AN INTRODUCTION TO ADA (TRADEMARK) FOR SCIENTISTS AND ENGINEERS(U) ARMY MATERIEL SYSTEMS ANALYSIS ACTIVITY ABERDEEN PROVING GROUND MD  H E COHEN OCT 83
SHORT COURSE:
AN INTRODUCTION TO ADA® FOR
SCIENTISTS AND ENGINEERS
12-14 OCTOBER 1983

ORGANIZED BY: HERBERT E. COHEN

U.S. ARMY MATERIEL SYSTEMS ANALYSIS ACTIVITY
ABERDEEN PROVING GROUND, MARYLAND 21005
Title: An Introduction to Ada Programming for Engineers, Scientists and Programmers

Abstract: Provides an introduction to Ada programming for engineers, scientists, and programmers in the new standard higher order language of the Department of Defense.
SHORT COURSE (TEXT):

AN INTRODUCTION TO Ada®
FOR SCIENTISTS AND ENGINEERS

SPONSORED BY: Ada JOINT PROGRAM OFFICE
3D139 (400 A/N)
THE PENTAGON
WASHINGTON, DC 20301

AND

US ARMY MATERIEL SYSTEMS ANALYSIS ACTIVITY
ABERDEEN PROVING GROUND, MD 21005-5071

ORGANIZED BY: HERBERT E. COHEN
DARCOM MATHEMATICS PROGRAM OFFICE
US ARMY MATERIEL SYSTEMS ANALYSIS ACTIVITY
ABERDEEN PROVING GROUND, MD 21005-5071

©ADA IS A REGISTERED TRADEMARK OF THE US GOVERNMENT - Ada
JOINT PROGRAM OFFICE
ACKNOWLEDGEMENT

The US Army Materiel Systems Analysis Activity (AMSAA) wishes to acknowledge the support provided by the Data General Corporation in making available the MV-4000 computer and the ROLM-Ada compiler for this short course on Ada.

US Army Materiel Systems Analysis Activity would like to express its deep appreciation to LTC Vance Mall (AF) of the Ada Joint Program Office for his continued support throughout the development of this course.
LECTURER FOR

AN INTRODUCTION TO Ada®
FOR SCIENTISTS AND ENGINEERS

BILL CARLSON

CENTEC CORPORATION
MANAGEMENT SYSTEMS DIVISION
11260 ROGER BACON DRIVE
RESTON, VIRGINIA 22090-5281
LECTURER'S PREAMBLE

The Ada language brings together 30 years of computer science in a surprisingly well integrated and coherent package. There exist ways to use Ada which are relatively easy to learn and which give you all the power to conventional languages like FORTRAN. This course starts from that direction, so that you are writing Ada programs as quickly as possible. Then we start the process of exploring the broader capabilities of Ada. You will most fully appreciate Ada when you apply it to large programs.

Bill Carlson
October 1983
This course has been designed as a practical introduction to Ada programming and software design for practicing engineers, mathematicians, operations research analysts, statisticians and other professionals. The objective is to communicate the essence of Ada so that students leave confident that they can use Ada effectively. Specific concepts to be taught are the following:

- There is a way to do everything in Ada that you can do in FORTRAN.
- Ada satisfies the requirements which originally caused the DoD to develop a common language.
- Ada programs are structured as one or more packages.
- The Ada compiler can help you write correct programs if you tell it the "type" of each piece of data, and Ada provides a variety of tools for defining new data types, operations on those data types, and controlling the internal representation of data types.
- Generics are used when the same or similar operations are required for several different data types.
- Tasks and exception handling allow asynchronous events to be created and/or modeled.
- Separate compilation is essential for the construction of large systems, and is provided by Ada.

An important goal is to become comfortable with the mechanical aspects of writing an Ada program. This course explains how to write Ada statements, use Ada's control structures, do input/output, and use the Ada development system.

The development system is a Data General Eclipse with the Rolm Ada compiler. That compiler has been validated by the Ada Joint Program Office. The course will introduce the general concept of an Ada Program Support Environment (APSE) and distinguish between concepts which are unique to the particular Data General implementation and those which should be true of all Ada implementations.
ACQUIRING VIDEO TAPES

Title of Tape: "Ada® Programming Language"

1. DoD organizations can obtain free copies of tapes and text by writing to:

   Commander
   Tobyhanna Army Depot
   DAVA
   ATTN: DAVA-TLW
   Warehouse #3, Bay #3
   Tobyhanna, PA 18466

   Tapes will be in standard DoD 3/4 inch video cassette; however, 1/2 inch VHS and Beta formats are also available on request.

2. Non-DoD organizations and the general public can obtain tapes at minimal cost, in any of the formats specified above, by writing to:

   National Audio Visual Center
   GSA
   ATTN: Order Section
   Washington, DC 20409

3. For additional information, contact:

   Ada Joint Program Office
   3D139 (400 A/N)
   The Pentagon
   Washington, DC 20301
   (202) 694-0209

   or:

   Director
   US Army Materiel Systems Analysis Activity
   ATTN: DRXS-Y-MP (Herbert E. Cohen)
   Aberdeen Proving Ground, MD 21005-5071
   (301) 278-6577/6597
<table>
<thead>
<tr>
<th>TAPE NO.</th>
<th>TIME MIN</th>
<th>TITLE</th>
<th>PAGE OF TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>54:48</td>
<td>Introduction/Getting Started</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>35:51</td>
<td>Running A Program</td>
<td>33</td>
</tr>
<tr>
<td>3</td>
<td>12:13</td>
<td>Records, Arrays &amp; Enumeration Types</td>
<td>53</td>
</tr>
<tr>
<td>4</td>
<td>33:12</td>
<td>Flow of Control</td>
<td>71</td>
</tr>
<tr>
<td>5</td>
<td>31:20</td>
<td>Defining Program Components</td>
<td>105</td>
</tr>
<tr>
<td>6</td>
<td>54:47</td>
<td>Making Components More General</td>
<td>147</td>
</tr>
<tr>
<td>7</td>
<td>37:07</td>
<td>Design Guidelines</td>
<td>169</td>
</tr>
<tr>
<td>8</td>
<td>32:53</td>
<td>Scope &amp; Visibility</td>
<td>197</td>
</tr>
<tr>
<td>9</td>
<td>29:56</td>
<td>Record Abstraction</td>
<td>217</td>
</tr>
<tr>
<td>10</td>
<td>46:14</td>
<td>Numeric Abstraction</td>
<td>243</td>
</tr>
<tr>
<td>11</td>
<td>40:29</td>
<td>Review</td>
<td>281</td>
</tr>
<tr>
<td>12</td>
<td>23:47</td>
<td>Generics</td>
<td>287</td>
</tr>
<tr>
<td>13</td>
<td>13:04</td>
<td>Access Types</td>
<td>307</td>
</tr>
<tr>
<td>14</td>
<td>46:26</td>
<td>Task/Task Types</td>
<td>323</td>
</tr>
<tr>
<td>15</td>
<td>21:51</td>
<td>Machine Dependent Programs/</td>
<td>379</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Summary</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Questionnaire</td>
<td>407</td>
</tr>
</tbody>
</table>

Next page: intentionally left blank.
### READING ASSIGNMENTS

<table>
<thead>
<tr>
<th>TAPE</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>#2</td>
<td>Data General, &quot;Ada Work Center&quot;, pages 35-42 of text under Tape #2.</td>
</tr>
<tr>
<td>#5</td>
<td>Review Chapter 7 - &quot;Programming in Ada&quot; by Barnes. Chapters 3 &amp; 9 - &quot;Programming in Ada&quot; by Barnes.</td>
</tr>
<tr>
<td>#7</td>
<td>Review Chapters 7, 9, 11 - &quot;Programming in Ada&quot; by Barnes. Chapters 6, 7 - Military Standard, ANSI/MIL-STD-1815A. Read Section 16.5 - &quot;Programming in Ada&quot; by Barnes.</td>
</tr>
<tr>
<td>#8</td>
<td>Review Chapters 7, 6 - &quot;Programming in Ada&quot; by Barnes. Chapter 8 - Military Standard, ANSI/MIL-STD-1815A.</td>
</tr>
<tr>
<td>#10</td>
<td>Chapter 12 - &quot;Programming in Ada&quot; by Barnes. Military Standard, ANSI/MIL-STD-1815A, 3.5.5 to 3.5.10. 4.5.7, 4.6 and Annex A.</td>
</tr>
<tr>
<td>#11 &amp; #12</td>
<td>Chapters 13, 11.3 to 11.5 - &quot;Programming in Ada&quot; by Barnes. Chapter 12 and 3.8 - Military Standard, ANSI/MIL-STD-1815A.</td>
</tr>
<tr>
<td>#15</td>
<td>No reading assignment.</td>
</tr>
</tbody>
</table>
TAPE #1

INTRODUCTION/GETTING STARTED
DEFINING COMPONENTS
GENERALIZING COMPONENTS
DESIGN GUIDELINES
SCOPE AND VISIBILITY
RECORD ABSTRACTION
NUMERIC ABSTRACTION
DERIVED TYPES
pragma MAIN:
with TEXT_IO; use TEXT_IO; with SORT; use SORT;
procedure Roots is
  A, B, C: FLOAT;
  D: Float;
begin

  -- COLLECT PARAMETERS
  PUT ("to solve AX**2+BX+C=0");
  NEW_LINE;
  PUT ("A="); GET (A);
  PUT ("B="); GET (B);
  PUT ("C="); GET (C);
  NEW_LINE (2);
  -- COMPUTE DISCRIMINANT
pragma MAIN;
with TEXT_IO; use TEXT_IO;
with SORT;
procedure ROOTS is
    -- \( ax^2 + bx + c = 0 \)
    A, B, C: FLOAT;
    D: FLOAT; -- DISCRIMINANT
begin
EXAMPLE #2
WRITE A PROGRAM TO COMPUTE
THE ROOTS OF A QUADRATIC
EQUATION
pragma MAIN;
with TEXT_10; use TEXT_10;
procedure ADD is
    A,B,C: FLOAT;
begin
    GET (A); GET (B); GET (C);
    PUT ("SUM="); PUT (A+B+C);
end ADD;
pragma MAIN;
with TEXT_IO: use TEXT_IO;
procedure ADD is
   A,B,C: INTEGER;
begin
   GET(A); GET(B); GET(C);
   PUT ("SUM="); PUT (A+B+C);
end ADD;
GETTING STARTED
Ada® STANDARD
REFERENCE MANUAL pub. July 1980
MIL-STD 1815 Designated Dec. 1980
ANSI Canvass initiated Apr. 1980
ANSI Recanvass initiated Oct. 1980
ANSI Recanvass completed Sept. 1982
ANSI/MIL-STD 1815 Ada Jan 1983
Ada LANGUAGE SPECIFICATION

REQUEST FOR PROPOSAL (APR 77)
17 LANGUAGE PROPOSALS RECEIVED

PHASE 1 (AUG 77 - FEB 78)
SOFTECH INTERMETRICS
SRI HONEYWELL

PHASE 2 (APR 78 - APR 79)
INTERMETRICS HONEYWELL

PHASE 3 (MAY 79 - JULY 80)
HONEYWELL
Ada REQUIREMENTS

- STRONG TYPING
- ENCAPSULATED DEFINITION
- COMPOSITE TYPES
- GENERIC DEFINITIONS
- NUMERIC PRECISION
- PARALLEL PROCESSING
- EXCEPTION HANDLING
- DECLARATION OF MACHINE DEPENDENCE
ADA® REQUIREMENTS DEFINED IN A SERIES OF DRAFT SPECIFICATIONS

- STRAWMAN (1975)
- WOODMAN (1975)
- TINMAN (1976)
- IRONMAN (1977)
- STEELMAN (1978)
EMBEDDED COMPUTER SYSTEMS
SOFTWARE CHARACTERISTICS

○ LARGE
○ LONG LIVED
○ CONTINUOUS CHANGE
• EMBEDDED COMPUTER SYSTEMS APPLICATIONS CHARACTERISTICS
• REAL-TIME CONSTRAINTS
• AUTOMATIC ERROR RECOVERY
• CONCURRENT CONTROL
• NON-STANDARD INPUT-OUTPUT
THE MOTIVATION FOR

Ada
- GENERICS
- TASKING
- ACCESS TYPES
- TASK TYPES
- MACHINE DEPENDENT CODE
PUT ("C="); GET (c);
NEW_LINE (2);

--COMPUTE DISCRIMINANT
D:=B**2 - A.0*A*C;
--REAL ROOTS?
if D > = 0
   then
      PUT ("POSITIVE ROOT =");
      PUT ((-B + SQRT(D)) / (2.0*A))
      NEW_LINE;
      PUT ("NEGATIVE ROOT =");
      PUT ((-B - SQRT(D)) / (2.0*A))
      NEW_LINE;
   else
      PUT ("IMAGINARY ROOTS");
   end if;
end ROOTS;
if $ \Delta \geq 0$

    then
        PUT ("POSITIVE ROOT = ");
        PUT ($(-B + \text{SORT}(D))/(2.0A)$);
        NEW_LINE;
        PUT ("NEGATIVE ROOT = ");
        PUT ($(-B - \text{SORT}(D))/(2.0A)$);
        NEW_LINE;
    else
        PUT ("IMAGINARY ROOTS");

end if;
MAIN PROCEDURE TEMPLATE

\begin{verbatim}
pragma MAIN; --FOR ADE
with COMPONENTS; --LIBRARY
use COMPONENTS;
procedure NAME is
  \{DECLARATIVE_PART\}
begin
  \{SEQUENCE_OF_STATEMENTS\}
end NAME;
\end{verbatim}
### BUILT-IN TYPES

<table>
<thead>
<tr>
<th>Variables</th>
<th>Literals</th>
</tr>
</thead>
<tbody>
<tr>
<td>I: INTEGER;</td>
<td>2 or 3 or 789_123</td>
</tr>
<tr>
<td>A: FLOAT;</td>
<td>2.0 or 1.0E35</td>
</tr>
<tr>
<td>S: STRING (1..5);</td>
<td>&quot;HELLO&quot;</td>
</tr>
</tbody>
</table>
TYPE CONVERSION
FOR "CLOSELY RELATED" TYPES

- I := INTEGER (A); -- WILL ROUND
- A := FLOAT (I);
- STRING := "123_456" -- CHARACTER STRING
- I := INTEGER (STRING) -- ILLEGAL
ASSIGNMENT STATEMENT

VARIABLE := EXPRESSION;
EXPRESSIONS

- OPERATOR PRECEDENCE
  
  +   -

  * / mod rem

- LEFT TO RIGHT

- EXAMPLES

  B**2 - 4.0*A*C

  (-B+SQRT(D)) / 2.0*A
package TEXT_IO
GET (A); --ASSUMES FLOAT_IO (FLOAT)
PUT (A);
GET (I); --ASSUMES INTEGER_IO (INTEGER)
PUT (I);
GET (STRING); --VARIABLE LENGTH
PUT ("HELLO");
OVERLOADING
THE SAME IDENTIFIER HAS MORE
THAN ONE MEANING AT A GIVEN
POINT IN PROGRAM TEXT
pragma MAIN;
with TEXT_IO; use TEXT_IO;
procedure ADD is
    A,B,C : FLOAT;
begin
    GET (A); GET (B); GET (C);
    PUT ("SUM ="); PUT (A+B+C);
end ADD;
LEXICAL ELEMENTS
(CHAPTER 2)

- IDENTIFIER ::= letter
  {underline! letter_or_digit}
  -- USED AS NAMES AND RESERVED
- WORDS

- DELIMITERS ::= & ' () * + , - . / : ; = > |
  => . * := /= >= <= << >> <>

- SEPARATOR ::= SPACE | FORMAT_EFFECTOR | EOL
COMPOUND DELIMITERS

=>  ARROW
..  DOUBLE DOT
**  DOUBLE STAR
:_  BECOMES
/=  NOT EQUAL
>=  GT OR EQ
<=  LT OR EO
<<  LEFT LABEL BRACKET
>>  RIGHT LABEL BRACKET
<>  BOX
if STATEMENT

IF_STATEMENT ::= if CONDITION then
                      SEQUENCE_OF_STATEMENTS
                      ...
            [else
                      SEQUENCE_OF_STATEMENTS]
end if;
IF STATEMENT
(EXAMPLE)

if (COLD and SUNNY) or WARM
    and then STATE = "VA"
    and MONTH in WINTER
then
    . . .
end if;
MV FAMILY OVERVIEW

ECLIPSE MV Family of Systems incorporate an advanced 32-bit architecture with up to 16 Megabytes of physical main memory. Efficient demand paging techniques, cache structures, and instruction pipeline timing let the system make use of its 4 Gigabyte logical address space with maximum efficiency. Individual program user space can be as large as 2 Gigabytes. This gives the system the high capacity and performance needed to support multi-user Ada program development as well as real-time multiprogramming Ada applications.

