Modern Programming Languages

Introduction and Historical Background

Lecture 1-8

Course Objectives

Thousands of different programming languages have been designed by different people over the last 60 years. The questions that come to mind are:

- Why are there so many different programming languages?
- How and why they are developed?
- What is the intended purpose of a language?
- In what ways are they similar?
- What are the differences among them?
- Why wouldn't we simply continue to use what we have today?
- What kinds of programming languages may be developed in future?

We will try to answer some of these questions in this course.

In addition, we will discuss the design issues of various languages, design choices and alternatives available, historical context and specific needs, and specific implementation issues.

Text Book

The main text book for this course is:
Concepts of Programming Languages, 6th Ed. by Robert Sebesta.

Reasons to study concepts of Programming Languages

The first question is: why should we study programming languages. There are many reasons for that and some of them are enumerated in the following paragraphs.

1. Increased capacity to express programming concepts

Study of programming languages helps in increasing the capacity of express programming concepts.

Dijkstra has put it as follows:

> *The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.*

That is, one is limited in his/her thinking by the tools used to express his/her ideas.

Depth at which we can think is influenced by the expressive power of the language. It includes the kind of algorithms you can develop. The range of software development thought process can be increased by learning new languages as those constructs can be simulated.

2. Improved background for choosing appropriate languages

Study of programming languages also helps one in choosing the right language for the given task. Abraham Maslow says, "To the man who only has a hammer in the toolkit, every problem looks like a nail."

That is, if the only tool you have is a hammer, then you will treat every problem like a nail. Sometimes, some programming languages are more suitable for a specific task. There are many special purpose languages. In this course we will study one such language by the name of Snobol.

3. Increased ability to learn new languages

Study of different programming languages also helps one in learning new languages by learning the syntax and semantics of different languages and understanding different design methodologies.

4. Understanding the significance of implementation

In some cases, an understanding of implementation issues leads to an understanding of why languages are designed the way they are. This ultimately leads to efficient use of the language. One such example is Row vs. column major. If a programmer knows that two-dimensional arrays are stored column-wise (column major) in FORTRAN (where in most other languages it is row major) then he will be careful to process it column-wise, hence making it more efficient.

Same is the case with recursion. If the programmer knows how recursion is implemented and the associated cost of recursive programs, he can use this knowledge to come-up with more efficient programs if needed.

Also, certain bugs can only be found and fixed if the programmer knows some related implementation details.

5. Increased ability to design new languages

By learning a number of programming languages, one gets to know the pros and cons of different language features and issues related to these features. This knowledge will therefore help if one has to design a new language for any purpose.

Language Evaluation Criterion

In order to evaluate and compare different languages, we need some mechanism for their evaluation. The first criterion that comes to mind is: how long it takes to develop a program in a given programming language.

Capers Jones has developed the following Programming Languages Table which relates languages with the productivity.

Language Level Relationship to Productivity

LANGUAGE LEVEL	PRODUCTIVITY AVERAGE PER STAFF MONTH
1 - 3	5 to 10 Function Points
4 - 8	10 to 20 Function Points
9 - 15	16 to 23 Function Points
16 - 23	15 to 30 Function Points
24 - 55	30 to 50 Function Points
Above 55	40 to 100 Function Points

As can be seen, a higher-level language will yield more productivity as compared to a lower level language. Some of the common languages with their levels are listed below.

Assembly(1), C(2.5), Pascal(3.5), LISP(5), BASIC(5), C++(6)

As can be seen, C++ has the highest level in this list.

Let us now try to use this criterion to evaluate different languages.

When a programmer starts to learn a new language, a typical first exercise is to program the computer to display the message "Hello World". A compilation of “Hello World programs” designed by various categories of “developer” follows.

Hello World Programs – adapted from: Infiltec Humor Page
www.infiltec.com

A compilation of *Hello World programs* designed by various categories of *developer* follows.

High School/Jr.High - BASIC Language

```
10 PRINT "HELLO WORLD"
20 END
```

First year in College - Pascal

```
program Hello(input, output)
  begin
    writeln('Hello World')
  end.
```

Senior year in College - LISP

```
(defun hello
  (print
    (cons 'Hello (list 'World))))
```

New professional - C

```
#include
void main(void)
{
  char *message[] = {"Hello ", "World"};
  int i;

  for(i = 0; i < 2; ++i)
    printf("%s", message[i]);
  printf("\n");
}
```

Seasoned professional – C++

```
==========================
#include
#include

class string
{
private:
  int size;
  char *ptr;

public:
  string() : size(0), ptr(new char('\0')) {}

  string(const string &s) : size(s.size)
  {
    ptr = new char[size + 1];
    strcpy(ptr, s.ptr);
  }

  ~string()
  {
    delete [] ptr;
  }

  friend ostream &operator <<(ostream &, const string &);
  string &operator=(const char *);
};

ostream &operator<<(ostream &stream, const string &s)
{
  return(stream << s.ptr);
}

string &string::operator=(const char *chrs)
{
  if (this != &chrs)
  {
    delete [] ptr;
   size = strlen(chrs);
    ptr = new char[size + 1];
    strcpy(ptr, chrs);
  }
  return(*this);
}

int main()
{
  string str;

  str = "Hello World";
  cout << str << endl;

  return(0);
}
```

```
[
uuid(2573F8F4-CFEE-101A-9A9F-00AA00342820)
]
library LHello
{
    // bring in the master library
    importlib("actimp.tlb");
    importlib("actexp.tlb");

    // bring in my interfaces
    #include "pshlo.idl"

    [
    uuid(2573F8F5-CFEE-101A-9A9F-00AA00342820)
    ]
    cotype THello
 {
 interface IHello;
 interface IPersistFile;
 };
};

[
exe,
uuid(2573F890-CFEE-101A-9A9F-00AA00342820)
]
module CHelloLib
{

    // some code related header files
    importheader();
    importheader();
    importheader();
    importheader("pshlo.h");
    importheader("shlo.hxx");
    importheader("mycls.hxx");

    // needed typelibs
    importlib("actimp.tlb");
    importlib("actexp.tlb");
    importlib("thlo.tlb");

    [
    uuid(2573F891-CFEE-101A-9A9F-00AA00342820),
    aggregatable
    ]
    coclass CHello
 {
 cotype THello;
 };
};

#include "ipfix.hxx"

extern HANDLE hEvent;
```

```
class CHello : public CHelloBase
{
public:
    IPFIX(CLSID_CHello);

    CHello(IUnknown *pUnk);
    ~CHello();

    HRESULT  __stdcall PrintSz(LPWSTR pwszString);

private:
    static int cObjRef;
};

#include
#include
#include
#include
#include "thlo.h"
#include "pshlo.h"
#include "shlo.hxx"
#include "mycls.hxx"

int CHello::cObjRef = 0;

CHello::CHello(IUnknown *pUnk) : CHelloBase(pUnk)
{
    cObjRef++;
    return;
}

HRESULT  __stdcall  CHello::PrintSz(LPWSTR pwszString)
{
    printf("%ws\n", pwszString);
    return(ResultFromScode(S_OK));
}

CHello::~CHello(void)
{

// when the object count goes to zero, stop the server
cObjRef--;
if( cObjRef == 0 )
    PulseEvent(hEvent);

return;
}

#include
#include
#include "pshlo.h"
#include "shlo.hxx"
#include "mycls.hxx"

HANDLE hEvent;
```

```
 int _cdecl main(
int argc,
char * argv[]
) {
ULONG ulRef;
DWORD dwRegistration;
CHelloCF *pCF = new CHelloCF();

hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);

// Initialize the OLE libraries
CoInitializeEx(NULL, COINIT_MULTITHREADED);

CoRegisterClassObject(CLSID_CHello, pCF, CLSCTX_LOCAL_SERVER,
    REGCLS_MULTIPLEUSE, &dwRegistration);

// wait on an event to stop
WaitForSingleObject(hEvent, INFINITE);

// revoke and release the class object
CoRevokeClassObject(dwRegistration);
ulRef = pCF->Release();

// Tell OLE we are going away.
CoUninitialize();

return(0); }

extern CLSID CLSID_CHello;
extern UUID LIBID_CHelloLib;

CLSID CLSID_CHello = { /* 2573F891-CFEE-101A-9A9F-00AA00342820 */
    0x2573F891,
    0xCFEE,
    0x101A,
    { 0x9A, 0x9F, 0x00, 0xAA, 0x00, 0x34, 0x28, 0x20 }
};

UUID LIBID_CHelloLib = { /* 2573F890-CFEE-101A-9A9F-00AA00342820 */
    0x2573F890,
    0xCFEE,
    0x101A,
    { 0x9A, 0x9F, 0x00, 0xAA, 0x00, 0x34, 0x28, 0x20 }
};

#include
#include
#include
#include
#include
#include "pshlo.h"
#include "shlo.hxx"
#include "clsid.h"

int _cdecl main(
int argc,
char * argv[]
```

```
) {
HRESULT  hRslt;
IHello        *pHello;
ULONG  ulCnt;
IMoniker * pmk;
WCHAR  wcsT[_MAX_PATH];
WCHAR  wcsPath[2 * _MAX_PATH];

// get object path
wcsPath[0] = '\0';
wcsT[0] = '\0';
if( argc  1) {
    mbstowcs(wcsPath, argv[1], strlen(argv[1]) + 1);
    wcsupr(wcsPath);
    }
else {
    fprintf(stderr, "Object path must be specified\n");
    return(1);
    }

// get print string
if(argc  2)
    mbstowcs(wcsT, argv[2], strlen(argv[2]) + 1);
else
    wcscpy(wcsT, L"Hello World");

printf("Linking to object %ws\n", wcsPath);
printf("Text String %ws\n", wcsT);

// Initialize the OLE libraries
hRslt = CoInitializeEx(NULL, COINIT_MULTITHREADED);

if(SUCCEEDED(hRslt)) {

    hRslt = CreateFileMoniker(wcsPath, &pmk);
    if(SUCCEEDED(hRslt))
 hRslt = BindMoniker(pmk, 0, IID_IHello, (void **)&pHello);

    if(SUCCEEDED(hRslt)) {

 // print a string out
 pHello->PrintSz(wcsT);

 Sleep(2000);
 ulCnt = pHello->Release();
 }
    else
 printf("Failure to connect, status: %lx", hRslt);

    // Tell OLE we are going away.
    CoUninitialize();
    }

return(0);
}
```

Apprentice Hacker
==========

```
#!/usr/local/bin/perl
$msg="Hello, world.\n";
if ($#ARGV >= 0) {
  while(defined($arg=shift(@ARGV))) {
    $outfilename = $arg;
    open(FILE, ">" . $outfilename) || die "Can't write $arg: $!\n";
    print (FILE $msg);
    close(FILE) || die "Can't close $arg: $!\n";
  }
} else {
  print ($msg);
}
1;
```

Experienced Hacker
==========

```
#include
#define S "Hello, World\n"
main(){exit(printf(S) == strlen(S) ? 0 : 1);}
```

Seasoned Hacker
==========

```
% cc -o a.out ~/src/misc/hw/hw.c
% a.out
```

Guru Hacker
==========

```
% cat
Hello, world.
^D
```

New Manager – Back to BASIC
==========

```
10 PRINT "HELLO WORLD"
20 END
```

Middle Manager
==========

```
mail -s "Hello, world." bob@b12
Bob, could you please write me a program that prints "Hello,
world."?
I need it by tomorrow.
^D
```

Senior Manager
==========

```
% zmail jim
I need a "Hello, world." program by this afternoon.
```

Chief Executive
==========

```
% letter
letter: Command not found.
% mail
To: ^X ^F ^C
% help mail
help: Command not found.

% damn!
!: Event unrecognized
% logout
```

Language Evaluation Criterion

Jokes aside, this set of programs gives the readers some initial ideas about the different types of languages. Anyway, the question is: what happened to the programming language levels? Why is there so much variation for a simple program like "Hello World". The answer to this question lies in the question itself – "Hello World" is not the program that should be analyzed to compare different languages. It is so simple that the important features of different programming languages are not highlighted and wrong conclusions can be drawn.

A fair comparison needs much deeper analysis than this type of programs. Bjarne Stroustrup – the designer of C++ language – makes the following observation in this regards:

> *I have reaffirmed a long-standing and strongly held view: Language comparisons are rarely meaningful and even less often fair. A good comparison of major programming languages requires more effort than most people are willing to spend, experience in a wide range of application areas, a rigid maintenance of a detached and impartial point of view, and a sense of fairness.*

The question remains then how to compare and examine languages?

For this purpose we will use Readability, Writability, Reliability as the base criteria.

Readability

Readability is directly related to the cost of maintenance. The different factors that impact readability are:

- Simplicity
- Control Statements
- Data types and data structures
- Syntax Considerations
- Orthogonality

Simplicity

A simple language will be easy to learn and hence understand. There are a number of factors that account for simplicity. These are:

- Number of basic components

Learning curve is directly proportional to the number of basic components. If a language has more basic components, it will be difficult to learn and vice-versa. It is important to note that one can learn a subset of a programming language for writing but for reading you must know everything.

- Feature multiplicity
  If a language has more than one way to accomplish the same task, then it can cause confusion and complexity. For example, a number can be incremented in C in any of the following manners.

  ```
  i = i + 1;
  i++;
  ++i;
  i += 1;
  ```

  If a programmer has used all these different constructs to increment a variable at different locations, the reader may get confused. It is simpler to adopt one structure and use it consistently.

- Operator overloading

  Operator overloading can be problematic if its use is inconsistent or unconventional.

- How much simple should it be?

  Simplicity has another dimension as well. If a language is very simple then its level may also be decreased as it may loose abstraction. In that case, it will be difficult to write and read algorithms in that language.

Control Statements

Control statements also play an important role in readability. We are all aware of the hazards of goto statement. If a language relies on goto statements for control flow, the logic becomes difficult to follows and hence understand and maintain. Restricted use of goto in extreme was needed and useful as well but with the emergence of new language constructs to handle those situations, it probably not an issue any more.

Data types and data structures

Data types and data structures also play an important role in improving the abstraction level of a programming language, hence making it more readable. For example, a simple omission of Boolean type in C resulted in many problems as integers were instead of Boolean which made the program difficult to follow and debug in many cases.

Similarly, in FORTRAN there was no record (or struct) type. So related data had to be put in different data structures, making it very difficult to read the logic and maintain.

Syntax

Syntax of a language plays a very important role in programming languages. There are many different considerations that we will discuss throughout this course. Here we give a brief overview of some of these considerations.

- Variable names
  The first consideration is the restriction of variable names. In the beginning it was a huge issue as some languages restricted the length of a variable name up to 6 or 8 characters making it difficult to choose meaningful names for the variable. Other considerations include the type of special characters allowed in the variable names to improve readability.

- Special keywords for signaling the start and end of key words.

  Some languages allow special keywords for signaling the start and end of a construct, making it more readable and less prone to errors. This can be elaborated with the help of following examples:

  Let us first consider the following C if statement.

```
if (some condition)
        // do this
        // now do this
```

  It may be noted that the second part "// now do this" is not part of the body of the if statement and it will be executed regardless of the truth value of the condition. This may cause problems as it is easy for the reader to miss this point because of indentation.

  Ada programming language solves this problem by using the keywords "then" and "end if" in the if statement. The resulting code is shown below:

```
if (some condition) then
        -- do this
end if
        -- now do this
```

  Now it can be easily seen that a reader of this program will not be confused by the indentation.

Another similar example is when the if has an else part also. Let us first consider the following C code.

```
if (cond1)
        if (cond2)
          // do 1
    else if (cond3)
      // do 2
else
    // do 3
```

Is the last else part of third if or the first one? Once again the indentation has caused this confusion and the reader can easily miss this point. Ada has solved this problem by introducing an elsif keyword. The Ada code is shown below and it can be seen very easily that the confusion has been removed.

```
if (cond1) then
        if (cond2) then
           -- do 1
        elsif (cond3) then
           -- do 2
        end if;
else
    -- do 3
end if;
```

Writability

Writability is a measure of support for abstraction provided by a language. It is the ability to define and use complicated structures without bothering about the details. It may be noted that the degree of abstraction is directly related to the expressiveness. For example, if you are asked to implement a tree, it will be a lot difficult in FORTRAN as compared to C++. It also is a measure of expressivity: constructs that make it convenient to write programs. For example, there is a set data type in Pascal which makes it easier to handle set operations as compared to a language such as C++ where this support is not available.

Orthogonality

Orthogonality is a very important concept. It addresses how relatively small number of components that can be combined in a relatively small number of ways to get the desired results. It is closely associated with simplicity: the more orthogonal the design the fewer exceptions and it makes it easier to learn, read, and write programs in a programming language. The meaning of an orthogonal feature is independent of the context. The key parameters are symmetry and consistency. For example pointers is an orthogonal concept.

Here is one example from IBM Mainframe and VAX that highlights this concept. In IBM main frame has two different instructions for adding the contents of a register to a memory cell or another register. These statements are shown below:

```
A       Reg1, memory_cell
AR      Reg1, Reg2
```

In the first case, contents of Reg1 are added to the contents of a memory cell and the result is stored in Reg1; in the second case contents of Reg1 are added to the contents of another register (Reg2) and the result is stored in Reg1.

In contrast to the above set of statement, VAX has only one statement for addition as shown below:

```
ADDL operand1, operand2
```

In this case the two operands, operand1 and operand2, can be registers, memory cells, or a combination of both and the instruction simply adds the contents of operand1 to the contents of operand2 and store the result in operand1.

It can be easily seen that the VAX's instruction for addition is more orthogonal than the instructions provided by IBM and hence it is easier for the programmer to remember and use it as compared to the one provided by IBM.

Let us now study the design of C language from the perspective of orthogonality. We can easily see that C language is not very consistent in its treatment of different concepts and language structure and hence makes it difficult for the user to learn and use the language. Here are a few examples of exceptions:

- You can return structures but not arrays from a function
- An array can be returned if it is inside a structure
- A member of a structure can be any data type except void or the structure of the same type
- An array element can be any data type except void
- Everything is passed by value except arrays
- void can be used as type in a structure but you cannot declare a variable of this type in a function.

Orthogonality The Other side

Too much orthogonality is also troublesome. Let us again have a look at C language and try to understand this concept:

- In C
  - All statements (including assignment, if, while, etc) return some value and hence can be used in expressions.
  - Logic and arithmetic expressions can be intermixed.

  This can cause side effects and cryptic code.

  Since languages need large number of components, too much orthogonality can cause problems. From a language designer's point of view, the most difficult task is to strike a balance which obviously is not trivial.

Reliability

Reliability is yet another very important factor. A programming language should enable the programmers to write reliable code. The important attributes that play an important role in this respect are listed below:

- Type Checking

  Type checking is related with checking the type of the variables used as operands in different. The question that needs to be addressed is whether this type checking is done at compile time or at run-time. This also includes checking types of parameters and array bounds. Traditionally, these two have been the major sources of errors in programs which are extremely difficult to debug. A language that does type checking at compile time generates more reliable code than the one that does it at run-time.

- Exception handling

  Exception handling enables a programmer to intercept run-time errors and take corrective measure if possible and hence making it possible to write more reliable code.

Cost

Cost is also a very important criterion for comparing and evaluating programming language. In order to understand the cost involved, one has to consider the following:

- Training cost – how much does it cost to train a new programmer in this language.
- What is the cost of writing programs in the language – this is a measure of productivity
- What is the cost of the development environment and tools?
- What is the compilation cost? That is, how long does it take to compile a program. This is related to productivity as the more time it takes to compile a program, the more time a programmer will be sitting idle and hence there will be a reduction in productivity.
- Execution cost is also an important factor. If the program written in a language takes more execution time then the overall cost will be more.
- A related is whether to have a more optimized code or to increase the compilation speed
- Cost of language implementation deals with the level of difficulty in terms of writing a compiler and development environment for the language.
- If the program written in a particular language is less reliable than the cost of failure of the system may be significant.
- The most important of all of these factors is the maintenance cost. It is a function of readability.

Portability

Portability deals with how easy it is to port a system written in a given programming language to a different environment which may include the hardware and the operating system. In today's heterogeneous environment, portability is a huge issue. Portability has to do a lot with the standardization of a language. One of the reasons the programs written in COBOL were less portable than C was because C was standardized very early whereas there was not universal standard available for COBOL.

Generality

Generality is also an important factor and deals with the applicability of the language to a range of different domains. For example, C is more general purpose than LISP or FORTRAN and hence can be used in more domains than these two languages.

Issues and trade-offs

Like all design problems, in the case of programming language design, one has to deal with competing criterion such as execution versus safety, readability versus writability, and execution versus compilation. It would be nice if one could assign weights to different criteria and then compare the different options. Unfortunately, this kind of help is not available and hence the balancing act, as usual, is a very difficult job.

Influence of computer architecture on language design

Over the years, development in the computer architecture has had a major impact on programming language design. With the decreasing cost and increasing speed of hardware we simply can afford the programming languages which are more complex and tolerate the programming language to be less efficient. We shall now study the impact of the hardware architecture on the evolution of the programming languages.

Babbage's Analytical Engine

Charles Babbage is considered to be the inventor of the first computer. This machine, known as the Analytical Engine (or Difference Engine), was invented in 1820's. In the beginning, it could only be made to execute tasks by changing the gears which executed the calculations. Thus, the earliest form of a computer language was physical motion.
The key features of this machine included Memory (called store), jump, loop, and the concept of subroutines. It was motivated by the success of power looms. It had limited capability and because of the technological limitations this design could not be fully implemented.

ENIAC (Electronic Numerical Integrator and Calculator)

Physical motion was eventually replaced by electrical signals when the US Government built the ENIAC (Electronic Numerical Integrator and Calculator) in 1942. It followed many of the same principles of Babbage's engine and hence, could only be "programmed" by presetting switches and rewiring the entire system for each new "program" or calculation. This process proved to be very tedious.

Von Neumann Architecture - 1945

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

The second concept was also extremely important to the development of programming languages. Von Neumann called it "conditional control transfer". This idea gave rise to the notion of subroutines, or small blocks of code that could be jumped to in any order, instead of a single set of chronologically ordered steps for the computer to take. The second part of the idea stated that computer code should be able to branch based on logical statements such as IF (expression) THEN, and looped such as with a FOR statement. "Conditional control transfer" gave rise to the idea of "libraries," which are blocks of code that can be reused over and over.

In March 1949, Popular Mechanics made the following prediction:

"Where a calculator on the ENIAC is equipped with 18 000 vacuum tubes and weighs 30 tons, computers of the future may have only 1 000 vacuum tubes and perhaps weigh 1½ tons."

In 1949, a few years after Von Neumann's work, the language Short Code appeared. It was the first computer language for electronic devices and it required the programmer to change its statements into 0's and 1's by hand. Still, it was the first step towards the complex languages of today. In 1951, Grace Hopper wrote the first compiler, A-0. A compiler is a program that turns the language's statements into 0's and 1's for the computer to understand. This lead to faster programming, as the programmer no longer had to do the work by hand.

A most important class of programming languages, known as the imperative languages, is based upon the von Neumann Architecture. This includes languages like FORTAN, COBOL, Pascal, Ada, C, and many more.

Other major influences on programming language design have been mentioned as below:

Programming Methodologies

- 1950s and early 1960s:
  - *Simple applications; worry about machine efficiency*
- Late 1960s:
  - *People efficiency became important*
  - *readability, better control structures*
- Late 1970s:
  - *Data abstraction*
- Middle 1980s:
  - *Domain and data complexity - Object-oriented programming*
- Today
  - Web and networked environment; distributed computing

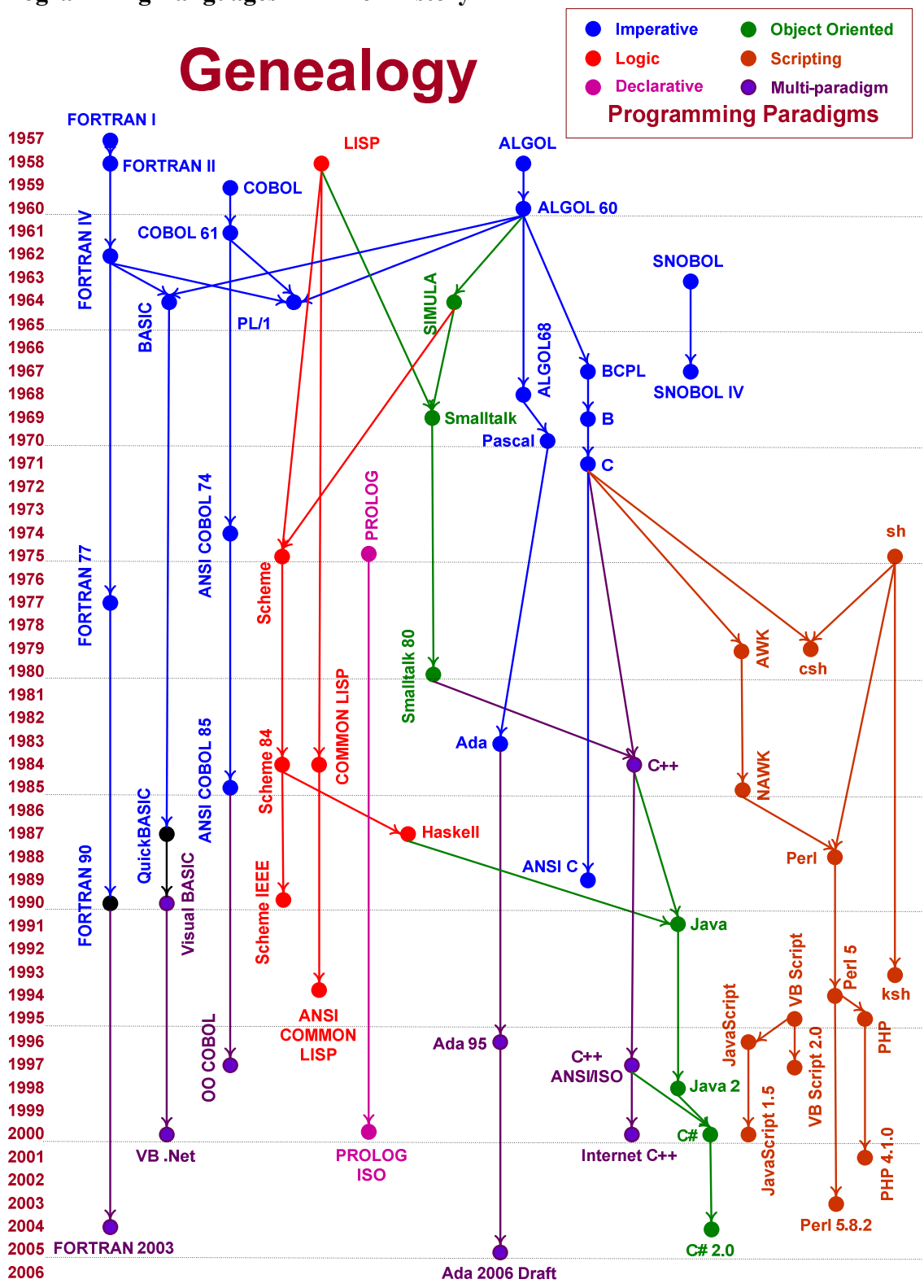
Language Categories

- Imperative
  - variables, assignment, iteration
- Functional (applicative)
  - set of functions
- Logic (Declarative)
  - rule based
  - Order of execution in not defined
- Object Oriented
  - close relatives of imperative

Application Domains

1. Scientific applications
    - simple data structures, large FP operations - FORTRAN
2. Business applications
    - reports, decimal arithmetic, character operations - COBOL
3. Artificial intelligence
    - Symbol processing, logic programming – Lisp, Prolog
4. Embedded systems
    - Concurrent programming - Ada
5. Systems programming
    - execution efficiency, low-level features – PL/1, BLISS, C
6. Scripting languages
    - list of commands – batch files, ksh, Perl
7. Special purpose languages
    - Hundreds of languages

Programming Languages - A Brief History



The First Programmer was a Lady

Charles Babbage and Ada Lovelace, lived in London of Dickens and Prince Albert (and knew them both). A hundred years before some of the best minds in the world used the resources of a nation to build a digital computer, these two eccentric inventor-mathematicians dreamed of building their "Analytical Engine". He constructed a practical prototype and she used it, with notorious lack of success, in a scheme to win a fortune at the horse races. Despite their apparent failures, Babbage was the first true computer designer, and Ada was history's first programmer.

