13 Concurrency

Introduction

- Concurrency can occur at four levels:
  - Machine instruction level
  - High-level language statement level
  - Unit level
  - Program level
- Because there are no language issues in instruction- and program-level concurrency, they are not addressed here

Multiprocessor Architectures

- Late 1950s - one general-purpose processor and one or more special-purpose processors for input and output operations
- Early 1960s - multiple complete processors, used for program-level concurrency
- Mid-1960s - multiple partial processors, used for instruction-level concurrency
- Single-Instruction Multiple-Data (SIMD) machines
- Multiple-Instruction Multiple-Data (MIMD) machines
  - Independent processors that can be synchronized (unit-level concurrency)

Categories of Concurrency

- A *thread of control* in a program is the sequence of program points reached as control flows through the program
- Categories of Concurrency:
  - *Physical concurrency* - Multiple independent processors (multiple threads of control)
  - *Logical concurrency* - The appearance of physical concurrency is presented by time-sharing one processor (software can be designed as if there were multiple threads of control)
- Coroutines (*quasi-concurrency) have a single thread of control*

Motivations for Studying Concurrency

- Involves a different way of designing software that can be very useful—many real-world situations involve concurrency
- Multiprocessor computers capable of physical concurrency are now widely used

Introduction to Subprogram-Level Concurrency

- A *task* or *process* is a program unit that can be in concurrent execution with other program units
- Tasks differ from ordinary subprograms in that:
  - A task may be implicitly started
  - When a program unit starts the execution of a task, it is not necessarily suspended
  - When a task’s execution is completed, control may not return to the caller
- Tasks usually work together

Two General Categories of Tasks

- *Heavyweight tasks* execute in their own address space
- *Lightweight tasks* all run in the same address space – more efficient
- A task is *disjoint* if it does not communicate with or affect the execution of any other task in the program in any way

Task Synchronization

- A mechanism that controls the order in which tasks execute
- Two kinds of synchronization
  - *Cooperation* synchronization
  - *Competition* synchronization
- Task communication is necessary for synchronization, provided by:
  - Shared nonlocal variables
  - Parameters
  - Message passing

Kinds of synchronization

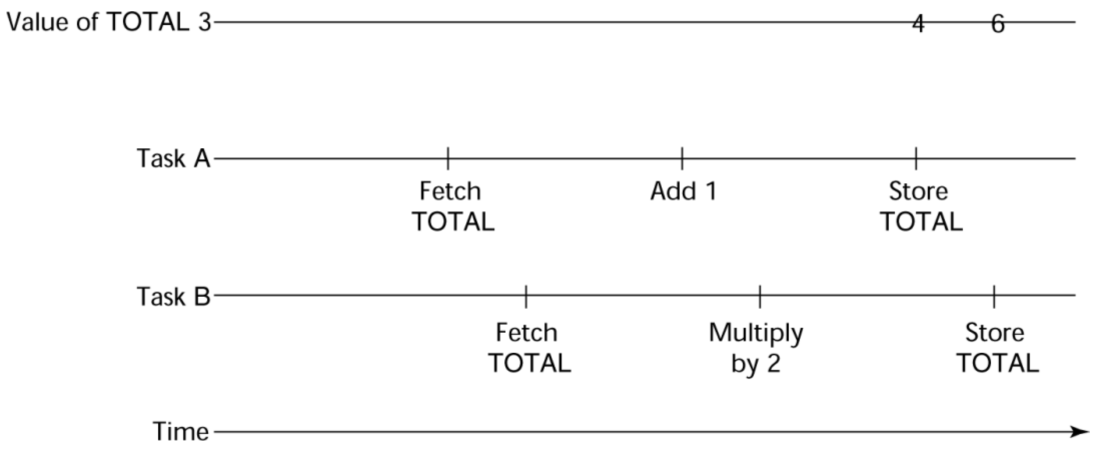
Cooperation synchronization

- Task B must wait for task A to complete some specific activity before task B can continue its execution, e.g., the producer-consumer problem

Competition synchronization

- Two or more tasks must use some resource that cannot be simultaneously used, e.g., a shared counter
  - Competition is usually provided by mutually exclusive access (approaches are discussed later)

Need for Competition Synchronization



Scheduler

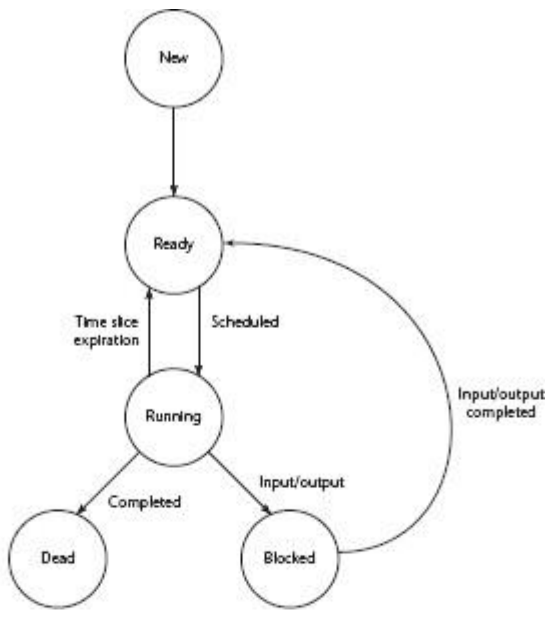
- Providing synchronization requires a mechanism for delaying task execution
- Task execution control is maintained by a program called the *scheduler*, which maps task execution onto available processors

Task Execution States

- *New* - created but not yet started
- *R*eady - ready to run but not currently running (no available processor)
- *Running*

- *Blocked* - has been running, but cannot now continue (usually waiting for some event to occur)
- *Dead* - no longer active in any sense

Figure 13.2
Flow diagram of task states



Liveness and Deadlock

- *Liveness* is a characteristic that a program unit may or may not have
  - In sequential code, it means the unit will
    eventually complete its execution
- In a concurrent environment, a task can easily lose its liveness
- If all tasks in a concurrent environment lose their liveness, it is called *deadlock*

Design Issues for Concurrency

- Competition and cooperation synchronization
- Controlling task scheduling
- How and when tasks start and end execution
- How and when are tasks created

Methods of Providing Synchronization

- Semaphores
- Monitors
- Message Passing

Semaphores

- Dijkstra - 1965
- A *semaphore* is a data structure consisting of a counter and a queue for storing task descriptors
- Semaphores can be used to implement guards on the code that accesses shared data structures
- Semaphores have only two operations, *wait* and *release* (originally called *P* and *V* by Dijkstra)
- Semaphores can be used to provide both competition and cooperation synchronization

Cooperation Synchronization with Semaphores

- Example: A shared buffer
- The buffer is implemented as an ADT with the operations DEPOSIT and FETCH as the only ways to access the buffer
- Use two semaphores for cooperation: emptyspots and fullspots
- The semaphore counters are used to store the numbers of empty spots and full spots in the buffer
- DEPOSIT must first check emptyspots to see if there is room in the buffer
- If there is room, the counter of emptyspots is decremented and the value is inserted
- If there is no room, the caller is stored in the queue of emptyspots
- When DEPOSIT is finished, it must increment the counter of fullspots
- FETCH must first check fullspots to see if there is a value
    - If there is a full spot, the counter of fullspots is decremented and the value is removed
    - If there are no values in the buffer, the caller must be placed in the queue of fullspots
    - When FETCH is finished, it increments the counter of emptyspots
- The operations of FETCH and DEPOSIT on the semaphores are accomplished through two semaphore operations named *wait* and *release*

Semaphores: Wait Operation

```
wait(aSemaphore)
if aSemaphore’s counter > 0 then
  decrement aSemaphore’s counter
else
  put the caller in aSemaphore’s queue
  attempt to transfer control to a ready task
    -- if the task ready queue is empty,
    -- deadlock occurs
end
```