ECLIPSE MV Family systems feature security mechanisms which complement the object orientation of Ada. The system's 4 Gigabyte virtual address space is divided into 6 processing segments of 512 Megabytes each for efficient memory management. These processing segments are identical to the hierarchical ring structure that is used to protect system resources.

These systems are designed for availability, reliability, and maintainability. The System Control Processor (SCP) performs self-diagnostics on internal subsystems, maintains a log of system errors, and identifies hardware faults to the field-replaceable unit level. It lets operators inspect memory and step through programs in order to monitor both hardware and software performance. In addition, the mechanical design of MV Family systems facilitates accessibility and efficient maintenance.

MV FAMILY HARDWARE OVERVIEW

<table>
<thead>
<tr>
<th></th>
<th>MV 4000</th>
<th>MV 8000</th>
<th>MV 10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum Address Space</td>
<td>4GB</td>
<td>4GB</td>
<td>4GB</td>
</tr>
<tr>
<td>Maximum User Program Size</td>
<td>2GB</td>
<td>2GB</td>
<td>2GB</td>
</tr>
<tr>
<td>Whetstone Performance</td>
<td>400</td>
<td>995</td>
<td>1900</td>
</tr>
<tr>
<td>On-Line Storage</td>
<td>4.7GB</td>
<td>9.6GB</td>
<td>18.5GB</td>
</tr>
<tr>
<td>Maximum Main Memory</td>
<td>12MB</td>
<td>16MB</td>
<td>16MB</td>
</tr>
<tr>
<td>System Cache</td>
<td>N/A</td>
<td>16KB</td>
<td>16KB</td>
</tr>
<tr>
<td>Instruction Cache</td>
<td>N/A</td>
<td>std</td>
<td>std</td>
</tr>
</tbody>
</table>
DEPLOYMENT STRATEGIES

ADE™ SOFTWARE

MV 10000
AOS RT32 ACS VS

MV 9000
AOS RT32 ACS VS

MV 4000
AOS RT32 ACS VS

MSE 800
AOS VS Ada
ARTS 32 Ada

DATA GENERAL MV FAMILY INFORMATION SYSTEMS

ROLM MIL SPEC COMPUTERS FOR HOSTILE ENVIRONMENTS
Procedures—Often a procedure is an operation performed on any type of data. For example, sorting an array. It is a general procedure in Ada, and it is likely to be used only once. However, if a specific procedure is needed, it is usually a single line of code that can be reused and modified for different targets.

User Interface—Command Line Interpreter (CLI)
The CLI is the primary interface between the programmer and the Ada Development Environment. The CLI controls terminal sessions, access to tools, and provides a user "help" facility.

Data Base Control Tools
The Ada Development Environment has four data control tools: the Data Base Manager (DBM), consisting of the Configuration Control Manager (CCM), the Mapper, and the Librarian. The DBM preprocesses data base primitives and allows definition of user primitives. It also provides services for creating, accessing, modifying, relating and deleting all ADE data base objects. The CCM provides control over the manipulation of ADE data base objects, including archiving and revision control services. The Mapper provides a means by which library objects can be specified and located. Finally, the Librarian is responsible for controlling the logical groupings of objects comprising Ada library units and subunits, as well as controlling access to those objects.

Application Development Tools
These tools include the Editor, Formatter, Pretty Printer, File Maintainer, and Debugger. The Editor is used by programmers to enter Ada source text, as well as other textual material. It is capable of Ada indenting and format control. The Formatter processes text files and re formats them into documentation files. The Pretty Printer is responsible for printing ADE programs in a logical Ada format. The File Maintainer allows comparisons of object programs: text files and typeless text files can each be compared. The Debugger provides a symbolic debugging facility to aid in the testing of Ada application programs.

Target Development Tools
Several of the tools are configured to support specific target machines. These tools include the Ada compilers themselves, Runtime Support Packages, Assemblers, Object Importers, Linkers, and Exporters. Ada compilers with unique code generators will be available for each of the target CPU's.

Unique Runtime Support Packages are supplied for each of the target environments. Each target also requires its own Assembler, which will be available as a cross-development tool. The Object Importer is used to bring into the Ada Development Environment binary modules produced by other language compilers such as FORTRAN 77. The Linker combines Ada binary with Libraries and Runtime Support Packages to create Ada Program Files. The Exporter tools are responsible for formatting and transferring Ada Program Files from the host environment to the target environments.

Libraries
The Ada Work Center includes Libraries of Packages that are useful to application programs. The STANDARD Package includes all of the basic language definitions, such as data types, allowable operations, and predefined exceptions. I/O Packages define fundamental data input/output capabilities such as sequential and direct I/O. A Math Package provides users with programs for computing commonly used math functions, such as sine and cosine.

SOFTWARE OPTIONS
The Ada Work Center supports non-Ada language program development concurrently with Ada program development. Any software available under the AOS/VS operating system is available on the Ada Work Center.
ADE™ SOFTWARE

A DE-Ada Development Environment features full DOD-Spec. ANSI Standard Ada Compiler and the most complete set of integrated Ada environment tools available.
Object Orientation—Ada introduced the concept of packages. A package consists of a specification part and an implementation part, which can be compiled separately. References to a package are only made to the specification part. This concept also supports information hiding, i.e., the forms of data structures need not be known to other packages and sub-routines which use that data.

Concurrency and Multitasking—In programming language Ada features.

Complex Data Structures—The compiler allows the user to create complex data structures. Arrays of arbitrary dimensions and dynamic size can be created. Strings are treated as arrays of characters. Records—collections of data of dissimilar types—are supported. Variant records, where the bounds varies as a function of values of data within individual records, can be treated. Data structures can be composed in arbitrarily complex ways and records and arrays within records.

Exception Handling—Ada has the ability to recover from execution time errors such as arithmetic faults, hardware faults, and array bounds errors. Further, the user may specify at what level in the program this error should be handled. Errors are handled where they logically should be handled, rather than allowing their effects to propagate destructively, or allowing the operating system to deal with them in an arbitrary way. This gives the user a great deal of control over execution conditions.
The Ada Work Center is a complete configuration that includes hardware, software, and support and allows users to become immediately productive in the development of applications in the Ada language.


The Ada Work Center also includes a comprehensive Ada programming environment to support development and maintenance activities. Based upon guidelines described in the DoD STONEMAN specification, taking full advantage of the proven power and versatility of the Data General AOS/VS virtual memory operating system, the Ada Development Environment will substantially increase programmer productivity and minimize costs associated with program development, testing, and maintenance.

ECLIPSE MV Family Systems provide ideal hosts for the Ada compiler and Ada Development Environment. Four packaged Ada Work Center systems are available. Two ECLIPSE MV 4000* based systems are excellent entry level systems for up to 8 concurrent Ada developers. For up to 8 users, an ECLIPSE MV 5000* based Ada Work Center is a cost effective choice. The ECLIPSE MV 10000* based Ada Work Center is the most powerful configuration and supports up to 16 simultaneous users. Each Ada Work Center comes complete with a system console, disk storage, a magnetic tape unit, and an optional line printer. The system may be expanded by the addition of more processors, memory, and peripherals. In addition, the following ECLIPSE hardware options are available:

- Fully integrated hardware and software Ada* Work Center
- Also available as full Ada software development system
- Extensively tested against the available DoD ACVC Validation Suite
- Compilers, code generators, and major environment tools written in Ada
- Hosted on powerful, 32-bit ECLIPSE* MV/Family Systems
- Produces executable Ada object programs which can be targeted at 32-bit ECLIPSE MV/Family and PCLM* Main Spec computer systems
- Host Target hardware and software J.O compatibility
- Ada Development Environment (ADE* software) using guidelines of the DoD STONEMAN Specification is built on AOS/VS Operating System
- AOS VS FORTRAN 77 object code can be integrated and supported in the Ada Development Environment
- Supports from 1-15 interactive Ada program development workstations.
READING ASSIGNMENT - TAPE #2

DATA GENERAL - ADA WORK CENTER

NEXT PAGE INTENTIONALLY LEFT BLANK.
TAPE #2

RUNNING A PROGRAM

NEXT PAGE INTENTIONALLY LEFT BLANK.
REVIEW

- ASSIGNMENT
- I/O - GET, PUT, NEW_LINE
- FLOW OF CONTROL - IF, LOOP
- DECLARATIONS - TYPE AND OBJECT
- EXPRESSIONS - ARITHMETIC AND LOGICAL
- TYPE CONVERSIONS
- MAIN PROCEDURE
ELABORATION OF STRING DECLARATIONS

S: STRING (1..5);

T: constant STRING := "HELLO";

-- STORAGE, AND HENCE SIZE,
-- FIXED WHEN DECLARATION
-- IS ELABORATED
ELABORATION

DECLARATIONS INVOLVE
RUN-TIME ACTIVITY :
1) STORAGE ALLOCATION -
   CREATE OBJECTS
2) INITIALIZE OBJECTS

EXAMPLE -
  I : INTEGER := 0;
for COLOR in COLOR_CHART
    loop
        --TRY COLOR
        exit when GOOD;
    end loop;
LOOP STATEMENT

LOOP_STATEMENT ::= [LOOP_SIMPLE_NAME:] [INTERATION_SCHEME] loop 
SEQUENCE_OF_STATEMENTS 
end loop [LOOP_SIMPLE_NAME];

INTERATION_SCHEME ::= while CONDITION |
for LOOP_PARAMETER_SPECIFICATION
LOGICAL EXPRESSIONS

- and | or | xor
  --ALLOW ARGUMENTS TO BE
  --EVALUATED IN EITHER ORDER
- and then | or else
  --LEFT ARGUMENT FIRST
- RELATIONAL OPERATORS PLUS in,
  not in
**MV FAMILY SOFTWARE OVERVIEW**

Data-Driven Information Systems

A Family System for A.W.C.

- A family system is a set of computer systems that are designed to work together to achieve a common goal.
- It is a modular approach to software development that allows for the easy integration of new systems with existing ones.

**LEOPEMENT STRATEGIES**

- A deployment strategy is a plan for how and when to implement a software system.
- It includes the selection of the appropriate system and the necessary training and support.

**ADO WORK CENTER PACKAGED CONFIGURATIONS**

<table>
<thead>
<tr>
<th>Model</th>
<th>A01</th>
<th>B01</th>
<th>C01</th>
<th>D01</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- A table showing the configurations of different models of the work center.
Ada

DEVELOPMENT ENVIRONMENT

<table>
<thead>
<tr>
<th>ADE</th>
<th>DEBUGGING</th>
<th>ADE</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOST</td>
<td>TOOLS</td>
<td>TARGET</td>
</tr>
</tbody>
</table>

LIBRARIAN   ISA
SOURCE EDITOR RUN-TIME
COMPILER    LOADER
LINKER      DEVICES

... ...
LOGGING ON

<CR>
USERNAME: bill<CR>
PASSWORD: x<CR>
AOS CLI
) aenter
=>
ADE FILE SYSTEM

<table>
<thead>
<tr>
<th>ADF</th>
</tr>
</thead>
<tbody>
<tr>
<td>YOUR_LIB</td>
</tr>
<tr>
<td>WORK</td>
</tr>
<tr>
<td>F1</td>
</tr>
<tr>
<td>F2</td>
</tr>
<tr>
<td>SYSTEM</td>
</tr>
<tr>
<td>TEXT_10</td>
</tr>
</tbody>
</table>

...
AEDIT

INSERT 2
MODIFY 4
DELETE 4 6
MOVE 3 5 BEFORE 7
DUP 3 5 BEFORE 7
SCREEN AHEAD Func 2
SCREEN BACK Func 1
BYE
FILE NAMES

OBJECT NAME::=

   IDENTIFIER (1..30) / \{QUALIFIERS\}

QUALIFIERS:

   CATEGORY    (ADA, BIN, PROG, LIST)
   VERSION     1.0, 2.0, ...
   TARGET      ADE, AOS_VS, ...
<table>
<thead>
<tr>
<th>Command</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALIB</td>
<td>ALIB: YOUR_LIB:WORK</td>
</tr>
<tr>
<td>ACREATE</td>
<td>FIRST/CAT=ADA</td>
</tr>
<tr>
<td>AEDIT</td>
<td>FIRST</td>
</tr>
<tr>
<td>ACOMP</td>
<td>FIRST/TAR=ADE</td>
</tr>
<tr>
<td>ATYPE</td>
<td>FIRST/CAT=LIST</td>
</tr>
<tr>
<td>ALINK</td>
<td>FIRST</td>
</tr>
<tr>
<td>AEXEC</td>
<td>FIRST</td>
</tr>
</tbody>
</table>
STATEMENTS DISCUSSED THUS FAR

- DECLARATIONS
- GET, PUT, NEW_LINE
- ARITHMETIC EXPRESSIONS
- ASSIGNMENT
- MAIN PROCEDURES
- IF...THEN...ELSE
- TYPE CONVERSION
LABORATORY #1
LABORATORY #1

--READ AN INTEGER (N)
--READ N INTEGERS
--ADD THEM
--PRINT RESULT
--MODIFY YOUR PROGRAM TO ADD
-- N REAL NUMBERS
--EXPERIMENT WITH GET AND PUT
--FOR OTHER TYPES
OBJECTIVES OF LAB #1

1) LEARN TO USE ADE
2) COMPILIE, LINK, AND EXECUTE AN ADA PROGRAM
3) EXPERIMENT WITH LITERALS
4) BUILD FAMILIARITY WITH TEXT_IO
   (SEE SECTION 14.3)
1) PROGRAM NAME
   SAME AS FILE NAME

2) GET AND PUT
   OPTION 1:
       with TTY_IO
       use TTY_IO
   OPTION 2:
       instantiate TEXT_IO
       for INTEGER
       $OR FLOAT
pragma MAIN
with TEXT_IO
procedure TEST 1 is
  package REAL_IO
    is new
      TEXT_IO. FLOAT_IO
        (FLOAT);
  use REAL_IO;
GENERICS

- ENUMERATION_10 is generic

  A new version must be instantiated for each enumeration type
ENUMERATION I/O

--SEE 14.3

with TEXT_IO; use TEXT_IO;
package DAY_IO is new
    ENUMERATION IO (DAY);
IN_DAY: DAY;
GET (IN_DAY);
PUT ("DAY=");
PUT (IN_DAY);
ENUMERATION TYPES

cdef DAY is
(MON,
TUES,
WED,
THURS,
FRI,
SAT,
SUN);
INDEX VALUES

- DISCRETE TYPE
  - INTEGER
  - ENUMERATION

- DISCRETE RANGE
  - CLOSED INTERVAL OF
  - VALUES OF A
  - DISCRETE TYPE
UNCONSTRAINED ARRAY

subtype POSITIVE is INTEGER
  range 1..INTEGER'LAST;

type STRING is array
  (POSITIVE range <>)
  of CHARACTER;

