JIT compilation affords the opportunity to perform certain kinds of code improvement that are usually not feasible in traditional compilers. It is customary, for example, for software vendors to ship a single compiled version of an application for a given instruction set architecture, even though implementations of that architecture may differ in important ways, including pipeline width and depth; the number of physical (renaming) registers; and the number, size, and speed of the various levels of cache. A JIT compiler may be able to identify the processor implementation on which it is running, and generate code that is tuned for that specific implementation. More important, a JIT compiler may be able to *in-line* calls to dynamically linked library routines. This optimization is particularly important in object-oriented programs, which tend to call many small methods. For such programs, dynamic in-lining can have a dramatic impact on performance.

Dynamic Compilation

Language implementation may choose to delay JIT compilation to reduce the impact on program start-up latency. In some cases, compilation *must* be delayed, either because the source or byte code was not created or discovered until run time, or because we wish to perform optimizations that depend on information gathered during execution. In these cases we say the language implementation employs *dynamic compilation*. Common Lisp systems have used dynamic compilation for many years: the language is typically compiled, but a program can extend itself at run time. Optimization based on run-time statistics is a more recent innovation.

Most programs spend most of their time in a relatively small fraction of the code. Aggressive code improvement on this fraction can yield disproportionately large improvements in program performance. A dynamic compiler can use statistics gathered by run-time *profiling* to identify *hot paths* through the code, which it then optimizes in the background. By rearranging the code to make hot paths contiguous in memory, it may also improve the performance of the instruction cache.

In some situations, a dynamic compiler may even be able to perform optimization that would be unsafe if implemented statically. In some cases, a dynamic compiler may choose to perform optimizations that may be unsafe even in the current program, provided that profiling suggests they will be profitable and run-time checks can determine whether they are safe

2. Binary Translation

Just-in-time and dynamic compilers assume the availability of source code or of byte code that retains all of the semantic information of the source. There are times, however, when it can be useful to recompile object code. This process is known as *binary translation*. It allows already-compiled programs to be run on a machine with a different instruction set architecture.

The principal challenge for binary translation is the loss of information in the original source-to-object code translation. Object code typically lacks both type information and the clearly delineated subroutines and control-flow constructs of source code and byte code. While most of this information appears in the compiler's symbol table, and may sometimes be included in the object file for debugging purposes, vendors usually delete it before shipping commercial products, and a binary translator cannot assume it will be present.

The typical binary translator reads an object file and reconstructs a control flow graph This task is complicated by the lack of explicit information about basic blocks.While branches (the ends of basic blocks) are easy to identify, beginnings are more difficult: since branch targets are sometimes computed at run time or looked up in dispatch tables or virtual function tables, the binary translator must consider the possibility that control

may sometimes jump into the middle of a "probably basic" block. Since translated code will generally not lie at the same address as the original code, computed branches must be translated into code that performs some sort of table lookup, or falls back on interpretation.
Static binary translation is not always possible for arbitrary object code. In addition to computed branches, problems include self-modifying code (programs that write to their own instruction space), dynamically generated code.

Dynamic Optimization
In a long-running program, a dynamic translator may revisit hot paths and optimize them more aggressively. A similar strategy can also be applied to programs that don't need translation—that is, to programs that already exist as machine code for the underlying architecture. This sort of *dynamic optimization* has been reported to improve performance by as much as 20% over already-optimized code, by exploiting run-time profiling information.

By identifying and optimizing traces, Dynamo is able to significantly improve locality in the instruction cache, and to apply standard code improvement techniques across the boundaries between separately compiled modules and dynamically loaded libraries.

IN the figure given below -- For example, it will perform register allocation jointly across print matchings and the predicate p. It can even perform instruction scheduling across basic blocks if it inserts appropriate *compensating code* on branches out of the trace. An instruction in block test2, for example, can be moved into the loop footer if a copy is placed on the branch to the right. Traces have proven to be a very powerful technique.

