Lesson 17

• Concepts of programming languages: an introduction
Programming linguistics

- **Programming linguistics** is the study of programming languages (PLs).
- This is by analogy with *linguistics*, the study of natural languages (NLs):
  - Both PLs and NLs have syntax (form) and semantics (meaning).
- However, NLs are far broader, more expressive, and more subtle than PLs.
- Also, linguists are limited to studying existing NLs. Computing scientists can design, specify, and implement new PLs.
Expected properties of PLs

• A PL must be **universal** – capable of expressing any computation.
  – A language without iteration or recursion is not universal.
  – A language of recursive functions (and nothing else) is universal.
• A PL should be reasonably *natural* for expressing computations in its intended application area.
• A PL must be **implementable** – it must be possible to run every program on a computer.
• A PL should be capable of acceptably *efficient* implementation.
Concepts

• **Concepts** are building blocks of programs and PL’s:
  – Names, bindings and scope
  – Values and data types
  – Variables and storage management
  – Control abstraction
  – Data abstraction
  – Generic abstraction
  – Concurrency
Paradigms

• A **paradigm** is a style of programming, characterized by a particular selection of key concepts.

• **Imperative programming**: variables, commands, procedures.

• **Object-oriented (OO) programming**: objects, methods, classes.

• **Concurrent programming**: processes, communication.

• **Functional programming**: values, expressions, functions.

• **Logic programming**: assertions, relations.
Syntax, semantics, and pragmatics

• A PL’s **syntax** is concerned with the form of programs: how expressions, commands, declarations, and other constructs must be arranged to make a well-formed program.

• A PL’s **semantics** is concerned with the meaning of (well-formed) programs: how a program may be expected to behave when executed on a computer.

• A PL’s **pragmatics** is concerned with the way in which the PL is intended to be used in practice. Pragmatics include the paradigm(s) supported by the PL.
Objectives of this part of PLP

- Improve the background for choosing appropriate programming languages
- Enhance the ability to learn new programming languages
- Increase the capacity to express general programming concepts
- Increase ability to choose among alternative ways to express things in a particular programming language
- Simulate useful features in languages that lack them
- Be able, in principle, to design a new programming language
Why study programming languages?

• **Help you choose a language.**
  – C vs. Modula-3 vs. C++ for systems programming
  – Fortran vs. APL vs. Ada for numerical computations
  – Ada vs. Modula-2 for embedded systems
  – Common Lisp vs. Scheme vs. ML for symbolic data manipulation
  – Java vs. C/CORBA for networked PC programs
Why study programming languages?

• Make it easier to learn new languages
  – some languages are similar: easy to walk down family tree
  – concepts have even more similarity; if you think in terms of iteration, recursion, abstraction (for example), you will find it easier to assimilate the syntax and semantic details of a new language than if you try to pick it up in a vacuum.
  – Think of an analogy to human languages: good grasp of grammar makes it easier to pick up new languages (at least Indo-European).
Why study programming languages?

• Help you make better use of whatever language you use

  – understand obscure features:

    • In C, help you understand unions, arrays & pointers, separate compilation, varargs, catch and throw

    • In Common Lisp, help you understand first-class functions/closures, streams, catch and throw, symbol internals
Why study programming languages?

- Help you make better use of whatever language you use (2)
  - understand implementation costs: choose between alternative ways of doing things, based on knowledge of what will be done underneath:
    - use simple arithmetic equal (use $x^2$ instead of $x^{**2}$)
    - use C pointers or Pascal "with" statement to factor address calculations
    - avoid call by value with large data items in Pascal
    - avoid the use of call by name in Algol 60
    - choose between computation and table lookup (e.g. for cardinality operator in C or C++)
Why study programming languages?

• Help you make better use of whatever language you use (3)
  – figure out how to do things in languages that don't support them explicitly:
    • lack of suitable control structures in Fortran
      – use comments and programmer discipline for control structures
    • lack of recursion in Fortran, CSP, etc
      – write a recursive algorithm then use mechanical recursion elimination (even for things that aren't quite tail recursive)
Why study programming languages?

- Help you make better use of whatever language you use (4)
  - figure out how to do things in languages that don't support them explicitly:
    - lack of named constants and enumerations in Fortran
      » use variables that are initialized once, then never changed
    - lack of modules in C and Pascal
      » use comments and programmer discipline
    - lack of iterators in just about everything
      » fake them with (member?) functions
A brief Programming Language History

• 1940s: The first electronic computers were monstrous contraptions
  – Programmed in binary machine code by hand via switches and later by card readers and paper tape readers
  – Code is not reusable or relocatable
  – Computation and machine maintenance were difficult: machines had short mean-time to failure (MTTF) because vacuum tubes regularly burned out
  – The term “bug” originated from a bug that reportedly roamed around in a machine causing short circuits

ENIAC (1946)
Assembly Languages

- **Assembly languages** were invented to allow machine operations to be expressed in mnemonic abbreviations
  - Enables larger, reusable, and relocatable programs
  - Actual machine code is produced by an assembler
  - Early assemblers had a one-to-one correspondence between assembly and machine instructions
Assembly Language Example