Semaphores: Release Operation

```
release(aSemaphore)
if aSemaphore’s queue is empty then
  increment aSemaphore’s counter
else
  put the calling task in the task ready queue
  transfer control to a task from aSemaphore’s queue
end
```

Cooperation Synchronization with semaphores

Producer Code

```
semaphore fullspots, emptyspots;
```

```
fullstops.count = 0;
emptyspots.count = BUFLEN;
task producer;
        loop
        -- produce VALUE –-
        wait (emptyspots); {wait for space}
        DEPOSIT(VALUE);
        release(fullspots); {increase filled}
        end loop;
end producer;
```

Consumer Code

```
task consumer;
        loop
        wait (fullspots);{wait till not empty}}
        FETCH(VALUE);
        release(emptyspots); {increase empty}
        -- consume VALUE –-
        end loop;
end consumer;
```

Competition Synchronization with Semaphores

- A third semaphore, named access, is used to control access (competition synchronization)
  - The counter of access will only have the values 0 and 1
  - Such a semaphore is called a *binary semaphore*
- Note that wait and release must be atomic!

producer code

```
semaphore access, fullspots, emptyspots;
access.count = 0;
fullstops.count = 0;
emptyspots.count = BUFLEN;
task producer;
        loop
        -- produce VALUE –-
        wait(emptyspots); {wait for space}
        wait(access);     {wait for access)
        DEPOSIT(VALUE);
        release(access); {relinquish access}
        release(fullspots);  {increase  filled}
        end loop;
end producer;
```

Consumer Code

```
task consumer;
    loop
    wait(fullspots);{wait till not empty}
    wait(access); {wait for access}
    FETCH(VALUE);
    release(access); {relinquish access}
    release(emptyspots); {increase empty}
    -- consume VALUE –-
    end loop;
end consumer;
```

Evaluation of Semaphores

- Misuse of semaphores can cause failures in cooperation synchronization, e.g., the buffer will overflow if the wait of fullspots is left out
- Misuse of semaphores can cause failures in competition synchronization, e.g., the program will deadlock if the release of access is left out

Monitors

One solution to some of the problems of semaphores in a concurrent environment is to encapsulate shared data structures with their operations and hide their representations—that is, tomake shared data structures abstract data types with some special restrictions

- The idea: encapsulate the shared data and its operations to restrict access
- A monitor is an abstract data type for shared data

Competition Synchronization

- Shared data is resident in the monitor (rather than in the client units)
- All access resident in the monitor
  - Monitor implementation guarantee synchronized access by allowing only one access at a time
  - Calls to monitor procedures are implicitly queued if the monitor is busy at the time of the call

Cooperation Synchronization

- Cooperation between processes is still a programming task
  - Programmer must guarantee that a shared buffer does not experience underflow or overflow

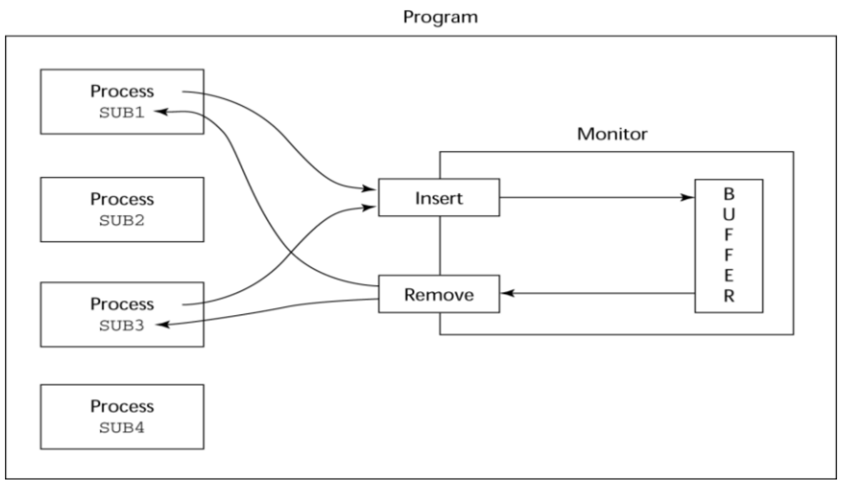


Fig: A program using a monitor to control access to a shared buffer

Evaluation of Monitors

- A better way to provide competition synchronization than are semaphores
- Semaphores can be used to implement monitors
- Monitors can be used to implement semaphores
- Support for cooperation synchronization is very similar as with semaphores, so it has the same problems

Message Passing

- Message passing is a general model for concurrency
  - It can model both semaphores and monitors
  - It is not just for competition synchronization
- Central idea: task communication is like seeing a doctor--most of the time she waits for you or you wait for her, but when you are both ready, you get together, or *rendezvous*

Message Passing Rendezvous

- To support concurrent tasks with message passing, a language needs:

  - A mechanism to allow a task to indicate when it is willing to accept messages

-If task A is waiting for a message at the time task B sends that message, the message can be transmitted. This actual transmission of the message is called a **rendezvous**.

Ada Support for Concurrency

- The Ada 83 Message-Passing Model
  - Ada tasks have specification and body parts, like packages; the spec has the interface, which is the collection of entry points:

```
task Task_Example is
        entry ENTRY_1 (Item : in Integer);
end Task_Example;
```

Task Body

- The body task describes the action that takes place when a rendezvous occurs
- A task that sends a message is suspended while waiting for the message to be accepted and during the rendezvous
- Entry points in the spec are described with accept clauses in the body

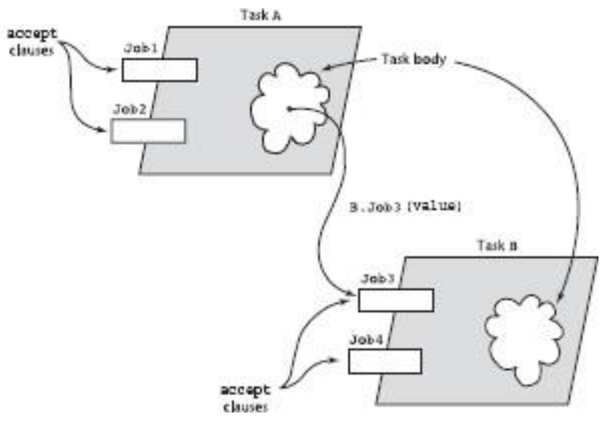
Example of a Task Body

```
task body Task_Example is
   begin
   loop
   accept Entry_1 (Item: in Float) do
   ...
   end Entry_1;
   end loop;
  end Task_Example;
```

The task executes to the top of the accept clause and waits for a message
During execution of the accept clause, the sender is suspended

Figure 13.5

Graphical representation of a rendezvous caused by a message sent from task A to task B



The Concept of Synchronous Message Passing

Message passing can be either synchronous or asynchronous.

The basic concept of synchronous message passing is that tasks are often busy, and when busy, they cannot be interrupted by other units. Suppose task A and task B are both in execution, and A wishes to

send a message to B. Clearly, if B is busy, it is not desirable to allow another task to interrupt it. That would disturb B's current processing. . The alternative is to provide a Linguistic mechanism that allows a task to specify to other tasks when it is ready to receive messages.
.
If task A is waiting for a message at the time task B sends that message, the message can be transmitted. This actual transmission of the message is called a **rendezvous**. Note that a rendezvous canoccur only if both the sender and receiver want it to happen. During a rendezvous, the information of the message can be transmitted in either or both directions.