V:STRING (1..5);
MULTIDIMENTIONAL ARRAYS

RECTANGLE: array
( 1..20, 1..30 )
of FLOAT;

type SCHEDULE is
array ( WEEK, --1..52
       DAY, --MON..SUN
       HOUR) --1..24
of STRING;
ONE DIMENSIONAL ARRAYS

type VECTOR is
    array (1..10)
    of INTEGER;

type LINE is
    array (1..MAX_LINE_SIZE)
    of CHARACTER;

type SCHEDULE is array (DAY)
    of BOOLEAN;
RECORDS

type TIME is
record
    DAY : INTEGER
        range 0..366*(2099-1901+1);
    SECOND : DURATION --see 9.6
        range 0.0..86_400.0;--one day
end record;
TAPE #3

RECORDS, ARRAYS & ENUMERATION TYPES

NEXT PAGE INTENTIONALLY LEFT BLANK.
This page blank
TTY_IO
Is package in ADE Library
provides:
  INTEGER_IO
  CHARACTER_IO
  STRING_IO
to terminal
use TTY_IO
INSTANTIATE FLOAT_10

pragma MAIN
with TEXT_10
procedure TEST 1 is
package INT_10
  is new
TEXT_10. INTEGER_10
  for INTEGER
use INT_10
SUBTYPES

subtype WEEKDAY is
    DAY range MON..FRI;

subtype WEEK is
    INTEGER range 1..52;

subtype HOUR is
    INTEGER range 1..24;
SUBTYPE

- SUBSET OF VALUES OF BASE TYPE
- DETERMINED BY A CONSTRAINT
OVERLOADING

type COLOR is (WHITE,
      RED,
      YELLOW,
      GREEN);

type LIGHT is (RED
      AMBER,
      GREEN);

COLOR'GREEN /= LIGHT'GREEN
OVERLOADING

- THE SAME IDENTIFIER
  HAS MORE THAN ONE
  MEANING AT A GIVEN
  POINT IN PROGRAM TEXT
TYPE

- A SET OF VALUES
  and
- A SET OF OPERATIONS
  APPLICABLE TO
  THOSE VALUES
SUBTYPE
- SUBSET OF VALUES OF
- BASE TYPE
- DETERMINED BY A
- CONSTRAINT
OBJECT
  - OBJECTS CONTAIN VALUES
  - CREATED BY ELABORATING A DECLARATION (OR ...)
  - TYPE BOUND AT ELABORATION
TAPE #4

FLOW OF CONTROL

NEXT PAGE INTENTIONALLY LEFT BLANK.
FLOW OF CONTROL STATEMENTS

1. SEQUENTIAL
2. IF . . THEN . . ELSE
3. CASE . . IS . .
4. WHILE . .LOOP
5. FOR . . LOOP
6. EXIT . . WHEN . .
7. PROCEDURE CALL
8. FUNCTION INVOCATION
9. RETURN
OTHER STATEMENTS AFFECTING FLOW OF CONTROL

10. RAISE -- EXCEPTION
11. TASK INITIATION
12. ENTRY CALL --
   RENDEZVOUS FROM USER TASK
   LOOKS LIKE A PROCEDURE CALL
13. ACCEPT --
    RENDEZVOUS FROM SERVER TASK
14. ABORT
15.  -- DON'T USE
IF STATEMENT

if D >= 0
    then
        PUT ("POSITIVE ROOT =");
--ETC.
else
    PUT ("IMAGINARY ROOTS");
end if:

if STATEMENT

IF_STATEMENT ::= 
  if CONDITION then 
    SEQUENCE_OF_STATEMENTS 
    ... 
  [else 
    SEQUENCE_OF_STATEMENTS] 
  end if;
ELSIF
SYNTACTIC CONVENIENCE

if CONDITION
  then ... 
else if COND_2 then ... 
  else if COND_3 then ... 
    else 
      end if; --COND_3 
    end if; --COND_2 
  end if; 
end if;
CASE STATEMENT

case SENSOR is
    when ELEVATION =>
        RECORD_ELEVATION
            (SENSOR_VALUE);
    when AZIMUTH =>
        RECORD_AZIMUTH
            (SENSOR_VALUE);
    when others => null;
end case;
CASE STATEMENT

case DISCRETE_TYPE_EXPRESSION
is -- COVER VALUES ONCE
  when DISCRETE_RANGE =>
    ...
  when ANOTHER_DISCRETE =>
    ...
  when other =>
    ...
end case;
WHILE . . . LOOP
WHILE CONDITION EVALUATED
BEFORE EACH EXECUTION OF
THE SEQUENCE OF STATEMENTS

\texttt{while \ CONDITION \ loop}
\begin{itemize}
\item \texttt{SEQUENCE \ OF \ STATEMENTS}
\end{itemize}
\texttt{end \ loop;}

ONLY EXECUTES IF CONDITION TRUE
FOR . . . LOOP

    for I in DISCRETE_RANGE
        loop . . . end loop;

1. CREATES I
2. EVALUATES DISCRETE RANGE
3. IF DISCRETE RANGE NOT null
   EXECUTE, TREATING I AS CONSTANT
4. AFTER ALL VALUES, DESTROY
   I AND EXIT
LOOP STATEMENT

LOOP_STATEMENT ::= [LOOP_SIMPLE_NAME:] [ITERATION_SCHEME] loop
SEQUENCE_OF_STATEMENTS
end loop [LOOP_SIMPLE_NAME];

ITERATION_SCHEME ::= while CONDITION
| for LOOP_PARAMETER_SPEC
EXIT STATEMENT

OUTER_LOOP:
  for I in 1..10 loop
    for J
      in reverse 1..10 loop
        exit when J = I + 3;
        exit OUTER_LOOP
        when J = I;
      end loop;
  end loop OUTER_LOOP;
--I AND J NOT VISIBLE
EXIT STATEMENT

EXIT_STATEMENT ::= exit [LOOP_NAME]
[when CONDITION];
PROCEDURES AND FUNCTIONS

PROCEDURES
GET (A); PUT (A);
NEW_LINE;

FUNCTIONS
SORT (D); -- FROM USER LIBRARY
NAMED vs POSITIONAL PARAMETERS

procedure CREATE

(FILE: in out FILE_TYPE;
MODE: in FILE_MODE := default;
NAME: in STRING := " ";
FORM: in STRING := " ");
CREATE (WALDO); -- TEMP FILE
CREATE (JUDY, NAME => "JUDY");
OVERLOADING

PARAMETER_TYPE_PROFILE
   -- # PARAMETERS;
   -- BY POSITION, PARAMETERS
   -- HAVE SAME BASE TYPE
RESULT_TYPE_PROFILE
   -- SAME BASE TYPE
   -- NOTE: NOT NAMES, NOT MODES,
   -- NOT SUBTYPES, NOT DEFAULTS
OVERLOADING OPERATORS

- CAN OVERLOAD PREDEFINED OPERATOR SYMBOLS
- CANNOT OVERLOAD MEMBERSHIP TEST OR SHORT CIRCUIT CONTROL FORMS
- EXAMPLE:

```haskell
function "+" (LEFT, RIGHT: MATRIX)
  return MATRIX;
```
EXCEPTIONS

- Deal with errors or other exceptional situations
- Exception names associated with exceptions at compile time and stay same no matter how often declaration is elaborated
  -- e.g., recursive calls don't proliferate exceptions
RAISING EXCEPTIONS

- Draw attention to abnormal situation
- Abandon normal program execution
- Transfer control to user provided handler
- Or propagate
SUMMARY

FLOW OF CONTROL

1. SEQUENTIAL
2. IF . . THEN . . ELSE
3. CASE . . IS . .
4. WHILE . . LOOP
5. FOR . . LOOP
6. EXIT . . WHEN
7. PROCEDURE CALL
8. FUNCTION INVOCATION
9. RETURN
10. RAISE --EXCEPTION
SUMMARY OF EXCEPTIONS

- EXCEPTIONS ARE NOT OBJECTS, THEY ARE MERELY TAGS.
- PROPAGATED DYNAMICALLY
- THEY EXIST THROUGHOUT PROGRAM LIFE — CAN BE PROPAGATED OUT OF SCOPE AND THEN BACK IN AGAIN
- ONLY USE FOR ABNORMAL EVENTS
procedure AVERAGE is . . .

begin loop declare begin
GET (DATA);
if DATA = -1 then EXIT; end if;
SUM := SUM + DATA
COUNT := COUNT + 1;
exception when CONSTRAINT_ERROR
=> BAD := BAD + 1; end;
end loop; PUT ("AVERAGE =");
PUT (FLOAT(SUM)/FLOAT(COUNT));
end AVERAGE;
EXAMPLE

procedure AVERAGE is
  DATA: INTEGER range -1..99_999;
  SUM, COUNT, BAD: INTEGER:=0;
begin
  loop
    declare
      DATA
    begin ...
      exception
        COUNT BAD DATA
    end;
  end loop;
end;
<table>
<thead>
<tr>
<th>Where Raised</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subprogram body</td>
<td>Point of call</td>
</tr>
<tr>
<td>Block</td>
<td>After block</td>
</tr>
<tr>
<td>Package body</td>
<td>After body -</td>
</tr>
<tr>
<td></td>
<td>Within enclosing declarative part</td>
</tr>
<tr>
<td>Library unit</td>
<td>Abandon main</td>
</tr>
<tr>
<td>Task body</td>
<td>Task completed</td>
</tr>
</tbody>
</table>
SCOPE OF EXCEPTIONS

- EXCEPTIONS HAVE IDENTIFIERS
  A DEFINITION OF THE EXCEPTION IDENTIFIER MUST BE VISIBLE WHERE AN EXCEPTION IS RAISED AND WHERE A HANDLER IS DEFINED
- HANDLERS ARE INDEPENDENT OF EXCEPTION DECLARATIONS AND ARE OPTIONAL
EXAMPLE

begin
  --SEQUENCE_OF_STATEMENTS
exception
  when SINGULAR NUMERIC_ERROR =>
    PUT ("MATRIX IS SINGULAR");
  when others =>
    PUT("FATAL ERROR");
    raise; --PROPAGATE SAME
    EXCEPTION

end;
EXCEPTION HANDLER

EXCEPTION HANDLER ::= 
when EXCEPTION_CHOICE 
{EXCEPTION_CHOICE} 
⇒ SEQUENCE_OF_STATEMENTS

EXCEPTION_CHOICE ::= 
EXCEPTION_NAME | others
EXCEPTION HANDLERS

begin
    SEQUENCE_OF_STATEMENTS

exception
    EXCEPTION_HANDLER
    EXCEPTION_HANDLER

end
HANDLING EXCEPTIONS

declare
  N : INTEGER := 0;
begin
  N := N+J**A(K); --A&K GLOBAL
exception
  when others => PUT("AN ERROR");
end;
PREDEFINED EXCEPTIONS (CONT)

STORAGE-ERROR
- DYNAMIC TASK STORAGE EXCEEDED
- DURING ALLOCATION IF COLLECTION FULL
- INSUFFICIENT STORAGE TO ELABORATE A DECLARATION OR CALL A SUBPROGRAM
PREDEFINED EXCEPTIONS (CONT)

PROGRAM_ERROR

- CALL SUBPROGRAM BEFORE BODY
- ACTIVATE TASK IS ELABORATED
- ELABORATE GENERIC
- REACH END OF FUNCTION
- SELECTIVE WAIT WITHOUT OPEN BRANCHES
- ERRONEOUS ACTION
- INCORRECT ORDER DEPENDENCY
PREDEFINED EXCEPTIONS (CONT)

**NUMERIC_ERROR**
- EXECUTION OF PREDEFINED OPERATION CANNOT DELIVER CORRECT RESULT

**TASKING_ERROR**
- EXCEPTIONS DURING INTERTASK COMMUNICATIONS
PREDEFINED EXCEPTIONS

CONSTRAINT_ERROR

- VIOLATE RANGE CONSTRAINT
- VIOLATE INDEX CONSTRAINT
- VIOLATE DISCRIMINANT CONSTRAINT
- NON-EXISTENT RECORD COMPONENT
- NULL ACCESS VALUE
DEFINING PROGRAM COMPONENTS

TAPE #5

NEXT PAGE INTENTIONALLY LEFT BLANK.
COMPONENT
- A VALUE THAT IS PART OF A LARGER VALUE
- AN OBJECT THAT IS PART OF A LARGER OBJECT
FOLLOWING ARE PROGRAM COMPONENTS

- PROCEDURES
- FUNCTIONS
- PACKAGES
- TASKS
SUBPROGRAM DECLARATION

procedure IDENTIFIER [formal_part]
function DESIGNATOR [formal_part]
    return type_mark
formal_part ::= 
    (parameter_specification
    | ; parameter_specification)
parameter_specification ::= 
    identifier_list : mode
    type_mark [ := expression]
PROCEDURE EXAMPLES

procedure RIGHT _INDENT
(MARGIN: _out LINE_SIZE);

procedure SWITCH
(FROM, TO: in_out LINK);

procedure PRINT_HEADER
(PAGES: in NATURAL);
HEADER: in LINE
:= (1..LINE'LAST :='');
CENTER: in BOOLEAN := TRUE);
FUNCTION EXAMPLES

function RANDOM return PROBABILITY;
function MIN_CELL (X : LINK);
function DOT_PRODUCT
  (LEFT,RIGHT : VECTOR)
  return REAL;
function "*" (LEFT,RIGHT : MATRIX)
  return MATRIX;
PARAMETER PASSING MECHANISMS

<table>
<thead>
<tr>
<th>parameter</th>
<th>in</th>
<th>in out</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>scaler</td>
<td>copy</td>
<td>copy</td>
<td>copy</td>
</tr>
<tr>
<td>access</td>
<td>copy</td>
<td>copy</td>
<td>copy</td>
</tr>
<tr>
<td>array</td>
<td>either reference</td>
<td>or copy</td>
<td></td>
</tr>
<tr>
<td>record</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>task type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>private type</td>
<td>according to full type</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
PARAMETER PASSING MECHANISMS

-COPY-

procedure PUT (X: INTEGER) is . . .
--CALLED BY PUT (28)
PARAMETER PASSING MECHANISMS

-REFERENCE-

type TABLE is array(1..10) of INTEGER;

procedure OUT (X: TABLE) is . . .