Zuse's Plankalkül – 1945

The language was designed by the German scientist in isolation during the Second World War and was discovered and Published in 1972.

It was never implemented but included the following interesting features:

- The data type included floating point, arrays, records, nesting in records
- It had the notion of advanced data types and structures
- There was no explicit goto
- The concept of iteration was there
- It had selection without else part
- It also had the notion of invariants and assertions

Zuse used this language to solve many problems. The list of programs include sorting, graph connectivity, integer and floating point arithmetic, expressions with operator precedence, and chess playing. It may be noted that many of these problems remained a challenge for quite some time.

The language had a terse notation and hence was difficult to use. However, a number of programming language designer think that had this language be known at an earlier stage, we may never have seen languages like FORTRAN and COBOL.

Assemblers, Assembly Language, and Speed Coding

Initially, programs used to be written in machine code. It suffered from poor readability and poor modifiability – addition and deletion of instructions. Coding of expression was tedious. On top of that, machines did not support floating point and indexing and these things had to be coded by hand.

In 1954, John Backus developed what is known as Speedcoding for IBM 701. It had Pseudo operations for arithmetic and math functions, conditional and unconditional branching, and auto-increment registers for array access. It was slow as it was interpreted and after loading the interpreter, only 700 words left for user program. However, it made the job of programmer much easier as compared to the earlier situation.

The First Compiler - Laning and Zierler System - 1953

The first compiler was implemented on the MIT Whirlwind computer. It was the first "algebraic" compiler system which supported subscripted variables, function calls, and expression translation. It was never ported to any other machine.

FORTAN
The First High Level Language - FORTRAN I – John Backus 1957

FORTRAN stands FORmula TRANslating system. It was the first implemented version of FORTRAN as FORTRAN 0 (1954) was never implemented. It was the first compiled high-level language designed for the new IBM 704, which had index registers and floating point hardware. At that time computers were small and unreliable, applications were scientific, and no programming methodology or tools were available. Machine efficiency was most important and no need was felt for dynamic storage. So the language needed good array handling and counting loops. No string handling or decimal arithmetic was supported. Names could have up to six characters. It had support for formatted I/O and user-defined subprograms. There was no data typing statements and no separate compilation. The compiler was released in April 1957, after 18 man/years of effort. Unfortunately, programs larger than 400 lines rarely compiled correctly, mainly due to poor reliability of the 704 but code was very fast. Within one year, 50% of the code written for 704 was being developed in FORTRAN.

All statements of FORTRAN I were based on 704's instruction set. It included 3 way branch and computed if in the form of If (EXPRESSION) L1, L2, L3. It also had posttest counting loop as shown below:

```
DO L1 I = N, M
```

Fortran II (1958) and IV (1960)

Fortran II came after one year in 1958. It added independent compilation and had some bug fixes.

Fortran IV was released in 1960 and became the most popular language of its time. It had support for explicit type declarations and logical IF statement. Subprograms could also be passed as parameters. ANSI standard of FORTRAN IV was release in 1966 and remained mostly unchanged for the next 10 years.

FORTRAN 77 and 90

FORTRAN 77, released in 1977, was the next major release of FORTRAN after FORTRAN IV. It added support for structured Programming, character string handling, logical loop control statement, and IF-THEN-ELSE statement.

FORTRAN 90, released in 1990, added modules, dynamic arrays, pointers, recursion – stack frames, case statement, and parameter type checking. However, because of the

backward compatibility constraints, it could never gained the same popularity as its predecessors.

Functional Programming – LISP – McCarthy 1959

LISP stands for LISt Processing language. It was designed by John McCarthy in 1959 as part of AI research at MIT to deal with problems of linguistic, psychology, and mathematics. They needed a language to process data in dynamically growing lists (rather than arrays) and handle symbolic computation (rather than numeric). LISP has only two data types: atoms and lists and syntax is based on lambda calculus. It pioneered functional programming where there is no need for assignment and control flow is achieved via recursion and conditional expressions. It is still the dominant language for AI. COMMON LISP and Scheme are contemporary dialects of LISP and ML, Miranda, and Haskell are related languages.

ALGOL

ALGOL 58 – 1958 – Search for a "Universal Language"

ALGOL stands for ALGOrithmic Language. It was designed in 1958. At that time FORTRAN had (barely) arrived for IBM 70x and was owned by IBM. Many other languages were being developed, all for specific machines. There was no portable language as all were machine-dependent. Also, there was no universal language for communicating algorithms.

ALGOL was thus designed to be a language that was close to mathematical notation, good for describing algorithms, and was machine independent. That is, it was an algorithmic language for use on all kinds of computers.

Salient features of the language are:

- Concept of type was formalized
- Names could have any length
- Arrays could have any number of subscripts
- Lower bound of an array could be defined
- Parameters were separated by mode (in & out)
- Subscripts were placed in brackets
- Compound statements were introduced (begin ... end)
- Semicolon as a statement separator was used
- Assignment operator was :=
- if had an else-if clause

It was actually not meant to be implemented, but variations of it were (MAD, JOVIAL) implemented. Although IBM was initially enthusiastic but vested interest in FORTRAN resulted in taking back all support by mid-1959.

Algol 60 - 1960

ALGOL 60 was a modified version of ALGOL 58. It added new features to the language. These included block structure (local scope), two parameter passing methods – value and name, subprogram recursion, and stack-dynamic arrays – run time size definition and space allocation. It did not have built-in I/O facility.

It was quite successful as it was the standard way to publish algorithms for over 20 years. All subsequent imperative languages are based on it. It was the first machine-independent language and the first language whose syntax was formally defined in BNF. It also impacted the hardware design.

It was never widely used, especially in U.S because there was no i/o support and the character set made programs non-portable. Also, it was too flexible and hence was hard to understand and implement. On top of that, IBM never supported it on their machines because of their interest in FORTRAN. In addition, BNF was considered strange and complicated!

Algol 68 - 1968

ALGOL 68 was from the continued development of ALGOL 60, but it is not a superset of that language. Its design is based on the concept of orthogonality. Major contributions of this language include user-defined data structures, reference types, and dynamic arrays (called flex arrays).

From a practical point of view it had even less usage than ALGOL 60 but had strong influence on subsequent languages, especially Pascal, C, and Ada.

COBOL - 1960

COBOL was designed in 1960 to support business-oriented computation which requires fixed point arithmetic. It was designed to look like simple English to broaden the base of computer users. It was thus required to be easy to use, even if that means it will be less powerful. Another design consideration was that it must not be biased by current compiler.

It encountered problems because the design committee members were all from computer manufacturers and DoD branches and there were fights among manufacturers.

It was the first language to add macro facility in a high-level language. It also introduced hierarchical data structures (records) and nested selection statements. The language supported long variable names (up to 30 characters) with hyphens, data division, and fixed-point arithmetic.

It was the first language required by DoD and would probably have failed without support from DoD. It is still a very widely used business applications language and is very popular in business and government, much less at universities.

Basic - 1964

BASIC was designed by Kemeny & Kurtz at Dartmouth College with the following goals in mind: Easy to learn and use for non-science students; Must be "pleasant and friendly"; Fast turnaround for homework; Free and private access; User time is more important than computer time.

Current popular dialects include QuickBASIC and Visual BASIC.

PL/1 – 1965 – Everything for Everybody

From IBM's point of view there were two computing groups: scientific computing and business computing. The scientific computing was supported by IBM 1620 and 7090 computers and FORTRAN. Business computing was done on IBM 1401, 7080 computers and COBOL programming language.

By 1963 scientific users began to need more elaborate I/O, like COBOL had and business users began to need fl. pt. and arrays (MIS). It looked like many shops would begin to need two kinds of computers, languages, and support staff. This proposition was obviously too costly and the obvious solution was to build a new computer to do both kinds of applications and design a new language to do both kinds of applications. Hence PL/1 came into being.

PL/I was the first language to introduce unit-level concurrency, exception handling, pointer data type, and array cross sections.

The language was not a huge success because many new features were poorly designed and it was too large and too complex.

Early Dynamic Languages -Characterized by dynamic typing and dynamic storage allocation

APL (A Programming Language) 1962

It was designed as a hardware description language (at IBM by Ken Iverson). It was highly expressive (many operators, for both scalars and arrays of various dimensions) but programs are very difficult to read - commonly known as "write-only" language.

SNOBOL (1964)

It was designed as a string manipulation language (at Bell Labs by Farber, Griswold, and Polensky). It had powerful operators for string pattern matching but suffered from poot readability and maintainability.

Simula 67 – 1967 –The first Object-oriented language

It was designed in Norway by Nygaard and Dahl, primarily for system simulation. It was based on ALGOL 60 and SIMULA I. Its primary contributions include the concept of a class which was the basis for data abstraction. Classes are structures that include both local data and functionality.

Pascal – 1971 – Simplicity by Design

Pascal was designed by Niklaus Wirth, who quit the ALGOL 68 committee because he didn't like the direction of that work. It was designed for teaching structured programming. It was small and simple with nothing really new. Because of its simplicity and size it was, for almost two decades, the most widely used language for teaching programming in colleges.

C – 1972 – High-level system programming language

C was designed for systems programming at Bell Labs by Dennis Richie. It evolved primarily from B, but was also influenced by ALGOL 68. It had powerful set of operators, but poor type checking. It initially spread through UNIX but became very popular in the academic circles because of its easy portability. Many modern programming languages including C++, Java, and C# are direct descendents of C.

Prolog – 1972 – Logic Programming

It was developed at the University of Aix-Marseille, by Comerauer and Roussel, with some help from Kowalski at the University of Edinburgh. It is based on formal logic. It is a non-procedural declarative programming language with built-in backtracking mechanism. It has support for associative memory and pattern directed procedure invocation. After LISP, it is the second most widely used language in AI community, especially in Europe.

Ada – 1983 – History's largest design effort

It involved a huge design effort, involving hundreds of people, much money, and about eight years. It introduced Packages - support for data abstraction, elaborate exception handling, generic program units, and concurrency through the tasking model.

It was the outcome of a competitive design effort. It included all that was then known about software engineering and language design. Because of its size and complexity, first

compilers were very difficult and the first really usable compiler came nearly five years after the language design was completed

Smalltalk - 1972-1980 – The Purest Object-Oriented Langauge

It was developed at Xerox PARC, initially by Alan Kay and then later by Adele Goldberg. It is the first full implementation of an object-oriented language (data abstraction, inheritance, and dynamic type binding) and the purest object-oriented language yet! It pioneered the graphical user interface everyone now uses.

C++ - 1985

It was developed at Bell Labs by Stroustrup. It evolved from C and SIMULA 67. Facilities for object-oriented programming, taken partially from SIMULA 67, were added to C. It also has exception handling. It is a large and complex language, in part because it supports both procedural and OO programming. Rapidly grew in popularity, along with OOP and ANSI standard was approved in November, 1997.

Java - 1995

Java was developed at Sun in the early 1990s and is based on C++. It is significantly simplified as compared to C++ and supports only OOP. It eliminated multiple inheritance, pointers, structs, enum types, operator overloading, and goto statement and added support for applets and a form of concurrency.

C# - 2002

It is part of the .NET framework by Microsoft. It is based upon C++ and Java and support component-based software development. It has taken some ideas from Visual Basic. It brought back pointers, structs, enum types, operator overloading, and goto statement. It has safer enum types, more useful struct types, and modified switch statement.

Scripting Languages for Web

These languages were designed to address the need for computations associated with HTML documents. JavaScript and PHP are two representative languages in this domain.

JavaScript – client side scripting

Primary objective of JavaScript is to create dynamic HTML documents and check validity of input forms. It is usually embedded in an HTML document. It is not really related to Java

PHP (Personal Home Page) – server-side scripting
It is interpreted on the Web Server when the HTML document in which embedded is requested by the browser. It often produces HTML code as an output and is very similar

to JavaScript. It allows simple access to HTML form data and makes form processing easy. It also provides support for many different database management systems and hence provides Web access to databases.

Programming Language Evolution

Over the last five decades, programming languages have evolved after going through several phases. These phases are briefly described below:

- **1950's – Discovery and description of programming language concepts**

  Programming language design in this period took an empirical approach. Programming languages were regarded solely as tools for facilitating the specification of programs rather than as interesting objects of study in their own right. In this era we saw development of symbolic assembly language, macro-assembly, FORTRAN, Algol 60, COBOL, and LISP. Many of the basic implementation techniques were discovered which include symbol table construction and look-up, stack algorithms for evaluating arithmetic expressions, activation record stack, and marking algorithms for garbage collection.

- **1960's – Analysis and elaboration**

  This was the era when programming language design took a mathematical approach. Here we saw theoretical research as an end in itself and a lot of analysis was carried out for the purpose of constructing models and theories of programming languages. Representative languages of this era include PL/1, Simula, Algol 68, and Snobol. These languages elaborated the earlier languages and attempted to achieve greater richness by synthesis of existing features and generalization. This resulted in greater complexity. This was the time when formal languages and automata theory with application to parsing and compiler theory as well as theory of operational and mathematical semantics were defined. There was a lot of emphasis on program correctness and verification.

- **1970's – Effective software technology**

  Decreasing hardware cost and increasing software cost resulted in more complex software requiring support for software engineering in the programming languages. It also required development of tools and methodologies for controlling the complexities, cost and reliability of large programs Therefore, languages designed in this period had software engineering support in the form of structured design in the form of structured programming, modular design, and verification. It saw a transition from pure research to practical management of the environment. Verifiable languages such as Pascal and Modula were developed as a result of this effort.

- **1980's – Support for SE continued**
- **1990's and 2000's**
  - Support for OOP and Internet
- **2010's – Aspect-Oriented Programming**

Language Design Perspectives

Programming Language design is not a simple task. A designer has to consider the need for many different stake holders and balance their requirements. For example, software developers ask for more features and theoreticians ask for features with clean semantics. On the other hand developers of mission-critical software want features that are easier to verify formally. Compiler writers want features that are orthogonal, so that their implementation is modular. These goals are often in conflict and hence it is difficult to satisfy all of them.

From a language definition perspective, once again different users have different needs. Programmers need tutorials, reference manuals, and programming guides (idioms). Implementers demand precise operational semantics and verifiers require rigorous axiomatic or natural semantics. Language designers and language advocates want all of the above. This means that a language needs to be defined at different levels of detail and precision but none of them can be sloppy!

Modern Programming Languages

Lecture 9-12

An Introduction to SNOBOL Programming Language

SNOBOL stands for StriNg Oriented SymBOlic Language. It is a Special purpose language for string manipulation and handling. It was developed in 1962 at the Bell Labs by Farber, Griswold, and Polensky.

It was created out of the frustrations of working with the languages of that time because they required the developers to write large programs to do a single operation. It was used to do most of the work in designing new software and interpreting different language grammars. Toward the end of the eighties, newer languages were created which could also be used for string manipulation. They include the PERL and AWK. Unlike SNOBOL, Perl and Awk use regular expressions to perform the string operations. Today, roughly forty years after its release, the language is rarely used!

SIMPLE DATA TYPES

Initial versions of SNOBOL only supported Integers and Strings as the basic data types. Real numbers were originally not allowed but were added to the language later on.

Integers

Both positive and negative integers were supported. Following are some examples of integers in SNOBOL.

14 -234 0 0012 +12832 -9395 +0

It was illegal to use a decimal point or comma in a number. Similarly, numbers greater than 32767 (that is, $2^{15} – 1$) could not be used as well. For the negative numbers, the minus sign has to be used without any spaces between the sign and the number. Following are therefore some examples of illegal numbers:

13.4 49723 - 3,076

As mentioned earlier, real numbers were allowed in the later versions of the language and hence 13.4 is a legal number in SNOBOL4.

Strings

The second basic data type supported by the language is a string. A string is represented by a sequence of characters surrounded by quotes (single of double). The maximum length allowed for a string is 5,000 characters. Examples of strings are: “This is a string”, ‘this is also a string’,
“it’s an example of a string that contains a single quote”

SIMPLE OPERATORS

SNOBOL supports the usual arithmetic operators. This includes +, -, *, and /. In addition it also supports the exponentiation operator which is ** or !. The unary – and + are also supported. = operator is used for assignment. The precedence of the operators is as follows: the unary operations are performed first, then exponentiation, then multiplication, followed by division, and finally addition and subtraction. All arithmetic operations are left associative except the exponentiation which is right associative. Therefore, 2 ** 3 ** 2 is evaluated as 2 ** (3 ** 2) which is 512 and not (2 ** 3) ** 2 which would be 64. Like most languages, parentheses can be used to change the order of evaluation of an expression.

Just like C, if both the operands of an operator are integers then the result is also an integer. Therefore, 5 / 2 will be evaluated to be 2 and not 2.5.

VARIABLE NAMES

A variable name must begin with an upper or lower case letter. Unlike C, SNOBOL is case insensitive. That is, the literals Wager and WAGER represent the same variable. If the variable name is more than one character long, the remaining characters may be any combination of letters, numbers, or the characters period '.' and underscore '_'. The name may not be longer than the maximum line length (120 characters). Following are some of the examples of variable names:

WAGER P23 VerbClause SUM.OF.SQUARES Verb_Clause

SNOBOL4 STATEMENT Structure

A SNOBOL statement has three components. The structure of a statement is as follows:

Label Statement body :GOTOField

Label
The must begin in the first character position of a statement and must start with a letter or number.

The GOTO Field

The goto field is used to alter the flow of control of a SNOBOL program. A goto filed has the following forms:

:(label)
:S(label)
:F(label)
:S(label1) F(label2)

The first one is an command for unconditional jump to the label specified within parentheses. The second is a command for jump to the specified label if the statement in the body of the statement was executed successfully or resulted in true Boolean value. The third one is opposite to the second and the control jumps to the label if the statement is not executed successfully or resulted in false. The fourth is a combination of the second and the third and states that goto label1 in case of success and label2 in case of failure.

Statement Body

The body can be of any of the following types:

Assignment statement
Pattern matching statement
Replacement statement
End statement

Assignment Statement

Assignment statement has the following form.

variable = value

Following are some examples of assignment statement which involve arithmetic expressions.

v = 5
w.1 = v
*w = 14 + 3 * -2*
v = 'Dog'

It is important to note that the binary operators must have at least one space on both sides. For example 3+2 is not allowed.

It may also be noted that, unlike C, variable are not declared in SNOBOL. The assignment statement creates a variable if it is not already created and assigns a type to it which is the type of the expression in the right hand side of the assignment operator (r-value). So the type of v and w is integer and w.1 is of type real. If a new value is assigned to a variable, it type changes accordingly. For example the type of the variable v is integer and it is now assigned a value of 'Dogs' then its type will change to string. This is known as dynamic typing. That is, the type is determined at run time and changes during execution.

If there is nothing on the right hand side of the assignment statement, then a value of NULL string is assigned to the variable. This is demonstrated in the following example.

This.is.null.string =

Another very interesting aspect of SNOBOL is mixing numeric and string values in an expression. If a string can be interpreted as a number then its numeric value will be used in the expression as follows:

```
z = '10'
x = 5 * -z + '10.6'
```

Now a value of -39.4 will be assigned to x.

Strings Concatenation

SNOBOL was designed mainly for manipulating and handling strings. It has therefore rich string manipulation mechanism. The first one is string concatenation. A *Space* is used as an operator for concatenation as shown below:

```
TYPE = 'Semi'
OBJECT = TYPE 'Group'
```

The above set of statements will assign 'SemiGroup' to the variable OBJECT. Numbers can also be concatenated with strings, producing interesting results. For example, consider the following code segment:

```
ROW  = 'K'
NO.    = 22
SEAT = ROW NO.
```

In this case NO. is concatenated with ROW and a value of '*K22'* is assigned to SEAT.

The Space operator has a lower precedence than the arithmetic operators and hence we can have arithmetic expressions in string concatenation. For example,

```
SEAT = ROW NO. + 6 / 2
```

will assign the value '*K25'* to SEAT if ROW and No. have the values 'K' and 22 respectively.

Pattern Matching

Pattern matching and manipulation is another very important feature of SNOBOL. The first statement in this regards is the Pattern Matching Statement. Once again *Space* is used as the pattern matching operator and the statement has the following form:

subject pattern

Note that there is no assignment operator in this case. Both the *subject* and the *pattern* are strings and this statement tries to match the *pattern* in the *subject*. It will be successful if the match is found and will result in failure otherwise. This is demonstrated with the help of the following example:

```
TRADE = 'PROGRAMMER'
PART  = 'GRAM'
TRADE PART
```

Now 'GRAM' will be searched in 'PROGRAMMER'. Since it is present in it ('PROGRAMMER'), it will result in success.

We can use Space for string concatenation as well as for pattern matching in the same statement. In this case, the first space is used as pattern matching and the second one is used as concatenation with concatenation taking precedence over pattern matching. This is shown with the help of the following example:

```
ROW  = 'K'
NO.  = 2
'LUCK22'  ROW NO.
```

In this case, NO. is first concatenated with ROW resulting in 'K2' which will then be matched in 'LUCK22'. Once again the match will be successful.

Replacement

Replacement statement is used in conjunction with the pattern matching statement in the following manner.

subject pattern = object

In this case the pattern is searched in the subject and if found it is replaced by the object as demonstrated by the following example:

SENTENCE = ‘THIS IS YOUR PEN’
SENTENCE ‘YOUR’ = ‘MY’

Since ‘YOUR’ is present in the subject, it will be replaced by ‘MY’, resulting in changing the value of SENTENCE to ‘THIS IS MY PEN’.

If we now have the following statement

SENTENCE ‘MY’ =

then SENTENCE will be further modified to ‘THIS IS PEN’ as we are now replacing ‘MY’ with a NULL string, effectively deleting ‘MY’ from the subject.

Pattern

There are two type of statements for pattern building. These are Alternation and Concatenation.

Alternation

Vertical bar is used to specify pattern alternation as shown in the example below.

P1 | P2

This is example of a pattern that will match either P1 or P2.

Here are some more examples:

KEYWORD = ‘INT’ | ‘CHAR’

This statement assigns the pattern ‘INT’ | ‘CHAR’ to the variable KEYWORD.

KEYWORD = KEYWORD | ‘FLOAT’

KEYWORD will now get a new value which is ‘INT’ | ‘CHAR’ | ‘FLOAT’. So we can create new pattern by using the assignment statement.

Let us now use the KEYWORD pattern in pattern matching and replacement.

TEXT = ‘THIS IS AN INTEGER’
TEXT KEYWORD =

Since KEYWORD had the value ‘INT’ | ‘CHAR’ | ‘FLOAT’, that means any one of these strings, that is ‘INT’, ‘CHAR’, or ‘FLOAT’, will match. This matches with INT in ‘THIS IS AN INTEGER’ and will be replaced by the NULL string. So the new value of TEXT is ‘THIS IS AN EGER’.

Concatenation
Two or more patterns can be concatenated to create new patterns. For example we define P1 and P2 as follows:

P1 = ‘A’ | ‘B’
P2 = ‘C’ | ‘D’

Now if we concatenate these and assign the result to P3 as shown below:

P3 = P1 P2

This will result in assigning ‘AC’ | ‘AD’ | ‘BC’ | ‘BD’ to P3 which is concatenation of different alternatives of P1 and P2.

Therefore

“ABCDEFGH” P3

will match at “ABCDEFGH”

Conditional Assignment

At times we want to ensure that assignment occurs ONLY if the entire pattern match is successful. This is achieved by using the '.' (dot) operator as shown below.

pattern . variable

In this case upon successful completion of pattern matching, the substring matched by the pattern is assigned to the variable as the value.

Following is a more elaborative example of this concept.

```
KEYWORD = ('INT' | 'CHAR') . K
TEXT = "INT INT CHAR CHAR INT"
TEXT KEYWORD =
```

In this case, the variable K which was associated with the pattern will get the value INT.

Here is another example.

Let us define the following pattern where BRVAL has been associated to this pattern by the dot operator.

```
BR = ( 'B' | 'R' ) ( 'E' | 'EA' ) ( 'D' | 'DS' ) . BRVAL
```

Associates variable BRVAL with the pattern BR

It may be noted that the pattern BR is created by concatenating three different alternations as shown below.

```
BR = ( 'B' | 'R' ) ( 'E' | 'EA' ) ( 'D' | 'DS' )
```

Note that it is equivalent to

```
      BR = 'BED' | 'BEDS' | 'BEAD' | 'BEADS' | 'RED' | 'REDS' | 'READ' |
+           'READS'
```

This '+' in the first column of a line states that this is continuation of the last line and not a new statement. Anyway, let us come back to our example.

On successful completion of matching, the entire substring matched will be assigned as value of the BRVAL

So

‘BREADS’ BR

will assign ‘READS’ to BRVAL.

Let us now define the BR pattern as shown below and also associate FIRST, SECOND, and THIRD to the respective constituent sub-patterns of this larger pattern.

```
        BR = ( ( ‘B’ | ‘R’ ) . FIRST ( ‘E’ | ‘EA’ ) . SECOND
+                    ( ‘D’ | ‘DS’ ) . THIRD) . BRVAL
```

On successful completion of matching, the entire substring matched will be assigned as value of the BRVAL. In addition B or R will become the value of FIRST, E or EA will be assigned to SECOND, and THIRD will get either D or DS.

So

‘BREADS’ BR

will result in the following assignments.

FIRST – ‘R’ , SECOND – ‘EA’, THIRD – ‘DS’
BRVAL – ‘READS’

Immediate Value Assignment

The '.' (dot) operator is used for conditional assignment only when the entire pattern is matched.

The $ is used for immediate value assignment even if the entire pattern does not match. It is used as follows:

Pattern $ Variable

In this case whenever pattern matches a substring, the substring immediately becomes the new value of the Variable. This is demonstrated with the help of the following example.

```
BR = ( ( 'B' | 'R' ) $ FIRST ( 'E' | 'EA' ) $ SECOND
+                    ( 'D' | 'DS' ) $ THIRD) . BRVAL
```

Value assignment is done for those parts of the pattern that match, even if the overall match failed.

So the following statement

'BREAD' BR

will result in the following matches.

FIRST – B, FIRST – R, SECOND – E, SECOND – EA, THIRD – D,
BRVAL - READ

Note that FIRST and SECOND get a value as soon as a match is made and hence are assigned values more than once. Of course only the last value will be the value of these variables at the end of entire match.