Example MIPS assembly program to compute GCD

Example MIPS R4000 machine code of the assembly program

27bdff0d a6bf0014 0c1002a8 00000000
0c1002a8 a6f2001c 8fa4001c
00401825 10820008 0064082a 10200003
00000000 10000002 00832023
00641823 14b3fffa 0064082a 0c1002b2
00000000 8f6f0014 27bd0020
03e00008 00001025

Actual MIPS R4400MC
The First High-Level Programming Language

• Mid 1950s: development of FORTRAN (FORmula TRANslator), the arguably first higher-level language
  – Programs could be developed that were machine independent

• Main computing activity in the 50s: solve numerical problems in science and engineering

• Other high-level languages soon followed:
  – **Algol 58** was an improvement compared to Fortran
  – **COBOL** for business computing
  – **Lisp** for symbolic computing and artifical intelligence
  – **BASIC** for "beginners"
  – **C** for systems programming
Genealogy of Programming Languages
PROGRAM AVEX
INTEGER IN TLST(99)
ISUM = 0
C read the length of the list
READ (*, *) LSTLEN
IF ((LSTLEN .GT. 0) .AND. (LSTLEN .LT. 100)) THEN
C read the input in an array
DO 100 ICTR = 1, LSTLEN
READ (*, *) INTLST(ICTR)
ISUM = ISUM + INTLST(ICTR)
100 CONTINUE
C compute the average
IAVE = ISUM / LSTLEN
C write the input values > average
DO 110 ICTR = 1, LSTLEN
IF (INTLST(ICTR) .GT. IAVE) THEN
WRITE (*, *) INTLST(ICTR)
END IF
110 CONTINUE
ELSE
WRITE (*, *) 'ERROR IN LIST LENGTH'
END IF
END

• FORTRAN is still widely used for scientific, engineering, and numerical problems, mainly because very good compilers exist
• In the early days skeptics wrongly predicted that compilers could not beat hand-written machine code
• FORTRAN 77 has
  – Subroutines, if-then-else, do-loops
  – Types (primitive and arrays)
  – Variable names are upper case and limited to 6 chars
  – No recursion
  – No structs/classes, unions
  – No dynamic allocation
  – No case-statements and no while-loops
FORTRAN 90,95,HPF

PROGRAM AVEX
INTEGER INT_LIST(1:99)
INTEGER LIST_LEN, COUNTER, AVERAGE
C read the length of the list
READ (*, *) LISTLEN
IF ((LIST_LEN > 0) .AND. (LIST_LEN < 100)) THEN
C read the input in an array
DO COUNTER = 1, LIST_LEN
READ (*, *) INT_LIST(COUNTER)
END DO
C compute the average
AVERAGE = SUM(INT_LIST(1:LIST_LEN)) / LIST_LEN
C write the input values > average
DO COUNTER = 1, LIST_LEN
IF (INT_LIST(COUNTER) > AVERAGE) THEN
WRITE (*, *) INT_LIST(COUNTER)
END IF
END DO
ELSE
WRITE (*, *) 'ERROR IN LIST LENGTH'
END IF
END
(DEFINE (avex lis)  
  (filtergreater lis (/ (sum lis) (length lis))))  
)

(DEFINE (sum lis)  
  (COND  
    ((NULL? lis) 0)  
    (ELSE (+ (CAR lis) (sum (CDR lis)))))  
  )  
)

(DEFINE (filtergreater lis num)  
  (COND  
    ((NULL? lis) '())  
    (>(CAR lis) num) (CONS (CAR lis)  
      (filtergreater (CDR lis) num)))  
    (ELSE (filtergreater (CDR lis) num)  
      )  
  )  
)

- Lisp (LIst Processing)
- The original functional language developed by McCarthy as a realization of Church's lambda calculus
- Many dialects exist, including Common Lisp and Scheme
- Very powerful for symbolic computation with lists
- Implicit memory management with garbage collection
- Influenced functional programming languages (ML, Miranda, Haskell)
comment avex program
begin
    integer array intlist [1:99];
    integer listlen, counter, sum, average;
    sum := 0;
    comment read the length of the input list
readint (listlen);
    if (listlen > 0) L (listlen < 100) then
        begin
            comment read the input into an array
for counter := 1 step 1 until listlen do
begin
    readint (intlist[counter]);
    sum := sum + intlist[counter]
end;
            comment compute the average
        average := sum / listlen;
            comment write the input values > average
for counter := 1 step 1 until listlen do
    if intlist[counter] > average then
        printint (intlist[counter])
    end
else
    printstring ("Error in input list length")
end
• The original block-structured language
  – Local variables in a statement block
• First use of Backus-Naur Form (BNF) to formally define language grammar
• All subsequent imperative programming languages are based on it
• No I/O and no character set
• Not widely used in the US
• Unsuccessful successor Algol 68 is large and relatively complex
IDENTIFICATION DIVISION.
PROGRAM-ID. EXAMPLE.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-370.
OBJECT-COMPUTER. IBM-370.