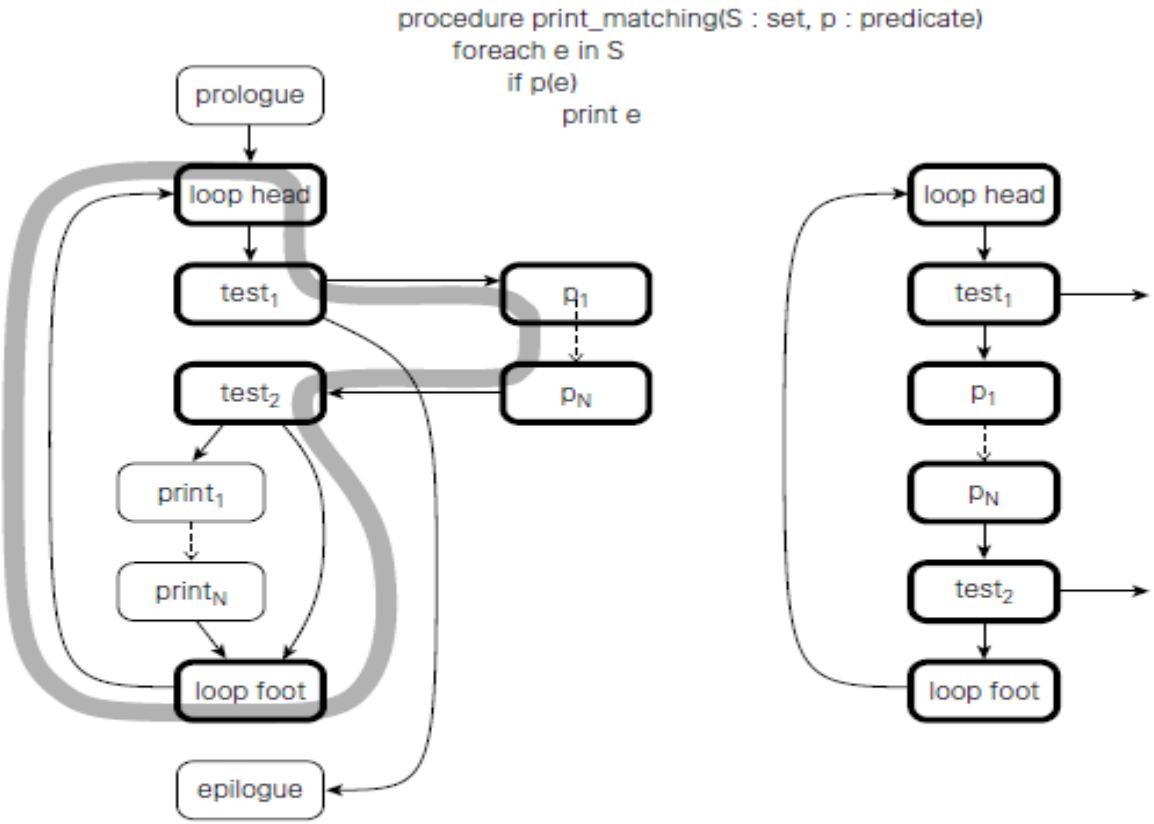


Figure 15.4 Creation of a partial execution trace. Procedure print_matching (shown at top) is often called with a particular predicate, p, which is usually false. The control flow graph (shown at *left*, with hot blocks in bold and the hot path in grey) can be reorganized at run time to improve instruction-cache locality and to optimize across abstraction boundaries (trace shown at *right*).

3. Binary Rewriting

While the goal of a binary optimizer is to improve the performance of a program without altering its behavior, one can also imagine tools designed to *change* that behavior. *Binary rewriting* is a general technique to modify existing executable programs, typically to insert instrumentation of some kind. The most common form of instrumentation collects profiling information. One might count the number of times that each subroutine is called, for example, or the number of times that each loop iterates

Such counts can be stored in a buffer in memory, and dumped at the end of execution. Alternatively, one might log all memory references. Such a log will generally need to be sent to a file as the program runs—it will be too long to fit in memory.

In addition to profiling, binary rewriting can be used to

- ➔ Simulate new architectures: operations of interest to the simulator are replaced with code that jumps into a special run-time library (other code runs at native speed).
- ➔ Evaluate the coverage of test suites, by identifying paths through the code that are not explored by a series of tests.
- ➔ Implement *model checking* for parallel programs, a process that exposes race conditions by forcing a program through different interleavings of operations in different threads.
- ➔ "Audit"the quality of a compiler's optimizations. For example, one might check whether the value loaded into a register is always the same as the value that was already there (such loads suggest that the compiler may have failed to realize that the load was redundant).
- ➔ Insert dynamic semantic checks into a program that lacks them. Binary rewriting can be used not only for simple checks like null-pointer dereference and arithmetic overflow, but for a wide variety of memory access errors as well, including uninitialized variables, dangling references, memory leaks, "double deletes" (attempts to deallocate an already deallocated block of memory), and access off the ends of dynamically allocated arrays.