Let us now try make the following match.

'BEATS' BR

In this case the statement fails as the whole pattern is not matched but parts of it are matched and hence get the values as shown below.

FIRST – 'B' , SECOND – 'E', SECOND – 'EA'

Input and Output

The I/O is accomplished through INPUT and OUTPUT statements. We will look at examples of I/O in the next sections.

Control Flow

Control flow is achieved through the ‘go to field’ in the statement body as shown in the following examples.

The first example simply keeps on reading the entire line from the INPUT and stops when there is end of input.

```
LOOP        PUNCH = INPUT     :S(LOOP)
END
```

The next example reads a line and prints it on the output and then goes back to read the next lines again.

```
LOOP        PUNCH = INPUT     :F(END)
            OUTPUT = PUNCH :(LOOP)
END
```

Here is a more interesting example that uses control flow. In this example it continues to delete a matched pattern in the subject and stops when no match is found.

```
            TEXT = “REDPURPLEYELLOWBLUE”

            COLOR = ‘RED’ | ‘BLUE’ | ‘GREEN’
BASIC       TEXT    COLOR =   :S(BASIC) F(OTHER)
OTHER
```

In the following example all the numbers for 1 to 50 are added and the result is stored in SUM and is printed on the output.

```
            SUM = 0
            N      = 0
ADD         N = LT(N, 50) N + 1         :F(DONE)
            SUM = SUM + N               :(ADD)
DONE        OUTPUT = SUM
```

Indirect Reference

Indirect reference is also an interesting feature of SNOBOL. It is similar to a pointer in concept but there are certain differences as well. These differences will be highlighted in the following code segments. Let us first look at the syntax:

The unary operator '$' is used for indirect reference. Note that '$' has been overloaded in this case. This says that now instead of using the operand as the variable, use its value as the variable. The rest will remain the same. Here is an example that elaborates this concept:

```
MONTH = 'APRIL'
$MONTH = 'FOOL'
```

MONTH is given the value 'APRIL' 1. In statement 2, indirect reference is used. Here, instead of MONTH, its value will be used as the variable. So it is equivalent to saying

```
APRIL = 'FOOL'
```

Here is another example that uses indirect reference in a more interesting manner:

```
RUN = 10
WORD = 'RUN'
$(WORD) = $(WORD) + 1
```

Because of indirect reference, it increments the value of RUN and not WORD.

The following example of indirect reference uses it in the go to field.

```
N = 3
N = N + 1      :$( 'LABEL' N)
```

This results in making the control jump to LABEL4

Functions
SNOBOL 4 supports two types of functions: (a) built-in function which are known as primitive functions and (b) user defined functions.

Primitive Functions

There are a number of primitive functions but we shall look at only a few. These include SIZE and REPLACE functions. The SIZE function returns the size of a string and the REPLACE function is used to replace one character with another in the entire string. Here are some examples of these functions:

```
SIZE(TEXT)

PART1 = “NUT”
OUTPUT = SIZE(PART1)
```

Since the string stored in PART1 is ‘NUT’, it has three characters and hence the result is 3.

```
PART2 = “BOLT”
N = 64
OUTPUT = SIZE(‘PART’ N + 36)
```

In this case, PART will be concatenated with 100 and will generate the string PART100. The output will thus display 7.

REPLACE function has the following syntax:

```
REPLACE(STRING, ST1, ST2)
```

In this case all occurrences of ST1 in STRING will be replaced by ST2. ST1 and ST2 must have the same number of characters. The corresponding characters of ST1 will be replaced by characters corresponding to them in ST2. This is shown in the following example.

```
TEXT = “A(I,J) = A(I,J) + 3”
OUTPUT = REPLACE(TEXT , ’()’ , ‘<>’)
```

The output will thus display:

```
A<I,J> = A<I,J> + 3
```

Arrays

SNOBOL4 has array data type just like other languages. However, unlike most languages, data stored in an array is not necessarily of the same type. That is, each element can be of different data type.

The following statement creates and assigns to V a one-dimensional array of 10 elements, each assigned to the real value of 1.0.

```
V = ARRAY(10, 1.0)
```

The next statement creates a one-dimensional array X of 8 elements with the lower index being 2 and the upper index being 9. As there is no initial value given, each cell is initialized to NULL string.

```
X = ARRAY('2:9')
```

The following statement creates a two dimensional array of 3 x 5 and each cell is initialized to NULL string.

```
N = ARRAY('3,5')
```

The size of the array can be determined at run time by using input from the system of using a variable. Following example shows this concept:

```
A = ARRAY(INPUT)
```

In this case size and initial value of the array is determined by the input.

Following example creates an array whose size is determined at run time. It then stores values in it from the INPUT. It is important to note here that when the array index 'I' gets a value, as the result of increment in the second last statement, greater than the value of the last index of the array, the statement with Label MORE fails and the control goes to the Label GO.

```
Arrays
                    &TRIM = 1
                    I = 1
                    ST = ARRAY(INPUT)
MORE                ST<I> = INPUT           :F(GO)
                    I = I + 1               :(MORE)
GO
```

The following example uses arrays in an interesting manner:

```
        &TRIM = 1
        WORDPAT = BREAK(&LCASE &UCASE) SPAN(&LCASE &UCASE "'-") . WORD
                COUNT = ARRAY('3:9',0)
READ            LINE = INPUT                                            :F(DONE)
NEXTW           LINE WORDPAT =                                          :F(READ)
                COUNT<SIZE(WORD)> = COUNT<SIZE(WORD)>+ 1                :(NEXTW)
DONE            OUTPUT = "WORD LENGTH NUMBER OF OCCURRENCES"
                 I = 2
PRINT           I = I + 1
                OUTPUT = LPAD(I,5) LPAD(COUNT<I>,20)                    :S(PRINT)
END
```

The first statement simply deletes leading spaces from the input.

In the second statement, a pattern WORDPAT is defined. It uses tow primitive functions BREAK and SPAN. BREAK takes the cursor to the first lowercase or uppercase character in the subject. The SPAN provides a mechanism to keep moving as long as we get uppercase, lowercase, or a hyphen in the subject string. So this pattern scans the subject string for complete words. Since we have associated variable WORD with this pattern, the matched value will be stored in it.

In the third statement an array COUNT is created whose first index is 3 and last index is 9. All elements are initialized to 0.

Fourth statement reads a line from the input. At the end of the input, the control the statement fails and the control goes to DONE.

In the fifth statement WORDPAT is matched in the LINE and match is replaced by the NULL string. The matched value is also stored in WORD. When there is no match found, the statement fails and the control goes to READ to read the next line from the input.

In the sixth statement, the size of the word is used as the array index and the value is incremented at that index. If it is less than 3 or greater than 9 then nothing happen. In fact the statement fails in this case but since there is unconditional jump, it goes back to read the next word from the input.

The rest of the statements are used to display the values at each array index. That is, this program simply counts the occurrences of words of lengths between 3 and 9 in the input and displays that number.

Tables
A table data structure maps pairs of associated data objects. Tables have varying lengths and are dynamically extended if necessary.

T = TABLE()

creates a table of unspecified length.

T = TABLE(N)

creates a table that initially has room for N elements

Once created, table can be used in a manner similar to arrays. The only difference is that they are indexed by the key value and not the index number. For example:

T<‘A’> = 1

Associates key ‘A’ with value 1.

SNOBOL Features

- Dynamic typing
- Mixing arithmetic and string operations
- String operations including concatenation
- GOTO control structure
- Overloaded operators
- Space as an operator
- Run-time compilation
- code can be embedded in data, allowing easy run-time extension of programs
- Variable length string
- Array tables and record type objects
- Absence of declaration
- Operator overloading.

Issues

Developers of SNOBOL had no or limited experience in software development. Griswold did not have any prior computing experience. Polonsky had also limited software development skills. This led to a number of issues:

- Crucial aspects of development process, such as documentation, were overlooked.
- The lack of formal documentation led the initial version of SNOBOL to minimal use.
- Another design problem was that developers treated language semantics casually. Such a poor practice hindered other programmers from understanding the meaning of the program.
- Choice of operators
- Poor readability and writability.
- Dynamic typing
- Mixing arithmetic and string operations
- String operations including concatenation
- GOTO control structure
- Overloaded operators. For example, the use of a blank as a concatenation operator caused confusions in coding and maintenance. Users had to be bewared of the proper use of the blank space operator, and not to confuse it with other operators.

To top it all the compiler was not sophisticated. Any string of characters was a legal SNOBOL program, and the compiler would accept everything as long as it was a string. The legend says that Griswold accidentally gave a non-SNOBOL program to the SNOBOL compiler, and the program was successfully compiled!

Modern Programming Languages

Lecture 13-17

Ada Programming Language
An Introduction

© Copyright Virtual University of Pakistan

Design Goals

Ada is a computer programming language originally designed to support the construction of long-lived, highly reliable software systems. Its design emphasizes readability, avoids error-prone notation, encourages reuse and team coordination, and it is designed to be efficiently implementable.
A significant advantage of Ada is its reduction of debugging time. Ada tries to catch as many errors as reasonably possible, as early as possible. Many errors are caught at compile-time by Ada that aren't caught or are caught much later by other computer languages.

Ada programs also catch many errors at run-time if they can't be caught at compile-time (this checking can be turned off to improve performance if desired).

In addition, Ada includes a problem (exception) handling mechanism so that these problems can be dealt with at run-time.

The main design goals of Ada were:

- Program reliability and maintenance
- Military software systems are expected to have a minimum lifetime of 30 years.
- Programming as a human activity
- Efficiency

Hence emphasis was placed on program readability over ease of writing.

© Copyright Virtual University of Pakistan

Ada History

The need for a single standard language was felt in 1975 and the draft requirements were given the code name strawman. Strawman was refined to Woodman and then Tinman in 1976. It was further refined to ironman. At that time proposals were invited for the design of a new language. Out of the 17 proposals received, four were selected and given the code names of green, red, blue, and yellow. Initial designs were submitted in 1978 and red and green short listed on the basis of these designs. Standard requirements were then refined to steelman. The designs were refined further and finally Green was selected in 1979. DoD announced that the language will be called Ada. The 1995 revision of Ada (Ada 95) was developed by a small team led by Tucker Taft. In both cases, the design underwent a public comment period where the designers responded to public comments.

© Copyright Virtual University of Pakistan

Ada Features

The salient features of Ada language are as follows:

- Packages (modules) of related types, objects, and operations can be defined.
- Packages and types can be made generic (parameterized through a template) to help create reusable components.
- It is strongly typed
- Errors can be signaled as exceptions and handled explicitly. Many serious errors (such as computational overflow and invalid array indexes) are automatically caught and handled through this exception mechanism, improving program reliability.
- Tasks (multiple parallel threads of control) can be created and communicate. This is a major capability not supported in a standard way by many other languages.
- Data representation can be precisely controlled to support systems programming.
- A predefined library is included; it provides input/output (I/O), string manipulation, numeric functions, a command line interface, and a random number generator (the last two were available in Ada 83, but are standardized in Ada 95).
- Object-oriented programming is supported (this is a new feature of Ada 95). In fact, Ada 95 is the first internationally standardized object-oriented programming language.
- Interfaces to other languages (such as C, Fortran, and COBOL) are included in the language.

© Copyright Virtual University of Pakistan

The first Example – Ada "Hello World"

```
with Ada.Text_Io;          -- intent to use
use Ada.Text_Io;           -- direct visibility

                           -- the first two statements are kind of include in C

procedure Hello is         -- procedure without parameters is the
                           -- starting point
begin
        Put_Line("Hello World!");
                           -- this statement prints "Hello World" on the output
end Hello;
```

It may be noted that Ada is not case sensitive.

© Copyright Virtual University of Pakistan

Ada Operators

Ada has a rich set of operators. The following table gives a list of these operators and also shown corresponding C++ operators for reference.

Operator	C/C++	Ada
Assignment	=	:=
Equality	==	=
Non Equality	!=	/=
Greater Than	>	>
Less Than	<	<
Greater Than Or Equal	>=	>=
Less Than Or Equal	<=	<=
PlusEquals	+=	
SubtractEquals	-=	
MultiplyEquals	*=	
DivisionEquals	/=	
OrEquals	|=	
AndEquals	&=	
Modulus	%	Mod
Remainder		Rem
AbsoluteValue		Abs
Exponentiation		**
Range		..
Membership		In
Logical And	&&	And
Logical Or	||	Or
Logical Not	!	Not
Bitwise And	&	And
Bitwise Or	|	Or
Bitwise Exclusive Or	^	Xor
Bitwise Not	~	Not
String Concatenation		&

It is important to note that Ada has not included operators like PlusEquals as such operators reduce readability.

© Copyright Virtual University of Pakistan

Operator Overloading

Ada allows a limited overloading of operators. The exception in Ada is that the assignment operator (**:=**) cannot be overridden. It can be overridden in case of inheritance from a special kind of “abstract class”. When you override the equality operator (**=**) you also implicitly override the inequality operator (***/=***).

Ada Types

Ada provides a large number of kinds of data types. Ada does not have a predefined inheritance hierarchy like many object oriented programming languages. Ada allows you to define your own data types, including numeric data types. Defining your own type in Ada creates a new type.

Elementary Types

The elementary Ada type are:

- Scalar Types
- Discrete Types
- Real Types
- Fixed Point Types
- Access Types

© Copyright Virtual University of Pakistan

Discrete Types

Discrete types include Integer types, Modular types, Character types, enumeration types, and Boolean type.

Integer Types

We first look at Signed Integer types. Ada, like other languages, supports signed integers. However, in this languages we can also define our own integer type with a limited set of values as shown in the following example:

type Marks **is range** 0..100;

This defines a type Marks with the property that a variable of this type can only have values between 0 and 100.

We can now create variable with this type as shown below:

finalScore : Marks;

You may also note that the syntax of Ada for variable declaration is different from C. In this case, the type comes after the variable name and is separated by a : from the variable names.

© Copyright Virtual University of Pakistan

Unsigned (Modular) Types

Modular types support modular arithmetic and have wrap around property. This concept is elaborated with the help of the following example:

```
type M is mod 7; -- values are 0,1,2,3,4,5,6

q : m := 6; -- initialization
…
q := q + 2; -- result is 1
```

It is most commonly used as conventional unsigned numbers where overflows and underflows are wrapped around.

```
type Uns_32 is mod 2 ** 32;
```

Remember that twos complement arithmetic is equivalent to mod 2**wordsize.

Character Types

There are two built in character types in Ada

- Simple 8-bit ASCII Characters
- Wide_Character that support 16-bit Unicode/ISO standard 10646.

These are good enough for most purposes, but you can define your own types just like the integer types:

```
type LetterGrades is ('A', 'B', 'C', 'D', 'F', 'I', 'W');
```

This can now be used to declare variables of this type.

Enumeration Types

Just like C, an enumeration type in Ada is a sequence of ordered enumeration literals:

```
type Colors is (Red, Orange, Yellow, Green, Blue, Indigo, Violet);
type State is (Off, Powering_Up, On);
```

It is however different from C in many respects:

1. There is no arithmetic defined for these types. For example:
   ```
   S1, S2 : State;
   S1 := S1 + S2; -- Illegal
   ```

© Copyright Virtual University of Pakistan

2. One can however add/subtract one (sort of increment and decrement) using the Pred and Succ as shown below:

   State'Pred (S1)
   State'Succ (S2)

3. Unlike C, the same symbolic literal can be used in two enumeration types. For example:

   type RainbowColors **is** (Red, Orange, Yellow, Green, Blue, Indigo, Violet);
   type BasicColors **is** (Red, Green, Blue);

The Ada compiler will use the type of the variable in question and there will be no ambiguity.

Enumeration types are used as attributes or properties of different objects.

Boolean Types

Boolean is a predefined enumeration type and has the following definition.

type Boolean **is** (False, True);

Expressions of type Boolean are used in logical statements and are used as conditions in the **if** statements, **while** loops, **exit** statements, etc.

© Copyright Virtual University of Pakistan

Floating Point Types

Like most other languages, Ada also supports floating point types. In this case, we can also explicitly declare the desired precision of the number. For example:

type Double **is digits** 15;

defines a floating point type with 15 decimal digits of precision.

type Sin_Values **is digits** 10 **range** -1.0..1.0;

defines a type with 10 decimal digits of precision and a valid range of values from -1.0 through 1.0.

Ordinary Fixed Point Types

Ada also supports fixed point real numbers that are used for more precise decimal arithmetic such as financial applications. In the Ordinary fixed point type, the distance between values is implemented as a power of 2. For example:

type Batting_Averages **is delta** 0.001 **range** 0.0..1.0;

The type *Batting_Averages* is a fixed point type whose values are evenly spaced real numbers in the range from 0 through 1. The distance between the values is no more than 1/1000. The actual distance between values may be 1/1024, which is 2^{-10}.

Decimal Fixed Point Types

In the case of decimal fixed point types, the distance between values is implemented as a power of 10.

type Balances **is delta** 0.01 **digits** 9 **range** 0.0 .. 9_999_999.99;

The important difference in the definition of a decimal fixed point type from an ordinary fixed point type is the specification of the number of digits of precision.

This example also shows how Ada allows you to specify numeric literals with underscores separating groups of digits. The underscore is used to improve readability.

© Copyright Virtual University of Pakistan

Arrays

Like most other languages, Ada also supports arrays. One major difference between the arrays in Ada and C is that the indexes of arrays in Ada do not start from 0. In fact, the range of indexes to be used can be defined by the programmer as shown below:

type Int_Buffer **is array** (1..10) **of** Integer;

type Normalized_Distribution **is array**(-100..100) **of** Natural;

An array type can also be defined without a specific size. In this case the size is defined at the time of instantiation of a variable of that type. For example:

type String **is array** (Positive **range** <>) **of** Character;

Can now be used to create variable of specific size as shown below:

Last_Name : String(1..20);

Another very interesting feature of Ada is that the array indexes can be of any discrete type. For example if we define Days as below:

type Days **is** (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday);

then we can create an array which uses Days (or any sub-range of Days) as the array indexes as shown below.

type Daily_Sales **is array**(Days) **of** Float;

Now the type Daily_Sales is an array type of size 7 and indexes Monday through Sunday. Also note that these literals will be used as indexes in the statements that reference array elements.

© Copyright Virtual University of Pakistan

Record Definitions

Record types are like structures in C. The following example shows a record type Address with three fields.

```
type Address is record
        Street : Unbounded_String;
        City :    Unbounded_String;
        Postal_Code : String(1..10);
end record;
```

Once defined, it can be used to declare variables of that type as shown below:

My_Address : Address;

The dot operator, '.', is used to access individual fields. This is demonstrated in the example below:

My_Address.Postal_Code := "00000-0000";

© Copyright Virtual University of Pakistan

Discriminated Records

Discriminated records are like union types in C. There are however major differences between C union types and Ada discriminated records. The union type in C is fundamentally unsafe, and therefore unacceptable. For example, consider the following definition:

union (**int**, **float**) puzzle;

Now puzzle has either an **int** or a **float.** But at runtime, the compiler and the environment cannot tell which particular value has been stored currently. So we have to trust the programmer. In Ada we are short on trust. The whole philosophy is that a programmer is a human and all humans make mistakes. So to make it safer, Ada uses discriminants in a record that tell what type of data is currently stored there. This is shown in the following example:

```
type IF is (Int, Float);
type Int_Float (V : IF := Int) is record
  case V is
    when Int => Int_Val : Integer;
    when Float => Float_Val : Float;
  end case;
end record;
```

Now the value of the discriminant V shows what type is currently present. Variables of a discriminated type are referenced as shown in the following example:

```
Puzzle : Int_Float;
…
Puzzle := (Float, 1.0);
F := Puzzle.Float_Val;          -- OK
I := Puzzle.Int_Val;            -- type mismatch - raise exception
…
Puzzle.V := SomeIntValue;       -- not allowed!
```

Discriminated records can be used for making subtypes

```
subtype PuzzleI is puzzle (Int);          -- this type only holds int values
```

These can also be used to specify array dimensions:

```
Subtype Vlen is Integer range 1 .. 10;
type Vstr (Vlen : Integer := 0) is record
  Data : String (1 .. Vlen);
end record;
VV : Vstr := (5, “hello”);
```

© Copyright Virtual University of Pakistan

Access Types

An *access* type roughly corresponds to a C++ pointer.

type Address_Ref **is access** Address;

```
A_Ref := new Address;
A_Ref.Postal_Code := "94960-1234";
```

Note that, unlike C, there is no notational difference for accessing a record field directly or through an access value.

To refer to the entire record accessed by an access value use the following notation:

```
Print(A_Ref.all);
```

© Copyright Virtual University of Pakistan

Statement Forms

Assignment statement

Like all imperative languages, Ada also supports the assignment statement. However, in Ada the assignment is *not* an expression like C. The syntax of the assignment statement is as follows:

Variable := *expression*;

Note that, in Ada, ':=' is used as the assignment operator whereas '=' is used as assignment operator in C.

© Copyright Virtual University of Pakistan

If Statement

The Ada if statement is fully blocked. In fact, all Ada control structures are fully blocked. That means, we do not have two separate forms if we need only one statement to be executed if the condition for the if statement is true and if we want more than one statements. The body of the if statements is always in side a a complete block that starts with the **then** keyword and end with **end if** as shown below:

```
if condition then
        -- statement(s)
end if;
```

If an if statement has multiple else parts with separate conditions, then Ada uses the keyword **elsif** to indicate that. The **else** part is optional and comes at the end. Each **if** statement must have a corresponding **end if** in it. The following examples demonstrate this concept.

```
if condition1 then
    -- statement(s)
elsif condition2 then
    if condition3 then
        -- statements
    end if;
    -- statement(s)
…
else
    -- statement(s)
end if;
```

```
if condition1 then
    -- statements
elsif condition2 then
    if condition3 then
        -- statements
    end if;
    -- statements
…
else if condition4 then
    -- this is a new if statement and hence must have its own end if
    -- statements
  end if;
    -- statements
end if;
```

© Copyright Virtual University of Pakistan

Case Statements

Case statement is like the switch statement in C, but as compared to C, it is safer and more structured. Its syntax is as shown below:

case *expression* **is**
    when *choice list* =>
        sequence-of-statements
    when *choice list* =>
        sequence-of-statements
    when others =>
        sequence-of-statements
end case;

As opposed to C, the case statement is fully structured; there is no equivalent of the break statement and the control does not fall through to the next choice after executing the set of statements for the selected choice. The choice _list can have more than one values specified in the form of a range specified with the .. operator like 1..10, discrete values separated by | such as a | e | i | o | u, or a combination of both. The following example elaborates this concept.

```
case TODAY is
    when MON .. THU =>
        WORK;
    when FRI =>
        WORK;       PARTY;
    when SAT | SUN =>
        REST;
end case;
```

All values in **when** branches must be static and all possible values of expression must be included exactly once. In Ada, the **case** expression must match one of the choices given in the **when** clause as an exception will be raised as run time if there is no match. **when others** => is like **default** in C and is used to cover all the rest of the choices not mentioned in the above when clauses.

© Copyright Virtual University of Pakistan

Looping

There are three types of loops in Ada:
1. Simple Loop – unconditional infinite loop
2. While loop
3. For Loop

Simple Loop

A simple loop is used to signify a potentially infinite loop. These loops are typically used in operating systems and embedded systems. Its syntax is as follows:

```
loop
        -- Loop body goes here
end loop;
```

The **exit** statement can be used to come out of the loop as shown below.

```
loop
        -- Loop body goes here
        exit when condition;
end loop;
```

It is equivalent to the following code segment.

```
loop
        -- Loop body goes here
        if condition then
                exit;
        end if;
end loop;
```

The **exit** statement is like the **break** statement in C but there are certain differences and will be discussed later.

© Copyright Virtual University of Pakistan

While and For Loops

Ada has only the pre-test while loop and does not support the post-test loop like the **do-while** statement in C. The **while** statement in Ada has the following structure.

while *condition* **loop**
    -- Loop body goes here
end loop;

The semantics of the **while** statement are exactly the same as its counterpart in C.

The **for** statement in Ada is however quite different here. In C, the **for** statement is virtually the same as the **while** statement as the condition in the C **for** statement is checked in each iteration and the number of iterations therefore cannot be determined upfront.

In Ada **for** statement, the number of iterations are fixed the first time the control hits the **for** statement and is specified by a range just like the choice_list in the **case** statement as shown below:

for variable **in** low_value .. high_value **loop**
    -- Loop body goes here
end loop;

The value of the loop variable can be used but cannot be changed inside the loop body. The loop counting can be done in the reverse order as well. This is shown below.

for variable **in reverse** high_value .. low_value **loop**
    -- Loop body goes here
end loop;

© Copyright Virtual University of Pakistan

Exit statement

The **exit** statement can be used with our without a **when** condition as mentioned in the case of the loop statement.
exit;
exit when condition;

It can only be used in a loop. It can use labels and can be used to specify the specific loop to exit as shown below.

```
Outer : loop
  Inner : loop
        …
        exit Inner when Done;
  end loop Inner;
end loop Outer;
```

Block Statement

Block statement in Ada is very similar to a block in C. It can be used anywhere and has the following structure:

```
declare          -- declare section optional
  declarations
begin
  statements
exception        -- exception section optional
  handlers
end;
```

Return statement

The **return** statement can only be used in subprograms and has the following form:

```
return;                    -- procedure
return expression;         -- function
```

Function must have at least one **return.**

Raise statement

Raise statement is like throw statement in C++ and is used to raise exception as shown below:

raise *exception-name*;

© Copyright Virtual University of Pakistan

Declaring and Handling Exceptions

Ada was the first language to provide a structured exception handling mechanism. C++ exception handling is very similar to Ada. In Ada, the programmer can declare, raise, and handle exception.

Declaring an exception

Exceptions are declared just like any other variable before they can be raised. This is shown in the following example:

Error, Disaster : **exception**;

Raising an exception

Like **throw** in C++, an Ada programmer can explicitly raise an exception and it strips stack frames till a handler is found.

raise Disaster;

Handling an exception

Like C++, an exception handling code may be written at the end of a block.

Anywhere we have **begin-end**, we can also have an exception handling code. The structure of exception handling part is shown in the following example.

```
begin
  statements
exception
  when exception1 => statements;
        when exception2 => statements;
end;
```

© Copyright Virtual University of Pakistan

Subprograms

In Ada, there are two types of subprograms: procedures and functions. Unlike C, we can also have nesting of subprograms. A subprogram defined inside another subprogram an be used by its parent, children, or sibling subprograms only.

This is demonstrated in the following example:

```
Procedure My_procedure(x, y : IN OUT integer) is
        …
        function aux_func(a : Integer) return integer is
                Temp : float;
        begin
                …
                return integer(temp) * a;  -- retrun temp * a is not allowed
        end aux_func;
        …
begin                   -- begin of My_procedure
        …
        aux_func(x);
        …
end My_procedure;
```

© Copyright Virtual University of Pakistan

Packages

The primary structure used for encapsulation in Ada is called a *package*. Packages are used to group data and subprograms. Packages can be hierarchical, allowing a structured and extensible relationship for data and code. All the standard Ada libraries are defined within packages.