Asynchronous message passing

- Provided through asynchronous select structures
- An asynchronous select has two triggering alternatives, and entry clause or a delay - The entry clause is triggered when sent a message; the delay clause is triggered when its time limit is reached

```
task WATER_MONITOR; -- specification
task body WATER_MONITOR is -- body
begin
loop
if WATER_LEVEL > MAX_LEVEL
then SOUND_ALARM;
end if;
delay 1.0; -- No further execution
-- for at least 1 second
end loop;
end WATER_MONITOR;
```

Both cooperation and competition synchronization of tasks can be conveniently handled with the message-passing model, as described in the following section.

Cooperation Synchronization with message Passing

- Provided by Guarded accept clauses
-Guarded commands are the basis of the construct designed for controlling message passing.

- Example:

```
task body buf_task is
   begin
   loop

when not FULL(BUFFER) =>
accept DEPOSIT (NEW_VALUE) do

- - - - -
end DEPOSIT;
end  loop;
end buf_task;
```

Def: A clause whose guard is true is called open.
Def: A clause whose guard is false is called closed.

Competition Synchronization with Message Passing:

- Example--a shared buffer
- Encapsulate the buffer and its operations in a task
- Competition synchronization is implicit in the semantics of accept clauses
- Only one accept clause in a task can be active at any given time

Ex: The tasks for the producer and consumer that could use buf_task have the following form

```
task Buf_Task is
entry Deposit(Item : in Integer);
entry Fetch(Item : out Integer);
end Buf_Task;

task body Buf_Task is
Bufsize : constant Integer := 100;
begin
loop
select
when Filled < Bufsize =>
accept Deposit(Item : in Integer) do
Buf(Next_In) := Item;
end Deposit;
or
when Filled > 0 =>
accept Fetch(Item : out Integer) do
Item := Buf(Next_Out);
end Fetch;
Filled := Filled - 1;
end select;
end loop;
end Buf_Task;
```

In this example, both **accept** clauses are extended. These extended clauses can be executed concurrently with the tasks that called the associated **accept** clauses.The tasks for a producer and a consumer that could use `Buf_Task` have the following form:

```
task Producer;
task Consumer;
task body Producer is
New_Value : Integer;
begin
loop
-- produce New_Value --
Buf_Task.Deposit(New_Value);
end loop;
end Producer;

task body Consumer is
Stored_Value : Integer;
begin
loop
Buf_Task.Fetch(Stored_Value);
-- consume Stored_Value --
```

```
end loop;
end Consumer;
```

Java Threads

- The concurrent units in Java are methods named run
  - A run method code can be in concurrent execution with other such methods
  - The process in which the run methods execute is called a *thread*

```
Class myThread extends Thread
        public void run () {…}
}
…
Thread myTh = new MyThread ();
myTh.start();
```

Controlling Thread Execution

- The Thread class has several methods to control the execution of threads
  - The **yield** is a request from the running thread to voluntarily surrender the processor
  - The **sleep** method can be used by the caller of the method to block the thread
  - The **join** method is used to force a method to delay its execution until the run method of another thread has completed its execution

Thread Priorities

- A thread's default priority is the same as the thread that create it
  - If main creates a thread, its default priority is NORM_PRIORITY
- Threads defined two other priority constants, MAX_PRIORITY and MIN_PRIORITY
- The priority of a thread can be changed with the methods setPriority

Competition Synchronization with Java Threads

- A method that includes the synchronized modifier disallows any other method from running on the object while it is in execution

```
…
public synchronized void deposit( int i) {…}
public synchronized int fetch() {…}
…
```

- The above two methods are synchronized which prevents them from interfering with each other
- If only a part of a method must be run without interference, it can be synchronized thru synchronized statement

synchronized (*expression*)
  statement

Cooperation Synchronization with Java Threads

- Cooperation synchronization in Java is achieved via wait, notify, and notifyAll methods
  - All methods are defined in Object, which is the root class in Java, so all objects inherit them
- The wait method must be called in a loop
- The notify method is called to tell one waiting thread that the event it was waiting has happened
- The notifyAll method awakens all of the threads on the object's wait list

Java's Thread Evaluation

- Java's support for concurrency is relatively simple but effective
- Not as powerful as Ada's tasks

C# Threads

- Loosely based on Java but there are significant differences
- Basic thread operations
  - Any method can run in its own thread
  - A thread is created by creating a Thread object
  - Creating a thread does not start its concurrent execution; it must be requested through the Start method
  - A thread can be made to wait for another thread to finish with Join
  - A thread can be suspended with Sleep
  - A thread can be terminated with Abort

Synchronizing Threads

- Three ways to synchronize C# threads
  - The Interlocked class
    - Used when the only operations that need to be synchronized are incrementing or decrementing of an integer
  - The lock statement
    - Used to mark a critical section of code in a thread

  lock (expression) {… }
  - The Monitor class
    - Provides four methods that can be used to provide more sophisticated synchronization

C#'s Concurrency Evaluation

- An advance over Java threads, e.g., any method can run its own thread
- Thread termination is cleaner than in Java
- Synchronization is more sophisticated

Statement-Level Concurrency

- Objective: Provide a mechanism that the programmer can use to inform compiler of ways it can map the program onto multiprocessor architecture
- Minimize communication among processors and the memories of the other processors

High-Performance Fortran

- A collection of extensions that allow the programmer to provide information to the compiler to help it optimize code for multiprocessor computers
- Specify the number of processors, the distribution of data over the memories of those processors, and the alignment of data

Primary HPF Specifications

- Number of processors
  !HPF$ PROCESSORS procs (n)
- Distribution of data
  !HPF$ DISTRIBUTE (*kind*) ONTO procs :: *identifier_list*
  - *kind* can be BLOCK (distribute data to processors in blocks) or CYCLIC (distribute data to processors one element at a time)
- Relate the distribution of one array with that of another
  ALIGN *array1_element* WITH *array2_element*

Statement-Level Concurrency Example

```
REAL list_1(1000), list_2(1000)
    INTEGER list_3(500), list_4(501)
!HPF$ PROCESSORS proc (10)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs ::list_1, list_2
!HPF$ ALIGN list_1(index) WITH list_2 (index)
!HPF$ ALIGN list_3(index) WITH list_4 (index+1)
    …
    list_1 (index) = list_2(index)
    list_3(index) = list_4(index+1)
```

- FORALL statement is used to specify a list of statements that may be executed concurrently
  ```
  FORALL (index = 1:1000)
  list_1(index) = list_2(index)
  ```

- Specifies that all 1,000 RHSs of the assignments can be evaluated before any assignment takes place

Summary

- Concurrent execution can be at the instruction, statement, or subprogram level
- Physical concurrency: when multiple processors are used to execute concurrent units
- Logical concurrency: concurrent united are executed on a single processor
- Two primary facilities to support subprogram concurrency: competition synchronization and cooperation synchronization
- Mechanisms: semaphores, monitors, rendezvous, threads
- High-Performance Fortran provides statements for specifying how data is to be distributed over the memory units connected to multiple processors

Introduction to Exception Handling

- In a language without exception handling
  - When an exception occurs, control goes to the operating system, where a message is displayed and the program is terminated
- In a language with exception handling
  - Programs are allowed to trap some exceptions, thereby providing the possibility of fixing the problem and continuing