4. Mobile Code and Sandboxing

Portability is one of the principal motivations for late binding of machine code. Code that has been compiled for one machine architecture or operating system cannot generally be run on another. Code in a byte code (JBC, CIL) or scripting language (JavaScript, Visual Basic), however, is compact and machine independent: it can easily be moved over the Internet and run on almost any platform. Such *mobile code* is increasingly common.

Every major browser supports JavaScript; most enable the execution of Java applets as well. Visual Basic macros are commonly embedded not only in pages meant for viewing with Internet Explorer, but also in Excel, Word, and Outlook documents distributed via email. Increasingly, cell phone platforms are using mobile code to distribute games, productivity tools, and interactive media that run on the phones themselves.

In some sense, mobile code is nothing new: almost all our software comes from other sources; we buy it on a DVD or download it over the Internet and install it on our machines. Historically, this usage model has relied on trust (we assume that software from a well-known company will be safe) and on the very explicit and occasional nature of installation. What has changed in recent years is the desire to download code frequently, from potentially untrusted sources, and often without the conscious awareness of the user. Mobile code carries a variety of risks. It may access and reveal confidential information (spyware). It may interfere with normal use of the computer in annoying ways (adware). It may damage existing programs or data, or save copies of itself that run without the user's intent (malware of various kinds). In the worst cases, it may use the host machine as a "zombie" from which to launch attacks on other users.

To protect against unwanted behavior, both accidental and malicious, mobile code must be executed in some sort of *sandbox*, Sandbox creation is difficult because of the variety of resources that must be protected. At a minimum, one needs to monitor or limit access to processor cycles, memory outside the code's own instructions and data, the file system, network interfaces, other devices (passwords, for example, may be stolen by snooping the keyboard), the window system (e.g., to disable pop-up ads), and any other potentially dangerous services provided by the operating system.

Sandboxing mechanisms lie at the boundary between language implementation and operating systems. Traditionally, OS-provided virtual memory techniques might be used to limit access to memory, but this is generally too expensive for many forms of mobile code. The two most common techniques today—both of which rely on technology discussed in this chapter—are binary rewriting and execution in an untrusting interpreter. Both cases are complicated by an inherent tension between safety and utility: the less we allow untrusted code to do, the less useful it can be. No single policy is likely to work in all cases. Applets may be entirely safe if all they can do is manipulate the image in a window, but macros embedded in a spreadsheet may not be able to do their job without changing the user's data. A major challenge for future work is to find a way to help users— who cannot be expected to understand the technical details—to make informed decisions about what and what not to allow in mobile code

INSPECTION/INTROSPECTION

Symbol table metadata makes it easy for utility programs—just-in-time and dynamic compilers, optimizers, debuggers, profilers, and binary rewriters—to *inspect* a program and reason about its structure and types There is no reason, however, why the use of metadata should be limited to outside tools, and indeed it is not: Lisp has long allowed a program to reason about its own internal structure and types (this sort of reasoning is sometimes called *introspection*).

Java and C# provide similar functionality through a *reflection* API that allows a program to peruse its own metadata. Reflection appears in several other languages as well, including Prolog and all the major scripting languages. In a dynamically typed language such as Lisp, reflection is essential: it allows a library or application function to type check its own arguments. In a statically typed language, reflection supports a variety of programming idioms that were not traditionally feasible.

REFLECTION

Trivially, reflection can be useful when printing diagnostics. Suppose we are trying to debug an old-style (nongeneric) queue in Java, and we want to trace the objects that move through it.

In the dequeue method, just before returning an object rtn of type Object, we might write

```
System.out.println("Dequeued a " + rtn.getClass().getName());
```

If the dequeued object is a boxed int,we will see

```
Dequeued a java.lang.Integer
```

More significantly, reflection is useful in programs that manipulate other programs. Most program development environments, for example, have mechanisms to organize and "pretty-print" the classes, methods, and variables of a program.

In a language with reflection, these tools have no need to examine source code: if they load the already-compiled program into their own address space, they can use the reflection API to query the symbol table information created by the compiler. Interpreters, debuggers, and profilers can work in a similar fashion. In a distributed system, a program can use reflection to create a general-purpose *serialization* mechanism, capable of transforming an almost arbitrary structure into a linear stream of bytes that can be sent over a network and reassembled at the other end.

In the increasingly dynamic world of Internet applications, one can even create conventions by which a program can "query" a newly discovered object to see what methods it implements, and then choose which of these to call.