Package definitions usually consist of two parts: package specification and package body. Package bodies can also declare and define data types, variables, constants, functions, and procedures not declared in the package specification. In the following example, a STACK package is defined. We first the have the package specification as follows:

```
package STACK is
        procedure PUSH (x : INTEGER);
        function POP return INTEGER;
end STACK;
```

The body of the package is shown below.

```
package body STACK is
        MAX: constant := 100;
        S : array (1..MAX) of  INTEGER;
        TOP : INTEGER range 0..MAX;

        procedure PUSH (x : INTEGER) is
        begin
                TOP := TOP + 1;
                S(TOP) := x;
        end PUSH;

        function POP return integer is
                TOP := TOP - 1;
                return S(TOP + 1);
        end POP;
begin
        TOP := 0;
end STACK;
```

© Copyright Virtual University of Pakistan

Once defined, a package can be used in the client program as follows:

```
with STACK;        use STACK;
procedure myStackExample is
        I, O : integer;
begin
        …
        push(i);
        …
        O := pop;
        …
end myStackExample;
```

© Copyright Virtual University of Pakistan

Object-Orientation – The Ada Way

Ada provides tools and constructs for extending types through inheritance. In many object oriented languages the concept of a class is overloaded. It is both the unit of encapsulation and a type definition. Ada separates these two concepts. Packages are used for encapsulation and Tagged types are used to define extensible types.

Tagged Type

A tagged type is like a record which can be used to declare objects. Following is an example of a tagged type:

```
type Person is tagged record
        Name : String(1..20);
        Age : Natural;
end record;
```

Such a type definition is performed in a package. Immediately following the type definition must be the procedures and functions comprising the *primitive operations* defined for this type. *Primitive operations* must have one of its parameters be of the tagged type, or for functions the return value can be an instance of the tagged type. It allows you to overload functions based upon their return type. It may be noted that C++ and Java do not allow you to overload based upon the return type of a function.

Extending Tagged Types

Just like classes in C++, the tagged type can be extended by making a *new* tagged record based upon the original type as shown below:

```
type Employee is new Person with record
        Employee_Id : Positive;
        Salary : Float;
end record;
```

In this example, type *Employee* inherits all the fields and primitive operations of type *Person*.

© Copyright Virtual University of Pakistan

Generics

Generics are like templates in C++ and allow parameterization of subprograms and packages with parameters which can be types and subprograms as well as values and objects. The following example elaborates the concept of generics.

```
generic
        MAX : positive;
        type item is private;
package STACK is
        procedure PUSH (x : item);
        function POP return item;
end STACK;
```

Note that the type of the item is left unspecified. The corresponding body of STACK package is as follows:

```
package body STACK is
        S : array (1..MAX) of  item;
        TOP : INTEGER range 0..MAX;

        procedure PUSH (x : item) is
        begin
        …
        end PUSH;

        function POP return item is
        …
        end POP;
begin
        TOP := 0;
end STACK;
```

Once we have the generic definition, we can instantiate our own STACK for the given type which is **integer** in the following example.

```
declare
        package myStack is new STACK(100, integer);
        use myStack;
begin
        …
        push (i);
        …
        O := pop;
        …
end;
```

© Copyright Virtual University of Pakistan

Concurrency

Ada Tasking allows the initiation of concurrent tasks which initiates several parallel activities which cooperate as needed. Its syntax is similar to packages as shown below:

```
task thisTask is
        …-- interfaces
end thisTask;

task body thisTask is
        …
end thisTask;

task simpleTask;   -- this task does not provide an interface to other tasks
```

Here is a very simple example that demonstrates the concept of parallel processing. In this example we specify COOKING a procedure composed of three steps.

```
procedure COOKING is
begin
        cookMeat;
        cookRice;
        cookPudding;
end COOKING;
```

As these steps can be carried out in parallel, we define tasks for these tasks as shown below:

```
procedure COOKING is
        task cookMeat;
        task body cookMeat is
        begin
                -- follow instructions for cooking meat
        end cookMeat;

        task cookRice;
        task body cookRice is
        begin
                -- follow instructions for cooking rice
        end cookRice;

begin
        cookPudding;
end COOKING;
```

© Copyright Virtual University of Pakistan

Other Features

Ada is a HUGE language and we have looked at may be only 25-30% of this language. It is to be remembered that the main objective of the Ada design was to incorporate the contemporary software engineering knowledge in it. In general, software development time is: 10% design, 10% coding, 60% debug and 20% test. Last 80% of the project is spent trying to find and eliminate mistakes made in the first 20% of the project. Therefore Ada promised to spend 20% time in design, 60% in coding, 10% in debug, and 10% in testing. Therefore cutting the entire cycle time in half!

I would like to finish with a quote from Robert Dewar:

> *When Roman engineers built a bridge, they had to stand under it while the first legion marched across. If programmers today worked under similar ground rules, they might well find themselves getting much more interested in Ada.*

© Copyright Virtual University of Pakistan

Modern Programming Languages

Lecture 18-21

LISP Programming Language
An Introduction

Functional Programming Paradigm and LISP

Functional programming is a style of programming that emphasizes the evaluation of expressions, rather than execution of commands. The expressions in these languages are formed by using functions to combine basic values. A functional language is a language that supports and encourages programming in a functional style.

LISP is a representative language of this style of programming.

Lisp stands for “LISt Process”. It was invented by John McCarthy in 1958 at MIT. It has simple data structure (atoms and lists) and makes heavy use of recursion. It is an interpretive language and has many variations. The two most popular versions are Scheme and Common Lisp (de facto industrial standard). It is the most widely used AI programming language.

Valid Objects

A LISP program has two types of elements: atoms and lists.

Atoms:

Atoms include numbers, symbols, and strings. It supports both real numbers and integers.

symbols: a symbol in LISP is a consecutive sequence of characters (no space). For example **a, x,** and **price-of-beef** are symbols. There are two special symbols: **T** and **NIL** for logical true and false.

strings: a string is a sequence of characters bounded by double quotes. For example **"this is red"** is a string.

S-expression
An **S-expression** (S stands for symbolic) is a convention for representing data or an expression in a LISP program in a text form. It is characterized by the extensive use of prefix notation with explicit use of brackets (affectionately known as Cambridge Polish notation).

S-expressions are used for both code and data in Lisp. S-expressions were originally intended only as machine representations of human-readable representation of symbols, but Lisp programmers soon started using S-expressions as the default notation.

S-expressions can either be single objects or atoms such as numbers, or lists.

Lists

List is the most important concept in LISP. A list is a group of atoms and/or lists, bounded by **(** and **)**. For example **(a b c)** and **(a (b c))** are two lists. In these examples the list **(a b c)** is a list of atoms **a, b,** and **c,** whereas **(a (b c))** is a list of two elements, the first one an atom **a**, and the second one a list **(b c)**.

Top elements of a list

The first-level elements in LISP are called top-level elements. For example top elements of list (a b c) are a, b, and c. Similarly, top elements of list (a (b c)) are a and (b c).

An empty list is represented by **nil.** It is the same as **()**.

Function calls

In LISP, a function and a function call is also a list. It uses prefix notation as shown below:

(function-name arg1 ... argn)

A function call returns function value for the given list of arguments. Functions are either provided by LISP function library or defined by the user.

Following are some examples of function calls. Note that the first symbol is the name of the function and the rest are its arguments.

```
>(+ 1 3 5)
9
```

Note that the arithmetic functions may take more than two arguments.

```
>(/ 3 5)
3/5
```

When we use two integers in division, the answer is converted into fraction.

```
>(/ 3.0 5)
0.59999999999999998
```

Note the difference between integer and real division.

```
>(sqrt 4)
2
```

Evaluation of S-expression

S-expressions are evaluated as follows:

1) Evaluate an atom.
   - numerical and string atoms evaluate to themselves;
   - symbols evaluate to their values if they are assigned values, return Error, otherwise;
   - the values of T and NIL are themselves.

2) Evaluate a list
   - evaluate every top element of the list as follows, unless explicitly forbidden:
   - the first element is always a function name; evaluating it means to call the function body;
   - each of the rest elements will then be evaluated, and their values returned as the arguments for the function.

Examples:

1.
```
>(+ (/ 3 5) 4)
23/5
```

The first element is + so we make a call to + function with (/ 3 5) and 4 as arguments. Now (/ 3 5) is evaluated and the answer is 3/5. After that 4 is evaluated to itself and thus returns 4. So, 3/5 and 4 are added to return 23/5.

2.
```
>(+ (sqrt 4) 4.0)
6.0
```

This example is similar to the first example.

3.
```
>(sqrt x)
Error: The variable
        X is unbound.
```

This code generates an error as the symbol x is not associated with any value and thus cannot be evaluated.

setq, set, and **setf** are used to assign a value to a symbol.

For example, the following code assigns a value 3 to symbol x.

```
>(setq x 3.0)
 3.0
```

setq is a special form of function (with two arguments). The first argument is a symbol which will not be evaluated and the second argument is a S-expression, which will be evaluated. The value of the second argument is assigned to be the value of the first argument.

To forbid evaluation of a symbol a **quote** or **'** is used.

For example

```
>(quote x)
 x
```

Similarly if we have

```
>(setq x 3.0)
 3.0
```

and then we do

```
>(setq y x)
3.0
```

then y will also have a value 3.0

But if we use a quote as shown below

```
<(setq y 'x)
x
```

then x will not be evaluated and y will get the value x instead of 3.0.

Functions

There are many built-in function is LISP. This includes math functions as well as functions for manipulating lists. The math functions include:

– +, -, *, /, exp, expt, log, sqrt, sin, cos, tan, max, min

with the usual meanings.

List Constructors
Lists can be constructed (created and extended) with the help of three basic functions. These are **cons, list** and **append.**

cons takes two arguments. The first argument is a symbol and the second one is a list. It inserts the first argument in front of the list provided by the second argument. It is shown in the following example:

```
>(cons 'x L)       ; insert symbol x at the front of list L, which is
(X A B C)          ; (A B C)
```

list takes arbitrary number of arguments and makes a list of these arguments as shown below:

```
>(list 'a 'b 'c)     ; making a list with the arguments as its elements
 (A B C)             ; if a, b, c have values A, B, C, then (list a b c) returns list (A B C)
```

append takes two lists as arguments and appends one list in front of the other as shown below:

```
>(append '(a b) '(c d))
(A B C D)          ; appends one list in front of another
```

List Selectors

In order to select elements from a list, selectors functions are used. There are two basic selector functions known as **first** (or **car**) and **rest** (or **cdr**). The rest can be build with the help of these functions.

first (or **car**) takes a list as an argument and returns the first element of that list as shown in the following examples:

```
>(first '(a s d f))
 a

>(first '((a s) d f))
 (a s)
```

```
>(setq L '(A B C))
(A B C)

>(car L)
A
```

rest (or **cdr**) takes a list as its argument and returns a new list after removing the first element from the list. This is demonstrated in the following examples:

```
>(rest '(a s d f))
(s d f).

>(rest '((a s) d f))
 (d f)

>(rest '((a s) (d f))
 ((d f))

>(setq L '(A B C))
(A B C)

>(cdr L)
(B C)
```

Some of the other useful list manipulation functions are:

```
>(reverse L)     ; reverses a list
   (C B A)
```

and

```
>(length L)      ; returns the length of list L
 3
```

Predicates

A predicate is a special function which returns NIL if the predicate is false, T or anything other than NIL, otherwise. Predicates are used to build Boolean expressions in the logical statements.

The following comparative operators are used as functions for numerical values and return a T or NIL. **=**, **>**, **<**, **>=**, **<=**;

For example:

- (= (+ 2 4) (* 2 3))
- T

- (> (- 5 2) (+ 3 1))
- NIL

For non numeric values you can only check for equality using **equal** or **eq.**

Some other useful predicates are listed below:
atom: test if x is an atom
listp: test if x is a list

Also **number**, **symbolp**, **null** can be used to test whether the operand is a number, symbol, or a null value.

Set Operations

A list can be viewed as a set whose members are the top elements of the list. The list membership function is a basic function used to check whether an element is a member of a list or not. It is demonstrated with the help of the following code:

```
>(setq L '(A B C))

>(member 'b L) ; test if symbol b is a member (a top element) of L
(B C)          ; if yes, returns the sublist of L starting at the
                ; first occurrence of symbol b

>(member ‘b (cons 'b L))
(B A B C)
```

Note here that the mathematical definition of a set is different from the LISP definition. In Mathematics, a symbol cannot be repeated in a set whereas in LIST there is no such restriction.

If an element is not present in the list, it returns NIL as shown below.

```
>(member x L)
NIL            ; if no, returns NIL
```

Some of the other set operations are defined as below:

```
>(union L1 L2)            ; returns the union of the two lists
>(intersection L1 L2)     ; returns the intersection of the two lists
>(set-difference L1 L2)   ; returns the difference of the two lists
```

Defining LISP functions

In LISP, defun is used to write a user-defined function. Note that different dialects of LISP may use different keywords for defining a function. The syntax of defun is as below:

(**defun** func-name (arg-1 ... Arg-n) func-body)

That is, a function has a name, list of arguments, and a body. A function returns the value of last expression in its body. This concept is demonstrated with the help of the following example:

```
>(defun y-plus (x) (+ x y))          ;definition of y-plus
```

This function adds x to y where x is the parameter passed to the function and y is a global variable since it is not defined inside the function. It work in the following manner:

```
>(setq y 2)
>(y-plus 23)
25
```

With this we introduce the concept of local and global variables. Local variables are defined in function body. Everything else is global.

Conditional control: if, when and **cond**

LISP has multiple conditional control statements. The set includes **if**, **when**, and **cond**. In the following pages we study these statements one by one.

if statement

The **if** statement has the following syntax:

```
 (if <test> <then> <else>)
```

That is, an if statement has three parts: the **test**, the **then** part, and the **else** part. It works almost exactly like the if statement in C++. If the test is TRUE then the **then** part will be executed otherwise the **else** part will be executed. If there is no else part then if the test is not true then the if statement will simply return NIL. Here is an example that shows the **if** statement:

```
> (setq SCORE 78)
> 78
> (if (> score 85) ‘HIGH
                   (if (and (< score 84) (> score 65)) ‘MEDIUM ‘LOW))
```

> MEDIUM

In the above **if** statement, the **then** part contains ‘HIGH and the **else** part is another **if** statement. So, with the help of nested **if** statements we can develop code with multiple branches.

cond statement

The **cond** statement in LISP is like the switch statement in C++. There is however a slight different as in this case each clause in the **cond** requires a complete Boolean test. That is, just like multiple else parts in the **if** statement where each needs a separate condition. Syntax of **cond** is as shown below:

>(**cond** (<test-1> <action-1>)
(<test-2> <action-2>)
...
(<test-k> <action-k>))

Each (<test-i> <action-i>) is called a clause. If test-i (start with i=1) returns T (or anything other than NIL), this function returns the value of action-i, else, it goes to the next clause. Usually, the last test is T, which always holds, meaning otherwise. Just like the **if** statement, **cond** can be nested (action-i may contain (cond ...))

This statement is explained with the help of the following example:

> (setf operation ‘area L 100 W 50)
> 50

> (cond ((eq operation ‘perimeter) (* 2 (+ L W)))
(eq operation ‘area) (* L W))
(t ‘i-do-not-understand-this)
)
)
> 5000

This program has three clauses. The first one checks whether the operation is the calculation of the perimeter or not. In this case it uses the length and width to calculate the perimeter. The control will come to the second clause if the first test is evaluated to be false or NIL. The second clause checks if the operation is about area and if it is true then it will calculate area. The third clause is the default case which will always be true. So if the first two tests fail, it will print **‘i-do-not-understand-this**

Recursion

Recursion is the main tool used for iteration. In fact, if you don't know recursion, you won't be able to go too far with LISP. There is a limit to the depth of the recursion, and it depends on the version of LISP. Following is an example of a recursive program in LISP. We shall be looking at a number of recursive functions in the examples.

```
(defun power (x y)
        (if (= y 0) 1 (* x (power x (1- y)))))
```

This function computes the power of x to y. That is, it computes x^y by recursively multiplying x with itself y number of times.

So

```
>(power 3 4)
81
```

Let us look at another example. In this example, we compute the length of a list, that is, we compute the number of elements in a list. The function is given as follows:

```
(defun length (x)
        (if (null x) 0
           (+ length (rest x) 1)
   )
)
```

```
> (Length '(a b c d))
> 4
```

Here are some more examples: The first function determines whether a symbol is present in a list or not and the second function computes the intersection of two lists. These functions are given below:

```
(defun member (x L)              ; determine whether x is in L or not
    (cond ((null L) nil)         ; base case 1: L is empty
        ((equal x (car L)) L)    ; base case 2: x=first(L)
        (t (member x (cdr L)))   ; recursion: test if x is in rest(L)
    ))

  (defun intersection (L1 L2)    ; compute intersection of two lists
    (cond ((null L1) nil)
          ((null L2) nil)
          ((member (car L1) L2)
```

```
            (cons (car L1) (intersection (cdr L1) L2)))
        (t (intersection (cdr L1) L2))
    ))
```

It may be noted that the intersection function is different from the mathematical definition of set intersection. In this case the function looks at all the elements of the first list, one at a time, and checks whether that element is present in the second list. You may recall from our earlier discussion that our definition of set if different from the mathematical definition of a set where duplications are not allowed. This concept is elaborated with the help of the following examples where you pass same lists in different order and get different results.

```
> (intersection '(a b c) '(b a b c))
> (a b c)

> (intersection '(b a b c) '(a b c))
> (b a b c)
```

Following is yet another example to compute the set difference of two sets.

```
(defun set-difference (L1 L2)
      (cond ((null L1) nil)
            ((null L2) L1)
            ((not (member (car L1) L2))
              (cons (car L1) (set-difference (cdr L1) L2)))
            (t (set-difference (cdr L1) L2))
      ))
```

Iteration: **dotimes** and **dolist**

Apart from recursion, in LISP we can write code involving loops using iterative non-recursive mechanism. There are two basic statements for that purpose: **dotimes** and **dolist.** These are discussed in the following paragraphs.

dotimes

dotimes is like a counter-control for loop. Its syntax is given as below:

(**dotimes** (count n result) body)

It executes the **body** of the loop n times where count starts with 0, ends with n-1.

The **result** is optional and is to be used to hold the computing result. If ***result*** is given, the function will return the value of ***result***. Otherwise it returns NIL. The value of the **count** can be used in the loop body.

dolist

The second looping structure is **dolist.** It is used to iterate over the list elements, one at a time. Its syntax is given below:

(**dolist** (x L result) body)

It executes the **body** for each top level element x in L. x is not equal to an element of L in each iteration, but rather x takes an element of L as its value. The value of x can be used in the loop body. As we have seen in the case of **dotimes,** the **result** is optional and is to be used to hold the computing result. If ***result*** is given, the function will return the value of ***result***. Otherwise it returns NIL.

To understand these concepts, let us look at the following examples:

```
> (setf cold 15 hot 35)
> 35
```

The following function use **dolist** to computes the number of pleasant days, that is, between cold and hot.

```
> (defun count-pleasant (list-of-temperatures)
                    (let ((count-is 0))                    ; initialize
                      (dolist (element my-list count-is)
                            (when (and (< element hot)
                                       (> element cold))
                              (setf count-is (+ count-is 1))))))
```

```
> (count-pleasant '(30 45 12 25 10 37 32))
> 3
```

Here is a very simple example which uses **dotimes:**

```
> (defun product-as-sum (n m)
                  (let ((result 0))
                       (dotimes (count n result)
                          (setf result (+ result m)))))
```

```
> (product-as-sum 3 4)
> 12
```

Property Lists

Property lists are used to assign/access properties (attribute-value pairs) of a symbol. The following functions are used to manipulate a property list.

To assign a property: (**setf** (**get** object attribute) value)
To obtain a property: (**get** object attribute)
To see all the properties; (**symbol-plist** object)

Here is an example that elaborates this concept:

```
>(setf (get 'a 'height) 8)  ; cannot use "setq" here
                            ; setting the height property of symbol a.
8

>(get 'a 'height)  ; setting the height property of symbol a.
8

>(setf (get  'a 'weight) 20)
                            ; setting the weight property of symbol a.
20
```

Now we list all properties of **a**:

```
>(symbol-plist 'a)
(WEIGHT 20 HEIGHT 8)
```

We can remove a property by using the **remprop** function as shown below:

```
> (remprop 'a 'WEIGHT)
T

 >(symbol-plist 'a)
(HEIGHT 8)

> (remprop 'a 'HEIGHT)
T

 >(symbol-plist 'a)
NIL
```

We can use the property list to build more meaningful functions. Here is one example:

Assume that if the name of a person's father is known, the father's name is given as the value of the father property. We define a function GRANDFATHER that

returns the name of a person's paternal grandfather, if known, or NIL otherwise. This function is given below:

```
(defun grandfather (x)
        (if (get x 'father)
                (get (get x 'father) 'father)
                nil))
```

We now make a list of the paternal line:

```
(defun lineage (x)
        (if  x
                (cons (x (lineage (get x 'father))))))
```

The following would trace the family line from both father and mother's side:

```
(defun ancestors (x)
        (if x
                (cons x (append (ancestors (get x 'father))
                                (ancestors (get x 'mother))))))
```

Arrays

Although the primary data structure in LISP is a list, it also has arrays. These are data type to store expressions in places identified by integer indexes. We create arrays by using the **linear-**array function as shown below:

```
(setf linear-array (make-array '(4)))
```

This statement creates a single dimension array of size 4 by the name of **linear-array** with indices from 0 to 3.

Here is an example of a two-dimensional array:

```
(setf chess-board (make-array '(8 8)))
```

Once created, we can store data at the desired index as follows:

```
(setf (aref chess-board 0 1) 'WHITE-KNIGHT)
(setf (aref chess-board 7 4) 'BLACK-KING)
```

Here, **aref** is the array reference. The above statements say that store symbol **WHITE-KNIGHT** at **chess-board** position **0 1** and store **BLACK-KING** at position **7 4**.

aref can be used to access any element in the array as shown below:

(aref chess-board 0 2)
WHITE-BISHOP

(aref chess-board 7 2)
BLACK-BISHOP

What made Lisp Different?

LISP was one of the earliest programming language. It was designed at MIT for artificial intelligence and it has since been the defacto standard language for the AI community, especially in the US.

LISP program composed of expression. They are in fact trees of expression, each of which returns a value. It has Symbol types: symbols are effectively pointer to strings stored in a hash table. One very important aspect of LISP is that the whole language is there. That is, there is no real distinction between read-time, compile-time and runtime. You can compile or run code while reading, read or run code while compiling, and read or compile code at runtime. That is, programs are expressed directly in the parse trees that get build behind the scenes when other languages are parsed, and these trees are make of lists, which are Lisp date structures. This provides a very powerful feature allowing the programmer *to write programs that write programs*.

From a programming language design point of view, LISP was the first to introduce the following concepts:

- Conditionals: such as if-then-else construct.
- Function types: where functions are just like integers and strings
- Recursion: first language to support it.
- Dynamic typing: all variable are pointers.
- Garbage-Collection.
- Programs composed of expressions.
- A symbol type.
- The whole program is a mathematical function

Modern Programming Languages

Lecture 22-26

PROLOG - Programming in Logic

An Introduction

PROLOG stands for PROgramming in LOGic and was design in 1975 by Phillippe Roussell. It is a declarative programming language and is based upon Predicate Calculus. A program in PROLOG is written by defining predicates and rules and it has a built-in inference mechanism. Unfortunately, there is no effective standardization available.

PROLOG Paradigm

As mentioned earlier, PROLOG draws inferences from facts and rules. It is a declarative language in which a programmer only specifies facts and logical relationships. It is non-procedural. That is we only state "what" is to be done and do not specify "how" to do it. That is, we do not specify the algorithm. The language just executes the specification. In other words, program execution is carried out as a theorem proving exercise. It is similar to SQL used in databases with automated search and the ability to follow general rules.

One of the main features of this language is its ability to handle and process symbols. Hence it is used heavily in AI related applications. It is an interactive (hybrid compiled/interpreted) language and its applications include expert systems, artificial intelligence, natural language understanding, logical puzzles and games.

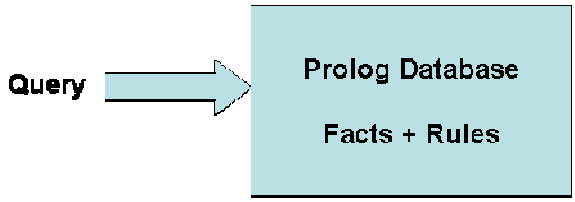
PROLOG Programming

PROLOG programming follows the following steps:

- Declaring some facts about objects and their relationships
- Defining some rules about objects and their relationships
- Asking questions about objects and their relationships

The PROLOG Programmer loads facts and rules into the database and makes queries to the database to see if a fact is in the database or can be implied from the facts and rules therein.

This is depicted in the following diagram.



Facts

- Facts are used to represent unchanging information about objects and their relationships.
- Only facts in the PROLOG database can be used for problem solving.
- A programmer inserts facts into the database by typing the facts into a file and loading (consulting) the file into a running PROLOG system.

Queries

Queries are used to retrieve information from the database. A query is a pattern that PROLOG is asked to *match* against the database and has the syntax of a compound query. It may contain variables. A query will cause PROLOG to look at the database, try to find a match for the query pattern, execute the body of the matching head, and return an answer.

Logic Programming

Logic programming is a form of declarative programming. In this form, a program is a collection of *axioms* where each axiom is a *Horn clause* of the form:

H :- B1, B2, ..., Bn.

where *H* is the *head term* and *Bi* are the *body terms*, meaning *H* is true if all *Bi* are true. A user of the program states a *goal* (a theorem) to be proven. The logic programming system attempts to find axioms using *inference steps* that imply the goal (theorem) is true. The basic proof technique is Modus Ponens i.e.

A -> B (If A is true then B is also true but it does not mean that if B is true then A is also true)

If Fact is given that

A is true

Then

B is also true

Resolution

To deduce a goal (theorem), the logic programming system searches axioms and combines sub-goals. For this purpose we may apply forward (from fact to goal) or backward (from goal to fact) chaining. For example, given the axioms:

C :- *A*, *B*.
D :- *C*.

Given that A and B are true, *forward chaining* (from fact to goal) deduces that *C* is true because *C* :- *A*, *B* and then *D* is true because *D* :- *C*.

Backward chaining (from goal to fact) finds that *D* can be proven if sub-goal *C* is true because *D* :- *C*. The system then deduces that the sub-goal is *C* is true because *C* :- *A*, *B*. Since the system could prove *C* it has proven *D*.

Prolog Uses backward chaining because it is more efficient than forward chaining for larger collections of axioms.

Examples

Let us look at some examples of facts, rules, and queries in Prolog.

Facts

Following table shows some example of facts.

English	PROLOG
"A dog is a mammal"	isa(dog, mammal).
"A sparrow is a bird"	isa(sparrow, bird).

Rules

Here are some examples of rules.

English	PROLOG
"Something is an animal	animal(X) :- isa(X, bird).
if it is a mammal or a bird"	animal(X) :- isa(X,mammal).

Queries

An now some queries:

English	PROLOG
"is a sparrow an animal?" answer: "yes"	?- animal(sparrow). yes
"is a table an animal?" answer: "no"	?- animal(table). no
"what is a dog?" answer: "a mammal"	?- isa(dog, X). X = mammal

----------END OF LECTURE 22----------

PROLOG syntax

Constants

There are two types of constants in Prolog: atoms and numbers.