-- CALLED BY

-- Y : TABLE := (others=>0);

-- OUT (Y);

![Diagram](CALLER\(\text{TABLE Y}\)\rightarrow\text{ACCESS}\rightarrow\text{OUT})
MATCHING ACTUALS & FORMALS

- ACTUALS AND FORMALS MUST BE OF THE SAME BASE TYPE.
- IF FORMAL IS CONSTRAINED
  --ACTUAL VALUE MUST OBEY
  --CONSTRAINT
  --(BUT NOT TYPE OF ACTUAL)
- IF ACTUAL out or in out
  --MORE CONSTRAINED THAN
  --FORMAL AND SCALER THEN
  --MUST OBEY AT COMPLETION
MATCHING ACTUALS & FORMALS

(CONT)

0. IF FORMAL IS UNCONSTRAINED ARRAY
   -- ACTUAL MUST BE CONSTRAINED
   -- DETERMINES BOUNDS
0. IF FORMAL IS RECORD OR PRIVATE
   -- WITH UNCONSTRAINED DISCRIMINANT
   -- THEN USE DISCRIMINANT OF
   -- ACTUAL INCLUDING UNCONSTRAINED
   -- IF ACTUAL IS UNCONSTRAINED
WARNINGS

- FOR ARRAYS AND RECORDS
  ASSIGNMENTS TO FORMAL MAY
  OR MAY NOT AFFECT ACTUAL IF
  SUBPROGRAM IS ABANDONED

- WHERE ACTUAL IS ACCESSIBLE
  BY MORE THAN ONE PATH
  (E.G. GLOBAL IDENTIFIER)
  VALUE IS UNDEFINED AFTER
  UPDATING BY ANY MECHANISM
  OTHER THAN ASSIGNING TO
  FORMAL AND RETURNING
SUBPROGRAM_DECLARATION ::= 
  SUBPROGRAM_SPECIFICATION;
SUBPROGRAM_SPECIFICATION ::= 
  procedure IDENTIFIER [FORMAL_PART]
  | function DESIGNATOR [FORMAL_PART]
  return TYPE_MARK
DESIGNATOR ::= IDENTIFIER
  | OPERATOR_SYMBOL
FORMAL_PART ::= 
  (PARAMETER_SPECIFICATION 
  {; PARAMETER_SPECIFICATION}) 
PARAMETER_SPECIFICATION ::= 
  IDENTIFIER_LIST : 
  MODE TYPE_MARK 
  { := EXPRESSION | 
  MODE ::= [IN] IN OUT | OUT
DISCRIMINANTS OF PRIVATE TYPES

\texttt{type TFXT (MAX\_LNG : INDEX) is limited private;}

\texttt{private type TEXT (MAX\_LNG : INDEX) is}
\texttt{record}
\texttt{  \hspace{1cm} POS : INDEX=0;}
\texttt{  \hspace{1cm} VALUE : STRING (1..MAX\_LNG);
  \hspace{1cm} end record;
end;
package KEY_MANAGER is
    type KEY is private;
    NULL_KEY : constant KEY;
    procedure GET_KEY
        (K : out KEY);
private
    type KEY is new NATURAL;
    NULL_KEY : constant := ∅;
end;
PRIVATE TYPES (CONT)

- PRIVATE PART OF INTERFACE
  -- AFFECTS SPEARATE COMPILATION

- LIMITED PRIVATE
  -- NO IMPLICIT ASSIGNMENT
  -- NO IMPLICIT EQUALITY
  -- NO IMPLICIT INEQUALITY

- DEFERRED CONSTANT
  -- VALUE IS IN PRIVATE PART
PRIVATE TYPES (CONT)

IMPLICITLY DEFINED OPERATIONS

- ASSIGNMENT
- MEMBERSHIP TESTS
- DISCRIMINANT SELECTION
- EXPLICIT CONVERSIONS
- T'BASE, T'SIZE, A'SIZE
  A'ADDRESS
- [A'CONSTRAINED --
  IF DISCRIMINANT]
- EQUALITY AND INEQUALITY
PRIVATE TYPES

- PACKAGE DEFINES A SET OF OPERATIONS
- PRIVATE TYPE DECLARATION CREATES A TYPE FOR OBJECTS TO WHICH THE OPERATIONS APPLY
- DETAILS OF THE PRIVATE TYPE HIDDEN FROM USER
PRIVATE TYPES

- HIDE DETAILS OF TYPE DEFINITION
- ONLY OPERATIONS DEFINED BY
  PACKAGE AND THE IMPLICITLY
  DECLARED OPERATIONS CAN
  AFFECT PRIVATE TYPES
package WORK_DATA is
    type DAY is (MON, TUES, WED, THU, FRI, SAT, SUN);
    type HOURS is delta 0.25 range 0.0 .. 24.0;
    type TIME_TABLE is
        array (DAY) of HOURS
    NORMAL : constant TIME_TABLE
        (MON..THR=> 8.25, FRI=> 7.0, OTHERS=> 0.0);
end WORK_DATA;
PACKAGE BODY

PACKAGE_BODY ::= 
   package_body SIMPLE_NAME is 
       [DECLARATIVE_PART]
       [begin
           SEQUENCE_OF_STATEMENTS
            [exception
                EXCEPTION_HANDLER
                [EXCEPTION_HANDLER]]]
       end [SIMPLE_NAME];
PACKAGE SPECIFICATION

PACKAGE_SPECIFICATION ::= 
  package IDENTIFIER is
  {BASIC_DECLARATIVE_ITEM?}
  [LIMITED] PRIVATE
  {BASIC_DECLARATIVE_ITEM?}
end [IDENTIFIER];
PACKAGE DECLARATION

PACKAGE_DECLARATION ::=  
  PACKAGE_SPECIFICATION;

-- PACKAGE_BODY

-- IS OPTIONAL
PACKAGES

- A COLLECTION OF RELATED TYPES, SUBPROGRAMS, AND OBJECTS
ACTUAL PARAMETERS
MODE
"IN"

function

VALUE
ACTUAL PARAMETERS

MODE "IN OUT"

ACTUAL PARAMETERS

FORMAL

procedure

ACTUAL PARAMETERS

MODE

"IN"

INTERNAL STATE

"OUT"

GLOBAL OBJECTS}
PARAMETER MODES

- **IN**: Formal parameter is a constant.
- **IN OUT**: Can both read and update actual.
- **OUT**: Can update actual. Can read bounds and discriminant of actual (only).
FOLLOWING ARE PROGRAM COMPONENTS

- PROCEDURES
- FUNCTIONS
- PACKAGES
- TASKS
LABORATORY #2

OBJECTIVES:
1. EXPERIMENT WITH Ada
   CONSTRUCTS COVERED ON
   FIRST DAY
2. BUILD EXAMPLE FOR
   USE IN LABS 3 - 5
ASSIGNMENT
SIMULATE A SIMPLE
HAND CALCULATOR
IN Ada
CALCULATOR SPECIFICATIONS

```plaintext
type OPERATIONS is
    (+, -, *, /, C);

REGISTER : FLOAT;
--GET AN OPERATOR
--GET A NUMBER
--OPERATOR (REGISTER, NUMBER)
--PUT REGISTER
```

138
IMPROVE CALCULATOR

1. ROBUST HANDLING OF BLANK CHARACTERS
2. ACCEPT NUMBERS WITHOUT DECIMAL POINT
3. EXCEPTION HANDLER FOR INVALID INPUT
TYPE LEGAL_OPS
IS ('+', '-', '*', '/', c)
package OPS_IO
is new
TEXT_IO, ENUMERATION_IO
(LEGAL_OPS);
THIS PAGE BLANK
TYPE

- A SET OF VALUES
  and
- A SET OF OPERATIONS
  APPLICABLE TO
  THOSE VALUES
MAKING COMPONENTS MORE GENERAL
SUBTYPE
- SUBSET OF VALUES OF BASE TYPE
- DETERMINED BY A CONSTRAINT
OBJECT

- OBJECTS CONTAIN VALUES
- CREATED BY ELABORATING A DECLARATION (OR ...)
- TYPE BOUND AT ELABORATION
DISCRIMINANTS

type SQUARE (SIDE : INTEGER) is
record
  MAT : MATRIX (1..SIDE,
    1..SIDE);
end record;
VARIANT RECORDS AND DISCRIMINANTS

subtype DRUM_UNIT
  is PERIPHERAL (DRUM);
subtype DISK_UNIT
  is PERIPHERAL (DISK);

WRITER : PERIPHERAL
  (UNIT => PRINTER);
ARCHIVE : DISK_UNIT;
VARIANT RECORDS (CONT)

type PERIPHERAL
  (UNIT : DEVICE := DISK) is
record STATUS : STATE;
  case UNIT is
    when PRINTER =>
      LINE COUNT : . . .
    when OTHERS =>
      CYLINDER : . . .
      TRACK : . . .
  end case;
end record;
VARIANT RECORDS

type DEVICE is
  (PRINTER, DISK, DRUM);

type STATE is
  (OPEN, CLOSED);
UNCONSTRAINED ARRAYS

function ROW_TOTAL (MATRIX)
returns VECTOR is
begin
if (MATRIX'LAST(2)-MATRIX'FIRST(2))
/=(VECTOR'LAST-VECTOR'FIRST)
then raise SOME_ERROR;
--etc
REVIEW: UNCONSTRAINED ARRAYS

WRITE SUBPROGRAMS THAT DETERMINE
SIZE OF FORMAL ARRAYS FROM ACTUAL

type MATRIX is array
  (INTEGER range <<>, INTEGER range<> of INTEGER);

type VECTOR is array
  (INTEGER range<> of INTEGER);
QUESTIONS ON EXAMPLE
1. WHY NO NEW LINE IN LOOP
2. WHAT IF DELETE exit
3. WHY NOT USE EXCEPTION AS TERMINAL CONDITION
TYPE ATTRIBUTES (CONT)

NEW_PAGE;

X := P'FIRST;

PRINT : while X <= P'LAST loop
  for I in 1..NO_COL loop
    SET_COL ((I-1)*HT+1);
    PUT (P'IMAGE (X));
    exit PRINT when X=P'LAST;
    X := P'SUCCE(X);
  end loop; end loop PRINT;
TYPE ATTRIBUTES (CONT)

P'WIDTH MAXIMUM LENGTH OF IMAGES OF TYPE P

function LINE_LENGTH return COUNT;

NO_COL : constant INTEGER := LINE_LENGTH/(P'WIDTH+5);

HT := P'WIDTH+5;
TYPE ATTRIBUTES (CONT)

P'IMAGE (X);

P IS DISCRETE SUBTYPE
X IS A VALUE OF TYPE P

function P'IMAGE (X) returns STRING;
RESULT IS THE PRINT IMAGE OF X
TYPE ATTRIBUTES

for J in BUFFER’RANGE loop
    if BUFFER (J) /= SPACE then
        PUT (BUFFER (J));
    end if;
end loop;
TOOLS FOR GENERALIZATION

- TYPE ATTRIBUTES
- VARIANT RECORDS
- UNCONSTRAINED ARRAYS
- UNCONSTRAINED DISCRIMINANTS
- PACKAGES ARE ABSTRACT TYPES
- GENERICS
OTHER STATEMENTS AFFECTING
FLOW OF CONTROL
10. RAISE — EXCEPTION
11. TASK INITIATION
12. ENTRY CALL —
RENDEZVOUS FROM USER TASK
LOOKS LIKE A PROCEDURE CALL
13. ACCEPT —
RENDEZVOUS FROM SERVER TASK
14. ABORT
15. [MARKED] — DON'T USE
FLOW OF CONTROL STATEMENTS
1. SEQUENTIAL
2. IF . . THEN . . ELSE
3. CASE . . IS . .
4. WHILE . . LOOP
5. FOR . . LOOP
6. EXIT . . WHEN . .
7. PROCEDURE CALL
8. FUNCTION INVOCATION
9. RETURN
DISCRIMINANTS

type TEXT (MAX_LNG : INDEX) is
  record
    POS : INDEX := 0;
    VALUE : STRING (1..MAX_LNG);
  end record;
UNCONSTRAINED DISCRIMINANTS

- Discriminants must always have a value.
- Normally, discriminant is a constant determined when the object is created.
- However, if default initial value given and object created by normal declaration, discriminant can be changed by whole record assignment.
STORAGE REQUIREMENT
KNOWN AT ALLOCATION

\texttt{type} \texttt{TEXT (MAX\_LNG:INDEX)} . . .
\texttt{LINE : TEXT(132);}
\texttt{type TEXT (MAX\_LNG: INDEX:=80)} . . .
\texttt{CARD : TEXT;}
\texttt{LINE : TEXT(132); . . .}
\texttt{CARD :=LINE; \texttt{-CHANGES MAX\_LNG}}
PACKAGES AS ABSTRACT TYPES

package RATIONAL_NUMBERS is

type RATIONAL is private

function EQUAL (X,Y:RATIONAL)
  return BOOLEAN;

function "+" (X,Y:RATIONAL)
  return RATIONAL;

function "-" (X,Y:RATIONAL)
  return RATIONAL;

-- etc
PACKAGES AS ABSTRACT TYPES

package RATIONAL_NUMBERS is
  -- etc (CONT.)

  procedure GET (ITEM:out RATIONAL;
                  WIDTH:in FIELD:=0);

  procedure PUT (ITEM:RATIONAL;
                 NUM_LNG:FIELD:=DEFAULT_NUM;
                 DENOM_LNG:FIELD:=DEFAULT_DENOM)
GENERICS

generic
  type P is ⟨⟩; --DISCRETE
procedure LIST;
procedure LIST is
  --CODE TO LIST ALL
  --VALUES OF P
end LIST;
POWERFUL TOOLS FOR
THE LIBRARY BUILDER

- ATTRIBUTES
- VARIANT RECORDS
- UNCONTRAINED FORMAL ARRAYS
- UNCONSTRAINED RECORD
  DISCRIMINANTS
- PACKAGES
- GENERICS

NEXT PAGE INTENTIONALLY
LEFT BLANK.
TAPE #7

DESIGN GUIDELINES
RULE #1
TRY TO STRUCTURE YOUR PROGRAM
AS A COLLECTION OF LIBRARY UNITS
RATIONALE:
1. LIBRARY UNITS ARE INDEPENDENT COMPONENTS
2. SIMPLIFIES SCOPE AND VISIBILITY
COMPILATION ORDER

STANDARD

A B C D E F

MAIN with with with with with with with D E C F TEXT_IO B F C

6 5 3 4 2 1
RULE #2

PUT THE BODIES OF LIBRARY UNITS IN SECONDARY UNITS

RATIONALE:
CHANGE IMPLEMENTATION OF A LIBRARY UNIT WITHOUT RECOMPILING UNITS USING IT
CREATING SECONDARY UNITS

LIBRARY UNITS:
  SUBPROGRAM DECLARATION/BODY
  PACKAGE DECLARATION
  GENERIC DECLARATION/INSTANTIATION
CREATING SECONDARY UNITS (CONT)

procedure EXAMPLE(X: in INTEGER,
    Y: out STRING)
    is separate;
function SAMPLE(X: INTEGER)
    return STRING
    is separate;
CREATING SECONDARY UNITS (CONT)

```vba
package PROBLEM is
  {DECLARATIONS}
  [private
   {OTHER DECLARATIONS}
  end

package body PROBLEM is
  -- etc

  SAME
  SOURCE
  FILE
  SAME

  COMPILATION

?```
RULE #3

CREATE A SYSTEM OF
MEANINGFUL TYPES
FOR YOUR APPLICATION

RATIONALE:
1. REDUCES CODING ERRORS
2. SIMPLIFIES MAINTENANCE
3. IMPROVES EXECUTION EFFICIENCY
CREATING MEANINGFUL TYPES:
DEFINING YOUR TYPES

package MY_TYPES is
  type STOP_LIGHT is
    (READ, YELLOW, GREEN);
  type DAY is
    (MON, TUE, WED, THU,
     FRI, SAT, SUN);
end MY_TYPES;
USING YOUR TYPES:

MAKING package MY_TYPES VISIBLE

with MY_TYPES;

procedure C is
    SIGNAL: MY_TYPES.STOPLIGHT;

begin
    --etc.

end C;
USING YOUR TYPES:

DIRECT VISIBILITY FOR INTERFACE OF

MY_TYPES

with MY_TYPES;
use MY_TYPES;
procedure C is
  SIGNAL : STOPLIGHT;
--etc.
USING YOUR TYPES:
OPERATIONS AVAILABLE ON USER DEFINED TYPES

1. EXPLICITLY DECLARED SUBPROGRAMS HAVING A PARAMETER OR RESULT OF THE TYPE
2. BASIC OPERATIONS
3. PREDEFINED OPERATORS
4. ENUMERATION LITERALS
5. ATTRIBUTES

*No change required*
USING YOUR TYPES:

IMPLICITLY DECLARED OPERATIONS

BASIC OPERATIONS INCLUDE:

1. ASSIGNMENT AND INITIALIZATION
2. MEMBERSHIP TEXTS
3. QUALIFICATION - EG DAY'(MON)
4. EXPLICIT TYPE CONVERSION
5. IMPLICIT TYPE CONVERSION FOR
   UNIVERSAL_REAL AND
   UNIVERSAL_INTEGER
USING YOUR TYPES:

EXPLICITLY DECLARED OPERATIONS

package RATIONAL_NUMBERS is

  type RATIONAL is ..

  function EQUAL (X,Y: RATIONAL)...

  function "/" (X,Y: INTEGER)...

  function "+" (X,Y: RATIONAL)...

  -- etc
DERIVED TYPES
ARE A CONVENIENT WAY TO
CREATE DISTINCT TYPES
EG:

```plaintext
type MEASURES is new RATIONAL;
derives subprograms declared
with the parent type
```

183
PITFALLS
RECOMPILING PACKAGE SPEC FORCES RECOMPILATION OF ALL USING PROGRAMS
--TO ADD MORE RATIONAL OPERATIONS
_with RATIONAL_NUMBERS;
_use RATIONAL_NUMBERS;
package MORE_RATIONAL_OPS is
  function "*" . . .
OUTSIDE THE LANGUAGE

- DIRECTORY SYSTEM FOR ORGANIZING
  LIBRARY UNITS
  (HIERARCHY OR RELATIONAL)

- LIBRARY UNIT MANAGEMENT
  SOURCE, OBJECT, EXECUTABLE
  MODULE
  NEW AND DERIVED VERSIONS
  WHICH PROGRAMS USE A UNIT
ASSUMPTIONS ABOUT COMPILER

- UNUSED SUBPROGRAMS AND OBJECTS ELIMINATED
- NO SPACE FOR ENUMERATION LITERALS UNLESS DISPLAYED
- CONSTRAINTS CHECKED AT COMPILE TIME IF POSSIBLE
RULE #4

USE A PACKAGE TO ACHIEVE
EFFECT OF FORTRAN COMMON
(GLOBAL OBJECTS)

package COMMON is
  A,B,C: INTEGER;
end COMMON;
PACKAGE DECLARATIVE REGION

package WALDO is
  X,Y: ...
end WALDO;

package body WALDO is
  X,Y: ...
begin
  SEQUENCE_OF_STATEMENTS
end;

EXTEND TO SCOPE OF
ENCLOSING DECLARATION
X,Y
EXIST
THROUGHOUT
WALDO
<table>
<thead>
<tr>
<th>DECLARATIVE REGIONS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBPROGRAM</td>
<td>ENTRY &amp; ACCEPT</td>
</tr>
<tr>
<td>PACKAGE</td>
<td>RECORD</td>
</tr>
<tr>
<td>TASK</td>
<td>RENAMING</td>
</tr>
<tr>
<td>GENERIC</td>
<td>BLOCK OR LOOP</td>
</tr>
</tbody>
</table>
OVERLOADING

OVERLOADING:
USING THE SAME IDENTIFIER FOR
TWO OR MORE (HOPEFULLY RELATED)
MEANINGS, WHERE THE MEANING
IN A PARTICULAR SITUATION IS
DETERMINED FROM THE CONTEXT
EXAMPLE:
"+"(LEFT, RIGHT : INTEGER)
    return INTEGER;
"+"(LEFT, RIGHT : FLOAT)
    return FLOAT;
SCOPE VS VISIBILITY

SCOPE OF A DECLARATION:
The lifetime of any entity declared by the declaration

VISIBILITY OF AN IDENTIFIER:
In combination with overloading, establishes the meaning of the occurrence of an identifier
TAPE #8

SCOPE & VISIBILITY
ASSIGNMENT

1. CREATE A PACKAGE THAT IMPLEMENTS THE CALCULATOR FUNCTIONS

2. CREATE A MAIN PROCEDURE THAT HANDLES TERMINAL I-O AND USES THE CALCULATOR PACKAGE

3. SEPARATELY COMPILe BODIES OF ALL FUNCTIONS AND PROCEDURES
OBJECTIVES:

1. EXPERIMENT WITH PROCEDURE AND FUNCTION DEFINITIONS
2. DEFINE A PACKAGE
3. EXPERIMENT WITH SEPARATE COMPILATION
LABORATORY #3
SUMMARY

1. MAKE SUBSYSTEMS LIBRARY UNITS
2. SEPARATELY COMPILE LIBRARY BODIES
3. USE MEANINGFUL TYPES
4. PUT GLOBALS IN PACKAGE(S)
5. COMMUNICATE VIA PARAMETERS
RESTRICTED VISIBILITY

STANDARD

MAIN

COMMON OBJECTS

A

B

\(\text{WITH COMMON A B}\)

\(\text{COMMON NOT VISIBLE}\)
RULE #5
PASS INFORMATION THROUGH PARAMETERS RATHER THAN USING GLOBAL OBJECTS

COMMENTS:
1. DON'T ASSUME COMMUNICATING THROUGH GLOBAL OBJECTS IS MORE EFFICIENT
2. RESTRICT VISIBILITY OF GLOBAL DATA
USING NESTED PACKAGES (CONT)

with COMMON;
use COMMON;

procedure R is
use BLOCK_1;
--etc
USING NESTED PACKAGES

with COMMON;
procedure R is
use COMMON.BLOCK_1;
--COULDN'T SAY IN CONTEXT
--CLAUSE BECAUSE NOT
--LIBRARY_UNIT_SIMPLE_NAME
NESTED PACKAGES

package COMMON is
  package BLOCK_1 is
    A, B, C : INTEGER;
  end BLOCK_1;
  package BLOCK_2 is
    X, Y, Z : FLOAT;
  end BLOCK_2
end COMMON;
WALDO : declare

A : ..... SCOPE

B : ..... SCOPE

begin ...

exception ...

end WALDO;


QUALIFIED NAMES

procedure P is
A, B : BOOLEAN;

procedure Q is
C : BOOLEAN;
B : BOOLEAN; -- HOMOGRAPH OF P.B

begin ...
B := A; -- Q.B := P.A;
C := P.B; -- Q.C := P.B;
NESTING HIERARCHY

PACKAGE STANDARD

BOOLEAN, CHARACTER
INTEGER, ...
SIMPLE_NAMES_LIBRARY_UNITS

exception--IMPLEMENTATION DEFINED
NESTING HIERARCHY

package STANDARD is
  BOOLEAN, CHARACTER, INTEGER, . . .
package CALENDAR is separate
package TEXT_IO is separate

MAIN  TEXT-IO  CALENDAR
COMPONENT LIBRARY
LIBRARY UNITS ARE IMPLICITLY DECLARED IN PACKAGE STANDARD

--TO MAKE A LIBRARY UNIT VISIBLE:
with LIBRARY_UNIT_SIMPLE_NAME;

--NOTE IMPLICATION THAT STANDARD HAS UNUSUAL BEHAVIOR
COMPONENT LIBRARY (CONT)

- Program library includes library units and secondary units.
- Secondary units are separately compiled sub-units of library units or of other secondary units.
- Library unit simple names must be unique.
package STANDARD is
  ...
  package LIB 1
  package LIB 2

package LIB 1 is
  package LIB 3

package LIB 2 is
  package LIB 3

LIB1.LIB3

LIB2.LIB3

THESE ARE DIFFERENT
USE CLAUSE

ACHIEVES DIRECT VISIBILITY OF DECLARATIONS IN VISIBLE PARTS OF NAMED PACKAGES

package D is
  T, U, V : BOOLEAN;
  VISIBLE
  procedure P . . .
  PART
  procedure O . . .
end D;
USE CLAUSE (CONT)

- USE CLAUSE AFFECTS VISIBILITY BUT NOT SCOPE!

- SPECIAL CASES GUARANTEE THAT USE CANNOT HIDE AN OTHERWISE DIRECTLY VISIBLE DECLARATION:
  --DIRECTLY VISIBLE HOMOGRAPHS HAVE PREFERENCE
USE AND RENAMES

0 USE CLAUSE SHOULD BE VIEWED
AS A CONVENIENT SHORTHAND

RENAMEING DECLARATION ::= 
  IDENTIFIER : TYPE MARK
    renames OBJECT_NAME;
  IDENTIFIER : exception
    renames EXCP_NAME;
  package IDENTIFIER
    renames PACK_NAME;
  SUBPROG_SPEC
    renames SUB_PROG_OR_ENTRY;
RENAMEING (CONT)

- RENAMING DOES NOT HIDE THE OLD NAME
- OBJECTS OF ANONYMOUS TYPE CANNOT BE RENAMED
- SUBTYPE CAN BE USED TO ACHIEVE EFFECT OF RENAMING TYPE:

```
subtype MODE is TEXT_TO_FILE_MODE;
```
OVERLOADING RULES

- Determine actual meaning of identifier when visibility rules show more than one meaning is acceptable.
- Err on conservative side - give error if not one clearly correct interpretation.
RESOLVING OVERLOADING CONFLICTS

- USE EXPANDED NAMES WITH DOT NOTATION, E.g.
  
  `MY_ARITH.SINE (X : FLOAT) . . .`

- USE QUALIFIED NAMES
  
  `REAL'SINE (X : FLOAT) . . .`
SUMMARY OF SCOPE AND VISIBILITY

- Declarations associate identifiers with entities
- Often, storage allocation is a side effect
- Identifiers can hide other identifiers
- Goal is to minimize both scope and visibility
  -- Scope restrictions save memory
  -- Visibility restrictions minimize errors
Ada RECORDS VS FORTRAN, COBOL, DMS RECORDS

- "RECORD" HISTORICALLY USED IN A PHYSICAL SENSE - A RECORD ON A STORAGE DEVICE
- Ada RECORDS ARE AN ABSTRACTION - A LOGICAL GROUPING OF COMPONENTS
MODELING REALITY

type HOUSEHOLD
    (NUMBER_MEMBERS:POSITIVE := 1)
    is record
    HOME : DWELLING ;
    MEMBERS : array (1 ..
        NUMBER_MEMBERS)
        of PERSON;
end record;
USE COMPOSITE TYPES TO MODEL REALITY

*type* PERSON is record...

subtype STUDENT is PERSON;

*type* CLASS is array (POSITIVE range<> of STUDENT;

type HOUSEHOLD (NUMBER_MEMBERS:POSITIVE := 1)

is record...
FIXED LENGTH VS VARIABLE LENGTH RECORDS

- RECORD TYPES WITH NO DISCRIMINANT OR A CONSTRAINED DISCRIMINANT DEFINE A SET OF IDENTICAL OBJECTS
- RECORD TYPES WITH UNCONSTRAINED DISCRIMINANT DEFINE LOGICALLY RELATED BUT (POSSIBLY) DIVERSE SET OF OBJECTS
- UNCONSTRAINED RECORDS CAN BE ARRAY COMPONENTS
FIXED LENGTH VS VARIABLE LENGTH ARRAYS

- NUMBER OF ARRAY COMPONENTS DETERMINED AT:
  - COMPILE TIME IF INDEX CONSTRAINT IS STATIC
  - TYPE DECLARATION ELABORATION IF INDEX CONSTRAINT IS DYNAMIC
  - OBJECT CREATION FOR UNCONSTRAINED ARRAYS

- ALL ARRAY COMPONENTS ARE SAME SUBTYPE
COMPOSITE TYPES

A TYPE WHOSE VALUES

HAVE COMPONENTS

- ARRAYS
- RECORDS
package X is
    procedure Y is
    begin
        declare
            declare
                procedure Z is
                declare
                    procedure Z is
package COMMON IS
    I, J, K: INTEGER;
end COMMON;

with COMMON;
use COMMON;
procedure MINE is
    I: INTEGER;
begin
    I:=I+1;
with TEXT_IO; TTY_IO;
use TEXT_IO; TTY_IO;
procedure P is
  package REAL_IO is
    new FLOAT_IO
      (FLOAT);
  I: INTEGER;
begin
  GET (I);
procedure X is
  I : INTEGER
for I in 1..10 loop
  exit when.......
end loop
  PUT (I);
package TEST IS

  X : INTEGER;
procedure y is

  X : INTEGER;
begin
  end
  end y;
Ada RECORDS VS
FORTRAN, COBAL, DMS RECORDS
TERMINOLOGY FROM THE PAST
- FIXED LENGTH RECORD
- VARIABLE LENGTH RECORD
- RECORD TYPE
- REPEATING GROUP
- MASTER AND DETAIL RECORDS
Ada RECORDS VS FORTRAN, COBOL, DMS RECORDS

1. EACH "OLD" RECORD IS A POSSIBLE IMPLEMENTATION OF AN Ada ABSTRACT TYPE
2. EACH "OLD" RECORD CAN BE DESCRIBED IN Ada
EXAMPLES:

VARIABLE LENGTH RECORD

type VAR (LENGTH :
    POSITIVE := 1)

is record
V : STRING (1..LENGTH)
end record;

add
EXAMPLES:

MULTIPLE RECORD TYPES

subtype RECORD_ID is
  INTEGER range 1..99;

type RECORDS (KIND : RECORD_ID)
  is record
  case KIND is
    when 1 => . . .
    when 2 => . . .
  end case; end record
EXAMPLES

subtype RECORD_ID is
  (FAMILY, HOUSE, HEAD_OF_HOUSE,
   SPOUSE, CHILD, JOB, ...)

type RECORDS (KIND:RECORD-ID)
  is record ...
INPUT - OUTPUT

FOR FILES CONTAINING ELEMENTS
OF A GIVEN TYPE, USE GENERIC

SEQUENTIAL_IO

DIRECT_IO
SPECIFICATION OF THE
package DIRECT_IO
with IO_EXCEPTIONS;
generic
  type ELEMENT_TYPE is private;
package DIRECT_IO is
  type FILE_TYPE is limited private;
  type FILE_MODE is
    (IN_FILE, INOUT_FILE, OUT_FILE);
  type COUNT is range 0..IMPLEMENTATION_DEFINED;
  subtype POSITIVE_COUNT is
    COUNT range 1..COUNT'LAST;
FILE MANAGEMENT

procedure CREATE
(FILE : in out FILE_TYPE;
MODE : in FILE_MODE := INOUT_FILE;
NAME : in STRING := " ";
FORM : in STRING := " ");

procedure OPEN
(FILE : in out FILE_TYPE;
MODE : in FILE_MODE;
NAME : in STRING;
FORM : in STRING := " ");
FILE MANAGEMENT (CONT)

procedure CLOSE
(FILE : in out FILE_TYPE);

procedure DELETE
(FILE : in out FILE_TYPE);

procedure RESET
(FILE : in out FILE_TYPE;
MODE : in FILE_MODE);

procedure RESET
(FILE : in out FILE_TYPE)
FILE MANAGEMENT (CONT)

function MODE
  (FILE : in FILE_TYPE)
  return FILE_MODE;

function NAME
  (FILE : in FILE_TYPE)
  return STRING;

function FORM
  (FILE : in FILE_TYPE)
  return STRING;

function IS_OPEN
  (FILE : in FILE_TYPE)
  return BOOLEAN;
INPUT AND OUTPUT OPERATIONS

procedure READ (FILE : in FILE_TYPE;
ITEM : out ELEMENT_TYPE;
FROM : POSITIVE_COUNT);

procedure READ (FILE : in FILE_TYPE;
ITEM : out ELEMENT_TYPE);

procedure WRITE (FILE : in FILE_TYPE;
ITEM : in ELEMENT_TYPE;
TO : POSITIVE_COUNT);

procedure WRITE (FILE : in FILE_TYPE;
ITEM : in ELEMENT_TYPE);
procedure SET_INDEX
  (FILE : in FILE_TYPE;
   TO : in POSITIVE_COUNT);
function INDEX(FILE : in FILE_TYPE)
  return POSITIVE_COUNT;
function SIZE (FILE : in FILE_TYPE)
  return COUNT;
function END_OF_FILE
  (FILE : in FILE_TYPE)
  return BOOLEAN;
EXCEPTIONS
STATUS ERRORS: exception renames
   IO_EXCEPTIONS.STATUS_ERROR;
MODE_ERROR : exception renames
   IO_EXCEPTIONS.MODE_ERROR
NAME ERROR: exception renames
   IO_EXCEPTIONS.NAME_ERROR;
USE_ERROR : exception renames
   IO_EXCEPTIONS.USE_ERROR;
DEVICE_ERROR : exception renames
   IO_EXCEPTIONS.DEVICE_ERROR;
EXCEPTIONS (CONT)

END_ERROR : exception renames
            IO_EXCEPTIONS.END_ERROR;
DATA_ERROR : exception renames
            IO_EXCEPTIONS.DATA_ERROR;
private
            -- implementation-dependent
end DIRECT_IO;
ATTRIBUTES (CONT)

\[ \text{\( P'\text{SMALL} = 2.0^{*\left(-P'\text{EMAX}-1\right)} \times 2.0^{-P'\text{EMAX}} \)} \]

\[ P'\text{MANTISSA} \]
\[ .1000...00 \quad \text{--} \quad 0.5 \]
\[ = 2.0^{-1} \times 2.0^{-P'\text{EMAX}} \]
\[ - 2.0^{*\left(-P'\text{EMAX}-1\right)} \]
ATTRIBUTES (CONT)

- $P'\text{LARGE} = 2.0^{\ast\ast}P'\text{EMAX}$
  
  --MAX REQ EXP
  
  $(1.0-2.0^{\ast\ast}(-P'\text{MANTISSA}))$

  $P'\text{MANTISSA} \quad P'\text{MANTISSA}$

  $1.0-0.000\ldots01 = .111\ldots11$

  $1.0-2.0 (-P'\text{MANTISSA})$
ATTRIBUTES (CONT)

- $\rho \varepsilon = 2.0^{(1 - \rho \text{ mantissa})}$
  
  \[
  (1.0 + \varepsilon) = \underbrace{.100 \ldots 1}_{2^1}
  \]

  \[
  1.0 = \underbrace{.100 \ldots 0}_{2^1}
  \]

  \[
  \varepsilon = \underbrace{.000 \ldots 1}_{2^1}
  \]

  \[
  = \frac{1}{2} \rho \text{ mantissa} \times 2^1
  \]

  \[
  = 2.0^{(1 - \rho \text{ mantissa})}
  \]
ATTRIBUTES

- **P'DIGITS**: # DECIMAL DIGITS IN MANTISSA
- **P'MANTISSA**: # BINARY DIGITS IN MANTISSA
- **P'EMAX**: 4*P'MANTISSA - REQ EXPONENT RANGE
- **P'EPSILON**: MODEL NUMBERS ARE 1.0 & 1.0 + EPSILON
- **P'LARGE**: LARGEST NUMBER OF SUBTYPE P
- **P'SMALL**: SMALLEST NUMBER OF SUBTYPE P
DEFINING DECIMAL REALS

type REAL is digits 7;

-- \( B = D \cdot \log_2 10 \times D \times 3.3219... \)

-- \( B = 23.2533... \rightarrow 24 \text{ bit MANTISSA} \)

-- \( 4 \times B = 93 \rightarrow 7 \text{ bit EXPONENT} \)
HOLE AROUND ZERO

\[-.1001*2^{-16}\]
\[-.1000*2^{-16}\]
\[0\]
\[.1000*2^{-16}\]
\[.1001*2^{-16}\]

\[\{\quad \triangle = 2^{-17}\}
\[\quad \triangle = 2^{-20}\]

BE CAREFUL SUBTRACTING TWO NEARLY EQUAL NUMBERS
FLOATING POINT

MODEL NUMBERS ARE

\[
\left[ \text{sign} \times \text{MANTISSA} \times 2^{\text{EXPONENT}} \right]
\]

\[
\frac{1}{2} \leq \text{MANTISSA} < 1
\]
INTEGRERS

\texttt{type MY\_INTEGER}
\texttt{is range-32\_768..+32\_767--16 BITS}
\texttt{MIGHT\_MEAN}
\texttt{type X is new INTEGER; --32 BITS}
\texttt{type MY\_INTEGER is X range}
\texttt{-32\_768..+32\_767;}
\texttt{MODEL INTEGERS EXACTLY}
\texttt{REPRESENTED IN HARDWARE}

247
THEORY AND IMPLEMENTATION

ABSTRACT NUMBERS

APPROX OR EXACT

PREDEFINED TYPES

MODEL NUMBERS

EXACTLY REPRESENT

RANGE CONSTRAINTS

SUBTYPES OF PREDEFINED NUMBERS

HARDWARE REPRESENTATION
NUMERIC TYPES

UNIVERSAL_INTEGER
  INTEGER -- REQUIRED
  LONG_INTEGER -- OPTIONAL
  SHORT_INTEGER -- OPTIONAL

UNIVERSAL_REAL
  FLOAT
  FIXED
SUMMARY

Ada RECORDS ARE VERY FLEXIBLE
USEFUL FOR MODELING REALITY
CAN BE USED FOR FILE I-O
ATTRIBUTES

X' CONSTRANGED
TRUE IF X IS CONSTANT
OR HAS CONSTRANGED
DISCRIMINANT, FALSE
OTHERWISE

ALSO: BASE, FIRST BIT, LAST BIT,
POSITION, SIZE
NAMING AND COMPONENT SELECTION

X.Y.Z

TO SELECT COMPONENT Z

FROM RECORD Y, WHICH

IS A COMPONENT OF X
ATTRIBUTES (CONT)

- `P'BASE'` etc -- EMAX, LARGE, SMALL
  -- FOR BASE TYPE
- `P'MACHINE` -- EMAX, EMIN
  -- MANTISSA
  -- MACHINE OVERFLOWS
  -- MACHINE RADIX
  -- MACHINE_rounds
- `P'SAFE` -- EMAX, LARGE, SMALL
ATTRIBUTES (CONT)

- P'SAFE_ --EMAX, LARGE, SMALL
- P'BASE'EMAX <= P'SAFE_EMAX
- P'BASE'SMALL >= P'SAFE_SMALL
- P'BASE'LARGE <= P'SAFE_LARGE
DERIVED TYPES

- DERIVED TYPE DEFINITION ::= 
  new SUBTYPE INDICATION
- SET OF VALUES FOR DERIVED TYPE
  IS COPY OF VALUES OF PARENT TYPE
- CORRESPONDING BASIC OPERATIONS,
  LITERALS, PREDEFINED OPERATORS,
  DERIVABLE SUBPROGRAMS
- EXPlicit TYPE CONVERSIONs
DERIVED TYPES VS SUBTYPES

- SUBTYPES RESTRICT THE SET OF VALUES;
  IMPLICIT CONVERSION AVAILABLE
- DERIVED TYPES ERECT A WALL
  SAME SET OF VALUES
  DIFFERENT ABSTRACT MEANING
  EXPLICIT CONVERSION REQUIRED
WHY DERIVED TYPES FOR NUMBERS

- USER SPECIFIES ABSTRACT VALUES
- COMPILER Chooses most appropriate model type
- Two types derived from same model type on one machine may use different models on another machine
- Conversions between hardware types may be expensive should be explicitly requested
MANY OBJECTS OF SAME TYPE

type MEASUREMENTS

is digits 3 --is FLOAT

range 0.0 .. MEASUREMENTS' LARGE

A, B : MEASUREMENTS;

C : MEASUREMENTS:
OBJECTS OF DIFFERENT TYPES

type LENGTH

is digits 5 --is FLOAT
range 0.0 .. 100.0

A : MEASUREMENTS;  { BOTH DERIVED FROM
B: LENGTH;  } FLOAT ON THIS
              } MACHINE
FIXED POINT TYPES

\[
\begin{array}{cccc}
L & O & R \\
\end{array}
\]

\[
\begin{array}{ccc}
\text{type } T \text{ is } \text{delta } D \text{ range } L..R; \\
\text{CANONICAL FORM ::=} \text{SIGN}\times\text{MANTISSA}\times\text{SMALL} \\
\text{MANTISSA IS POSITIVE INTEGER} \\
\text{MODEL NUMBERS CHOOSE SMALL AS} \text{SMALL} = 2^D \leq D \leq 2^{D+1} \\
\text{--OR AS DEFINED IN LENGTH CLAUSE}
\end{array}
\]

262
MODEL FIXED POINT NUMBERS

MODEL NUMBERS INCLUDE ZERO

\( \text{SMALL} = 2^I \leq D \leq 2^{I+1} \)

BOTH 'L AND 'R MUST BE WITHIN SMALL
OF A MODEL NUMBER

CHOOSE SMALLEST NUMBER OF BITS IN MANTISSA

\(- (B) \text{ SUCH THAT } -1 \times 2^{B-1} \times \text{SMALL} \leq L \)

\(2^{B-1} \times \text{SMALL} \geq R\)

AND (B) IS SMALLEST SUCH INTEGER
### FIXED POINT VS FLOATING POINT

<table>
<thead>
<tr>
<th>FIXED POINT</th>
<th>FLOATING</th>
</tr>
</thead>
<tbody>
<tr>
<td>W/DELTA=2</td>
<td>POINT</td>
</tr>
<tr>
<td>1 ( (*2^{-4}) )</td>
<td>.100.. ( *2^{-3} )</td>
</tr>
<tr>
<td>2 ( (*2^{-4}) )</td>
<td>.100.. ( *2^{-2} )</td>
</tr>
<tr>
<td>3 ( (*2^{-4}) )</td>
<td>.110.. ( *2^{-2} )</td>
</tr>
<tr>
<td>0 or 1</td>
<td>( { .1 } ) ( *2^{-4} )</td>
</tr>
<tr>
<td>0</td>
<td>( { .1 } ) ( *2^{-5} )</td>
</tr>
<tr>
<td>0</td>
<td>0.000</td>
</tr>
</tbody>
</table>
WORKING WITH VERY SMALL NUMBERS

FLOATING NUMBER NEAREST ZERO

$= P'\text{SAFE}_\text{SMALL}$

CHOOSE $\Delta = \left\{ 2^{-\left( P'\text{SAFE}_\text{EMAX} + 3 \right)} \right\}$

THEN $P'\text{SAFE}_\text{SMALL} = 50*\Delta$
WORKING WITH VERY LARGE NUMBERS

LARGEST FLOATING NUMBER = 
\[ p'\text{SAFE}\_\text{LARGE} \]

CHOOSE DELTA = \[ 2^{p'\text{SAFE}\_\text{EMAX}} - 2 \]

THEN \[ p'\text{SAFE}\_\text{LARGE} = 2^{\text{DELTA}} \]

\[ 10 \times p'\text{SAFE}\_\text{LARGE} = 20^{\text{DELTA}} \]
PREDEFINED OPERATIONS

- **ADDITION AND SUBTRACTION**
  Operands must be same type

- **MULTIPLICATION AND DIVISION**
  Operands of same type
  - \texttt{ANY\_FIXED\_PT} * \texttt{INTEGER}
  - \texttt{ANY\_FIXED} * \texttt{ANY\_FIXED} => \texttt{UNIVERSAL\_FIXED}

- **EXPONENTIATION**
  - Integer to integer power
  - Float to integer power
CONVERSION

- EXPLICIT TYPE CONVERSION DEFINED AMONG ALL NUMERIC TYPES
- LITERALS ARE EITHER
  - UNIVERSAL INTEGER OR
  - UNIVERSAL REAL AND IMPLICITLY CONVERTED TO TYPE OF
  - EXPRESSION WHERE USED
### Attributes of Fixed Point Types

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T'DELTA</td>
<td>Requested fixed accuracy</td>
</tr>
<tr>
<td>T'SMALL</td>
<td>Smallest positive model number</td>
</tr>
<tr>
<td>T'MANTISSA</td>
<td>Number binary digits in model numbers</td>
</tr>
<tr>
<td>T'LARGE</td>
<td>Largest model number</td>
</tr>
<tr>
<td>T'FORE</td>
<td>Representation before</td>
</tr>
<tr>
<td>T'AFT</td>
<td>AND after decimal point</td>
</tr>
</tbody>
</table>

269
### Attributes of Fixed Point Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T'safe Small</td>
<td>Smallest positive number of base type of T</td>
</tr>
<tr>
<td>T'safe Large</td>
<td>Largest number of base type</td>
</tr>
<tr>
<td>T'safe Small = T'base'Small</td>
<td></td>
</tr>
<tr>
<td>T'safe Large = T'base'LARGE</td>
<td></td>
</tr>
<tr>
<td>T'machine Rounds</td>
<td>Boolean</td>
</tr>
<tr>
<td>T'machine Overflows</td>
<td>True if always recognizes numeric error</td>
</tr>
</tbody>
</table>

270
WHY POWER OF TWO FOR DELTA

```pascal
type RULER is delta 2**(-4)
  range 0.0 .. 36.0;

type INCHES is new INTEGER
  range 0 .. 36;

FINE_MEASURE : RULER;
APPROX_MEASURE : INCHES;

FINE_MEASURE (APPROX_MEASURE)
  --LEFT SHIFT 4
APPROX_MEASURE (FINE_MEASURE)
  --ADD 8/16 AND RIGHT SHIFT 4
```
ALTERNATE VALUES FOR T'SMALL

```plaintext
type ENGLISH is delta 1.0/(12.0*16.0)
  range 0.0 .. MAX_INT/1024;
for ENGLISH'SMALL use 1.0/(12.0*16.0)
  delta 1.0/12.0;

type FEET is new INTEGER
--EVERY SIXTEENTH MODEL NUMBER
--OF ENGLISH IS A MODEL NUMBER
--OF INCHES
```

---

272
ALTERNATE VALUES FOR T'SMALL