There are dangers, of course, associatedwith the undisciplined use of reflection. Because it allows an application to peek inside the implementation of a class (e.g., to list its private members), reflection violates the normal rules of abstraction and information hiding. It may be disabled by some security policies (e.g., in sandboxed environments). By limiting the extent to which target code can differ from the source, it may preclude certain forms of code improvement.

SYMBOLIC DEBUGGING

Most programmers are familiar with symbolic debuggers: they are built into most programming language interpreters, virtual machines, and integrated program development environments. They are also available as stand-alone tools, of which the best known is GNU's gdb. The adjective *symbolic* refers to a debugger's understanding of high-level language syntax—the symbols in the original program. Early debuggers understood assembly language only.

In a typical debugging session, the user starts a program under the control of the debugger, or *attaches* the debugger to an already running program. The debugger then allows the user to perform two main kinds of operations. One kind inspects or modifies program data; the other controls execution: starting, stopping, stepping, establishing ***breakpoints*** and ***watchpoints***. A breakpoint specifies that execution should stop if it reaches a particular location in the source code. A watchpoint specifies that execution should stop if a particular variable is read or written. Both breakpoints and watchpoints can typically be made *conditional*, so that execution stops only if a particular Boolean predicate evaluates to true. Both data and control operations depend critically on symbolic information. Both data and control operations also depend on the ability to manipulate a program from outside: to stop and start it, and to read and write its data.

This control can be implemented in at least three ways. The easiest occurs in interpreters. Since an interpreter has direct access to the program's symbol table and is "in the loop" for the execution of every statement, it is a straightforward matter to move back and forth between the program and the debugger, and to give the latter access to the former's data.

For compiled programs, the third implementation of debugger control is by far the most common. It depends on support from the operating system. In Unix, it employs a kernel service known as ptrace. The ptrace kernel call allows a debugger to "grab" (attach to) an existing process or to start a process under its control. The tracing process (the debugger) can intercept any signals sent to the traced process by the operating system and can read and write its registers and memory. If the traced process is currently running, the debugger can stop it by sending it a signal. If it is currently stopped, the debugger can specify the address at which it should resume execution, and can ask the kernel to run it for a single instruction (a process known as *single stepping*) or until it receives another signal.

Perhaps the most mysterious parts of debugging from the user's perspective are the mechanisms used to implement breakpoints, watchpoints, and single stepping. The default implementation, which works on any modern processor, relies on the ability to modify the memory space of the traced process—in particular, the portion containing the program's code.

PERFORMANCE ANALYSIS

Before placing a debugged program into production use, one often wants to understand and if possible improve—its performance Perhaps the simplest way to measure, at least approximately, the amount of time spent in each part of the code is to *sample* the program counter (PC) periodically.

This approach was exemplified by the classic prof tool in Unix. By linking with a special prof library, a program could arrange to receive a periodic timer signal—once a millisecond, say—in response to which it would increment a counter associated with the current PC. After execution, the prof post-processor would correlate the counters with an address map of the program's code and produce a statistical summaryof the percentage of time spent in each subroutine and loop.

While simple, prof had some serious limitations. Its results were only approximate, and could not capture fine-grain costs. It also failed to distinguish among calls to a given routine frommultiple locations. If we want to know which of A, B,

Call graph profiling and C is the biggest contributor to program run time, it is not particularly helpful to learn that all three of them call D where most of the time is actually spent. If we want to know whether it is A's Ds, B's Ds, or C's Ds that are so expensive, we can use the (slightly) more recent gprof tool, which relies on compiler support to instrument procedure prologues. As the instrumented program runs, it logs the number of times that D is called from each location. The gprof post-processor then assumes that the total time spent in D can accurately be apportioned among the call sites according to the relative number of calls. More sophisticated tools log not only the caller and callee but also the stack *backtrace* (the contents of the dynamic chain), allowing them to cope with the case in which D consumes twice as much time when called from A as it does when called from B or C.

If our program is underperforming for algorithmic reasons, it may be enough to know where it is spending the bulk of its time. We can focus our attention on improving the source code in the places it will matter most. If the program is underperforming for other reasons, however, we generally need to know *why*. Is it cache misses due to poor locality, perhaps? Branch mispredictions? Poor pipeline performance? Tools to address these and similar questions generally rely on more extensive instrumentation of the code or on some sort of hardware support.