Atoms:

- Alphanumeric atoms - alphabetic character sequence starting with a lower case letter. Examples: apple a1 apple_cart
- Quoted atoms - sequence of characters surrounded by single quotes. Examples: 'Apple' 'hello world'
- Symbolic atoms - sequence of symbolic characters. Examples: & < > * - + >>
- Special atoms. Examples: ! ; [ ] {}

Numbers:

Numbers include integers and floating point numbers.

Examples: 0 1 9821 -10 1.3 -1.3E102.

Variable Names

In Prolog, a variable is a sequence of alphanumeric characters beginning with an upper case letter or an underscore. Examples: Anything _var X _

Compound Terms (structures)

A compound term is an atom followed by an argument list containing terms. The arguments are enclosed within brackets and separated by commas.
Example: isa(dog, mammal)

Comments

In Prolog % is used to write a comment just like // in C++. That is after % everything till the end of the line is a comment.

Important points to note

- The names of all relationships and objects must begin with a lower case letter.
  - For example ***studies, ali, programming***
- The relationship is written first and the objects are enclosed within parentheses and are written separated by commas.
  - For example ***studies(ali, programming)***
- The full stop character '.' must come at the end of a fact.
- Order is arbitrary but it should be consistent.

An Example Prolog Program

This program has three facts and one rule. The facts are stated as follows:

```
rainy(columbo).
rainy(ayubia).
cold(ayubia).
```

The rule is:

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

Some queries and responses:

```
?- snowy(ayubia).
yes

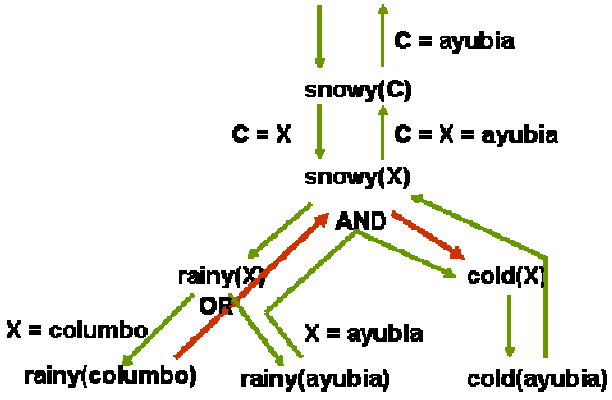
?- snowy(columbo).
no

?- snowy(lahore).
No

?- snowy(C).
C = ayubia
```

Because **rainy(ayubia)** and **cold(ayubia)** are sub-goals that are both true facts in the database, **snowy(X)** with **X=columbo** is a goal that fails, because **cold(X)** fails, triggering *backtracking.*

This inference mechanism based upon back tracking is depicted in the following diagram:



Logic Representation : Propositional Calculus

Propositional Logic is a set of Boolean statements. It involves logic connectives such as $\neg \wedge \vee \Leftrightarrow \Rightarrow$.

Example – a family

If we had the following atomic statements:

p: Ahmed is father of Belal
q: Ahmed is brother of Aslam
r : Aslam is uncle of Belal

Then the following table shows the meanings in English for some logical statements in propositional calculus:

Statement	Logical meaning
$\neg p$	Ahmed is not father of Belal
$p \vee q$	Ahmed is father of Belal or Ahmed is brother of Aslam
$p \wedge q \Rightarrow r$	If Ahmed is father of Belal and brother of Aslam then Aslam is uncle of Belal

Statements in predicate calculus can be mapped almost directly into prolog as shown below:

Predicate Calculus

- Predicates
  - man(Ahmed)
  - father(Ahmed, Belal)
  - brother(Belal, Chand)
  - brother(Chand, Delawar)
  - owns(Belal, car)
  - tall(Belal)
  - hates(Ahmed, Chand)
  - family()
- Formulae
  - $\forall X,Y,Z(man(X) \wedge man(Y) \wedge father(Z,Y) \wedge father(Z,X) \Rightarrow brother(X,Y))$
- Variables
  - X, Y and Z
- Constants
  - Ahmed, Belal, Chand, Delawar and car

Prolog code

```
Predicates
        man(symbol)
        father(symbol, symbol)
        brother(symbol, symbol)
        owns(symbol, symbol)
        tall(symbol)
        hates(symbol, symbol)
        family()
Clauses
        man(ahmed).
        father(ahmed, belal).
        brother(ahmed, chand).
        owns(belal, car).
        tall(belal).
        hates(ahmed, chand).
        family().

        brother(X,Y):-
              man(X),
              man(Y),
              father(Z,Y),
              father(Z,X).
Goal
        brother(ahmed,belal).
```

Sections of Prolog Program

Predicates
Predicates are declarations of relations or rules. They are just like function prototypes (declaration). A predicate may have zero or more arguments. Example of predicates is:

```
man(symbol)
family()
a_new_predicate (integer, char)
```

Clauses
Clauses are definition(s) of Predicate sentence or phrase. It has two types: rules and facts. A rule is a function definition. It may have 0 or more conditions. A fact has all parameters as constants and it cannot contain any condition.

Example – Fact:

```
brother(ahmed, chand).
```

Saying that ahmed and chand are brothers.

Example – Rule:

```
brother(X,Y):-
        man(X),
        man(Y),
        father(Z,Y),
        father(Z,X).
```

It says that two symbols, X and Y, are brothers if X is a man and Y is a man and their father is same.

Goal

- Goal is the objective of the program.
- There can be only and exactly one instance of the goal in a program.
- It is the only tentative section of the program.
- It may be thought as the main() function of prolog.
- The truth value of the goal is the output of the program.
- Syntactically and semantically, it is just another **clause.**
- It may be empty, simple (one goal), or compound (sub-goals).

Examples

1. brother(X,chand). % In this case the output will be the name of Chand's brother.
2. brother(ahmed,chand).% The output will be true or false.
3. brother(X,chand), father(X, Y). % The output will be Chand's and his children.

----------END OF LECTURE 23----------

Prolog Variables

Prolog variable are actually constant placeholders and NOT really variables. That is, a value to a variable can be bound only once and then it cannot be changed. These variables are loosely typed. That is, type of the variable is not defined at compile time. As mentioned earlier, a variable name starts with capital letter or underscore. Here are some examples of variable names:

- brother(ahmed, Ahmed) % ahmed is a constant whereas Ahmed is a variable
- brother(ahmed, _x) % _x is a variable
- Brother(ahmed, X) % X is a variable

Anonymous variable:
The _ is a special variable. It is called the anonymous variable. It is used as a place holder for a value that is not required. For example _ is used as a variable in the following clause:

brother(ahmed, _)

Other Syntactic Elements

Prolog has a number of other syntactic elements. Some of these are listed below:

- Special predicates
  - cut <or> !
  - not(predicate)
- Predefined predicates
  - write(arg1[,arg2[,arg3[,arg4[…..]]]]) % for output
  - nl % new line
  - readint(var) % read integer
  - readchar(var) % read char
  - readln(var) % read line into var
  - … many more related to i/o, arithmetic, OS etc
- Operators
  - Arithmetic
    - + - * div / mod
  - Relational
    - < <= => > = <> ><

PROLOG Lists

In Prolog, a list is a sequence of terms of the form

[t1, t2, t3, t4, ..., tn]

where term ti is the ith element of the list

Examples:
1. [a,b,c] is a list of three elements a, b and c.

2. [[a,list,of,lists], and, numbers,[1,2,3]]
is a four element list, the first and last are themselves lists.

3. [ ] is the 'empty list'. It is an atom not a list data type. This is a fixed symbol. And it will always have the same meaning that's why it is treated as atom not list.

Working with Lists

1. Lists are used to build compound data types.
2. A list can have elements of arbitrary type and size.
3. There is no size limits.
4. Representation in prolog is very concise (short).
5. **head – tail separator**
    a. The vertical bar '|' is used to separate the head and tail of a list as shown below:
        [<head>|<tail>]
    b. In this case, head is an element whereas tail is list of the same sort.
    c. The | has dual purpose in Prolog; it is used for list construction as well as list dismantling.

Here as some examples of lists:

```
[a, bc, def, gh, i]   % a simple list of five elements
[ ]                   % an empty list
[1,2,3]               % a simple list of three elements
[1, 2 | [3] ]         % same as above; a list of three elements with 1 and 2 as head and
                      % [3] as tail. When you join head and tail you get [1 2 3].
[1 | [2, 3] ]         % another way of writing the same thing again. This time the head
                      % is only 1 and the tail is [2 3]. Concatenation of these two gives
                      % [1 2 3].
[1, 2, 3 | [ ] ]      % same thing again but with a different head and tail formation
```

Recursion is the basic tool for writing programs in Prolog. Let us now look at some example programs in Prolog using lists.

Example 1: Calculating the length of a list by counting the first level elements

```
Domains
        list = Integer *
Predicates
        length (list, integer)
Clauses
        length([],0).                       %length of an empty list is zero

        length ([_|Tail], Len):-
                Length (Tail, TailLen),     %getting the tail length
                Len = TailLen + 1.          %the length of list is 1 plus the
                                            % length of tail
Goal
        X = [1,2,3],
        length(X,Len).
```

Example 2: Finding the last and second last element in a list

```
lastElement(X,[X]).                         % if the list has only one element
                                            % then it is the last element
lastElement(X,[_|L]) :- lastElement(X,L).   % recursive step

last_but_one(X,[X,_]).                      % similar to the above example
last_but_one(X,[_,Y|Ys]) :- last_but_one(X,[Y|Ys]).
```

----------END OF LECTURE 24----------

Example 3: Eliminate consecutive duplicates of list elements.

If a list contains repeated elements they should be replaced with a single copy of the element. The order of the elements should not be changed.

Example:

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

Here is the code for this function:

```
compress([],[]).                    % there is nothing to be compressed in
compress([X],[X]).                  % an empty list or a list with only one
                                    % element

compress([X,X|Xs],Zs) :-            % if the first element is repeated, copy
        compress([X|Xs],Zs).        % it once in the new list
```

```
compress([X,Y|Ys],[X|Zs]) :-          % inspect the rest of the list and
        X \= Y,compress([Y|Ys],Zs). % repeat the process recursively
```

Sorting

Remember, in Prolog, we only specify what we want and not how to do it. Here is a classical example of this specification. In the following program segment, in order to sort a list, we only specify what is meant by sorting and not how to do it.

```
sorted ([ ]).                % an empty list or list with only one element
sorted ([X]).                % is already sorted
sorted ([X, Y | list]) :-    % recursive step
        X <= Y,              % a list is sorted if every element in the list
        sorted ([Y | list]). % is smaller than the next element in the list.
```

List Membership

```
member(X, [X|T]).            % X is a member of the list if it is the first
                             % element

member(X, [H|T]) :- member(X, T).
                             % otherwise recursively check it in the rest of
                             % the list
```

subset

```
subset( [ ], Y ).            % The empty set is a subset of every set.
subset( [ A | X ], Y ) :-    % recursive step
        member( A, Y ), subset( X, Y ).
```

intersection

```
% Assumes lists contain no duplicate elements.
intersection( [ ], X, [ ] ).
intersection( [ X | R ], Y, [ X | Z ] ) :-
                member( X, Y ), !, intersection( R, Y, Z ).
intersection( [ X | R ], Y, Z ) :- intersection( R, Y, Z ).
```

union

```
union( [ ], X, X ).
union( [ X | R ], Y, Z ) :- member( X, Y ), !, union( R, Y, Z ).
union( [ X | R ], Y, [ X | Z ] ) :- union( R, Y, Z ).
```

Puzzles- Who owns the zebra

According to an e-mail source, this puzzle originated with the German Institute of Logical Thinking, Berlin, 1981. [It's possible.]

1. There are five houses.
2. Each house has its own unique color.
3. All house owners are of different nationalities.
4. They all have different pets.
5. They all drink different drinks.
6. They all smoke different things.
7. The English man lives in the red house.
8. The Swede has a dog.
9. The Dane drinks tea.
10. The green house is on the left side of the white house.
11. In the green house, they drink coffee.
12. The man who smokes cigarette has birds.
13. In the yellow house, they smoke pipe.
14. In the middle house, they drink milk.
15. The Norwegian lives in the first house.
16. The man who smokes cigar lives in the house next to the house with cats.
17. In the house next to the house with the horse, they smoke pipe.
18. The man who smokes hukka drinks soda.
19. The German smokes bedi.
20. The Norwegian lives next to the blue house.
21. They drink water in the house that is next to the house where they smoke cigar.
22. Who owns the zebra?

Try solving this puzzle on your own. Now imagine writing a computer program to solve it. Which was more difficult?

This is an example of a completely specified solution which doesn't appear to be specified at all. The constraints are such that the answer is unique, but they are stated in such a way that it is not at all obvious (to this human, at least) what the answer is.

In Prolog, we could express each of these specifications, then let Prolog's search strategy (database search engine, automated theorem prover) search for a solution. We don't have to worry about *how* to solve the problem -- we only have the specify *what* is to be solved.

```
%  -*-prolog-*-
%
%  The Zebra puzzle:  Who owns the zebra?
%
%  According to a recent e-mail source, this puzzle originated with
%  the German Institute of Logical Thinking, Berlin, 1981. [It's
possible.]
%  The terms of the puzzle are included as comments in the code below.
%
%  Solution by Jonathan Mohr (mohrj@augustana.ca)
%  Augustana University College, Camrose, AB, Canada  T4V 2R3


%  Invoke this predicate if you just want to see the answer to the
%  question posed at the end of the puzzle.
solve :-
    solve(_).


%  Invoke this predicate (with a variable parameter) if you want to
%  see a complete solution.
solve(S) :-

% There are five houses.
% Each house has its own unique color.
% All house owners are of different nationalities.
% They all have different pets.
% They all drink different drinks.
% They all smoke different cigarettes.

% (The constraints that all colors, etc., are different can only be
% applied after all or most of the variables have been instantiated.
% See below.)

% S = [[Color1, Nationality1, Pet1, Drink1, Smoke1] |_]
% The order of the sublists is the order of the houses, left to right.
    S = [[C1,N1,P1,D1,S1],
         [C2,N2,P2,D2,S2],
         [C3,N3,P3,D3,S3],
         [C4,N4,P4,D4,S4],
         [C5,N5,P5,D5,S5]],

% The English man lives in the red house.
    member([red, 'English man', _, _, _], S),
% The Swede has a dog.
    member([_, 'Swede', dog, _, _], S),
% The Dane drinks tea.
    member([_, 'Dane', _, tea, _], S),
% The green house is on the left side of the white house.
    left_of([green |_], [white |_], S),
% In the green house, they drink coffee.
    member([green, _, _, coffee, _], S),
% The man who smokes cigarette has birds.
    member([_, _, birds, _, cigarette], S),
% In the yellow house, they smoke pipe.
    member([yellow, _, _, _, pipe], S),
% In the middle house, they drink milk.
```

```
    D3 = milk,
% The Norwegian lives in the first house.
    N1 = 'Norwegian',
% The man who smokes cigar lives in the house next to the house with
cats.
    next_to([_, _, _, _, cigar], [_, _, cats |_], S),
% In the house next to the house with the horse, they smoke pipe.
    next_to([_, _, _, _, pipe], [_, _, horse |_], S),
% The man who smokes hukka drinks soda.
    member([_, _, _, soda, hukka], S),
% The German smokes bedi.
    member([_, 'German', _, _, bedi], S),
% The Norwegian lives next to the blue house.
    next_to([_, 'Norwegian' |_], [blue |_], S),
% They drink water in the house that is next to the house
% where they smoke cigar.
    next_to([_, _, _, water,_], [_, _, _, _, cigar], S),

%
% The puzzle is so constrained that the following checks are not really
% required, but I include them for completeness (since one would not
% know in advance of solving the puzzle if it were fully constrained
% or not).
%
% Each house has its own unique color.
    C1 \== C2, C1 \== C3, C1 \== C4, C1 \== C5,
    C2 \== C3, C2 \== C4, C2 \== C5,
    C3 \== C4, C3 \== C5, C4 \== C5,
% All house owners are of different nationalities.
    N1 \== N2, N1 \== N3, N1 \== N4, N1 \== N5,
    N2 \== N3, N2 \== N4, N2 \== N5,
    N3 \== N4, N3 \== N5, N4 \== N5,
% They all have different pets.
    P1 \== P2, P1 \== P3, P1 \== P4, P1 \== P5,
    P2 \== P3, P2 \== P4, P2 \== P5,
    P3 \== P4, P3 \== P5, P4 \== P5,
% They all drink different drinks.
    D1 \== D2, D1 \== D3, D1 \== D4, D1 \== D5,
    D2 \== D3, D2 \== D4, D2 \== D5,
    D3 \== D4, D3 \== D5, D4 \== D5,
% They all smoke different cigarettes.
    S1 \== S2, S1 \== S3, S1 \== S4, S1 \== S5,
    S2 \== S3, S2 \== S4, S2 \== S5,
    S3 \== S4, S3 \== S5, S4 \== S5,

% Who owns the zebra?
    member([_, Who, zebra, _, _], S),
    write('The '), write(Who), write(' owns the zebra.\n').

% Or, replace the last line by:
%   format("The ~w owns the zebra.", Who).

left_of(L1, L2, [L1, L2 |_]).
left_of(L1, L2, [_| Rest ]) :- left_of(L1, L2, Rest).

next_to(L1, L2, S) :- left_of(L1, L2, S).
next_to(L1, L2, S) :- left_of(L2, L1, S).
```

Expert system

One of the main application domains for Prolog has been the development of expert systems. Following is an example of a simple medical expert system:

```
% clauses for relieves(Drug, Symptom). That is facts about drugs and relevant symptoms

relieves(aspirin, headache).
relieves(aspirin, moderate_pain).
relieves(aspirin, moderate_arthritis).
relieves(aspirin_codine_combination, severe_pain).
relieves(cough_cure, cough).
relieves(pain_gone, severe_pain).
relieves(anti_diarrhea, diarrhea).
relieves(de_congest, cough).
relieves(de_congest, nasal_congestion).
relieves(penicilline, pneumonia).
relieves(bis_cure, diarrhea).
relieves(bis_cure, nausea).
relieves(new_med, headache).
relieves(new_med, moderate_pain).
relieves(cong_plus, nasal_congestion).

% now we have clauses for side effects in the form of aggravates(Drug, Condition).

aggravates(aspirin, asthma).
 aggravates(aspirin, peptic_ulcer).
aggravates(anti-diarrhea, fever).
aggravates(de_congest, high_blood_pressure).
aggravates(de_congest, heart_disease).
aggravates(de_congest, diabetes).
aggravates(de_congest, glaucoma).
aggravates(penicilline, asthma).
aggravates(de_congest, high_blood_pressure).
aggravates(bis_cure, diabetes).
aggravates(bis_cure, fever).
```

```
% now some rules for prescribing medicine using symptoms and side effects
should_take(Person, Drug) :-
        complains_of(Person, Symptom),
        relieves(Drug, Symptom),
        not(unsuitable_for(Person, Drug)).

unsuitable_for(Person, Drug) :-
        aggravates(Drug, Condition),
        suffers_from(Person, Condition).

% we now have some specific facts about a patient named Ali
complains_of(ali, headache).
suffers_from(ali, peptic_ulcer).

% now the query about the proper medicine and the answer by the expert system
?- should_take(ali, Drug).
Drug = new_med;
```

As mentioned earlier, Prolog has been used to write expert systems and expert system shells. Because of its inference mechanism and independence of rules and clauses it is relatively easy to achieve such task as compared to any imperative language.

Conclusions

Prolog is a declarative programming language where we only specify what we need rather than how to do it. It has a simple concise syntax with built-in tree formation and backtracking which generates readable code which is easy to maintain. It is relatively case and type insensitive and is suitable for problem solving / searching, expert systems / knowledge representation, language processing / parsing and NLP, and game playing. Efficiency is definitely is negative point. While writing programs one has to be aware of the left recursion, failing to do that may result in infinite loops. It has a steep learning curve and suffers from lack of standardization.

Modern Programming Languages

Lecture 27-30

Java Programming Language

An Introduction

Java was developed at Sun in the early 1990s and is based on C++. It looks very similar to C++ but it is significantly simplified as compared to C++. That is why Professor Feldman says that Java is C++--. It supports only OOP and has eliminated multiple inheritance, pointers, structs, enum types, operator overloading, goto statement from C++. It includes support for applets and a form of concurrency.

The First Program

Here is the famous Hello World program in Java.

```
class HelloWorld {
        public static void main( String [ ] args )
 {
                System.out.println("Hello world!");
 }
}
```

It may be noted that as Java supports only OOP, in Java, every variable, constant, and function (including *main*) must be inside some class. Therefore, there are no global variables or functions in Java. In addition, the following may be noted:

- Function main is member of the class.
- Every class may have a main function; there may be more than one main function in a Java program.
- The main must be public static void.
- The main may have one argument: an array of String. This array contains the command-line arguments.
- There is no final semi-colon at the end of the class definition.

Java Files

In Java the source code is written in the .java file. The following restrictions apply:
(1) Each source file can contain at most one *public* class.
(2) If there is a public class, then the class name and file name must match.

Furthermore, every function must be part of a class and every class is part of a package. A *public* class can be used in *any* package and a non-public class can only be used in its own package.

The Java Bytecode is created by the Java compiler and is stored in .class file. This file contains ready to be exected Java bytecode which is executed by the Java Virtual Machine. For each class in a source file (both public and non-public classes), the compiler creates one ".class" file, where the file name is the same as the class name. When compiling a program, you type the full file name, including the ".java" extension;

When running a program, you just type the name of the class whose *main* function you want to run.

Java Types

Java has two "categories" of types: *primitive types* and *reference* types.

Primitive Types

All the primitive types have specified sizes that are machine independent for portability. This includes:

boolean	same as bool in C++
char	holds one 16 bit unicode character
byte	8-bit signed integer
short	16-bit signed integer
int	32-bit signed integer
long	64-bit signed integer
float	floating-point number
double	double precision floating-point number

Reference Types

arrays classes

There are no struct, union, enum, unsigned, typedef, or pointers types in Java.

C++ Arrays vs Java Arrays

In C++, when you declare an array, storage for the array is allocated. In Java, when you declare an array, you are really only declaring a pointer to an array; storage for the array itself is not allocated until you use "new". This difference is elaborated as shown below:

C++

```
int A[10];          // A is an array of length 10
A[0] = 5;           // set the 1st element of array A
```

JAVA

```
int [ ] A;          // A is a reference to an array
A = new int [10];   // now A points to an array of length 10
A[0] = 5;           // set the 1st element of the array pointed to by A
```

In both C++ and Java you can initialize an array using values in curly braces. Here's example Java code:

```
int [ ] myArray = {13, 12, 11};
                        //  myArray points to an array of length 3
                        // containing the values 13, 12, and 11
```

In Java, a default initial value is assigned to each element of a newly allocated array if no initial value is specified. The default value depends on the type of the array element as shown below:

Type	**Value**
boolean	false
char	'\u0000'
byte, int, short, long, float, double	0
any reference	null

In Java, array bounds are checked and an out-of-bounds array index always causes a runtime error.

In Java, you can also determine the current length of an array (at runtime) using ".length" operator as shown below:

```
int [ ] A = new int[10];
...
A.length ...          // this expression evaluates to 10
A = new int[20];
...
A.length ...          // now it evaluates to 20
```

In Java, you can copy an array using the *arraycopy* function. Like the output function *println*, arraycopy is provided in java.lang.System, so you must use the name System.arraycopy. The function has five parameters:

src: the source array (the array from which to copy)
srcPos: the starting position in the source array
dst: the destination array (the array into which to copy)
dstPos: the starting position in the destination array
count: how many values to copy

Here is an example:

```
int [ ] A, B;
A = new int[10];

// code to put values into A
B = new int[5];
System.arraycopy(A, 0, B, 0, 5)   // copies first 5 values from A to B
System.arraycopy(A, 9, B, 4, 1)   // copies last value from A into last
                                  // element of B
```

Note that the destination array must already exist (i.e., *new* must already have been used to allocate space for that array), and it must be large enough to hold all copied values

(otherwise you get a runtime error). Furthermore, the source array must have enough values to copy (i.e., the length of the source array must be at least srcPos+count). Also, for arrays of primitive types, the types of the source and destination arrays must be the same. For arrays of non-primitive types, System.arraycopy(A, j, B, k, n) is OK if the assignment B[0] = A[0] would be OK.

The arraycopy function also works when the source and destination arrays are the *same* array; so for example, you can use it to "shift" the values in an array:

```
int [ ] A = {0, 1, 2, 3, 4};
System.arraycopy(A, 0, A, 1, 4);
```

After executing this code, A has the values: [0, 0, 1, 2, 3].

As in C++, Java arrays can be *multidimensional*. For example, a 2-dimensional array is an array of arrays. However, a two-dimensional arrays need not be rectangular. Each row can be a different length. Here's an example:

```
int [ ][ ] A;              // A is a two-dimensional array
A = new int[5][];          // A now has 5 rows, but no columns yet
A[0] = new int [1];        // A's first row has 1 column
A[1] = new int [2];        // A's second row has 2 columns
A[2] = new int [3];        // A's third row has 3 columns
A[3] = new int [5];        // A's fourth row has 5 columns
A[4] = new int [5];        // A's fifth row also has 5 columns
```

C++ Classes vs Java Classes

In C++, when you declare a variable whose type is a class, storage is allocated for an object of that class, and the class's constructor function is called to initialize that instance of the class. In Java, you are really declaring a pointer to a class object; no storage is allocated for the class object, and no constructor function is called until you use "new".

This is elaborated with the help of the following example:

Let us assume that we have the following class:

```
class MyClass {
        ...
}
```

Let us first look at C++

```
MyClass m;   // m is an object of type MyClass;
             //  the constructor function is called to initialize M.
```

```
MyClass  *pm;
                // pm is a pointer to an object of type MyClass
                // no object exists yet, no constructor function
                // has been called

pm = new MyClass;
                // now storage for an object of MyClass has been
                // allocated and the constructor function has been
                // called
```

Now the same thing in Java

```
MyClass m;   // pm is a pointer to an object of type MyClass
             // no object exists yet, no constructor function
             // has been called

m = new MyClass();
             // now storage for an object of MyClass has been allocated
             // and the constructor function has been called. Note
             // that you must use parentheses even when you are not
             // passing any arguments to the constructor function

             // Also note that there is a simple '.' (dot) operator used to
             // access members or send message. Java does not use
             // -> operator.
```

Whereas, as opposed to Java, in C++ use have the following:

```
MyClass m;          // m is an object of type MyClass;
                    // the constructor function is called to initialize M.

MyClass  *pm;       // pm is a pointer to an object of type MyClass
                    // no object exists yet, no constructor function has
                    // been called

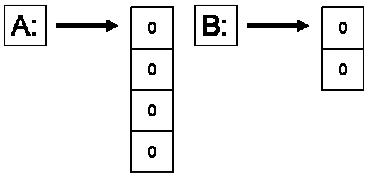
pm = new MyClass; // now storage for an object of MyClass has been
                    // allocated and the constructor function has been
                    // called
```

Aliasing Problems in Java

The fact that arrays and classes are really pointers in Java can lead to some problems. Here is a simple assignment that causes *aliasing*:

```
int [] A = new int [4];
Int [] B = new int [2];
```

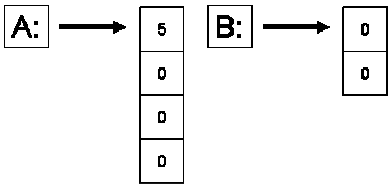
This is depicted as below:



Now, when we say:

```
A[0] = 5;
```

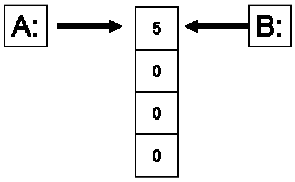
We get the following:



Now when we say:

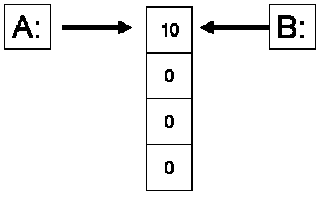
```
B = A;
```

B points to the same array as A and creates an alias. This is shown below:



Now if we make a simple assignment in B, we will also change A as shown below:

```
B[0] = 10;
```



This obviously creates problems. Therefore, as a programmer you have to be very careful when writing programs in Java.