```plaintext
type MONEY is delta 1.0/1000.0
  range 0.0 .. MAX_INT/1024;
for MONEY'SMALL use 1.0/1000.0;
subtype PENNIES is MONEY
  delta 1.0/100.0;
subtype DIMES is MONEY
  delta 1.0/10.0;
subtype QUARTERS is MONEY
  delta 1.0/4.0;
```
SUMMARY OF FIXED POINT

- FIXED POINT PROVIDES FOR RATIONAL ARITHMETIC ON MODEL NUMBERS
- COMMON DENOMINATOR IS $1/T'\text{BASE}'\text{SMALL}$
- ADDITION AND SUBTRACTION ARE EXACT AS LONG AS ACTUAL VALUES ARE MODEL NUMBERS
SUMMARY OF FIXED POINTS (CONT)

- DEFAULT T'BASE'SMALL IS POWER OF TWO, WHICH IS EQUIVALENT TO FLOATING POINT WITH CONSTANT EXPONENT
- MULTIPLICATION AND DIVISION PRODUCE UNIVERSAL_REAL, WHICH MUST BE CONVERTED
- CONVERSION CAN INTRODUCE ROUNDOFF ERRORS UNLESS SOURCE AND TARGET DERIVED FROM SAME BASE TYPE
SUMMARY

- NUMERIC TYPES ARE:
  
  INTEGER
  FLOAT   } REAL
  FIXED

- USER DEFINED TYPES ARE
  DERIVED FROM BUILT-IN
  (HARDWARE) NUMERIC TYPES
LABORATORY #4

OBJECTIVES:

1. IMPLEMENT PARENTHETICAL EXPRESSIONS IN YOUR HAND CALCULATOR USING A RECORD WITH DISCRIMINANT TO SAVE INTERMEDIATE RESULTS

2. ADJUST "IS DIGITS" UNTIL YOU CAN DEMONSTRATE ROUND-OFF ERROR
ASSIGNMENT

1. IMPLEMENT PARENTHETICAL
   EXPRESSIONS IN YOUR HAND
   CALCULATOR USING A RECORD
   WITH DISCRIMINANT TO SAVE
   INTERMEDIATE RESULTS

2. ADJUST "IS DIGITS" UNTIL
   YOU CAN DEMONSTRATE
   ROUND-OFF ERROR
TAPE #11

REVIEW

NEXT PAGE INTENTIONALLY LEFT BLANK.
STACK (CONT)

package body STACK is
  type TABLE is array
    (POSITIVE range<>) of ITEM;
  SPACE : TABLE(1..SIZE);
  INDEX : NATURAL := 0);
procedure PUSH(E: in ITEM) is
begin
GENERIC STACK

generic
    SIZE : POSITIVE;

    type ITEM is private;

package STACK is

    procedure PUSH (E: in ITEM);
    procedure POP (E: out ITEM);

    OVERFLOW, UNDERFLOW : exception

end STACK;
function SIGMA(A: VECTOR)
  return ITEM is
  TOTAL : ITEM := A(A'FIRST);
  --THE FORMAL type ITEM
  begin
    for N in A'FIRST+1..A'LAST loop
      TOTAL := SUM(TOTAL,A(N));
      --THE FORMAL function SUM
    end loop;
  return TOTAL;
end:,
PACKAGE ON_VECTORS (CONT)
PACKAGE ON VECTORS (CONT)

function SUM (A, B: VECTOR)
  return VECTOR is
  RESULT : VECTOR (A'LENGTH);
  -- THE FORMAL type VECTOR
  BIAS : constant
    INTEGER := B'FIRST - A'FIRST;
GENERIC INSTANTIATION

package INT_VECTOR is new
ON_VECTORS
(INTEGER,
TABLE,
"+");
package ON_VECTORS is
    function SUM (A,B: VECTOR)
        return VECTOR;
    function SIGMA (A:VECTOR)
        return ITEM;
    LENGTH_ERROR: exception;
end;
GENERIC PACKAGE EXAMPLE

generic
  type ITEM is private;
  type VECTOR is array
    (POSITIVE range <>)
    of ITEM;
  with function SUM (X,Y: ITEM)
  return ITEM;

package ONVECTOR is
GENERIC EXAMPLE

generic
  type ELEM is private;
procedure EXCHANGE
    (U,V: in out ELEM);
procedure EXCHANGE
    (U,V: in out ELEM) is
  T: ELEM;
begin T:=U; U:=V; V:=T;
end;
GENERICS

DEFINE A TEMPLATE FOR EITHER
SUBPROGRAMS OR PACKAGES

GENERIC FORMAL PARAMETERS CAN BE
- OBJECTS
- TYPES
- SUBPROGRAMS
TAPE #12

GENERICS
KEY TERMS (CONT)

- Abstract vs Physical Representation
- Numeric Exceptions
- Attributes of Numbers
- Binding - Binding Time
- Software Component
KEY TERMS (CONT)

- DISCRETE RANGE
- CONSTRAINED VS UNCONSTRAINED
- WHOLE RECORD ASSIGNMENT
- UNIVERSAL REAL
- UNIVERSAL INTEGER
KEY TERMS

- SYNTAX DIAGRAMS
- DERIVED TYPES VS SUBTYPES
- OVERLOADING VS HIDING
- USE VS WITH
- DISCRIMINANT
- TYPE MARK
STACK (CONT)

procedure PUSH(E: in ITEM) is
begin
  if INDEX >= SIZE then
    raise OVERFLOW;
  end if;
  INDEX := INDEX + 1;
  SPACE(INDEX) := E;
end PUSH;
STACK (CONT)

procedure POP(E: out ITEM) is
begin
  if INDEX = 0 then
    raise UNDERFLOW;
  end if;
  E := SPACE(INDEX);
  INDEX := INDEX - 1;
end POP;
end STACK
STACK (CONT)

package STACK_INT is new
STACK(SIZE => 200,
ITEM => INTEGER);
package STACK_BOOL is new
STACK(100, BOOLEAN);
STACK_INT.PUSH(N);
STACK_BOOL.PUSH(TRUE);
STACK (CONT)

generic
        type ITEM is private
package ON_STACKS is
        type STACK(SIZE : POSITIVE)
            is limited private;
        procedure PUSH(S : in out STACK;
                        E : in ITEM);
        procedure POP(S : in out STACK;
                        E : out ITEM);
OVERFLOW, UNDERFLOW : exception ;
STACK (CONT)

private

type TABLE is array
  (POSITIVE range<>) of ITEM;

type STACK(SIZE : POSITIVE) is
  record
    SPACE : TABLE(1..SIZE);
    INDEX : NATURAL := 0;
  end record;

end;
STACK (CONT)

declare
package STACK_REAL is new
  ON_STACKS(REAL);
  use STACK_REAL;
S : STACK(100);
begin
  ...  
  PUSH(S, 2.54);
  ...  
end;
GENERIC INSTANTIATION
INSTANTIATED IN SCOPE OF
DECLARATION, NOT WHERE
IT IS INSTANTIATED
SUMMARY OF GENERICS

- GENERICS ARE USED TO CREATE TEMPLATES FOR COMMON FUNCTIONS
- GENERICS WILL BE WIDELY USED IN Ada
- ALREADY, GENERICS USED FOR:
  - INTEGER_IO
  - FLOAT_IO
  - FIXED_IO
  - ENUMERATION_IO
ACCESS TYPES
USED WHEN BLOCK STRUCTURED
ALLOCATION OF OBJECTS AND
OBJECT NAMES IS TOO RESTRICTIVE
REVIEW OF BLOCK STRUCTURED OBJECTS:

- NAMES BOUND TO OBJECTS
- OBJECTS DESTROYED WHEN CONTROL LEAVES DECLARING UNIT
- NUMBER OF OBJECTS DETERMINED WHEN DECLARATIONS ARE ELABORATED
ACCESS TYPES

OBJECTS OF ACCESS TYPES

ACCESS TYPES

OBJECTS

ACCESS OBJECTS

311
SCOPE OF ACCESSED OBJECTS

SCOPE IS THAT OF
THE ACCESS TYPE

ACCESSED OBJECTS
FORM A "COLLECTION"

OBJECTS BECOME INACCESSIBLE
WHEN NO VARIABLES REFER
TO THEM DIRECTLY
OR INDIRECTLY
REFERRING TO ACCESSED OBJECTS

- P.NEXT  --POINTER FIELD OF
  --RECORD ACCESSED BY P
- P.I1    --COMPLETE RECORD
  --ACCESSED BY P
- Q:POINTER :=P;
  --P & Q POINT TO SAME RECORD
procedure SHOPPING is
  task MOTHER;
  task body MOTHER is
    begin BUY_MEAT; end MOTHER;
  task Boy;
  task body BOY is
    begin BUY_Beer; end BOY;
    begin BUY_Gas; end SHOPPING;
procedure SHOPPING is
  task GET_MEAT; --BY MOTHER
  task body GET_MEAT is
    begin BUY_MEAT; end GET_MEAT;
  task GET_BEER; --BY SON
  task body GET_BEER is
    begin BUY_BEER; end GET_BEER;
begin BUY_GAS; end SHOPPING;
task T is --SPECIFICATION
   ...
end T;
task body T is --BODY
   ...
end T;
TASKS

- Tasks are used to express independent activities.
- Tasks can (but may not) be executed in parallel with each other.
- The rendezvous is used to synchronize tasks and to exchange data between tasks.
TAPE #14

TASK/TASK TYPES

NEXT PAGE INTENTIONALLY LEFT BLANK.
SUMMARY OF ACCESS TYPES

- ACCESS TYPES PROVIDE VERY FLEXIBLE CAPABILITY TO CREATE AND DESTROY OBJECTS
- STRONG TYPING IS ENFORCED, SINCE EACH ACCESS TYPE IS TIED TO A SINGLE COLLECTION OF ACCESSED OBJECTS
- CANNOT BE DANGLING POINTERS SINCE INITIALIZED TO null AND ASSIGNMENTS TYPE CHECK (UNLESS ERRONEOUS UNCHECKED DEALLOCATION)
or accept RECEIVE (RESPONSE: out
    STRING(<>) do
    if LAST-null then
        RESPONSE:="b";
    else RESPONSE := LAST.TEXT;
    X := LAST; LAST := LAST.PRIOR;
    FREE(X); end if;
end select; end loop; end MAILBOX;
begin loop select
accept SEND(MESSAGE:STRING
(<>)) do
X := FIRST;
FIRST := new NODE'
(MESSAGE, null);
if LAST = null
then LAST := FIRST;
else X.PRIOR := FIRST
end if;
MAILBOX EXAMPLE (CONT)

X, FIRST, LAST: POINTER :
    --INITIALLY null
procedure FREE is new
    UNCHECKED_DEALLOCATION
    (NODE, POINTER);
MAILBOX EXAMPLE (CON'T)

type NODE (SIZE: INTEGER

range 1..132) is

record

TEXT : STRING (1..SIZE);
PRIOR : POINTER;

end record;
UNBOUNDED MAILBOX EXAMPLE

task MAILBOX is
  entry SEND (MESSAGE: STRING (<>));
  entry RECEIVE (RESPONSE:out
      STRING (<>));
end MAILBOX;

task body MAILBOX is
  type NODE;
  type POINTER is access NODE;
  --etc.
QUALIFIED_EXPRESSION ::= TYPE_MARK' (EXPRESSION) TYPE_MARK'AGGREGATE
INITIALIZING A NODE

P1 : POINTER := new NODE'
    (21, null);
-- ALLOCATOR new
-- TAKES QUALIFIED EXPRESSION
-- RETURNS ACCESS VALUE
-- THAT DESIGNATES OBJECT
LIST PROCESSING EXAMPLE

type NODE;

type POINTER is access NODE;

type NODE is
    record
        VALUE : INTEGER;
        NEXT : POINTER;
    end record

P : POINTER; --DEFAULT VALUE null
TASKS : JOBS TO BE DONE
GET_MEAT
GET_BEER
GET_GAS
PROCESSORS : RESOURCES FOR
PERFORMING TASKS
MOTHER
CHILDREN
DECLARATIONS ARE ELABORATED DURING PROGRAM EXECUTION
- INTRODUCE IDENTIFIERS
- ALLOCATE STORAGE
- INITIALIZE OBJECTS
OBJECTS DESTROYED AT END OF BLOCK WHERE DECLARED
DECLARATION,
ACTIVATION,
TERMINATION

- TASKS ARE COMPONENTS, DECLARED
  IN SUBPROGRAM, BLOCK,
  PACKAGE, TASK_BODY
- ACTIVATE WHEN PARENT HITS begin
- TERMINATE AT FINAL end
RENGEVOUS

ONE TASK CALLS ENTRY IN ANOTHER

```
task T is
  entry E (...);
  end;
-- etc
  T.E (...);
```
IMPLEMENTING ENTRIES

```vhdl
task T is
    entry E (...);
end;
task body T is
    entry E (...);
begin;
    accept E (...) do
        --SEQUENCE_OF_STATEMENTS
        end E;
end T;
```
ENTRY QUEUES

task BUFFER is
  entry PUT(X:in ITEM);
  entry GET(X:out ITEM);
end;
task BUFFER is
  entry PUT (X:in ITEM);
  entry GET (X:out ITEM);
end;

  task body BUFFER is
    V:ITEM;
    begin loop
      accept PUT (X:in ITEM) do
        V:=X; end PUT;
      accept GET (X:out ITEM) do
        X:=V; end GET;
    end loop;
  end BUFFER;
ENTRY QUEUES

PRODUCING Tasks

PUT

BUFFER

X

CONSUMING Tasks

GET

PUT

GET

PUT

GET
ENTRY QUEUES

- FIRST IN - FIRST OUT (FIFO)
- WITHIN TASK BODY, ATTRIBUTE
  \texttt{COUNT TELLS NUMBER OF WAITING TASKS}

337
ASYMMETRIES

- Caller must name called task.
- Entries are available to any task.
- Entry can have several queued tasks.
- Task can only be on one queue.
- Task names cannot appear in use clause, so need dot notation.
- Source text for a task can contain more than one accept for an entry; they all draw from same queue.
TIMING AND SCHEDULING

IF TWO TASKS WITH DIFFERENT PRIORITIES ARE ELIGIBLE FOR EXECUTION AND COULD SENSIBLY USE SAME RESOURCES, LOWER PRIORITY TASK CANNOT EXECUTE WHILE HIGHER PRIORITY TASK WAITS

#pragma PRIORITY (integer);
SCHEDULING

- IMPLEMENTATION SHOULD MAKE AN "ARBITRARY" CHOICE AMONG EQUAL PRIORITY TASKS
- IMPLEMENTATION CAN CHOOSE AN EFFICIENT (BUT FAIR) MECHANISM
- PROGRAMS THAT DEPEND ON SELECTION ALGORITHM ARE ERRONEOUS
PRIORITY

- The PRIORITY IN a PRIORITY must be a static expression that evaluates to an integer.
- Larger integers imply greater urgency.
- The range of subtype PRIORITY is implementation defined.
PRIORITY

- PRIORITY IS OPTIONAL
- PRIORITY OF RENDEZVOUS IS GREATER OF TWO PRIORITIES
- IF NO EXPLICIT PRIORITY, SCHEDULING RULES NOT DEFINED

- TASK PRIORITIES
  STATIC = CONSTANT
select
  CONTROLLER.REQUEST
  (MEDIUM) (SOME_ITEM);

or
  delay 45.0;
  -- give up
end select;
CONDITIONAL ENTRY CALL EXAMPLE

-- BUSY_WAIT

procedure SPIN (R: RESOURCE) is
begin
loop
select
R.SEIZE;
return;
else;
null; -- busy wait
end select;
end loop;
end;
USER SELECTS

○ CONDITIONAL ENTRY CALL
  CANCEL CALL IF RENDEZVOUS
  NOT IMMEDIATELY POSSIBLE

○ TIMED ENTRY CALL
  CANCEL CALL IF NO RENDEZVOUS
  BEFORE TIMER RUNS OUT

ONLY ON ONE ENTRY AT A TIME!!
MECHANICS OF SELECT

- GUARDS (WHEN CONDITION => )
  EVALUATED WHEN SELECT IS EXECUTED
- ABSENT GUARD IS TRUE
- ORDER OF EVALUATION OF GUARDS IS UNDEFINED
- SELECT ERROR IF ALL GUARDS ARE FALSE
select
  when COUNT < N =>
    accept PUT ... .
  or
  when COUNT > 0 =>
    accept GET (X : out ITEM) do
      x := A(J);
      end;
      J := J mod N+1;
    end select;
  COUNT := COUNT - 1;
task body BUFFER is
  N : constant := 8; --EXAMPLE
  A : array (1..N) of ITEM;
  I,J : INTEGER range 1..N := 1;
  COUNT : INTEGER range 0..N :=0;
begin loop
  select
    when COUNT < N =>
    accept PUT (X : in ITEM) do
      A(I) := X;
    end;
    I := I mod N+1;
    COUNT := COUNT+1;
  or
task body BUFFER is
N : constant := 8; --EXAMPLE
A : array (1..N) of ITEM;
I,J : INTEGER range 1..N := 1;
COUNT : INTEGER range 0..N := 0;
begin loop
BUFFER EXAMPLE

task BUFFER is
  entry PUT (X : in ITEM);
  entry GET (X : out ITEM);
end;
SERVER SELECTS

    select
        SELECT_ALTERNATIVE
    [or]
        SELECT_ALTERNATIVE
    [else]
        SEQUENCE_OF_STATEMENTS
    end select;

EXAMPLE (SELECT STATEMENT)

select
  accept DRIVER_AWAKE_SIGNAL;

{or
  delay 30.0 * SECONDS;
  STOP_THE_TRAN;
end select;
--SELECTIVE WAIT CAN HAVE MORE
--THAN ONE DELAY ALTERNATIVE
SELECT STATEMENT
- SELECT FIRST OF SEVERAL ENTRIES TO BE CALLED
- MAKE AVAILABILITY OF AN ENTRY CONDITIONAL
- CONTROL WAITING TIMES
SYNCHRONOUS EXECUTION

declare use CALENDAR;

    INTERVAL: constant DURATION:=1.0;

    NEXT_TIME: TIME:=FIRST_TIME;

begin loop

    delay NEXT_TIME - CLOCK ();

    ACTION;

    NEXT_TIME:=NEXT_TIME+INTERVAL;

end loop; end;
DELAY STATEMENT

delay DURATION;

type DURATION

-- IMPLEMENTATION DEFINED
-- FIXED POINT TYPE
-- IN SECONDS
-- AT LEAST ONE DAY (36,400)
delay 3.0; -- AT LEAST 3 SECONDS
SYNCHRONIZATION

- Use rendezvous for synchronization
- Use priorities to indicate relative urgency and fine tune responsiveness
TIMED ENTRY CALL

TIMED_ENTRY_CALL ::= 

\texttt{select} 

\texttt{ENTRY\_CALL\_STATEMENT} 

[\texttt{SEQUENCE\_OF\_STATEMENTS}] 

or 

\texttt{DELAY\_ALTERNATIVE} 

end select;
SUMMARY OF TASKING

TASKS VS PROCESSORS

TASK DECLARATIONS

TASKS INITIATION

(DECLARATION ELABORATION)

TASK TERMINATION

RENDEZVOUS_SYNCHRONIZATION
TASK TYPES
TASK TYPES

- Are used to create several similar but different tasks
- Task objects behave as constants (but not declared as constant since no initial value expression)
- Task types are limited private (no assignment, no equality compare)
SIMPLE TASK DECLARATIONS

task SIMPLE is task type ANON is
  ...
  ...
end SIMPLE;  end ANON;
  SIMPLE:ANON;
USING TASK OBJECTS IN STRUCTURES

\[ \text{AT : array(1..10) of T; --AT(1).E(...);} \]

\[ \text{type REC is} \]
\[ \text{record} \]
\[ \text{CT : T;} \]
\[ \ldots \]
\[ \text{end record;} \]

\[ \text{R : REC;} \]

362
TASKS VIA ACCESS TYPES