As an example of instrumentation, consider the task of identifying basic blocks that execute an unusually small number of instructions per cycle. To find such blocks we can combine

(1) the aggregate time spent in each block (obtained by statistical sampling),

(2) a count of the number of times each block executes (obtained via instrumentation), and
(3) static knowledge of the number of instructions in each block. If basic block *i* contains *ki*

$N = \sum_i k_i n_i$ be the total number of instructions in the run. If statistical sampling indicates that block i accounts for x_i% of the time in the run and x_i is significantly larger than $(k_i n_i)/N$, then something strange is going on—probably an unusually large number of cache misses.

instructions and executes *ni* times during a run of a program, it contributes *kini* dynamic instructions to that run. Let

Most modern processors provide a set of *performance counters* that can be used to good effect by performance analysis tools. The Intel PentiumM processor, for example, has two performance counters that can be configured by the kernel to count any of 47 different kinds of *events*, including branch mispredictions; TLB (address translation) misses; and various kinds of cache misses, interrupts, executed instructions, and pipeline stalls. Unfortunately, performance counters are generally a scarce resource (one might often wish for many more of them). Their number, type, and mode of operation varies greatly from processor to processor; direct access to them is usually available only in kernel mode; and operating systems do not always export that access to user-level programs with a convenient or uniform interface. Portable tools that make use of performance counters are an active topic of research.

CONTENT BEYOND SYLLABUS

Python programming paradigms

Python supports three types of Programming paradigms

- Object Oriented programming paradigms
- Procedure Oriented programming paradigms
- Functional programming paradigms

Object Oriented programming paradigms

In the object-oriented programming paradigm, objects are the key element of paradigms. Objects can simply be defined as the instance of a class that contains both data members and the method functions. Moreover, the object-oriented style relates data members and methods functions that support encapsulation and with the help of the concept of an inheritance, the code can be easily reusable but the major disadvantage of object-oriented programming paradigm is that if the code is not written properly then the program becomes a monster.

Advantages

- Relation with Real world entities
- Code reusability
- Abstraction or data hiding

Disadvantages

- Data protection
- Not suitable for all types of problems
- Slow Speed

Example:

```
# class Emp has been defined here

class Emp:

    def __init__(self, name, age):

        self.name = name

        self.age = age
```

```
    def info(self):
        print("Hello, % s. You are % s old." % (self.name, self.age))


# Objects of class Emp has been
# made here
Emps = [Emp("John", 43),
    Emp("Hilbert", 16),
    Emp("Alice", 30)]


# Objects of class Emp has been
# used here
for emp in Emps:
    emp.info()
```

Output:

Hello, John. You are 43 old.

Hello, Hilbert. You are 16 old.

Hello, Alice. You are 30 old.

Procedural programming paradigms

In Procedure Oriented programming paradigms, series of computational steps are divided modules which means that the code is grouped in functions and the code is serially executed

step by step so basically, it combines the serial code to instruct a computer with each step to perform a certain task. This paradigm helps in the modularity of code and modularization is usually done by the functional implementation. This programming paradigm helps in an easy organization related items without difficulty and so each file acts as a container.

Advantages

- General-purpose programming
- Code reusability
- Portable source code

Disadvantages

- Data protection
- Not suitable for real-world objects
- Harder to write

Example:

```
# Procedural way of finding sum

# of a list



mylist = [10, 20, 30, 40]



# modularization is done by

# functional approach

def sum_the_list(mylist):

    res = 0

    for val in mylist:

        res += val

    return res
```

```
print(sum_the_list(mylist))
```

Output:
100

Functional programming paradigms

Functional programming paradigms is a paradigm in which everything is bind in pure mathematical functions style. It is known as declarative paradigms because it uses declarations overstatements. It uses the mathematical function and treats every statement as functional expression as an expression is executed to produce a value. Lambda functions or Recursion are basic approaches used for its implementation. The paradigms mainly focus on "what to solve" rather than "how to solve". The ability to treat functions as values and pass them as an argument make the code more readable and understandable.

Advantages

- Simple to understand
- Making debugging and testing easier
- Enhances the comprehension and readability of the code

Disadvantages

- Low performance
- Writing programs is a daunting task
- Low readability of the code

Example:

```
# Functional way of finding sum of a list

import functools

 mylist = [11, 22, 33, 44]

 # Recursive Functional approach

def sum_the_list(mylist):

   if len(mylist) == 1:
```

```
        return mylist[0]
    else:
        return mylist[0] + sum_the_list(mylist[1:])


# lambda function is used
print(functools.reduce(lambda x, y: x + y, mylist))
```

Output:

110