In Java, all parameters are passed by value, but for arrays and classes the actual parameter is really a pointer, so changing an array element, or a class field inside the function *does* change the actual parameter's element or field.

This is elaborated with the help of the following example:

```
void f( int [ ] A ) {
        A[0] = 10;      // change an element of parameter A
        A = null;       // change A itself
}

void g() {
        int [ ] B = new int [3];
        B[0] = 5;
        f(B);
                // B is not null here, because B itself was passed by value
                // however, B[0] is now 10, because function f changed the
                // first element of the array
}
```

Note that, in C++, similar problems can arise when a class that has pointer data members is passed by value. This problem is addressed by the use of copy constructors, which can be defined to make copies of the values pointed to, rather than just making copies of the pointers. In Java, the solution is to use the *arraycopy* operation, or to use a class's *clone* operation. Cloning will be discussed later.

Type Conversion

Java is much stronger than C++ in the type conversions that are allowed.
Here we discuss conversions among primitive types. Conversions among class objects will be discussed later.

Booleans cannot be converted to other types. For the other primitive types (char, byte, short, int, long, float, and double), there are two kinds of conversion: *implicit* and *explicit*.

Implicit conversions:

An implicit conversion means that a value of one type is changed to a value of another type without any special directive from the programmer. A char can be implicitly converted to an int, a long, a float, or a double. For example, the following will compile without error:

```
char c = 'a';
int k = c;
long x = c;
float y = c;
double d = c;
```

For the other (numeric) primitive types, the basic rule is that implicit conversions can be done from one type to another if the range of values of the first type is a subset of the range of values of the second type. For example, a byte can be converted to a short, int, long or float; a short can be converted to an int, long, float, or double, etc.

Explicit conversions:

Explicit conversions are done via *casting*: the name of the type to which you want a value converted is given, in parentheses, in front of the value. For example, the following code uses casts to convert a value of type double to a value of type int, and to convert a value of type double to a value of type short:

```
double d = 5.6; int k = (int)d; short s = (short)(d * 2.0);
```

Casting can be used to convert among any of the primitive types except boolean. Note, however, that casting can lose information; for example, floating-point values are truncated when they are cast to integers (e.g., the value of k in the code fragment given above is 5), and casting among integer types can produce wildly different values (because upper bits, possibly including the sign bit, are lost).

JAVA CLASSES

Java classes contain *fields* and *methods*. A field is like a C++ data member, and a method is like a C++ member function. Each field and method has an *access level*:

- private: accessible only in this class
- (package): accessible only in this package
- protected: accessible only in this package and in all subclasses of this class
- public: accessible everywhere this class is available

Similarly, each class has one of two possible access levels:

- (package): class objects can only be declared and manipulated by code in this package
- public: class objects can be declared and manipulated by code in any package

Note: for both fields and classes, package access is the default, and is used when *no* access is specified.

A Simple Example Class

In the following example, a List is defined to be an ordered collection of items of any type:

```
class List {
        // fields
        private Object [ ] items;      // store the items in an array
        private int   numItems;        // the current # of items in the list
                                       // methods

        // constructor function
        public List()
        {
                items = new Object[10];
                numItems = 0;
        }

        // AddToEnd: add a given item to the end of the list
        public void AddToEnd(Object ob)
        {
        ...
        }
}
```

In Java, all classes (built-in or user-defined) are (implicitly) subclasses of the class **Object**. Using an array of Object in the List class allows any kind of Object (an instance of any class) to be stored in the list. However, primitive types (int, char, etc) cannot be stored in the list as they are not inherited from Object.

Constructor function:

As in C++, constructor functions in Java are used to initialize each instance of a class. They have no return type (not even void) and can be overloaded; you can have multiple constructor functions, each with different numbers and/or types of arguments. If you don't write any constructor functions, a default (no-argument) constructor (that doesn't do anything) will be supplied.

If you write a constructor that takes one or more arguments, no default constructor will be supplied (so an attempt to create a new object without passing any arguments will cause a compile-time error). It is often useful to have one constructor call another (for example, a constructor with no arguments might call a constructor with one argument, passing a default value). The call must be the *first* statement in the constructor. It is performed using *this* as if it were the name of the method. For example:

```
this( 10 );
```

is a call to a constructor that expects one integer argument.

Initialization of fields:

If you don't initialize a field (i.e., either you don't write any constructor function, or your constructor function just doesn't assign a value to that field), the field will be given a default value, depending on its type. The values are the same as those used to initialize newly created arrays (see the "Java vs C++" notes).

Access Control:

Note that the access control must be specified for every field and every method; there is no grouping as in C++. For example, given these declarations:

```
public
        int x;
        int y;
```

only x is public; y gets the default, package access.

Static Fields and Methods

Fields and methods can be declared *static*. If a field is static, there is only one copy for the entire class, rather than one copy for each instance of the class. (In fact, there is a copy of the field even if there are *no* instances of the class.) A method should be made static when it does not access any of the non-static fields of the class, and does not call any non-static methods. (In fact, a static method *cannot* access non-static fields or call non-static methods.) Methods that would be "free" functions in C++ (i.e., not members of any class) should be static methods in Java. A public static field or method can be accessed from outside the class.

Final Fields and Methods

Fields and methods can also be declared *final*. A final method cannot be overridden in a subclass. A final field is like a constant: once it has been given a value, it cannot be assigned to again.

Some Useful Built-in Classes

Following is a list of some useful classes in Java. These classes are not really part of the language; they are provided in the package *java.lang*

1. String

String Creation:

```
String S1 = "hello",          // initialize from a string literal
S2 = new String("bye"),       // use new and the String constructor
S3 = new String(S1);          // use new and a different constructor
```

String concatenation

```
String S1 = “hello ” + “world”;
String S2 = S1 + “!”;
String S3 = S1 + 10;
```

2. Object

Object is the Base class for all Java Classes.

3. Classes for primitive types

In Java, we have classes also for primitive types. These are: Boolean, Integer, Double, etc. Note that variable of bool, int, etc types are not objects whereas variable of type Boolean, Integer, etc are objects.

Packages

Every class in Java is part of some *package*. All classes in a file are part of the same package. You can specify the package using a *package declaration*:

package *name* ;

as the first (non-comment) line in the file. Multiple files can specify the same package name. If no package is specified, the classes in the file go into a special unnamed package (the same unnamed package for all files). If package *name* is specified, the file must be in a subdirectory called *name* (i.e., the directory name must match the package name).

You can access public classes in another (named) package using:

package-name.class-name

You can access the public fields and methods of such classes using:

package-name.class-name.field-or-method-name

You can avoid having to include the *package-name* using import *package-name*.* or import *package-name.class-name* at the beginning of the file (after the package declaration). The former imports all of the classes in the package, and the second imports just the named class.

You must still use the *class-name* to access the classes in the packages, and *class-name.field-or-method-name* to access the fields and methods of the class; the only thing you can leave off is the package name.

Inheritance

Java directly supports single inheritance. To support concept of a **interface** class is used. Inheritance is achieved as shown below:

```
class SuperClass {
        …
}

class SubClass extends SuperClass {
        …
}
```

When a class is inherited, all fields and methods are inherited. When the **final** reserved word is specified on a class specification, it means that class cannot be the parent of any class. *Private* fields are inherited, but cannot be accessed by the methods of the subclass. fields can be defined to be *protected*, which means that subclass methods can access them

Each superclass method (except its constructors) can be inherited, overloaded, or overridden. These are elaborated in the following paragraphs:

inherited: If no method with the same name is (re)defined in the subclass, then the subclass has that method with the same implementation as in the superclass.

overloaded: If the subclass defines a method with the same name, but with a different number of arguments or different argument types, then the subclass has *two* methods with that name: the old one defined by the superclass, and the new one it defined.

overridden: If the subclass defines a method with the same name, and the same number and types of arguments, then the subclass has only *one* method with that name: the new one it defined. A method cannot be overridden if it is defined as final in the superclass

Dynamic Binding

In C++, a method must be defined to be virtual to allow dynamic binding. In Java all method calls are dynamically bound unless the called method has been defined to be **final**, in which case it cannot be overridden and all bindings are static.

Constructors

A subclass's constructors *always* call a super class constructor, either explicitly or implicitly. If there is no explicit call (and no call to another of the subclass constructors), then the no-argument version of the superclass constructor is called before executing any statements.

Abstract Classes

An *abstract method* is a method that is declared in a class, but not defined. In order to be instantiated, a subclass must provide the definition. An *abstract class* is any class that includes an abstract method. It is similar to Pure virtual in C++.

If a class includes an abstract method, the class *must* be declared abstract, too. For example:

```
abstract class AbstractClass {
        abstract public void Print();
                // no body, just the function header
}
class MyConcreteClass extends AbstractClass {
        public void Print() { // actual code goes here } }
```

An abstract class cannot be instantiated. A subclass of an abstract class that does not provide bodies for all abstract methods must also be declared abstract. A subclass of a

non-abstract class can override a (non-abstract) method of its superclass, and declare it abstract. In that case, the subclass must be declared abstract.

Interface

As mentioned earlier, multiple inheritance is achieved through Interface in Java. Inheritance implements the "is-a" relationship whereas an interface is similar to a class, but can only contain public, static, final fields (i.e., constants) and public, abstract methods (i.e., just method headers, no bodies).

An interface is declared as shown below:

```
public interface Employee {
        void RaiseSalary( double d );
        double GetSalary();
}
```

Note that both methods are implicitly public and abstract (those keywords can be provided, but are not necessary).

A class can *implement* one or more interfaces (in addition to extending one class). It must provide bodies for all of the methods declared in the interface, or else it must be abstract. For example:

```
public class TA implements Employee {
        void RaiseSalary( double d ) {
                // actual code here
        }
        double GetSalary() {
                // actual code here
        }
}
```

Public interfaces (like public classes) must be in a file with the same name. Many classes can implement the same interface thus achieving multiple inheritance. An interface can be a "marker" with *no* fields or methods, is used only to "mark" a class as having a property, and is testable via the *instanceof* operator.

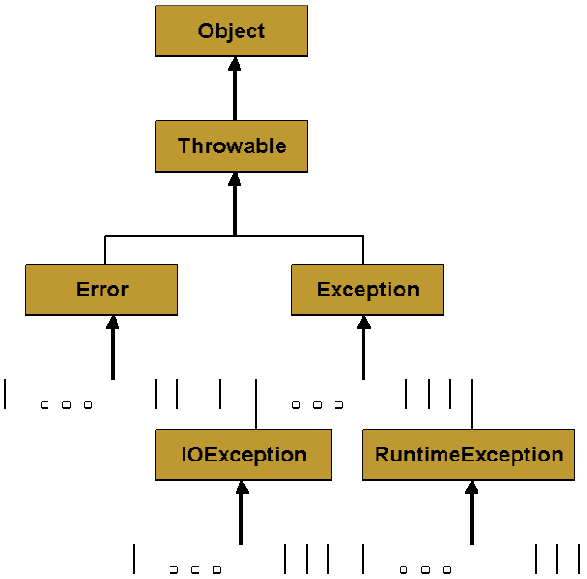
Exception Handling in Java

Exception handling in Java is based on C++ but is designed to be more in line with OOP. It includes a collection of predefined exceptions that are implicitly raised by the JVM. All java exceptions are objects of classes that are descendents of Throwable class. There are two predefined subclasses of Throwable: Error and Exception.

Error and its descendents are related to errors thrown by JVM. Examples include out of heap memory. Such an exception is never thrown by the user programs and should not be handled by the user.

User programs can define their own exception classes. Convention in Java is that such classes are subclasses of Exception. There are two predefined descendents of Exception: IOException and RuntimeException. IOException deals with errors in I/O operation.
In the case of RuntimeException there are some predefined exceptions which are, in many cases, thrown by JVM for errors such as out of bounds, and Null pointer.

The following diagram shows the exception hierarchy in Java.



Checked and Unchecked Exceptions

Exceptions of class Error and RuntimeException are called unchecked exceptions. They are never a concern of the compiler. A program can catch unchecked exceptions but it is not required. All other are checked exceptions. Compiler ensures that all the checked exceptions a method can throw are either listed in its throws clause or are handled in the method.

Exception Handlers

Exception handler in Java is similar to C++ except that the parameter of every **catch** must be present and its class must be descendent of Thrwoable. The syntax of **try** is exactly same as C++ except that there is **finally** clause as well. For example:

```
class MyException extends Exception {
```

```
        public MyException() { }
        public MyException(String message) {
                super (message);
        }
}
```

This exception can be thrown with

```
throw new MyException();
```

Or

```
MyException myExceptionObject = new MyException();
…
throw myExceptionObject;
```

Binding of exception is also similar to C++. If an exception is thrown in the compound statement of try construct, it is bound to the first handler (catch function) immediately following the try clause whose parameter is the same class as the thrown object or an ancestor of it. Exceptions can be handled and then re-thrown by including a throw statement without an operand at the end of the handler. To ensure that exceptions that can be thrown in a try clause are always handled in a method, a special handler can be written that matches all exceptions. For example:

```
catch (Exception anyException) {
        …
}
```

Other Design Choices

The exception handler parameter in C++ has a limited purpose. During program execution, the Java runtime system stores the class name of every object. getClass can be used to get an object that stores the class name. It has a getName method. The name of the class of the actual parameter of the throw statement can be retrieved in the handler as shown below.

```
anyException.getClass().getName();
```

In the case of a user defined exception, the thrown object could include any number of data fields that might be useful in the handler.

throws Clause

throws clause is overloaded in C++ and conveys two different meanings: one as specification and the other as command. Java is similar in syntax but different in semantics. The appearance of an exception class name in the **throws** clause of Java method specifies that the exception class or any of its descendents can be thrown by the method.

A C++ program unit that does not include a throw clause can throw any exceptions. A Java method that does not include a throws cannot throw any checked exception it does not handle. A method cannot declare more exceptions in its throws clause than the methods it overrides, though it may declare fewer. A method that does not throw a particular exception, but calls another method that could throw the exception, must list the exception in its throws clause.

The finally clause

A Java exception handler may have a finally clause. A finally clause *always* executes when its try block executes (whether or not there is an exception). The finally clause is written as shown below:

```
try {
        …
}
catch (…) {
        …
}
…
finally {
        …
}
```

A finally clause is usually included to make sure that some clean-up (e.g., closing opened files) is done. If the finally clause includes a transfer of control statement (return, break, continue, throw) then that statement overrides any transfer of control initiated in the try or in a catch clause.

First, let's assume that the finally clause does not include any transfer of control. Here are the situations that can arise:

- No exception occurs during execution of the try, and no transfer of control is executed in the try. In this case the finally clause executes, then the statement following the try block.
- No exception occurs during execution of the try, but it *does* execute a transfer of control. In this case the finally clause executes, then the transfer of control takes place.
- An exception *does* occur during execution of the try, and there is no catch clause for that exception. Now the finally clause executes, then the uncaught exception is "passed up" to the next enclosing try block, possibly in a calling function.
- An exception *does* occur during execution of the try, and there *is* a catch clause for that exception. The catch clause does not execute a transfer of control. In this case the catch clause executes, then the finally clause, then the statement following the try block.

- An exception *does* occur during execution of the try, there *is* a catch clause for that exception, and the catch clause *does* execute a transfer of control. Here, the catch clause executes, then the finally clause, then the transfer of control takes place.

If the finally block *does* include a transfer of control, then that takes precedence over any transfer of control executed in the try or in an executed catch clause. So for all of the cases listed above, the finally clause would execute, then *its* transfer of control would take place. Here's one example:

```
try {
        return 0;
} finally {
        return 2;
}
```

The result of executing this code is that 2 is returned. Note that this is rather confusing! The moral is that you probably do *not* want to include transfer-of-control statements in both the try statements and the finally clause, or in both a catch clause and the finally clause.

Java Threads

Java supports concurrency through threads which are lightweight processes. A thread is similar to a real process in that a thread and a running program are threads of execution. A thread takes advantage of the resources allocated for that program instead of having to allocate those resources again. A thread has its own stack and program counter. The code running within the thread works only within the context implied by the stack and PC so also called an execution context.

Creating Java Threads

There are two ways to create our own **Thread** object:

1. Subclassing the **Thread** class and instantiating a new object of that class
2. Implementing the **Runnable** interface

In both cases the **run()** method should be implemented. This is elaborated with the help of following examples:

Example of Extending Thread

```
public class ThreadExample extends Thread {
        public void run () {
        for (int i = 1; i <= 100; i++) {
```

```
                    System.out.println("Thread: " + i);
                }
            }
        }
```

Example of Implementing Runnable

```
    public class RunnableExample implements Runnable {
        public void run () {
            for (int i = 1; i <= 100; i++) {
            System.out.println ("Runnable: " + i);
        }
        }
    }
```

It may be noted that both of these are very similar to each other with minor syntactic and semantic differences.

Starting the Threads

A thread is started by simply sending the **start** message to the thread. This is shown in the following example:

```
public class ThreadsStartExample {
    public static void main (String argv[]) {
        new ThreadExample ().start ();
            new Thread(new RunnableExample ()).start ();
    }
}
```

Thread Methods

Some of the common thread methods are listed below:

- start()
- sleep()
- yield()
- run()
- wait()
- notify()
- notifyAll()
- setPriority()

Thread Synchronization - Wait and Notify

Threads are based upon the concept of a Monitor. The **wait** and **notify** methods are used just like **wait** and **signal** in a Monitor. They allow two threads to cooperate and based on a single shared lock object. The following example of a producer-consumer problem elaborates this concept.

```
class Buffer {
        private int [ ] buf;
        private int head, tail, max, size;

        public Buffer (int buf_size) {
                buf = new int [buf_size];
                head =0; tail = 0; size = 0;
                max = buf_size;
        }

        public synchronized void deposit (int item) {
                try {
                        while (size ==max) wait();
                        buf [tail] = item;
                        tail = (tail + 1) % max;
                        size++;
                        notify();
                }
                catch
                (interruptedException e) {
                        // exception handler
                }
        }

        public synchronized int fetch() {
                int item = 0;
                try {
                        while (size == 0) wait();
                        item = buf [head] ;
                        head = (head + 1) % max;
                        size--;
                        notify();
                }
                catch
                   (interruptedException e) {
                        // exception handler
                }
                return item;
        }
```

```
class Producer extends Thread {
        private Buffer buffer;


        public Producer (Buffer buf) {
                buffer = buf;
        }

        public void run () {
                int item;
                while (true) {
                    // create a new item
                    buffer.deposit(item);
                }
        }
}

class Consumer implements Runnable  {
        private Buffer buffer;

        public Consumer (Buffer buf) {
                buffer = buf;
        }

        public void run () {
                int item;
                while (true) {
                    item = buffer.fetch();
                    // consume item
                }
        }
}
```

We can now instantiate these threads as shown below:

```
Buffer buffer = new Buffer(100);
Producer producer1 = new Producer(buffer);
Consumer comsumer = new Consumer(buffer);
Thread consumer1 = new Thread(consumer);
producer1.start();
consumer1.start();
```

In this case the **buffer** is a shared variable used by both producer and consumer.

notify and notifyAll

There is a slight difference between **notify** and **notifyAll**. As the name suggest, **notify()** wakes up a single thread which is waiting on the object's lock. If there is more than one thread waiting, the choice is arbitrary i.e. there is no way to specify which waiting thread should be re-awakened. On the other hand, **notifyAll()** wakes up ALL waiting threads; the scheduler decides which one will run.

Applet

Java also has support for web applications through Applets. An applet is a stand alone Java *application.* Java *applet* runs in a Java-enabled Web browser.

Java versus C++

Generally, Java is more robust than C++. Some of the reasons are:
- Object handles are initialized to **null** (a keyword)
- Handles are always checked and exceptions are thrown for failures
- All array accesses are checked for bounds violations
- Automatic garbage collection prevents memory leaks and dangling pointers
- Type conversion is safer
- Clean, relatively fool-proof exception handling
- Simple language support for multithreading
- Bytecode verification of network applets

PHP – Personal Home Page PHP: Hypertext Preprocessor (Lecture 35-37)

A Server-side Scripting Programming Language

An Introduction

What is PHP?

PHP stands for "**P**HP: **H**ypertext **P**reprocessor". It is a server-side scripting language. PHP scripts are executed on the server and it is especially suited for Web development and can be embedded into HTML. The main goal of the language is to allow web developers to write dynamically generated web pages quickly. Its syntax is very similar to C/C++ and it has support for almost all major OS, web-servers, and databases. PHP is an open source software (OSS) and is free to download and use.

What is a PHP File?

PHP files may contain text, HTML tags and scripts. PHP files are returned to the browser as plain HTML. PHP files have a file extension of ".php", ".php3", or ".phtml". As mentioned earlier, a PHP script is run on the web server, not on the user's browser. Therefore, client-side compatibility issues are not of concern of the programmer as simply return an HTML document for the browser to process. You need three things to make this work: the PHP parser (CGI or server module), a web server and a web browser.

PHP Hello World

Following is the PHP Hello World program.

```
<html>
  <head>
    <title>PHP Test</title>
  </head>
  <body>
        <?php // start of php code
        echo '<p>Hello World</p>';
        ?>
  </body>
</html>
```

The code written within "php" tags is parsed by the PHP processor and the rest is ignored. Hence, a PHP parser would convert the above code into the following HTML document and send it to the browser.

```
<html>
```

```
<head>
 <title>PHP Test </title>
</head>
<body>
        <p>Hello World</p>
</body>
</html>
```

It may be noted that PHP supports C++ type of comments and requires a semicolon ";" at the end of each statement.

PHP Opening and Closing Tags

PHP scripts are always enclosed in between two PHP tags. This tells your server to parse the information between them as PHP. The three different forms are as follows:

1.
```
<?php
        echo "this is the recommended style";
?>
```

2.
```
<script language="php">
        echo "this style is also available";
</script>
```

3.
```
<?
        echo "this is short tags style; it needs to be configured";
?>
```

There is also another form which may or may not be supported on some browsers. It is shown as below:

4.
```
<%
        echo 'this is ASP-style tags; it may not be available';
%>
```

Data types

The type of a variable is decided at runtime by PHP depending on the context in which that variable is used. PHP supports the following data types:

- Boolean
- integer
- float or double
- string
- array
- object
- resource
- NULL

Integers

The size of an integer is platform-dependent, although a maximum value of about two billion is the usual value (that's 32 bits signed). 2147483647 = $2^{31} – 1$. PHP does not support unsigned integers. It is important to note that, unlike C++ or Java, if you specify a number beyond the bounds of the integer type, it will be interpreted as a float instead. If an operation results in a number beyond the bounds of the **integer** type, a **float** will be returned instead. Also, there is no integer division operator in PHP. *1/2* yields the **float** *0.5*.

Strings

PHP strings are created using single quote or double quote. They can also be created by <<< which is called heredoc. One should provide an identifier after <<<, then the string, and then the same identifier to close the quotation. The closing identifier *must* begin in the first column of the line.

Variable

Variables in PHP are represented by a dollar sign followed by the name of the variable. The variable name is case-sensitive. A valid variable name starts with a letter or underscore, followed by any number of letters, numbers, or underscores. As a regular expression, it would be expressed thus: '[a-zA-Z_\x7f-\xff][a-zA-Z0-9_\x7f-\xff]*'.

Note: For our purposes here, a letter is a-z, A-Z, and any ASCII characters from 127 through 255 (0x7f-0xff). Therefore, $Inzi½ is a legal variable name. It is not necessary to initialize variables in PHP. Un-initialized variables have a default value of their type - FALSE, zero, empty string or an empty array.

String conversion to numbers

When a string is evaluated as a numeric value, the resulting value and type are determined as follows.

- The string will evaluate as a **float** if it contains any of the characters '.', 'e', or 'E'. Otherwise, it will evaluate as an integer.
- The value is given by the initial portion of the string. If the string starts with valid numeric data, this will be the value used. Otherwise, the value will be 0 (zero).

Here is an example:

```
<?php
        $foo = 2 + "10.5";                  // $foo is float (12.5)
        $foo = 1 + "-1.1e3";            // $foo is float (-1099)
        $foo = 1 + “Ali-1.3e3";             // $foo is integer (1)
        $foo = 1 + "10 Small Cats";   // $foo is integer (11)
        $foo = "10.0 cats " + 1;        // $foo is float (11)
?>
```

Using strings in numeric expressions provides flexibility but it should be obvious that it also is a source to numerous programming errors and mistakes. This kind of error is also very hard to debug and detect.

Arrays

An array in PHP is an ordered map. An **array** can be created by the **array()** construct. It takes a certain number of comma-separated *key => value* pairs. *key* may be an **integer** or **string.** The following example illustrates this concept:

```
<?php
        $arr = array("foo" => "bar", 12 => true);
        echo $arr["foo"];               // bar
        echo $arr[12];          // 1
?>
```

If no key is specified in the assignment, then the maximum of the existing integer indices is taken, and the new key will be that maximum value + 1. If the current maximum is negative then the next key created will be zero. If no integer indices exist yet, the key will be *0* (zero). Note that the maximum integer key used for this *need not currently exist in the array*. It simply must have existed in the array at some time since the last time the array was re-indexed. If the specified key already has a value assigned to it, that value will be overwritten. These things are demonstrated in the following example:

```
<?php
        $arr = array(5 => 1, 12 => 2);
```

```
        $arr[] = 56;     // This is the same as $arr[13] = 56;
                                // at this point of the script
        $arr["x"] = 42;          // This adds a new element to
                        // the array with key "x“
        unset($arr[13]);         // This removes the element from the array
        $arr[] = 56;     // This is the same as $arr[14] = 56;
                                // at this point of the script
        unset($arr);     // This deletes the whole array
?>
```

class

A class is defined as shown below:

```
<?php
        class Cart {
                var $items; // Items in our shopping cart
                                // Add $num articles of $artnr to the cart

                 function add_item($artnr, $num) {
                        $this->items[$artnr] += $num;
                }
                function remove_item($artnr, $num) {
                        if ($this->items[$artnr] > $num) {
                                $this->items[$artnr] -= $num;
                                return true;
                        }
                         elseif ($this->items[$artnr] == $num) {
                                unset($this->items[$artnr]);
                                return true;
                        }
                        else { return false; }
                }
        }
?>
```

Aliasing

Aliasing is used to assign by reference. To assign by reference, simply prepend an ampersand (&) to the beginning of the variable which is being assigned (the source variable) as shown below:

```
<?php
        $foo = ‘Fakhar';
        $bar = &$foo;                    // Reference $foo via $bar.
        $bar = "My name is $bar";     // Alter $bar...
        echo $bar;
```

```
    echo $foo;                          // $foo is altered too.
?>
```

Variable variable

A variable variable is like pointers that is, it maintains the address of a variable in it. This is shown in the example below:

```
<?php
        $a = ‘hello’;
        $$a = 'world';
        echo "$a ${$a}“;              // ‘hello world”
?>
```

Constants

A constant is an identifier (name) for a simple value that cannot change during the execution of the script. A constant is case-sensitive by default. By convention, constant identifiers are always uppercase. The name of a constant follows the same rules as any label in PHP. A valid constant name starts with a letter or underscore, followed by any number of letters, numbers, or underscores. As a regular expression, it would be expressed thusly: *[a-zA-Z_\x7f-\xff][a-zA-Z0-9_\x7f-\xff]** .