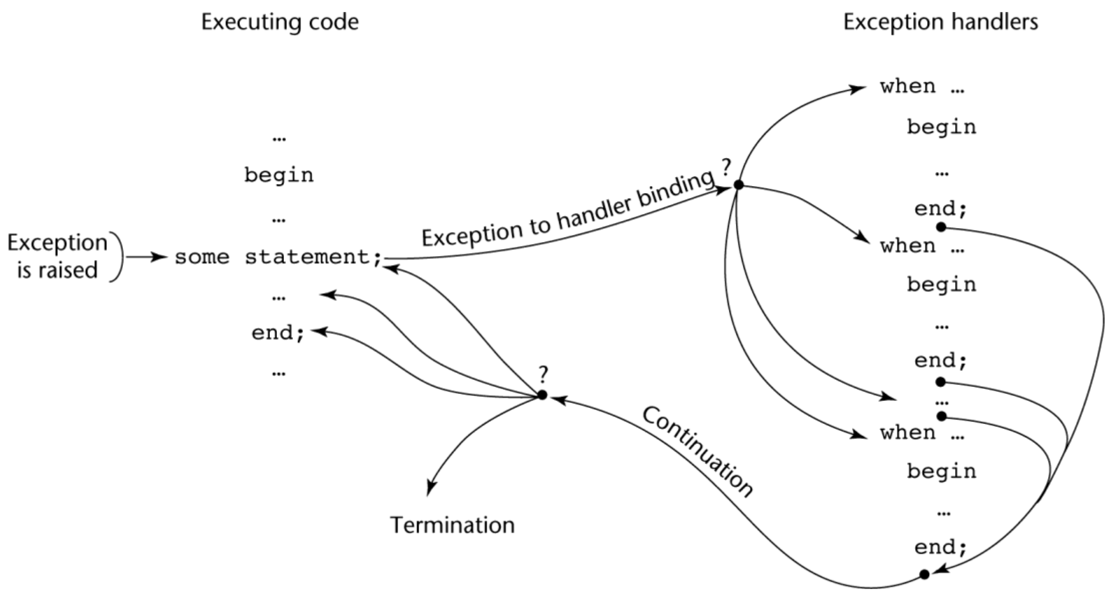
Basic Concepts

- Many languages allow programs to trap input/output errors (including EOF)
- An *exception* is any unusual event, either erroneous or not, detectable by either hardware or software, that may require special processing
- The special processing that may be required after detection of an exception is called *exception handling*
- The exception handling code unit is called an *exception handler*

Advantages of Built-in Exception Handling

- Error detection code is tedious to write and it clutters the program
- Exception handling encourages programmers to consider many different possible error

Exception Handling Control Flow



Exception Handling in Ada

- The frame of an exception handler in Ada is either a subprogram body, a package body, a task, or a block
- Because exception handlers are usually local to the code in which the exception can be raised, they do not have parameters

Ada Exception Handlers

- Handler form:
  when *exception_choice*{|*exception_choice*} => *statement_sequence*

 ...
 [when others =>
   statement_sequence]

 exception_choice form:
   exception_name | others
- Handlers are placed at the end of the block or unit in which they occur

Example:

```
type Age_Type is range 0..125;
type Age_List_Type is array (1..4) of Age_Type;
package Age_IO is new Integer_IO (Age_Type);
```

```
use Age_IO;
Age_List : Age_List_Type;
. . .
begin
for Age_Count in 1..4 loop
loop -- loop for repetition when exceptions occur
Except_Blk:
begin -- compound to encapsulate exception handling
Put_Line("Enter an integer in the range 0..125");
Get(Age_List(Age_Count));
exit;
exception
when Data_Error => -- Input string is not a number
Put_Line("Illegal numeric value");
Put_Line("Please try again");
when Constraint_Error => -- Input is < 0 or > 125
Put_Line("Input number is out of range");
Put_Line("Please try again");
end Except_Blk;
end loop; -- end of the infinite loop to repeat input
-- when there is an exception
end loop; -- end of for Age_Count in 1..4 loop
. . .
```

Predefined Exceptions

- CONSTRAINT_ERROR - index constraints, range constraints, etc.
- NUMERIC_ERROR - numeric operation cannot return a correct value (overflow, division by zero, etc.)
- PROGRAM_ERROR - call to a subprogram whose body has not been elaborated
- STORAGE_ERROR - system runs out of heap
- TASKING_ERROR - an error associated with tasks

Evaluation

- Ada was the only widely used language with exception handling until it was added to C++

Exception Handling in C++

- Added to C++ in 1990
- Design is based on that of Ada, and ML

C++ Exception Handlers

- Exception Handlers Form:

```
try {
-- code that is expected to raise an exception
}
catch (formal parameter) {
-- handler code
}
...
catch (formal parameter) {
-- handler code
}
```

C++ Exception Handling example

```
int main()
{
int a,b;
cout<<—enter a,b values:‖;
cin>>a>>b;
   try{
        if(b!=0)
          cout<<—result is:‖<<(a/b);
          else
        throw b;
       }
catch(int e)
  {
    cout<<—divide by zero error occurred due
          to b= — << e;
  }
}
```

The catch Function

- catch is the name of all handlers--it is an overloaded name, so the formal parameter of each must be unique
- The formal parameter need not have a variable
  - It can be simply a type name to distinguish the handler it is in from others
- The formal parameter can be used to transfer information to the handler
- The formal parameter can be an ellipsis, in which case it handles all exceptions not yet handled

Binding Exceptions to Handlers

- Exceptions are all raised explicitly by the statement:
  throw [*expression*];
- The brackets are metasymbols
- A throw without an operand can only appear in a handler; when it appears, it simply re-raises the exception, which is then handled elsewhere
- The type of the expression disambiguates the intended handler

Continuation

- After a handler completes its execution, control flows to the first statement after the last handler in the sequence of handlers of which it is an element

Exception Handling in Java

- Based on that of C++, but more in line with OOP philosophy
- All exceptions are objects of classes that are descendants of the Throwable class

Classes of Exceptions

- The Java library includes two subclasses of Throwable :
  - Error
    - Thrown by the Java interpreter for events such as heap overflow
    - Never handled by user programs
  - Exception
    - User-defined exceptions are usually subclasses of this
    - Has two predefined subclasses, IOException and RuntimeException (e.g., ArrayIndexOutOfBoundsException and NullPointerException

Java Exception Handlers

- Like those of C++, except every catch requires a named parameter and all parameters must be descendants of Throwable
- Syntax of try clause is exactly that of C++, except for the finally clause

Binding Exceptions to Handlers

- Exceptions are thrown with throw, as in C++, but often the throw includes the new operator to create the object as : throw new MyException();

Checked and Unchecked Exceptions

- The Java throws clause is quite different from the throw clause of C++

- Exceptions of class Error and RunTimeException and all of their descendants are called unchecked exceptions; all other exceptions are called checked exceptions
- Checked exceptions that may be thrown by a method must be either:
  - Listed in the throws clause, or
  - Handled in the method

finally clause

- Can appear at the end of a try construct
- Form:

```
finally {
...
}
```

Example:

- Purpose: To specify code that is to be executed, regardless of what happens in the try construct
- A try construct with a finally clause can be used outside exception handling

```
try {
        for (index = 0; index < 100; index++) {
                …
                if (…) {
                        return;
                }  //** end of if
}  //** end of try clause
finally {
        …
}  //** end of try construct
```

Example of Exception Handling in java

```
import java.io.*;
class Test
{
        public static void main(String args[]) throws IOException
        {
                int a[],b,c;
                DataInputStream dis=new DataInputStream(System.in);
                a=new int[5];
                for(int i=0;i<5;i++)
                {
                        a[i]=Integer.parseInt(dis.readLine());
                }
                //displaying the values from array
                try{
                        for(int i=0;i<7;i++)
                        {
                                System.out.println(a[i]);
                        }
                }
                catch(Exception e)
                {

                                System.out.println("The run time error is:"+e);
                }

                finally
            {

                        System.out.print("100% will be executed");
            }

        }
}
```

```
O/P: 1 2 3 4 5

        1 2 3 4 5
The runtime error is : ArrayIndexOutOfBoundsException: 5
100% wiil be executed
```

Assertions

- Statements in the program declaring a boolean expression regarding the current state of the computation
- When evaluated to true nothing happens
- When evaluated to false an AssertionError exception is thrown
- Can be disabled during runtime without program modification or recompilation
- Two forms
  - assert *condition*;
  - assert *condition*: *expression*;