```solidity
type REF_T is access T;
RX:REF_T := new T;

// REF_T is normal access type, so can do assignment and compare two objects for equality

// to call entry E
RX.E (...);
```
BINDING TIME OF
NUMBER OF TASKS

1) SIMPLE TASK DECLARATIONS
   NO RECURSION
   \[\Rightarrow\] COMPILATION / LINK TIME

2) NO ACCESS OBJECTS TO
   OBJECTS OF TASK TYPE
   \[\Rightarrow\] ELABORATION TIME

3) ACCESS OBJECTS USED
   \[\Rightarrow\] DYNAMIC DURING
   EXECUTION
TASK TERMINATION

1) EXECUTE FINAL END =>
   TERMINATE WHEN ALL DEPENDENT
   TASKS ARE TERMINATED

2) TERMINATE ALTERNATIVE IN A
   SELECT STATEMENT
TERMlNATE ALTERNATIVE

select
  when GUARD_FALSE => ...
  when GUARD_TRUE =>
    terminate;
-- ACCEPT PREFERRED
-- NO else OR delay CHOICE
TERMINATE ALTERNATIVE (CON'T)

1. ALL ENTRY QUEUES EMPTY
2. MASTER IS COMPLETED
3. DEPENDENT TASKS AND SIBLING TASKS ARE:
   A) COMPLETE OR
   B) AT TERMINATE ALTERNATIVE
EXCEPTIONS AND ABORT

abort TASK_NAME_LIST;

- Each named task and each task that depends on them "becomes abnormal"
- Abnormal tasks must terminate prior to next synchronization point
ABNORMAL TASKS
CALLER RECEIVES "TASKING_ERROR"
IF:

1. CALLED TASK IS ABNORMAL
2. BECOMES ABNORMAL DURING RENDEZVOUS
3. BECOMES ABNORMAL WHILE ENQUEUED
   o SERVER TASKS ARE PROTECTED FROM
     CALLER BECOMING ABNORMAL
SYNCHRONIZATION POINTS

- END OF ACTIVATION
- ACTIVATE ANOTHER TASK
- ENTRY CALL
- START & END OF ACCEPT
- SELECT STATEMENT
- DELAY STATEMENT
- EXCEPTION HANDLER
- ABORT STATEMENT
SHARED VARIABLES
RULES TO FOLLOW

- IF TWO (OR MORE) TASKS READ
  OR UPDATE A SHARED VARIABLE
  (VARIABLE ACCESSIBLE BY BOTH)
- BETWEEN SYNCHRONIZATION POINTS
  1) MULTIPLE READERS
  OR
  2) ONE WRITER
SYNCHRONIZATION POINTS

1) START OF RENDEZVOUS  
2) END OF RENDEZVOUS  
3) AT ACTIVATION, WITH PARENT  
4) AFTER COMPLETION, WITH ALL  
5) \texttt{pragma} SHARED (VARIABLE)  
   -- ONLY SCALERS AND ACCESS  
   -- ONLY IF R/W INDIVISIBLE  
   -- EVERY R/W IS SYNCHRONIZED
WHY PRAGMA SHARFD?

READ A STATUS VARIABLE
-- EG DATABASE SIZE
-- WITHOUT RENDEZVOUS
OVERHEAD
SUMMARY OF TASK TYPES

- Tasks are one of the four forms of program unit.
- Tasks are a type; (subprograms, packages, generics are not)
- Tasks suggest parallel execution by logical processors.
LABORATORY #5

OBJECTIVES:

1. WRITE, COMPILIE, AND EXECUTE
   A MULTITASK PROGRAM
2. SHOW THAT Ada PROGRAMS
   ARE EASY TO MODIFY
ASSIGNMENT
TRANSFORM EACH PROCEDURE
OR FUNCTION IN YOUR HAND
CALCULATOR PROGRAM INTO A
SEPARATE TASK ACCESSED VIA
AN ENTRY CALL

NEXT PAGE INTENTIONALLY
LEFT BLANK.
TAPE #15

MACHINE DEPENDENT PROGRAMS/SUMMARY
IMPLEMENTATION DEPENDENT FEATURES

- REPRESENTATION CLAUSES
- PRAGMAS
- LENGTH CLAUSES
- ADDRESS CLAUSES
- INTERRUPTS
- INTERFACE TO OTHER LANGUAGES
- UNCHECKED STORAGE DEALLOCATION
- UNCHECKED TYPE CONVERSIONS
PRAGMAS

A `pragma` CONVEYS ADVICE TO THE COMPILER
`pragma PACK (TEXT);`
`pragma INLINE (PROCEDURE_NAME);`
<table>
<thead>
<tr>
<th>Memory Management</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>Compile/Load</td>
</tr>
<tr>
<td>Block Entry</td>
<td>Stack</td>
</tr>
<tr>
<td>Dynamic Allocator</td>
<td>Collection</td>
</tr>
<tr>
<td>(new)</td>
<td>In a heap</td>
</tr>
</tbody>
</table>
BLOCK ALLOCATION

"THE STACK"

TOP OF STACK
FRAME POINTER

ADD_A_FRAME:
FRAME_POINTER := TOP_OF_STACK;
TOP_OF_STACK := TOP_OF_STACK + LENGTH (NEW_FRAME);
MACHINE CODE INSERTS
EACH MACHINE INSTRUCTION IS
A RECORD AGGREGATE OF A TYPE
THAT DEFINES THE INSTRUCTION
M : MASK;
procedure SET-MASK;
pragma INLINE (SET_MASK);
.
SI_FORMAT' (CODE => SSM,
          B => M'BASE REG,
          D => M'DISP);
--IMPLEMENTATION SPECIFIC ATTRIBUTES
FORTRAN INTERFACE

package FORT_LIB is
  function SORT (X:FLOAT)
    return FLOAT;
  function EXP (X:FLOAT)
    return FLOAT;
private
  pragma INTERFACE
    (FORTRAN, SORT);
  pragma INTERFACE
    (FORTRAN, EXP);
end FORT_LIB;
package SYSTEM

type ADDRESS is . . . --MEMORY

type NAME is . . . --ENUMERATION

SYSTEM_NAME : constant := . . .

STORAGE_UNIT : constant := . . .

MEMORY_SIZE : constant := . . .

--REPRESENTATION ATTRIBUTES
INTERRUPTS (CONT)

- Above semantics can be implemented by having hardware execute the appropriate accept statement.
- Queued interrupts map to ordinary entry calls.
- Interrupts that are lost if not processed correspond to conditional entry calls.
- Interrupts have higher priority than main program or any user task.
INTERRUPTS

- INTERRUPTS ACT AS ENTRY CALLS

ISSUED BY HARDWARE TASKS

task INTERRUPT_HANDLER is
    entry DONE;
    for DONE use at 16#40#;
end INTERRUPT_HANDLER;
RECORD REPRESENTATION (CONT)

for PROGRAM_STATUS_WORD use
  record at mod 8;
    SYSTEM MASK at O*WORD range 0..7;
    PROTECTION KEY at O*WORD range 10..11;
    -- BITS 8,9 UNUSED
    MACHINE STATE at O*WORD range 12..15;
    INTERRUPT_CLAUSE at O*WORD range 16..31;
  end record;
for PROGRAM_STATUS_WORD'SIZE
  use 4*SYSTEM.STORAGE_UNIT
RECORD REPRESENTATION (CONT)

type PROGRAM_STATUS_WORD is
record
    SYSTEM_MASK : BYTE_MASK;
    PROTECTION_KEY : INTEGER range 0..3;
    MACHINE_STATE : STATE_MASK;
    INTERRUPT_CLAUSE : INTERRUPTION_CODE;
end record;
RECORD REPRESENTATION CLAUSE

WORD : constant := 4;
type STATE is (A, M, W, P);
type MODE is
  (FIX, DEC, EXP, SIGNIF);
type BYTE_MASK is
  array (0..7) of BOOLEAN;
type STATE_MASK is
  array (STATE) of BOOLEAN;
type MODE_MASK is
  array (MODE) of BOOLEAN;
ENUMERATION REPRESENTATION CLAUSE

type MIX_CODE is
  (ADD, SUB, MUL, LOA, STA, STZ);
for MIX_CODE use
  (ADD => 1, SUB => 2, MUL => 3,
  LOA => 8, STA => 24, STZ => 33);
REPRESENTATION SPECS
USED TO CONTROL THE BIT
PATTERNS WHICH REPRESENT
ENUMERATION TYPES
AND
RECORDS
AND THE AMOUNT OF STORAGE
ASSOCIATED WITH A TYPE
SIZE OF OBJECTS IN A COLLECTION

- ALL OBJECTS MUST BE OF SAME BASE TYPE
- VARIABLE LENGTH RECORDS CAN OCCUR WITH UNCONSTRAINED DISCRIMINANTS
- VARIABLE LENGTH ARRAYS OCCUR WHEN BASE TYPE IS UNCONSTRAINED
CONTROLLING MEMORY ALLOCATION

for TASK_TYPE_NAME'STOORAGE_SIZE
  use INTEGER_EXPRESSION;
--STORAGE UNITS PER ACTIVATION

for ACCESS_TYPE_NAME'STOORAGE_SIZE
  use INTEGER_EXPRESSION;
--STORAGE UNITS FOR COLLECTION
HEAP ALLOCATION

ACCESS
OBJECTS

"THE HEAP"
UNCHECKED DEALLOCATION

generic
    type OBJECT is limited private;
    type NAME is access OBJECT;
procedure UNCHECKED_DEALLOCATION
    (X : in out NAME);
procedure FREE is new
    UNCHECKED_DEALLOCATION
    (OBJ_TYPE,ACCESS_TYPE);
UNCHECKED TYPE CONVERSION

generic
    type SOURCE is limited private;
    type TARGET is limited private;
function UNCHECKED_CONVERSION
    (S : SOURCE) return TARGET;

function UNSPEC is new
    UNCHECKED_CONVERSION
    (SAC_TYPE, RESULT_TYPE);
SUMMARY
SOME Ada IMPLEMENTATIONS WILL ALLOW YOU TO:
- AFFECT MEMORY MANAGEMENT
- CONTROL THE REPRESENTATION OF TYPES
- WRITE INTERRUPT HANDLERS
- WRITE EMBEDDED MACHINE CODE
- LINK TO OTHER LANGUAGES
SYSTEM ATTRIBUTES ARE USED FOR IMPLEMENTATION DEPENDENT ALGORITHMS
Ada

REVIEW AND CONCLUSIONS
CONFORMING TO THE Ada STANDARD
(a) TRANSLATE & CORRECTLY EXECUTE
LEGAL UNITS, PROVIDED DON'T
EXCEED CAPACITY
(b) REJECT UNITS IF EXCEED
CAPACITY
(c) REJECT ALL ERRORS AS REQUIRED
(d) SUPPLY ALL REQUIRED PREDEFINED
UNITS
(e) NO VARIATIONS, EXCEPT AS
PERMITTED
(f) SPECIFY PERMITTED VARIATIONS
PROPERLY
CLASSIFICATION OF ERRORS

(a) ERRORS THAT MUST BE DETECTED AT COMPILATION BY ALL COMPILERS

(b) ERRORS THAT MUST BE DETECTED AT RUN-TIME BY ALL IMPLEMENTATIONS. THESE ARE THE PREDEFINED EXCEPTIONS.

(c) ERRONEOUS EXECUTION - RULES THAT MUST BE OBeyed BUT THAT IMPLEMENTATIONS NEED NOT CHECK

(d) INCORRECT ORDER DEPENDENCIES
RUN-TIME PROCESSES

- **EXECUTION**: Process a sequence of statements
- **EVALUATION**: Compute value of expression
- **ELABORATION**: Achieve effect of declarations, such as creating objects
ORDER OF EXECUTION

DEFINED

- IN ABSENCE OF FLOW OF CONTROL
  COMMAND, STATEMENTS IN TEXTUAL ORDER
- DECLARATIONS ELABORATED IN TEXTUAL ORDER
- EVALUATION OF SHORT-CIRCUIT CONTROL
ORDER OF EXECUTION (CONT)

UNDEFINED
- OPERANDS OF AN EXPRESSION EVALUATED IN SOME (UNDEFINED) ORDER
- ORDER OF PARAMETER ASSOCIATIONS
- ORDER IN WHICH in out AND out PARAMETERS COPIED BACK
- ORDER OF TASKS OF EQUAL PRIORITY
Ada Joint Program Office
30139 (400 A/N)
The Pentagon
Washington, DC 20301
QUESTIONNAIRE

1. Briefly state your current computer background.

2. Brief comment on course format.

3. How many tapes did you watch at a time?

4. What tapes needed to be replayed for content. Please amplify reasons to replay.

5. Overall evaluation of the course.

6. Do you recommend future courses of this format?

7. What topics do you recommend?

8. Should any tapes be redone - which ones, and why. (Please provide constructive suggestions.)

9. How effective were the lecture notes?

10. What improvement can you recommend for lecture notes.

(If you desire) Name

408