Logical operators

PHP supports the following logical operators.

and, or, xor, !, &&, ||

The reason for the two different variations of "and" and "or" operators is that they operate at different precedence. This is demonstrated with the help of the following example:

```
$a and $b or $c     //      ($a and $b) or $c
$a && $b || $c
$a and $b || $c     //      $a and ($b || $c)
```

String operators

There are two **string** operators. The first is the concatenation operator ('.'), which returns the concatenation of its right and left arguments. The second is the concatenating assignment operator ('.='), which appends the argument on the right side to the argument on the left side.

```
<?php
        $a = "Hello ";
        $b = $a . "World!"; // now $b contains "Hello World!"
```

```
        $a = "Hello ";
        $a .= "World!";          // now $a contains "Hello World!"
?>
```

Array Operators

PHP provides a number of operators to manipulate arrays. These are:

- $a + $b Union – Union of $a and $b.
- $a == $b Equality – **TRUE** if $a and $b have the same key/value pairs.
- $a === $b Identity - **TRUE** if $a and $b have the same key/value pairs in the same order and of the same types.
- $a != $b Inequality - **TRUE** if $a is not equal to $b.
- $a <> $b Inequality - **TRUE** if $a is not equal to $b.
- $a !== $b Non-identity - **TRUE** if $a is not identical to $b.

Note the difference between == and ===. The former checks if the same key-value pairs are present in the two arrays and the order does not matter whereas the latter requires that in order to be identical the pairs have to be in the same order.

The + operator appends the right handed array to the left handed, whereas duplicated keys are NOT overwritten. This concept is elaborated as below:

```
$a = array("a" => "apple", "b" => "banana");
$b = array("a" => "pear", "b" => “date", "c" => “mango");
$c = $a + $b; // Union of $a and $b
```

This will result in an array c with the following values:

```
["a"]=> "apple" ["b"]=> "banana" ["c"]=> “mango"
```

Control Statements

PHP supports the following control statements:

- if
- while
- do while
- for
- switch
- foreach
- break
- continue

Most control structures work in a manner similar to C. The differences are highlighted in the following paragraphs.

If statements

The if statement in PHP has the following shape and form:

```
<?php
        if ($a > $b) { echo "a is bigger than b"; }
        elseif ($a == $b) { echo "a is equal to b"; }
        else { echo "a is smaller than b"; }
?>
```

There may be several *elseif*s within the same *if* statement. The first *elseif* expression (if any) that evaluates to **TRUE** would be executed. In PHP, you can also write 'else if' (in two words) and the behavior would be identical to the one of 'elseif' (in a single word).

foreach

foreach statement works only on arrays, and will issue an error when you try to use it on a variable with a different data type or an uninitialized variable. It gives an easy way to iterate over arrays. There are two syntaxes; the second is a minor but useful extension of the first:

- foreach (array_expression as $value) statement
- foreach (array_expression as $key => $value) statement

The first form loops over the array given by *array_expression*. On each loop, the value of the current element is assigned to *$value* and the internal array pointer is advanced by one. The second form does the same thing, except that the current element's key will also be assigned to the variable *$key* on each loop. As of PHP 5, it is possible to iterate objects too.

Note: Unless the array is referenced, *foreach* operates on a copy of the specified array and not the array itself. Therefore changes to the array element returned are not reflected in the original array. As of PHP 5, array's elements can be modified by preceding *$value* with &. This will assign reference instead of copying the value.

```
<?php
        $arr = array(1, 2, 3, 4);
        foreach ($arr as &$value)
                { $value = $value * 2; }
                        // $arr is now array(2, 4, 6, 8)
?>
```

This is possible only if iterated array can be referenced (i.e. is variable).

each

each return the current key and value pair from an array and advance the array cursor. It is used in the following manner:

array **each** (array &array)

Returns the current key and value pair from the array *array* and advances the array cursor. This pair is returned in a four-element array, with the keys *0*, *1*, *key*, and *value*. Elements *0* and *key* contain the key name of the array element, and *1* and *value* contain the data. If the internal pointer for the array points past the end of the array contents, **each()** returns **FALSE**.

break

break ends execution of the current *for*, *foreach*, *while*, *do-while* or *switch* structure. In PHP, *break* accepts an optional numeric argument which tells it how many nested enclosing structures are to be broken out of. The following example illustrates this concept:

```
<?php
        $i = 0;
        while (++$i) {
                switch ($i) {
                        case 5:
                                echo "At 5<br />\n";
                                break 1; // Exit only the switch.
                        case 10:
                                echo "At 10; quitting<br />\n";
                                break 2; // Exit the switch and the while.
                        default:
                                break;    // Exit only the switch.
                }
        }
?>
```

continue

continue is used within looping structures to skip the rest of the current loop iteration and continue execution at the condition evaluation and then the beginning of the next iteration. **Note:** Note that in PHP the switch statement is considered a looping structure for the purposes of *continue*. Like break, *continue* accepts an optional numeric argument which tells it how many levels of enclosing loops it should skip to the end of.

Omitting the semicolon after *continue* can lead to confusion as shown below:

```
<?php
        for ($i = 0; $i < 5; ++$i) {
                if ($i == 2) continue print "$i\n";
        }
?>
```

One can expect the result to be : 0 1 3 4 but this script will output : 2 because the return value of the **print()** call is *int(1)*, and it will look like the optional numeric argument mentioned above.

Alternative syntax for control statements

PHP offers an alternative syntax for some of its control structures; namely, *if*, *while*, *for*, *foreach*, and *switch*. In each case, the basic form of the alternate syntax is to change the opening brace to a colon (:) and the closing brace to *endif;*, *endwhile;*, *endfor;*, *endforeach;*, or *endswitch;*, respectively. An example is shown below:

```
<?php
        if ($a == 5):
                echo "a equals 5";
                echo "...";
        elseif ($a == 6):
                echo "a equals 6";
                echo "!!!";
        else: echo "a is neither 5 nor 6";
        endif;
?>
```

User defined functions

Just like all programming languages, one can define functions in PHP as shown in the following example:

```
<?php
        function foo($arg_1, $arg_2, /* ..., */ $arg_n)
        {
                echo "Example function.\n";
                return $retval;
        }
?>
```

PHP also allows functions to be defined inside functions. It may be noted that C and its descendents do not support this feature but descendants of Algol generally support it. The following example shows this concept:

```
<?php
	function foo() {
		function bar() {
		echo "I don't exist until foo() is called.\n";
		}
	}

	/* We can't call bar() yet since it doesn't exist. */

	foo();

	/* Now we can call bar(), foo()'s processesing has made it
	  accessible. */

	bar();
?>
```

PHP also supports recursive functions as shown below:

```
	<?php
		function recursion($a) {
			if ($a < 20) {
				echo "$a\n"; recursion($a + 1);
			}
		}
	?>
```

Function arguments

PHP supports passing arguments by value (the default), passing by reference, and default argument values. If you want an argument to a function to always be passed by reference, you can prepend an ampersand (&) to the argument name in the function definition:

```
	<?php
		function add_some_extra(&$string) {
			$string .= 'and something extra.';
		}
		$str = 'This is a string, ';
		add_some_extra($str);
		echo $str; // outputs 'This is a string, and
				// something extra.‘
	?>
```

You can't return multiple values from a function, but similar results can be obtained by returning a list as shown below:

```
	<?php
```

```
        function small_numbers() {
                return array (0, 1, 2);
        }
        list ($zero, $one, $two) = small_numbers();
?>
```

To return a reference from a function, you have to use the reference operator & in both the function declaration and when assigning the returned value to a variable:

```
<?php
        function &returns_reference() {
                return $someref;
        }
        $newref =& returns_reference();
?>
```

Classes and Objects

Classes and objects are similar to Java. A variable of the desired type is created with the *new* operator. It supports Single inheritance only and uses the keyword extends to inherit from a super class. The inherited methods and members can be overridden, unless the parent class has defined a method as final.

Databases

One of the strongest features of PHP is its support for providing web pages access to database. The following example shows database manipulation through ODBC in PHP:

```
<html>
<body>
<?php
        $conn=odbc_connect('northwind',",");
        if (!$conn)  {
                exit("Connection Failed: " . $conn);
        }
        $sql="SELECT * FROM customers";
        $rs=odbc_exec($conn,$sql);
        if (!$rs)  {exit("Error in SQL");}
        echo "<table><tr>";
        echo "<th>Companyname</th>";
        echo "<th>Contactname</th></tr>“;
        while (odbc_fetch_row($rs)){
                $compname=odbc_result($rs,"CompanyName");
                $conname=odbc_result($rs,"ContactName");
                echo "<tr><td>$compname</td>";
                echo "<td>$conname</td></tr>";
        }
        odbc_close($conn);
        echo "</table>";
?>
</body>
</html>
```

Modern Programming Languages-JavaScript
Lecture 38

The <script> Tag

The <script> tag (<script>….</script>) in all major browsers interprets contents as JavaScript unless one of the following occurs

Inclusion of language attribute

<script language="VBS">….</script>
Inclusion of type attribute
<script type="text/javascript">….</script>

The type attribute is W3C recommended, it makes the language more common and in many ways more useful

<script> tag is used to delimit the script code from the HTML

- **The script tag causes the browser's JavaScript interpreter to be invoked, the script run and any output produced**
- **The browser is considered the "host" environment**
- **There are other hosts for JavaScript and its variants**

Location of Code

JavaScript may be placed at three places
In the <head> element

Place scripts to be called or when event is triggered here
Ensures script is loaded before called

```
<html>
  <head>
    <script type="text/javascript">
                    //script statements
    </script>
  </head>
```

Location of Code

In the <body> element
Place scripts to be executed when the page loads here
Script generates some or all of the contents of the page

```
<body>
  <script type="text/javascript">
        //script statements
  </script>
</body>
```

Location of Code

External to the HTML file

```
<head>
  <script src="myfile.js">
  </script>
</head>
```

Could be in <head> or <body>
External script should not contain <script> tag

- You can use as many **<script>** tags as you like in both the **<head>** and **<body>** and they are executed sequentially.

-
```
<h1>Ready start</h1>
<script language="Javascript" type="text/javascript">
    alert("First Script Ran");
</script>
<h2>Running...</h2>
<script language="Javascript" type="text/javascript">
    alert("Second Script Ran");
</script>
<h2>Keep running</h2>
<script language="Javascript" type="text/javascript">
    alert("Third Script Ran");
</script>
</h1>Stop!</h1>
</body>
```

Statements

- A script is made up of individual statements
- Javascript statements are terminated by returns or semi-colons same as

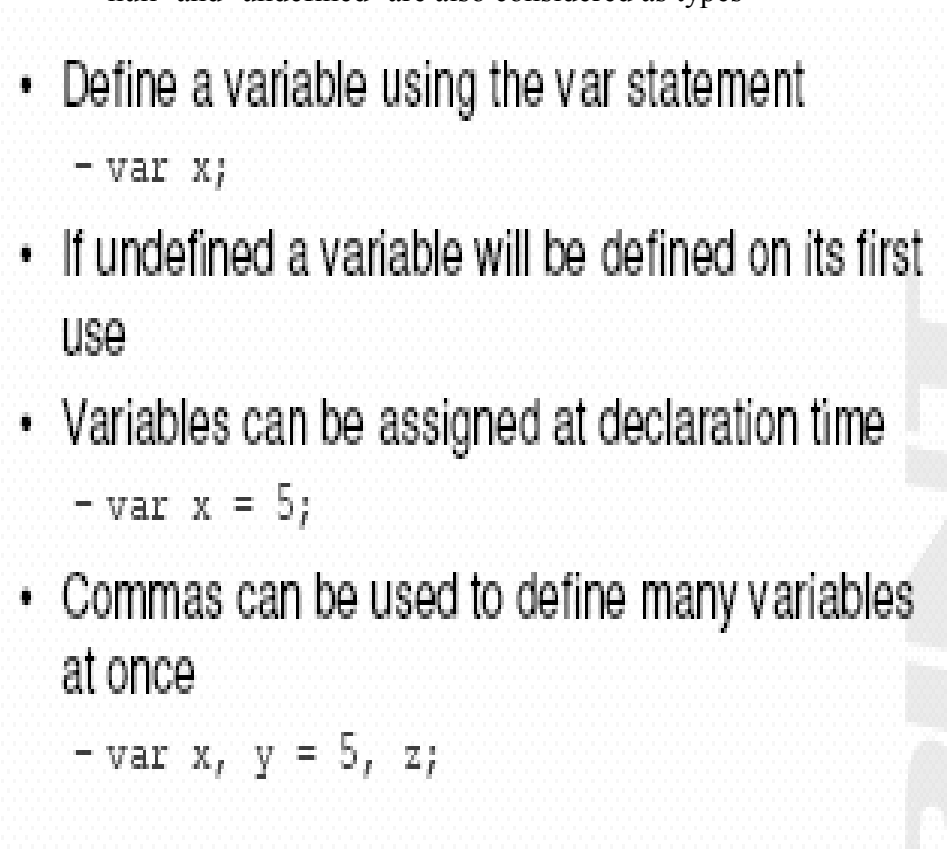
- So, prefer to use semi-colons

```
x=x+1    alert(x) //throws an error while
x=x+1;   alert(x); //does not
```

- Every variable has a data type that indicates what kind of data the variable holds
- Basic data types in JavaScript
- Strings

Strings may include special escaped characters

- Numbers (integers, floats)
- Booleans (true, false)
- 'null' and 'undefined' are also considered as types



- JavaScript is a weekly typed language meaning that the contents of a variable can change from one type to another
- Example

x = "hello"; x = 5; x=false;

While week typing seems beneficial to a programmer it can lead to problems

Arrays

- An ordered set of values grouped together with a single identifier
- Defining Arrays

- var myArray = [1,5,1968,3];
- var myArray2 = [“fakhar”,true,3,-47.2];
- var myArray3 = new Array ();
- var myArray4 = new Array (10);

Arrays

- Arrays in JavaScript are 0 based
- We access arrays by index values

- var myArray = [1,5,1968,3];
- myArray[3] is ‘3’

Difference

Array types (composite type as well) are reference types, so problem of aliasing is there

```
var firstArray = [“Mars”, “Jupiter”, “Saturn” ];
var secondArray = firstArray;
secondArray[0] = “Neptune”;
alert(firstArray); // it has been changed
```

Operators

- Basic Arithmetic
  + , - , / , * , %
- Increment decrement
  ++, - -
- Comparison
  < , > , >= , <= , != , == , === (type equality)
- Logical
  &&, ||, !
- Bitwise Operators
  & , | , ^
- String Operator

- + (used for concatenation)
- document.write("JavaScript" + "is" + "great!");

Type Conversion

- **Converting one type of data to another is both useful and a source of numerous errors in JavaScript**
- var x = "10" – 2 ; //result is 8
- var x = "2" – "2" ; //result is 0
- var x = "2" + "2" ; //result is "22"

Control Statements

- **If**
- **Switch**
- **While**
- **Do-while**
- **For**
- **Continue**
- **Break**
- **For (in)**

Labels and Flow Control

```
outerloop:
  for (var i=0; i < 3; i++)
  {
        document.write("Outerloop: "+i+"<br />");
        for (var j = 0; j < 5; j++)
          {
            if ( j == 3)
              break outerloop;
            document.write("Innerloop: "+j+"<br />");
          }
  }
  document.write("All loops done"+"<br>");
```

```
function name(parameter list)
 {
  function statement(s)
  return;
 }
```

Local Functions

```
function testFunction()
    {

    function inner1() {  document.write("testFunction-inner1<br />");  }

    function inner2() {  document.write("testFunction-inner2<br />");  }

     document.write("Entering testFunction<br />");
     inner1();
     inner2();
     document.write("Leaving testFunction<br />");
    }

    document.write("About to call testFunction<br />");
    testFunction();
    document.write("Returned from testFunction<br />");

  /* Call inner 1 or inner2 here and error */
    inner1();
```

Modern Programming Languages-JavaScript
Lecture 39

Objects

- **An object is a collection of data types as well as functions in one package**
- **The various data types called properties and functions called methods are accessed using the dot notation**

- **objectname.propertyname**

Objects

- **There are many types of objects in JavaScript**

- **Built-in objects (primarily type related)**
- **Browser objects (navigator, window etc.)**
- **Document objects (forms, images etc.)**
- **User defined objects**

Two Object Models

- **In JavaScript, two primary object models are employed**
- **Browser Object Model (BOM)**

- **The BOM provides access to the various characteristics of a browser such as the browser window itself, the screen characteristics, the browser history and so on.**
- **Document Object Model (DOM)**
- **The DOM on the other hand provides access to the contents of the browser window, namely the documents including the various HTML elements ranging from anchors to images as well as any text that may be enclosed by such element.**

The Big Picture

- Looking at the "big picture" of all various aspects of JavaScript including its object models. We see four primary pieces:
  1. The core JavaScript language (e.g. data types, operators, statements, etc.)
  2. The core objects primarily related to data types (e.g. Date, String, Math, etc.)
  3. The browser objects (e.g. Window, Navigator, Location, etc.)
  4. The document objects (e.g. Document, Form, Image, etc.)

Four Models

- By studying the history of JavaScript we can bring some order to the chaos of competing object models. There have been four distinct object models used in JavaScript including:
  1. Traditional JavaScript Object Model (NS 2 & IE 3)
  2. Extended Traditional JavaScript Object Model (NS 3)
  3. Dynamic HTML Flavored Object Models
     1. a. IE 4
     2. b. NS 4
  4. Traditional Browser Object Model + Standard DOM (NS6 & Explorer 5)

Overview of Core Objects

Object	Description
Window	The object that relates to the current browser window.
Document	An object that contains the various HTML elements and text fragments that make up a document. In the traditional JavaScript object model, the **Document** object relates roughly the HTML <body> tag.
Frames[ ]	An array of the frames in the Window contains any. Each frame in turn references another **Window** object that may also contain more frames.
History	An object that contains the current window's history list, namely the collection of the various URLs visited by the user recently.
Location	Contains the current location of the document being viewed in the form of a URL and its constituent pieces.
Navigator	An object that describes the basic characteristics of the browser, notably its type and version.

Document Object

- The **Document** object provides access to page elements such as anchors, form fields, and links as well as page properties such as background and text color.
- Consider
  - document.alinkColor, document.bgColor, document.fgColor, document.URL
  - document.forms[ ], document.links[ ], document.anchors[ ]
- We have also used the methods of the **Document** object quite a bit
  - document.write( ) , document.writeln( ), document.open( ), document.close( )

Object Access by Document Position

- HTML elements exposed via JavaScript are often placed in arrays or collections. The order of insertion into the array is based upon the position in the document.
- For example, the first **<form>** tag would be in document.forms[0], the second in document.forms[1] and so on.
- Within the form we find a collection of elements[ ] with the first **<input>**, **<select>** or other form field in document.forms[0].elements[0] and so on.
- As arrays we can use the length property to see how many items are in the page.
- The downside of access by position is that if the tag moves the script may break

Object Access by Name

- When a tag is named via the **name** attribute (HTML 4.0 - **<a>**, **<img>**, embedded objects, form elements, and frames) or by **id** attribute (pretty much every tag) it should be scriptable.
- Given

```
<form id="myform" name="myform">
  <input type="text" name="username" id="username">
</form>
```

we can access the form at window.document.myform and the first field as

window.document.myform.username

Object Access by Associative Array

- The collection of HTML objects are stored associatively in the arrays.
- Given the form named "myform" we might access it using

  window.document.forms["myform"]

- In Internet Explorer we can use the **item( )** method like so

  window.document.forms.item("myform")

Object Statement: With

document.write("Hello World");
document.write("<br />");
document.write("this is another write statement");

```
with (document) {
        write(“Hello World”);
         write(“<br />”);
         write(“this is another write statement”);
}
```

Object Statements: for..in

- The **for...in** statement is used to loop through the properties of an object (if they can be enumerated)
- Syntax

  for (*variablename* in *object*)

  statement or block to execute

- Example

```
var aProperty;
 document.write("<h1>Navigator Object Properties</h1>");
 for (aProperty in navigator)
 {
   document.write(aProperty);
   document.write("<br />");
 }
```

Events

- One of the primary uses of JavaScript is to make Web pages interactive.
  - Responsive to user actions
- JavaScript provides event handlers.
  - Execute segment of code based on events occurring within the application
  - E.g., `onLoad` or `onClick`
- Handlers associated with elements.
- Not all elements support all event handlers.

Events (cont.)

```
<input type="button"
    name="clickme"
    value="Click Here"
    onClick=
        "window.status='Thanks';
      return true;">
```


Events (cont.)

- Event handlers can be categorized into interactive and non-interactive.
- Interactive: Depends on a user action.
  - E.g., `onClick`
- Non-interactive: Non-user event.
  - E.g., `onLoad`

Events (cont.)

- `onAbort`: Image loading is interrupted.
- `onBlur`: Element loses input focus.
- `onChange`: User selects or deselects item.
- `onClick`: User clicks once.
- `onDragDrop`:
- `onError`: Image doesn't load properly.

Events (cont.)

- `onFocus`: Element is given input focus.
- `onKeyPress`:
- `onKeyUp`:
- `onLoad`:
- `onMouseDown`:
- `onMouseOver`:
- `onMouseOut`:
- `onMouseUp`:

Events
Supporting Events

1. Give the target HTML element a `name` attribute.

```
<input type="text"
       name="price"  />
```

2. Give activating HTML element event attribute that calls function.

```
<input type="submit"
    value="Calculate total."
    onClick="calcTotal()"/>
```

Modern Programming Languages
Lecture # 40

Names

- Design issues:
  - Maximum length?
  - Are connector characters allowed?
  - Are names case sensitive?
  - Are special words reserved words or keywords?

Special Words

- There are two types of special words
  - Keyword
  - Reserved word
- A keyword is a word that is special only in a certain context
  - REAL X
  - REAL = 44.7
- Disadvantage: poor readability

- A reserved word is a special word that cannot be used as a user-defined name
  - Type

- **Type:** Determines the range of values of variables and the set of operations that are defined for values of that type; in the case of floating point, type also determines the precision
  - Value
- **Value:** The contents of the location with which the variable is associated
  - Binding

- **Binding :** A binding is an association, such as between an attribute and an entity, or between an operation and a symbol

  - Binding Time
    It is the time at which a binding takes place

Possible binding times

1. Language design time - e.g. bind operator symbols to operations
2. Language implementation time - e.g. bind fl. pt. type to a representation
   1. Compile time - e.g. bind a variable to a type in C or Java
   2. Load time- e.g.
   3. Runtime - e.g. bind a non-static local variable to a memory cell

Static and Dynamic Binding

- A binding is static if it occurs before run time and remains unchanged throughout program execution

- A binding is dynamic if it occurs during execution or can change during execution of the program

Type Bindings

- How is a type specified?
- When does the binding take place?
- If static, type may be specified by either an explicit or an implicit declaration
  - An explicit declaration is a program statement used for declaring the types of variables
  - An implicit declaration is a default mechanism for specifying types of variables (the first appearance of the variable in the program)
- FORTRAN, PL/I, BASIC, provide implicit declarations
  - Advantage: writability
  - Disadvantage: reliability

Dynamic Type Binding

- Specified through an assignment statement e.g. in SNOBOL
  - LIST = 'my name'
  - LIST = 4 + 5
- Advantage
  - Flexibility (generic program units)
- Disadvantages:
  - Type error detection by the compiler is difficult
  - High cost (dynamic type checking and interpretation)

Storage Bindings

  - At what time we associate the memory location with a name

Categories of variables by lifetimes

1. **Static Storage Binding**
   - Bound to memory cells before execution begins and remains bound to the same memory cell throughout execution
   - Example
     - In FORTRAN 77 all variables are statically bound
     - C static variables
   - Advantage
     - Efficiency (direct addressing), history-sensitive, subprogram support

- Disadvantage
    - Lack of flexibility (no recursion)

Stack Dynamic Variables:

- Bound to storages when execution reaches the code to which the declaration is attached. (But, data types are statically bound.) That is, stack-dynamic variables are allocated from the **run-time stack**.
- Example
    - E.g. a Pascal procedure consists of a declaration section and a code section.
    - E.g. FORTRAN 77 and FORTRAN 90 use SAVE list for stack-dynamic list.
    - E.g. C and C++ assume local variables are static-dynamic.
- Advantage
    - allows recursion; conserves storage
- Disadvantages
    - Overhead of allocation and deallocation
    - Subprograms cannot be history sensitive
    - Inefficient references (indirect addressing)

Explicit Heap Dynamic Variables

- Allocated and de-allocated by explicit directives, specified by the programmer, which take effect during execution
- Referenced only through pointers or references
- Examples
    - Dynamic objects in C++ (via new and delete)
    - All objects in Java
- Advantage
    - Provide dynamic storage management
- Disadvantage
    - Inefficient and unreliable

Implicit Heap Dynamic Variables

- Allocation and de-allocation is caused by assignment statements
- Example
    - All variables in SNOBOL
- Advantage
    - Flexibility
- Disadvantages
    - Inefficient, because all attributes are dynamic
    - Loss of error detection

Modern Programming Languages Lecture 41

Type Checking

- Generalizes the concept of operands and operators to include subprograms and assignments
- Type checking is the activity of ensuring that the operands of an operator are of compatible types
- A compatible type is one that is either legal for the operator, or is allowed under language rules to be implicitly converted, by compiler generated code, to a legal type. This automatic conversion is called **coercion**
- A type error is the application of an operator to an operand of an inappropriate type
- If all type bindings are static, nearly all type checking can be static
- If type bindings are dynamic, type checking must be dynamic
- A programming language is strongly typed if type errors are always detected
- A programming language which does all the type checking statically then it is a strongly typed languages otherwise weakly typed language.

Strongly Typed Languages?

- FORTRAN 77 is not strongly typed
  - Parameters, EQUIVALENCE
- C and C++ are not
  - Parameter type checking can be avoided
  - Unions are not type checked
- Pascal is not
  - Variant records
- Modula-2 is not
  - Variant records
- Ada is, almost
  - UNCHECKED CONVERSION is a loophole
  - Coercion rules strongly affect strong typing; they can weaken it considerably (C++ versus Ada)
- Advantage of strong typing
  - Allows the detection of the misuse of variables that result in type errors

Type Compatibility

- Type compatibility by name means that the two variables have compatible types if they are in either the same declaration or in declarations that use the same type name
- Easy to implement but highly restrictive
  - Sub ranges of integer types are not compatible with integer types

Data Types

- Design Issues
  - What is the syntax of references to variables?
  - What operations are defined and how are they specified?