Evaluation

- The types of exceptions makes more sense than in the case of C++
- The throws clause is better than that of C++ (The throw clause in C++ says little to the programmer)

- The finally clause is often useful
- The Java interpreter throws a variety of exceptions that can be handled by user programs

Introduction to Event Handling

- Event handling is a basic concept of graphical user interfaces.
- An *event* is created by an external action such as a user interaction through a GUI
- The *event handler* is a segment of code that is called in response to an event

Java Swing GUI Components

- Text box is an object of class JTextField
- Radio button is an object of class JRadioButton
- Applet’s display is a frame, a multilayered structure
- Content pane is one layer, where applets put output
- GUI components can be placed in a frame
- Layout manager objects are used to control the placement of components

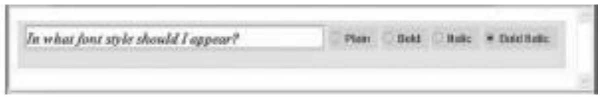
The Java Event Model

- User interactions with GUI components create events that can be caught by event handlers, called *event listeners*
- An event generator tells a listener of an event by sending a message
- An interface is used to make event-handling methods conform to a standard protocol
- A class that implements a listener must implement an interface for the listener
- One class of events is ItemEvent, which is associated with the event of clicking a checkbox, a radio button, or a list item
- The ItemListener interface prescribes a method, itemStateChanged, which is a handler for ItemEvent events
- The listener is created with addItemListener

```
/* RadioB.java
An example to illustrate event handling with interactive
radio buttons that control the font style of a textfield
*/
package radiob;
import  java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class RadioB  extends JPanel  implements
ItemListener {
private JTextField text;
private Font plainFont, boldFont, italicFont,
boldItalicFont;
private JRadioButton plain, bold, italic, boldItalic;
private ButtonGroup radioButtons;
// The constructor method is where the display is initially
// built
```

```
public RadioB() {
// Create the test text string and set its font
text = new JTextField(
"In what font style should I appear?", 25);
text.setFont(plainFont);
// Create radio buttons for the fonts and add them to
// a new button group
plain = new JRadioButton("Plain", true);
bold = new JRadioButton("Bold");
italic = new JRadioButton("Italic");
boldItalic = new JRadioButton("Bold Italic");
radioButtons = new ButtonGroup();
radioButtons.add(plain);
radioButtons.add(bold);
radioButtons.add(italic);
radioButtons.add(boldItalic);
// Create a panel and put the text and the radio
// buttons in it; then add the panel to the frame
JPanel radioPanel = new JPanel();
radioPanel.add(text);
radioPanel.add(plain);
radioPanel.add(bold);
radioPanel.add(italic);
radioPanel.add(boldItalic);
add(radioPanel, BorderLayout.LINE_START);
// Register the event handlers
plain.addItemListener(this);
bold.addItemListener(this);
italic.addItemListener(this);
boldItalic.addItemListener(this);
// Create the fonts
plainFont = new Font("Serif", Font.PLAIN, 16);
boldFont = new Font("Serif", Font.BOLD, 16);
italicFont = new Font("Serif", Font.ITALIC, 16);
boldItalicFont = new Font("Serif", Font.BOLD +
Font.ITALIC, 16);
} // End of the constructor for RadioB
// The event handler
public void itemStateChanged (ItemEvent e) {
// Determine which button is on and set the font
// accordingly
if (plain.isSelected())
text.setFont(plainFont);
else if (bold.isSelected())
text.setFont(boldFont);
else if (italic.isSelected())
text.setFont(italicFont);
else if (boldItalic.isSelected())
text.setFont(boldItalicFont);
} // End of itemStateChanged
// The main method
public static void main(String[] args) {
// Create the window frame
JFrame myFrame = new JFrame(" Radio button
example");
// Create the content pane and set it to the frame
JComponent myContentPane = new RadioB();
myContentPane.setOpaque(true);
myFrame.setContentPane(myContentPane);
// Display the window.
myFrame.pack();
myFrame.setVisible(true);
}
} // End of RadioB
```

Output oRadioB.java



<u>Functional and Logic Languages</u>:- Lambda Calculus, Overview of Scheme, Strictness and Lazy Evaluation, Streams and Monads, Higher-Order Functions, Logic Programming in Prolog, Limitations of Logic Programming.

Functional Languages

The imperative and functional models grew out of work undertaken by mathematicians Alan Turing, Alonzo Church, Stephen Kleene, Emil Post etc.
These individuals developed several very different formalizations of the notion of an algorithm, or effective procedure, based on automata, symbolic manipulation, recursive function definitions, and combinatorics.
Turing's model of computing was the Turing machine, an automaton reminiscent of a finite or pushdown automaton, but with the ability to access arbitrary cells of an unbounded storage "tape."
Church's model of computing is called the lambda calculus.
It is based on the notion of parameterized expressions ;with each parameter introduced by an occurrence of the letter λ—hence the notation's name.
Lambda calculus was the inspiration for functional programming: one uses it to compute by substituting parameters into expressions, just as one computes in a high level functional program by passing arguments to functions.

A constructive proof is the one that shows how to obtain a mathematical object with some desired property, and a nonconstructive proof is one that merely shows that such an object must exist, perhaps by contradiction, or counting arguments, or reduction to some other theorem whose proof is nonconstructive.

The logic programmer writes a set of axioms that allow the computer to discover a constructive proof for each particular set of inputs.

Functional Programming Concepts
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.
Features are:
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 in a language with side effects; 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, and are easily manipulated by operating on their first element and recursively the remainder of the list.
Python and Ruby provide aggregates capable of representing an unnamed functional value ;a
lambda expression, but few imperative languages are so expressive.
Functional languages tend to employ a garbage-collected heap for all dynamically allocated data.

Lambda Calculus
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.
The order in which these rules are applied captures the distinction between applicative and normal-order evaluation.
Conventions on the use of certain simple functions e.g., the identity function allow selection, structures, and even arithmetic to be captured as lambda expressions.

An Overview of Scheme

Scheme implementations employ an interpreter that runs a“read-eval-print” loop.
The interpreter repeatedly reads an expression from standard input –generally typed by the user

Functional Languages
The imperative and functional models grew out of work undertaken by mathematicians Alan Turing, Alonzo Church, Stephen Kleene, Emil Post etc.
These individuals developed several very different formalizations of the notion of an algorithm, or effective procedure, based on automata, symbolic manipulation, recursive function definitions, and combinatorics.
Turing’s model of computing was the Turing machine, an automaton reminiscent of a finite or pushdown automaton, but with the ability to access arbitrary cells of an unbounded storage “tape.”
Church’s model of computing is called the lambda calculus.
It is based on the notion of parameterized expressions ;with each parameter introduced by an occurrence of the letter λ—hence the notation’s name.
Lambda calculus was the inspiration for functional programming: one uses it to compute by substituting parameters into expressions, just as one computes in a high level functional program by passing arguments to functions.

A constructive proof is the one that shows how to obtain a mathematical object with some desired property, and a nonconstructive proof is one that merely shows that such an object must exist, perhaps by contradiction, or counting arguments, or reduction to some other theorem whose proof is nonconstructive.
The logic programmer writes a set of axioms that allow the computer to discover a constructive proof for each particular set of inputs.

Functional Programming Concepts
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.
Features are:
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 in a language with side effects; 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, and are easily manipulated by operating on their first element and recursively the remainder of the list.
Python and Ruby provide aggregates capable of representing an unnamed functional value ;a

lambda expression, but few imperative languages are so expressive.
Functional languages tend to employ a garbage-collected heap for all dynamically allocated data.
Lambda Calculus
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.
The order in which these rules are applied captures the distinction between applicative and normal-order evaluation.
Conventions on the use of certain simple functions e.g., the identity function allow selection, structures, and even arithmetic to be captured as lambda expressions.