Primitive Data Types

- Not defined in terms of other data types

1. Integer

- Almost always an exact reflection of the hardware, so the mapping is trivial
- There may be as many as eight different integer types in a language

2. Floating Point

- Models real numbers, but only as approximations
- Languages for scientific use support at least two floating-point types; sometimes more
- Usually exactly like the hardware, but not always; some languages allow accuracy specs in code e.g. (Ada)
  - type SPEED is digits 7 range 0.0..1000.0;
  - type VOLTAGE is delta 0.1 range -12.0..24.0;

3. Decimal

- For business applications (money)
- Store a fixed number of decimal digits
- Advantage
  - Accuracy
- Disadvantages
  - Limited range, wastes memory
- COBOL and ADA supports decimal
- C++ and java does not support decimal

4. Boolean

- Could be implemented as bits, but often as bytes
- Advantage
  - Readability

Character String Types

- Values are sequences of characters
- Design issues:
  - Is it a primitive type or just a special kind of array?
  - Is the length of objects static or dynamic?
- Operations:
  - Assignment
  - Comparison (=, >, etc.)
  - Concatenation
  - Substring reference
  - Pattern matching

- Examples
  - Pascal---Not primitive; assignment and comparisons supported only
  - Ada, FORTRAN 77, FORTRAN 90 and BASIC
    - Somewhat primitive
    - Assignment, comparison, catenation
    - Substring reference
  - C and C++
    - Not primitive
    - Use char arrays and a library of functions that provide these operations
  - SNOBOL4 (a string manipulation language)
    - Primitive
    - Many operations, including elaborate pattern matching
  - Java; string class (not arrays of char)

Ordinal Types (user defined)

- An ordinal type is one in which the range of possible values can be easily associated with the set of positive integers
- Enumeration Types - one in which the user enumerates all of the possible values, which are symbolic constants
- Design Issue
  - Should a symbolic constant be allowed to be in more than one type definition?
- Examples
  - Pascal
    - Cannot reuse constants
    - They can be used for array subscripts e.g. for variables, case selectors
    - NO input or output
    - Can be compared
  - Ada
    - Constants can be reused (overloaded literals)
    - Disambiguates with context or type_name
    - CAN be input and output
  - C and C++
    - Like Pascal, except they can be input and output as integers
  - Java does not include an enumeration types
  - C# includes them

- Evaluation (of enumeration types)
  - Useful for readability
    - e.g. no need to code a color as a number
  - Useful for reliability
    - e.g. compiler can check operations and ranges of values

- Subrange Type
  - An ordered contiguous subsequence of an ordinal type
  - Pascal
    - Subrange types behave as their parent types; can be used as for variables and array indices
    - E.g. type pos = 0 .. MAXINT;
  - Ada
    - Subtypes are not new types, they are just constrained existing types (so they are compatible)
    - Can be used as in Pascal, plus case constants
    - **E.g.** subtype POS_TYPE is
    - **E.g.** INTEGER range 0 ..INTEGER'LAST;
  - Evaluation (of enumeration types)
    - Aid to readability
    - Reliability - restricted ranges add error detection

- Implementation of user-defined ordinal types
  - Enumeration types are implemented as integers
  - Subrange types are the parent types with code inserted (by the compiler) to restrict assignments to subrange variables

5. Arrays

- An array is an aggregate of homogeneous data elements in which an individual element is identified by its position in the aggregate, relative to the first element
- Design Issues
  - What types are legal for subscripts?
  - Are subscripting expressions in element references range checked?
  - When are subscript ranges bound?
  - When does allocation take place?
  - What is the maximum number of subscripts?
  - Can array objects be initialized?

6. Subscript Types

- FORTRAN, C, C++
  - int only
- Pascal
  - Any ordinal type (int, boolean, char, enum)
- Ada
  - int or enum (includes boolean and char)
- Java
  - integer types only

Four Categories of Arrays (based on subscript binding and binding to storage)

- **Static -** range of subscripts and storage bindings are static e.g. FORTRAN 77, some arrays in Ada
  - Advantage: execution efficiency (no allocation or de-allocation)
- **Fixed stack dynamic -** range of subscripts is statically bound, but storage is bound at elaboration time e.g. C local arrays are not static
  - Advantage: space efficiency
- **Stack-dynamic -** range and storage are dynamic, but fixed from then on for the variable's lifetime e.g. Ada declare blocks declare

  - STUFF : array (1..N) of FLOAT;
    begin
    ...
    end;
  - **Advantage:** flexibility - size need not be known until the array is about to be used

Modern Programming Languages Lecture 42

Records-(like structs in C/C++)

- **Description:**
  - Are composite data types like arrays
  - The difference between array and record is that array elements are of same data type where as record is a logical grouping of heterogeneous data elements.
  - Another difference is that access to array elements is much slower than access to record fields, because subscripts are dynamic (field names are static)
  - Dynamic subscripts could be used with record field access, but it would disallow type checking and it would be much slower

- **Design Issues:**
  - What is the form of references?
  - What unit operations are defined?

- **Record Definition Syntax**
  - COBOL uses level numbers to show nested records; others use recursive definitions

  - Examples:
    - COBOL
      field_name OF record_name_1 OF ... OF
      record_name_n

    - Others (dot notation)
      record_name_1.record_name_2. ...
      .record_name_n.field_name

  - Better Approach:
    - According to readability point of view COBOL record definition syntax is easier to read and understand but according to writability point of view other languages record definition syntax is easier to write and less time consuming.

- **Record Field References**

  - Two types of record field references:
    - Fully qualified references must include all record names

    - Elliptical references allow leaving out record names as long as the reference is unambiguous

  - Pascal and Modula-2 provide a with clause to abbreviate references

- **Record Operations**

  - Assignment
    - Pascal, Ada, and C allow it if the types are identical
    - In Ada, the RHS can be an aggregate constant

  - Initialization
    - - Allowed in Ada, using an aggregate constant

  - Comparison
    - - In Ada, = and /=; one operand can be an aggregate constant

Pointers

- **Description:**
  - A pointer type is a type in which the range of values consists of memory addresses and a special value, nil (or null)

- **Uses:**
  - Addressing flexibility
  - Dynamic storage management

- **Fundamental Pointer Operations:**
  - Assignment of an address to a pointer
  - References (explicit versus implicit dereferencing)

- **Design Issues**
  - What is the scope and lifetime of pointer variables?
  - What is the lifetime of heap-dynamic variables?
  - Are pointers restricted to pointing at a particular type?
  - Are pointers used for dynamic storage management, indirect addressing, or both?
  - Should a language support pointer types, reference types, or both?
  - Pointer arithmetic?

- **Problems with Pointers**

  - Dangling pointers
    - A pointer points to a heap-dynamic variable that has been deallocated

  - Lost Heap-Dynamic Variables
    - A heap-dynamic variable that is no longer referenced by any program pointer
    - The process of losing heap-dynamic variables is called memory leakage

- Pointers are like goto's - they widen the range of cells that can be accessed by a variable
- Pointers are necessary - so we can't design a language without them

- **Examples:**
  - **Pascal:** used for dynamic storage management only
    - Explicit dereferencing
    - Dangling pointers are possible (dispose)
    - Dangling objects are also possible

  - **Ada:** a little better than Pascal and Modula-2
    - Some dangling pointers are disallowed because dynamic objects can be automatically deallocated at the end of pointer's scope
    - All pointers are initialized to null
    - Similar dangling object problem (but rarely happens)

  - **C/C++:** Used for dynamic storage management and addressing
    - Explicit dereferencing and address-of operator
    - Can do address arithmetic in restricted forms
    - Domain type need not be fixed (void *)
    - ```
      float stuff[100];
      float *p;
      p = stuff;
      ```
    - *(p+5) is equivalent to stuff [5] and p [5]
    - *(p+i) is equivalent to stuff[i] and p[i]
    - void * - can point to any type and can be type checked (cannot be dereferenced)

  - **Fortran90:** Can point to heap and non-heap variables
    - Implicit dereferencing
    - Special assignment operator for non dereferenced references

  - **C++ Reference Types**
    - Constant pointers that are implicitly dereferenced
    - Used for parameters
    - Advantages of both pass-by-reference and pass-by-value

  - **Java - Only references**
    - No pointer arithmetic
    - Can only point at objects (which are all on the heap)
    - No explicit deallocator (garbage collection is used)
    - Means there can be no dangling references
    - Dereferencing is always implicit

Unions

- **Description:**

  - A union is a type whose variables are allowed to store different type values at different times during execution
      ```

- **Design Issues for unions:**

  - What kind of type checking, if any, must be done?
  - Should unions be integrated with records?

- **Examples:**

  - FORTRAN
    - EQUIVALENCE
  - Algol has "discriminated unions"
    - Use a hidden tag to maintain the current type
    - Tag is implicitly set by assignment
    - References are legal only in conformity clauses
    - This runtime type selection is a safe method of accessing union objects
  - Pascal
    - Both discriminated and non-discriminated unions are used e.g.

```
type intreal = record tag : Boolean of
                 true  : (blint : integer);
                 false : (blreal : real);
               end;
```

    - Problem with Pascal's design is that type checking is ineffective

    - Reasons:
      - User can create inconsistent unions (because the tag can be individually assigned)

```
var blurb : intreal;
      x : real;
blurb.tagg := true;   { it is an integer }
blurb.blint := 47;    { ok }
blurb.tagg := false;  { it is a real }
x := blurb.blreal;    { assigns an integer
                        to a real }
```

      - The tag is optional!

  - Ada
    - Discriminated unions are safer than Pascal & Modula-2
    - Reasons
      - Tag must be present
      - It is impossible for the user to create an inconsistent union (because tag cannot be assigned by itself)
      - All assignments to the union must include the tag value)

  - C and C++ have free unions (no tags)
    - Not part of their records
    - No type checking of references

- Java has neither records nor unions but aggregate types can be created with classes, as in C++

- Unions are potentially unsafe in most languages (except Ada)

Arithmetic Expressions

- **Description:**

  - Arithmetic evaluation was one of the motivations for the development of the first programming languages
  - Arithmetic expressions consist of operators, operands, parentheses, and function calls
  - 15 * (a + b) / log(x)

- **Design issues for arithmetic expressions:**

  - What are the operator precedence rules?

  - What are the operator associativity rules?

  - What is the order of operand evaluation?

  - Are there restrictions on operand evaluation side effects?

  - Does the language allow user-defined operator overloading?

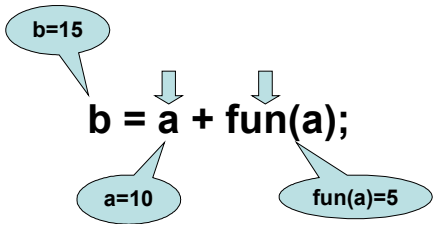
  - What mode mixing is allowed in expressions?

  - Conditional expressions?

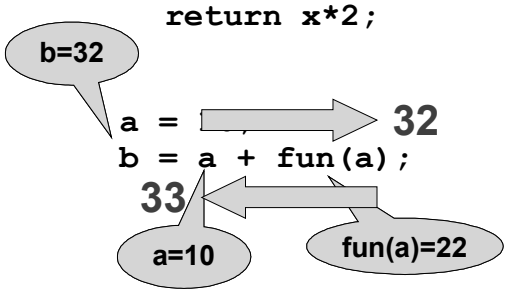
Modern Programming Languages Lecture 43

- **Arithmetic Expressions: Operators**
  - A unary operator has one operand
  - A binary operator has two operands
  - A ternary operator has three operands
- **Arithmetic Expressions: Operator Precedence Rules**
  - The operator precedence rules for expression evaluation define the order in which "adjacent" operators of different precedence levels are evaluated
  - Typical precedence levels
    - parentheses
    - unary operators
    - ** (if the language supports it)
    - *, /
    - +, -
- **Arithmetic Expressions: Potentials for Side Effects**
  - Functional side effects: when a function changes a two-way parameter or a non-local variable
  - Problem with functional side effects:
    - When a function referenced in an expression alters another operand of the expression; e.g., for a parameter change:

```
a = 10;
/* assume that fun changes its parameter */
b = a + fun(a);
```




```
int fun(int &x)
{
    x = x + 1;
    return x*2;
```



- **Functional Side Effects**
  - Two possible solutions to the problem
    - Write the language definition to disallow functional side effects
      - No two-way parameters in functions
      - No non-local references in functions
      - **Advantage:** it works!
      - **Disadvantage:** inflexibility of two-way parameters and non-local references
  - Write the language definition to demand that operand evaluation order be fixed
    - **Disadvantage:** limits some compiler optimizations

- **Operator Overloading**
  - Use of an operator for more than one purpose is called operator overloading
  - Some overloaded operators are common (e.g. '+' for int and float)
  - Some are potential trouble (e.g. '*' in C and C++)
    - * is also used for pointers
  - Loss of compiler error detection (omission of an operand should be a detectable error)
  - Some loss of readability
  - Can be avoided by introduction of new symbols (e.g., Pascal's div for integer division)
  - C++ and Ada allow user-defined overloaded operators
  - Potential problems
    - Users can define nonsense operations
    - Readability may suffer

- **Type Conversion**

  - Implicit Type Conversion
  - Explicit Type Conversion

- **Implicit Type Conversions**

  - A **narrowing conversion** is one that converts an object to a type that cannot include all of the values of the original type e.g., float to int
  - A **widening conversion** is one in which an object is converted to a type that can include at least approximations to all of the values of the original type e.g., int to float
  - A **mixed-mode expression** is one that has operands of different types
  - **Coercion** is an implicit type conversion
  - **Disadvantage of coercions:**
    - They decrease in the type error detection ability of the compiler
  - In most languages, all numeric types are coerced in expressions, using widening conversions
  - In Ada, there are virtually no coercions in expressions

- **Explicit Type Conversions**
  - Called casting in C-based language
  - Examples
    - C: (int) angle
    - Ada: Float (sum)
  - Note that Ada's syntax is similar to function calls

- **Errors in Expressions**
  - Causes
    - Coercions of operands in expressions
    - Inherent limitations of arithmetic e.g., division by zero
    - Limitations of computer arithmetic e.g. overflow or underflow
  - Division by zero, overflow, and underflow are run-time errors (sometimes called exceptions)

Relational Expressions

- Use relational operators and operands of various types

- Evaluate to some boolean representation

- Operator symbols used vary somewhat among different languages (!=, /=, .NE., <>, #)

Boolean Expressions

- Operands are Boolean and the result is also a Boolean

- Operators

FORTRAN 77	FORTRAN 90	C	Ada
AND	and	&&	and
OR	or	||	or
NOT	not	!	not
	xor		

Boolean Expressions

One odd characteristic of C's expressions
a < b < c

Short Circuit Evaluation

- A and B

- A or B

- *Example*

```
index := 1;
while (index <= length) and
      (LIST[index] <> value) do
  index := index + 1
```

C, C++, and Java*:* use short-circuit evaluation for the usual Boolean operators (&& and ||), but also provide bitwise Boolean operators that are not short circuit (& and |)

Ada: programmer can specify either (short-circuit is specified with 'and then' and 'or else')

FORTRAN 77: short circuiting is there, but any side affected place must be set to undefined

Problem with Short Circuiting

(a > b) || (b++ / 3)

Short-circuit evaluation exposes the potential problem of side effects in expressions

Modern Programming Languages Lecture 44

Control Structure

- Def: A *control structure* is a control statement and the statements whose execution it controls

- *Levels of Control Flow*
  1. Within expressions
  2. Among program units
  3. Among program statements

- *Overall Design Question*

What control statements should a language has, beyond selection and pretest logical loops?

Evolution

- FORTRAN I control statements were based directly on IBM 704 hardware
- Much research and argument in the1960s about the issue
- One important result: It was proven that all flowcharts can be coded with only two-way selection and pretest logical loops

Selection Statements
Design Issues

1. What is the form and type of the control expression?

2. What is the selectable segment form (single statement, statement sequence, compound statement)?

3. How should the meaning of nested selectors be specified?

Single-Way Selection Statement

FORTRAN IF
- IF (boolean_expr) statement

Problem
- *C*an select only a single statement;
- To select more, a goto must be used

FORTRAN example:

```
   IF (.NOT. condition) GOTO 20
     ...
     ...
20 CONTINUE
```

ALGOL 60 if:

```
if (boolean_expr) then
  begin
  ...
  end
```

Two-way Selector

***ALGOL 60* if:**

```
if (boolean_expr)
  then statement  (the then clause)
  else statement  (the else clause)
```

- The statements could be single or compound

Nested Selectors

Example (Pascal)

```
if ... then
if ... then
    ...
else ...
```

- Which then gets the else?

- Pascal's rule: else goes with the nearest then

Nested Selectors

ALGOL 60's solution - disallow direct nesting

```
if ... then          if ... then
  begin                begin
  if ...               if ... then ...
    then ...             end
    else ...           else ...
  end
```

FORTRAN 77, Ada, Modula-2 solution – closing special words

Example (Ada)

```
if ... then          if ... then
  if ... then          if ... then
    ...                  ...
  else                   end if
    ...              else
```

```
  end if              ...
end if              end if
```

- *Advantage:* flexibility and readability

- Modula-2 uses the same closing special word for all control structures (END)

- This results in poor readability

Multiple Selection Constructs

Design Issues

1. What is the form and type of the control expression?

2. What segments are selectable (single, compound, sequential)?

3. Is the entire construct encapsulated?

4. Is execution flow through the structure restricted
   to include just a single selectable segment?

5. What is done about un-represented expression
   values?

Early Multiple Selectors

1. FORTRAN arithmetic IF (a three-way selector)
   - IF (arithmetic expression) N1, N2, N3

   Bad aspects:
   - Not encapsulated
     (selectable segments could be anywhere)
   - Segments require GOTO's

Modern Multiple Selectors

1. Pascal case (from Hoare's contribution to ALGOL W)

```
case expression of
  constant_list_1 : statement_1;
  ...
  constant_list_n : statement_n
end
```

Design Choices

1. Expression is any ordinal type
   (int, boolean, char, enum)
2. Only one segment can be executed per

execution of the construct
3. In Wirth's Pascal, result of an un-represented control expression value is undefined
(In 1984 ISO Standard, it is a runtime error)

- Many dialects now have otherwise or else clause

The C and C++ switch

```
switch (expression)  {
   constant_expression_1 : statement_1;
   ...
   constant_expression_n : statement_n;
   [default: statement_n+1]
}
```

Any number of segments can be executed in one execution of the construct (there is no implicit branch at the end of selectable segments)

- Trade-off between reliability and flexibility (convenience)

- To avoid it, the programmer must supply a break statement for each segment

Ada's case is similar to Pascal's case, except:

1. Constant lists can include:
   - Subranges e.g., 10..15
   - Boolean OR operators
     e.g. 1..5 | 7 | 15..20

2. Lists of constants must be exhaustive
   - Often accomplished with others clause
   - This makes it more reliable

- Multiple Selectors can appear as direct extensions to two-way selectors, using else-if clauses (ALGOL 68, FORTRAN 77, Modula-2, Ada)

Example (Ada)

```
if ...
  then ...
elsif ...
  then ...
elsif ...
  then ...
  else ...
```

end if

- Far more readable than deeply nested if's
- Allows a boolean gate on every selectable group

Iterative Statements

- The repeated execution of a statement or compound statement is accomplished either by iteration or recursion; here we look at iteration, because recursion is a unit-level control

- *General design Issues for iteration control statements are:*
  1. How is iteration controlled?
  2. Where is the control mechanism in the loop?

Counter-Controlled Loops

Design Issues

1. What is the type and scope of the loop variable?

2. What is the value of the loop variable at loop termination?

3. Should it be legal for the loop variable or loop parameters to be changed in the loop body, and if so, does the change affect loop control?

4. Should the loop parameters be evaluated only once, or once for every iteration?

1. *FORTRAN 77 and 90*

- Syntax: DO label var = start, finish [, stepsize]

  - Stepsize can be any value but zero
  - Parameters can be expressions

- Design choices:
  1. Loop variables can be INTEGER, REAL, or DOUBLE
  2. Loop variable always has its last value
  3. The loop variable cannot be changed in the loop, but the parameters can; because they are evaluated only once, it does not affect loop control
  4. Loop parameters are evaluated only once

FORTRAN 90's 'Other DO'

- Syntax:
  [name:] DO variable = initial, terminal [, stepsize]
  …
  END DO [name]

- Loop variable must be an INTEGER

2. *ALGOL 60*

- Syntax: for var := <list_of_stuff> do statement
  where <list_of_stuff> can have:
  - List of expressions
  - expression step expression until expression
  - expression while boolean_expression

```
for index := 1 step 2 until 50,
        60, 70, 80,
        index + 1 until 100 do
```

(index = 1, 3, 5, 7, ..., 49, 60, 70, 80,
81, 82, ..., 100)

ALGOL 60 Design choices

1. Control expression can be int or real; its scope is whatever it is declared to be

2. Control variable has its last assigned value after loop termination

3. Parameters are evaluated with every iteration, making it very complex and difficult to read

4. The loop variable cannot be changed in the loop, but the parameters can, and when they are, it affects loop control

3. Pascal

- Syntax:
  for variable := initial (to | downto) final do
  statement

- Design Choices:
  1. Loop variable must be an ordinal type of usual scope
  2. After normal termination, loop variable is undefined
  3. The loop variable cannot be changed in the loop; the loop parameters can be changed, but they are evaluated just once, so it does not affect loop

control

4. *Ada*

- Syntax:

```
for var in [reverse] discrete_range loop
  ...
end loop
```

Ada Design choices

1. Type of the loop var is that of the discrete range; its scope is the loop body (it is implicitly declared)
2. The loop var does not exist outside the loop
3. The loop var cannot be changed in the loop, but the discrete range can; it does not affect loop control
4. The discrete range is evaluated just once

5. C

- Syntax:

```
for ([expr_1] ; [expr_2] ; [expr_3]) statement
```

- The expressions can be whole statements, or even statement sequences, with the statements separated by commas
- The value of a multiple-statement expression is the value of the last statement in the expression
  e.g.

```
for (i = 0, j = 10; j == i;  i++) ...
```

- If the second expression is absent, it is an infinite loop

-C Design Choices

1. There is no explicit loop variable
2. Irrelevant
3. Everything can be changed in the loop
4. Pretest
5. The first expression is evaluated once, but the other two are evaluated with each iteration

- This loop statement is the most flexible

6. C++

- Differs from C in two ways:
  1. The control expression can also be Boolean
  2. The initial expression can include variable definitions (scope is from the definition to the end of the function in which it is defined)

7. Java

- Differs from C++ in two ways:
  1. Control expression must be Boolean
  2. Scope of variables defined in the initial expression is only the loop body

Logically-Controlled Loops

- Design Issues
  1. Pre-test or post-test?
  2. Should this be a special case of the counting loop statement (or a separate statement)?

Examples

1. Pascal has separate pretest and post-test logical loop statements (while-do and repeat-until)

2. C and C++ also have both, but the control expression for the post-test version is treated just like in the pretest case (while - do and do - while)

3. Java is like C, except the control expression must be Boolean (and the body can only be entered at the beginning- Java has no goto)

4. Ada has a pretest version, but no post-test

5. FORTRAN 77 and 90 have neither

User-Located Loop Control Mechanisms

- Design issues

1. Should the conditional be part of the exit?
2. Should the mechanism be allowed in an already controlled loop?
3. Should control be transferable out of more than one loop?

1. C , C++, and Java - break

- Unconditional; for any loop or switch; one level only (Java's can have a label)
- There is also a continue statement for loops; it skips the remainder of this iteration, but does not exit the loop

2. FORTRAN 90 - EXIT

- Unconditional; for any loop, any number of levels
- FORTRAN 90 also has a CYCLE, which has the same semantics as C's continue

Examples:

1. Ada - conditional or unconditional exit; for any loop and any number of levels

```
for ... loop
  ...
  exit when ...
  ...
end loop
```

Example (Ada)

```
LOOP1:
  while ... loop
            ...
    LOOP2:
          for ... loop
    ...
    exit LOOP1 when ..
    ...
end loop LOOP2;
  ...
  end loop LOOP1;
```

Unconditional Branching

- All classical languages have it

- Problem: readability
  Edsger W. Dijkstra,
-Go To Statement Considered Harmful,
  CACM, Vol. 11, No. 3, March 1968, pp. 147-148
- Some languages do not have them e.g. Modula-2 and Java
- Advocates: Flexibility
  KNUTH D E, Structured programming with go to statements, ACM Computing Surveys 6, 4 (1974)

Conclusion

Choice of control statements beyond selection and logical pretest loops is a trade-off between language size and writability!

Connection between control statements and

program verification is intimate

- Verification is impossible with goto's
- Verification is possible with only selection and logical pretest loops
- Verification is relatively simple with only guarded commands

Modern Programming Languages Lecture 45

Actual/Formal Parameter Correspondence:

1. Positional
2. Keyword

e.g. SORT(LIST => A, LENGTH => N);

Default Parameter Values

```
procedure SORT(LIST : LIST_TYPE;
               LENGTH : INTEGER := 100);
...
SORT(LIST => A);
```

Parameters and Parameter Passing

Semantic Models

- in mode, out mode, in-out mode

Conceptual Models of Transfer

1. Physically move a value

2. Move an access path

Implementation Models

1. Pass-by-value (in mode)

- Either by physical move or access path

2. Pass-by-result (out mode)

- Local's value is passed back to the caller

- Physical move is usually used

- Disadvantages: Collision

Example:

```
procedure sub1(y: int, z: int);
            ...

sub1(x, x);
```

3. Pass-by-reference (in-out mode)

i. Actual parameter collisions

e.g.

```
procedure sub1(a: int, b: int);
```

```
...
sub1(x, x);
```

ii. Array element collisions

e.g.

```
sub1(a[i], a[j]);  /* if i = j  */
```

iii. Collision between formals and globals

Root cause of all of these is: The called subprogram is provided wider access to Non-locals than is necessary

Type checking parameters

- Now considered very important for reliability

FORTRAN 77 and original C: none

Pascal, Modula-2, FORTRAN 90, Java, and Ada: it is always required

- ANSI C and C++: choice is made by the user

Implementing Parameter Passing

ALGOL 60 and most of its descendants use the run-time stack

- Value copied to the stack; references are indirect to the stack

- Result : same

- Reference : regardless of form, put the address on the stack

Design Considerations for Parameter Passing

1. Efficiency
2. One-way or two-way

- These two are in conflict with one another!

Good programming =>
limited access to variables, which means one-way whenever possible

Efficiency =>
pass by reference is fastest way to pass structures of significant size

Concluding Remarks

- Different programming languages for different problems
  Imperative, Declarative, Functional, Object-Oriented, Scripting
- Readability – maintainability

- Side effects
- Mainly because of the assignment statement and aliasing
- Reduces readability and causes errors
- Operand evaluation order
- Data types and data sizes
- Static-Dynamic activation records
- Necessary for recursion