An Overview of Scheme

Scheme implementations employ an interpreter that runs a“read-eval-print” loop.
The interpreter repeatedly reads an expression from standard input –generally typed by the user, evaluates that expression, and prints the resulting value.
If the user types (+ 3 4)
the interpreter will print 7
To save the programmer the need to type an entire program verbatim at the keyboard, most Scheme implementations provide a load function that reads and evaluates input from a file:
(load "my_Scheme_program")
Scheme (like all Lisp dialects) uses Cambridge Polish notation for expressions.
Parentheses indicate a function application.
Extra parentheses change the semantics of Lisp/Scheme programs. (+ 3 4) ⇒ 7
((+ 3 4)) ⇒ error
Here the ⇒ means “evaluates to.”
This symbol is not a part of the syntax of Scheme itself. _
One can prevent the Scheme interpreter from evaluating a parenthesized expression by quoting it: (quote (+ 3 4)) ⇒ (+ 3 4)
Though every expression has a type in Scheme, that type is generally not determined until run time.
The expression,
(if (> a 0) (+ 2 3) (+ 2 "foo"))
will evaluate to 5 if a is positive, but will produce a run-time type clash error if a is negative or zero.

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

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$_%:&=*!) =⇒ #t

To create a function in Scheme one evaluates a lambda expression:
(lambda (x) (* x x)) ⇒ function
Scheme differentiates between functions and so-called special forms -lambda among them, which resemble functions but have special evaluation rules.
The value of the last expression -most often there is only one, becomes the value returned by the function:
((lambda (x) (* x x)) 3) ⇒ 9 _
Simple conditional expressions can be written using if:
If expressions
(if (< 2 3) 4 5) ⇒ 4
(if #f 2 3) ⇒ 3

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.
Scheme provides a special form called define that has the side effect of creating a global binding for a name:
(define hypot (lambda (a b)
(sqrt (+ (* a a) (* b b))))) (hypot 3 4)   ⇒ 5

Lists and Numbers
The three most important are car, which returns the head of a list, cdr (“coulder”), which returns the rest of the list (everything after the head), and cons, which joins a head to the rest of a list: (car ’(2 3 4)) ⇒ 2
(cdr ’(2 3 4)) ⇒ (3 4)
(cons 2 ’(3 4)) = (2 3 4)
The notation ’(2 3 4) indicates a proper list, in which the final element is the empty list:
(cdr ’(2)) ⇒ ()
(cons 2 3) ⇒ (2 . 3) ; an improper list

EqualityTesting and Searching

eqv? Performs a shallow comparison, while equal? performs a deep (recursive) comparison, using eqv? at the leaves.
The functions memq, memv, and member take an element and a list as argument, and return the longest suffix of the list (if any) 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.
The functions assq, assv, and assoc search for values in association lists -otherwise known as A-lists.
An A-list is a dictionary implemented as a list of pairs.
Control Flow and Assignment
Cond resembles a more general if. . . elsif. . . else:

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

The arguments to cond are pairs.
They are considered in order from first to last.
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)
```

Delay and force permit the lazy evaluation of expressions.
Call-with-current-continuation (call/cc;) allows the current program counter and referencing environment to be saved in the form of a closure, and passed to a specified subroutine.
Programs as Lists
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.
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.
There is an unevaluated expression (lambda (x) (f (g x))).
When passed to eval, this list evaluates to the desired function.
(eval (list ’lambda ’(x) (list f (list g ’x)))

Eval and Apply

Lisp included a self-definition of the language:code for a Lisp interpreter. The code is based on the functions eval and apply.
Apply, takes two arguments: a function and a list.
It achieves the effect of calling the function, with the elements of the list as arguments.
For Primitive special forms, built into the language implementation- lambda, if, define, set!, quote, etc. eval provides a direct implementation.
Formalizing Self-Definition: Self-definition” is that for all expressions E, we get the same result by evaluating E under the interpreter I that we get by evaluating E directly.
Consider,
M(I) =M
Suppose let H(F) = F(I) where F can be any function that takes a Scheme expression as its argument.
Clearly
H(M) =M
Function M is said to be a fixed point of H.
H is well defined we can use it to obtain a rigorous definition of M.

Extended Example: DFA Simulation
Consider the simulation of the execution of a DFA (deterministic finite automaton).
We represent a DFA as a list of three items: the start state, the transition function, and a list of final states.
To simulate this machine, we pass it to the function simulate along with an input string.
As it runs, the automaton accumulates as a list a trace of the states through which it has traveled, ending with the symbol accept or reject.
For example, if we type
(simulate
zero-one-even-dfa ; machine description ’(0 1 1 0 1)) ; input string
then the Scheme interpreter will print (q0 q2 q3 q2 q0 q1 reject)
If we change the input string to 010010, the interpreter will print
(q0 q2 q3 q1 q3 q2 q0 accept)

Evaluation Order Revisited

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.
Consider, the following function:
(define double (lambda (x) (+ x x))) Under normal-order evaluation we would have
(double (* 3 4))
⇒ (+ (* 3 4) (* 3 4))
⇒ (+ 12 (* 3 4))
⇒ (+ 12 12)
⇒ 24
Normal order causes us to evaluate (* 3 4) twice.
Special forms and functions are known as expression types in Scheme.
Some expression types are primitive, in the sense that they must be built into the language implementation.
Others are derived; they can be defined in terms of primitive expression types.

In an eval/apply–based interpreter, primitive special forms are built into eval; primitive functions are recognized by apply.
Syntax-rules, can be used to create derived special forms.
These can then be bound to names with define-syntax and let-syntax.
Derived special forms are known as macros in Scheme, but unlike most other macros, they are
hygienic—lexically scoped, integrated into the language's semantics.
Strictness and Lazy Evaluation
A sideeffect-free 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 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 strict if it is defined in such a way that functions are always strict. ML and Scheme are strict.
Miranda and Haskell are nonstrict.
Lazy evaluation gives us the advantage of normal-order evaluation -not evaluating unneeded subexpressions while running within a constant factor of the speed of applicative-order evaluation for expressions in which everything is needed.
The trick is to tag every argument internally with a"memo"that indicates its value, if known. (double (* 3 4)) will be compiled as (double (f)), where f is a hidden closure with an internal side effect:
(define f (lambda ()
(let ((done #f)   ; memo initially unset (memo '())

(code (lambda () (* 3 4))))
(if done memo ; if memo is set, return it (begin
(set! memo (code)) ; remember value memo))))) ; and return it
Here (* 3 4) will be evaluated only once.
Lazy evaluation is used for all arguments in Miranda and Haskell. It is available in Scheme through explicit use of delay and force. Lazy evaluation is sometimes said to employ “call-by-need.”
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.
Scheme requires that every use of a delay-ed expression be enclosed in force, making it relatively easy to identify the places where side effects are an issue.
I/O: Streams and Monads
Side effects can be found in traditional I/O, including the builtin functions read and display of Scheme: read will generally return a different value every time it is called, and multiple calls to display.
Though they never return a value, they must occur in the proper order if the program is to be considered correct.
One way to avoid these 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.
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)

Lazy evaluation would force things to happen in the proper order. The car of output is the first prompt.
The cadr of output is the first square, a value that requires evaluation of the car of input. The caddr of output is the second prompt.
Recent versions of Haskell employ a more general concept known as monads.
Monads are drawn from a branch of mathematics known as category theory, but one doesn’t need to understand the theory to appreciate their usefulness in practice.
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 following code calls random twice to illustrate its interface;

```
twoRandomInts gen = let (rand1, gen2) = random gen (rand2, gen3) = random gen2
in ([rand1, rand2], gen3)
```

gen2, one of the return values from the first call to random, has been passed as an argument to the second call.
Then gen3, one of the return values from the second call, is returned to main, where it could, if we wished, be passed to another function.
This is particularly complicated for deeply nested functions.
Monads provide a more general solution to the problem of threading mutable state through a functional program.
We use Haskell’s standard IO monad, which includes a random number generator:

```
twoMoreRandomInts = do rand1 <- randomIO
rand2 <- randomIO return [rand1, rand2]
```

The type of the twoMoreRandomInts function has become IO [Integer].
This identifies it as an IO action—a function that -in addition to returning an explicit list of integers invisibly accepts and returns the state of the IO monad -including the standard RNG.
The type of randomIO is IO Integer.
The return operator in twoMoreRandomInts packages an explicit return value -in our case, a two-element list together with the hidden state, to be returned to the caller.
The IO monad, however, is abstract : only part of its state is defined in library header files; the rest is implemented by the language run-time system.
This is unavoidable, because, in effect, the hidden state of the IO monad encompasses the real world.
IO state hiding means that a value of type IO T is permanently tainted: it can never be extracted from the monad to produce a “pure T.”
Given putChar, we can define putStr:

```
putStr :: String -> IO ()
putStr s = sequence_ (map putChar s)
```

The result of map putChar s is a list of actions, each of which prints a character: it has type [IO ()]. The bulk of the program—both the computation of values and the determination of the order in which I/O actions should occur—is purely functional.
For a program whose I/O can be expressed in terms of streams, the top-level structure may consist of a single line:

```
main = interact my_program
```

The library function interact is of type (String -> String) -> IO ().
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.
Examples of higher-order functions: call/cc, for-each, compose and apply. Map takes as argument a function and a sequence of lists.
Map calls its function argument on corresponding sets of elements from the lists:
(map * ’(2 4 6) ’(3 5 7)) ⇒ (6 20 42)

Where for-each is executed for its side effects, and has an implementation dependent return value, map is purely functional: it returns a list composed of the values returned by its function argument.
We would 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.
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 for logician Haskell Curry, is to replace a multiargument 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.
We can write a general-purpose function that “curries” its (binary) function argument:
(define curry (lambda (f) (lambda (a) (lambda (b) (f a b))))) (((curry +) 3) 4) ⇒ 7
(define curried-plus (curry +))
Consider the following function in ML:

fun plus (a, b) : int = a + b;
==> val plus = fn : int * int -> int
The ML definition says that all functions take a single argument.
What we have declared is a function that takes a two-element tuple as argument. To call plus, we juxtapose its name and the tuple that is its argument:
plus (3, 4);
==> val it = 7 : int
Now consider the definition of a curried function in ML:
fun curried_plus a = fn b : int => a + b;
==> val curried_plus = fn : int -> int -> int
Note the type of curried_plus: int -> int -> int groups implicitly as int ->(int -> int).
ML’s syntax for function calls—juxtaposition of function and argument—makes the

use of a curried function more intuitive and convenient than it is in Scheme:
curried_fold plus 0 [1, 2, 3, 4, 5]; (* ML *)
(((curried_fold +) 0) ’(1 2 3 4 5)) ; Scheme

Logic Languages

Prolog and other logic languages are based on first-order predicate calculus
Logic Programming Concepts
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.
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 ← B1, B2, . . . , Bn
The semantics of this statement are that when the Bi are all true,we can deduce that H is true as well.
We say “H, if B1, B2, . . . , and Bn.”
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 ← A, B
D ← C
D ← A, B
In general, terms like A, B, C, and D may consist not only of constants -“Rochester is rainy”, but also of predicates applied to atoms or to variables:
rainy(Rochester), rainy(Seattle), rainy(X). _
During resolution, free variables may acquire values through unification with expressions in
matching terms, much as variables acquire types in ML
flowery(X) ← rainy(X) rainy(Rochester)
flowery(Rochester)
Prolog
A 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.

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’

A variable looks like an identifier beginning with an uppercase letter:

Foo My_var X

Variables can be instantiated to (i.e., can take on) arbitrary values at run time as a result of unification.

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))

We use the term“predicate” to refer to the combination of a functor and an “arity” - number of arguments.

The predicate rainy has arity 1.

The predicate teaches has arity 2. _

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.

It looks like a single term -the implication symbol is implicit:

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 are entered with a special ?- version of the implication symbol. If we were to type the following:

rainy(seattle). rainy(rochester).

?- rainy(C).

the Prolog interpreter would respond with

C = seattle

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 C1 and C2 are Horn clauses and the head of C1 matches one of the terms in the body of C2, then we can replace the term in C2 with the body of C1.

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.
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. Formal parameter of type int * ’b list, for example, will unify with an actual parameter of type ’a * real list in ML by instantiating ’a to int and ’b to real. _
Equality in Prolog is defined in terms of “unifiability.”
The goal =(A, B) succeeds if and only if A and B can be unified.

?- a = a.
Yes % constant unifies with itself
?- a = b.
No % but not with another constant
?- foo(a, b) = foo(a, b).
Yes % structures are recursively identical
It is possible for two variables to be unified without instantiating them. If we type
?- A = B.
the interpreter will simply respond A = B
Suppose we are given the following rules:
takes_lab(S) :- takes(S, C), has_lab(C).
has_lab(D) :- meets_in(D, R), is_lab(R).
S takes a lab class if S takes C and C is a lab class.
Moreover D is a lab class if D meets in room R and R is a lab. Lists
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.
[a, b, c] could be expressed as [a | [b,c]], [a, b | [c]], or [a, b, c | []].
The vertical-bar notation is particularly handy when the tail of the list is a variable:
member(X, [X | _]).
member(X, [_ | T]) :- member(X, T). sorted([]). % empty list is sorted sorted([A, B | T]) :- A =< B, sorted([B | T]).

% compound list is sorted if first two elements are in order and % remainder of list -after first element is sorted
Here =< is a built-in predicate that operates on numbers.
The underscore is a placeholder for a variable that is not needed anywhere else in the clause. Note that [a, b | c] is the improper list .(a, .(b, c)).
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]).
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. It will not unify with 5:
?- (2 + 3) = 5.
No
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.

Search/Execution Order
We 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.
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.
The process of returning to previous goals is known as backtracking. It strongly resembles the control flow of generators in Icon. rainy(seattle).
rainy(rochester). cold(rochester).
snowy(X) :- rainy(X), cold(X).

Fig: Backtracking search in Prolog- An OR level consists of alternative database clauses whose head will unify with the subgoal above; one of these must be satisfied.
The notation _C = _X is meant to indicate that while both C and X are uninstantiated, they have been associated with one another in such a way that if either receives a value in the future it will be shared by both.
The binding of X to seattle is broken when we backtrack to the rainy(X) subgoal. The effect is similar to the breaking of bindings between actual and formal parameters in an imperative programming language.
The interpreter pushes a frame onto its stack every time it begins to pursue a new subgoal G.
If G succeeds, control returns to the "caller" (the parent in the search tree), but G's frame remains on the stack.
Later subgoals will be given space above this dormant frame.
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.
path(X, Y) :- path(X, Z), edge(Z, Y). path(X, X).
The interpreter first unifies path(a, a) with the left-hand side of path(X, Y):- path(X, Z), edge(Z, Y).
It then considers the goals on the right-hand side, the first of which (path(X, Z)), unifies with the left-hand side of the very same rule, leading to an infinite regression.

Extended Example: Tic-Tac-Toe
Ordering also allows the Prolog programmer to indicate that certain resolutions are preferred, and should be considered before other, "fallback" options.
Consider, for example, the problem of making a move in tic-tac-toe. Tic-tac-toe is a game played on a 3 × 3 grid of squares.
Two players, X and O, take turns placing markers in empty squares. Let us number the squares from 1 to 9 in row-major order.

Let us use the Prolog fact x(n) to indicate that player X has placed a marker in square n, and o(m) to indicate that player O has placed a marker in square m.
Issue a query ?- move(A) that will cause the Prolog interpreter to choose a good square A for the computer to occupy next.
Clearly we need to be able to tell whether three given squares lie in a row. One way to express this is:
ordered_line(1, 2, 3). ordered_line(4, 5, 6).

ordered_line(7, 8, 9). ordered_line(1, 4, 7).
line(A, B, C) :- ordered_line(A, B, C).
line(A, B, C) :- ordered_line(A, C, B).

The following rules work well. move(A) :- good(A), empty(A). full(A) :- x(A).
full(A) :- o(A).
empty(A) :- \+(full(A)).
% strategy:
good(A) :- win(A). good(A) :- block_win(A).
good(A) :- split(A). good(A) :- strong_build(A). good(A) :- weak_build(A).
The initial rule indicates that we can satisfy the goal move(A) by choosing a good, empty square. The \+ is a built-in predicate that succeeds if its argument -a goal cannot be proven;
If none of these goals can be satisfied, our final, default choice is to pick an unoccupied square, giving priority to the center, the corners, and the sides in that order:
good(5).
good(1). good(3). good(7). good(9).
good(2). good(4). good(6). good(8).

Imperative Control Flow
Prolog provides the programmer with several explicit control-flow features. The most important of these features is known as the cut.
The cut is a zero-argument predicate written as an exclamation point: !.
As a subgoal it always succeeds, but with a crucial side effect: it commits the interpreter to whatever choices have been made since unifying the parent goal with the lefthand side of the current rule, including the choice of that unification itself.
Definition of list membership:
member(X, [X | _]).
member(X, [_ | T]) :- member(X, T).
If a given atom a appears in list L n times, then the goal ?- member(a, L) can succeed n times.
This can lead to wasted computation, particularly for long lists, when member is followed by a goal that may fail:
prime_candidate(X) :- member(X, candidates), prime(X).
Suppose that prime(X) is expensive to compute.
If prime(a) fails, Prolog will backtrack and attempt to satisfy member(a, candidates) again. We can save substantial time by cutting off all further searches for a after the first is found: member(X, [X | _]) :- !.
member(X, [_ | T]) :- member(X, T).
The cut on the right-hand side of the first rule says that if X is the head of L, we should not attempt to unify member(X, L) with the left-hand side of the second rule; the cut commits us to the first rule.

member(X, [X | _]).
member(X, [H | T]) :- X \= H, member(X, T).
Here X \= H means X and H will not unify; that is, \+(X = H).
It turns out that \+ is actually implemented by a combination of the cut and two other built-in predicates, call and fail:
\+(P) :- call(P), !, fail.
\+(P).
The call predicate takes a term as argument and attempts to satisfy it as a goal -terms are first-class values in Prolog.
The fail predicate always fails.
Explicit use of the cut may actually make a program easier to read. We can cut off consideration of the others by using the cut:
move(A) :- good(A), empty(A), !.
Definition of append:
append([], A, A).
append([H | T], A, [H | L]) :- append(T, A, L).
If we use write append(A, B, L), where L is instantiated but A and B are not, the interpreter will find an A and B for which the predicate is true.
If backtracking forces it to return, the interpreter will look for another A and B; append will
generate pairs on demand.
Idiom—an unbounded generator with a test-cut terminator— is known as generate-and-test. Like the iterative constructs of Scheme, it is generally used in conjunction with side effects.
The built-in predicates consult and reconsult can be used to read database clauses from a file, so they don't have to be typed into the interpreter by hand.
The predicate get attempts to unify its argument with the next printable character of input, skipping over ASCII characters with codes below 32.
repeat.
repeat :- repeat.
Within the above definition of get, backtracking will return to repeat as often as needed to produce a printable character.
Database Manipulation
Clauses in Prolog are simply collections of terms, connected by the built-in predicates :- and ,, both of which can be written in either infix or prefix form.
The structural nature of clauses and database contents implies that Prolog, like Scheme, is
homoiconic: it can represent itself. It can also modify itself.
A running Prolog program can add clauses to its database with the built-in predicate assert, or remove them with retract:
?- assert(rainy(syracuse)). Yes
?- rainy(X). X = seattle ;
X = rochester ; X = syracuse ; No
?- retract(rainy(rochester)). Yes

There is also a retractall predicate that removes all matching clauses from the database.
Individual terms in Prolog can be created, or their contents extracted, using the built-in predicates functor, arg, and =...
The goal functor(T, F, N) succeeds if and only if T is a term with functor F and arity N:

?- functor(foo(a, b, c), foo, 3). Yes
?- functor(foo(a, b, c), F, N). F = foo
N = 3
?- functor(T, foo, 3).
T = foo(_10, _37, _24)
The goal arg(N, T, A) succeeds if and only if its first two arguments (N and T) are instantiated, N is a natural number, T is a term, and A is the Nth argument of T:
?- arg(3, foo(a, b, c), A). A = c
Using =.. and call, the programmer can arrange to pursue (attempt to satisfy) a goal created at run-time:
param_loop(L, H, F) :- natural(I), I >= L, G =.. [F, I], call(G),
I = H, !.
The goal param_loop(5, 10, write) will produce the following output:
5678910
Yes
The only mechanism we have for perusing the database i.e., to determine its contents is the built-in search mechanism.
To allow programs to "reason" in more general ways, Prolog provides a clause predicate that attempts to match its two arguments against the head and body of some existing clause in the database:
?- clause(snowy(X), B).
B = rainy(X), cold(X) ; No
A clause with no body (a fact) matches the body true:
?- clause(rainy(rochester), true). Yes
Note that clause is quite different from call
Various other built-in predicates can also be used to "deconstruct" the contents of a clause.
The var predicate takes a single argument; it succeeds as a goal if and only if its argument is an uninstantiated variable.
The atom and integer predicates succeed as goals if and only if their arguments are atoms and integers, respectively.
The name predicate takes two arguments.
It succeeds as a goal if and only if its first argument is an atom. Parts of the logic not covered
Horn clauses do not capture all of first-order predicate calculus.
They cannot be used to express statements whose clausal form includes a disjunction with more than one non-negated term.

In Prolog we use the \+ predicate, but the semantics are not the same. Execution Order

While logic is inherently declarative, most logic languages explore the tree of possible resolutions in deterministic order.

Prolog provides a variety of predicates, including the cut, fail, and repeat, to control that execution order.

One must often consider execution order to ensure that a Prolog search will terminate. Even for searches that terminate, naive code can be very inefficient.

Negation and the “Closed World” Assumption

A collection of Horn clauses, such as the facts and rules of a Prolog database, constitutes a list of things assumed to be true.

It does not include any things assumed to be false.

This reliance on purely “positive” logic implies that Prolog’s \+ predicate is different from logical negation.

Unless the database is assumed to contain everything that is true -this is the closed world assumption), the goal \+(T) can succeed simply because our current knowledge is insufficient to prove T.

Negation in Prolog occurs outside any implicit existential quantifiers on the right-hand side of a rule. Thus ,

?- \+(takes(X, his201)).

Logical limitations of Prolog

Prolog can do many things. But it has four fundamental logical weaknesses:

Prolog doesn't allow "or"d (disjunctive) facts or conclusions--that is, statements that one of several things is true, but you don't know which.

For instance, if a light does not come on when we turn on its switch, we can conclude that either the bulb is burned out or the power is off or the light is disconnected. Prolog doesn't allow "not" (negative) facts or conclusions--that is, direct statements that something is false.

For instance, if a light does not come on when we turn on its switch, but another light in the same room comes on when we turn on its switch, we can conclude that it is false that there is a power failure.

Prolog doesn't allow most facts or conclusions having existential quantification--that is, statements that there exists some value of a variable, though we don't know what, such that a predicate expression containing it is true.