DATA DIVISION.
WORKING-STORAGE SECTION.
77 FAHR PICTURE 999.
77 CENT PICTURE 999.

PROCEDURE DIVISION.
DISPLAY 'Enter Fahrenheit ' UPON CONSOLE.
ACCEPT FAHR FROM CONSOLE.
DISPLAY 'Celsius is ' CENT UPON CONSOLE.
GOBACK.

• Originally developed by the Department of Defense
• Intended for business data processing
• Extensive numerical formatting features and decimal number storage
• Introduced the concept of records and nested selection statement
• Programs organized in divisions:
  IDENTIFICATION: Program identification
  ENVIRONMENT: Types of computers used
  DATA: Buffers, constants, work areas
  PROCEDURE: The processing parts (program logic).
REM avex program
DIM intlist(99)
sum = 0
REM read the length of the input list
INPUT listlen
IF listlen > 0 AND listlen < 100 THEN
REM read the input into an array
FOR counter = 1 TO listlen
   INPUT intlist(counter)
   sum = sum + intlist(counter)
NEXT counter
REM compute the average
average = sum / listlen
REM write the input values > average
FOR counter = 1 TO listlen
   IF intlist(counter) > average THEN
      PRINT intlist(counter);
   NEXT counter
ELSE
   PRINT "Error in input list length"
END IF
END

• BASIC (Beginner’s All-Purpose Symbolic Instruction Code)
• Intended for interactive use (interpreted) and easy for "beginners"
• Goals: easy to learn and use for non-science students
• Structure of early basic dialects were similar to Fortran
• Classic Basic
• QuickBasic (see example)
• MS Visual Basic is a popular dialect
AVEX: PROCEDURE OPTIONS (MAIN);

DECLARE INTLIST (1:99) FIXED;
DECLARE (LISTLEN, COUNTER, SUM, AVERAGE) FIXED;
SUM = 0;
/* read the input list length */
GET LIST (LISTLEN);
IF (LISTLEN > 0) & (LISTLEN < 100) THEN DO;
    /* read the input into an array */
    DO COUNTER = 1 TO LISTLEN;
        GET LIST (INTLIST(COUNTER));
        SUM = SUM + INTLIST(COUNTER);
    END;
    /* compute the average */
    AVERAGE = SUM / LISTLEN;
    /* write the input values > average */
    DO COUNTER = 1 TO LISTLEN;
        IF INTLIST(COUNTER) > AVERAGE THEN
            PUT LIST (INTLIST(COUNTER));
    END;
ELSE
    PUT SKIP LIST ('ERROR IN INPUT LIST LENGTH');
END AVEX;

• Developed by IBM
  – Intended to replace FORTRAN, COBOL, and Algol
• Introduced exception handling
• First language with pointer data type
• Poorly designed, too large, too complex
C (ANSI C, K&R C)

- Statically typed, general purpose systems programming language
- Computational model reflects underlying machine
- Relationship between arrays and pointers
  - An array is treated as a pointer to first element
  - E1[E2] is equivalent to ptr dereference: *((E1)+(E2))
  - Pointer arithmetic is not common in other languages
- Not statically type safe
  - If variable has type float, no guarantee value is floating pt
- Ritchie quote
  - “C is quirky, flawed, and a tremendous success”
C (ANSI C, K&R C)

main()
{  
  int intlist[99], listlen, counter, sum, average;
  sum = 0;
  /* read the length of the list */
  scanf("%d", &listlen);
  if (listlen > 0 && listlen < 100)
    {  
      /* read the input into an array */
      for (counter = 0; counter < listlen; counter++)
        {   
          scanf("%d", &intlist[counter]);
          sum += intlist[counter];
        }
      /* compute the average */
      average = sum / listlen;
      /* write the input values > average */
      for (counter = 0; counter < listlen; counter++)
        if (intlist[counter] > average)
          printf("%d\n", intlist[counter]);
    }
  else
    printf("Error in input list length\n");
}
The most widely used logic programming language

- Declarative: states what you want, not how to get it
- Based on Resolution: 
  \((A \lor B) \land (\neg A \lor C) \rightarrow B \lor C\)
program avex(input, output);
  type
    intlisttype = array [1..99] of integer;
  var
    intlist : intlisttype;
    listlen, counter, sum, average : integer;
begin
  sum := 0;
  (* read the length of the input list *)
  readln(listlen);
  if ((listlen > 0) and (listlen < 100)) then
    begin
      (* read the input into an array *)
      for counter := 1 to listlen do
        begin
          readln(intlist[counter]);
          sum := sum + intlist[counter]
        end;
      (* compute the average *)
      average := sum / listlen;
      (* write the input values > average *)
      for counter := 1 to listlen do
        if (intlist[counter] > average) then
          writeln(intlist[counter])
      end
    else
      writeln('Error in input list length')
  end.

• Designed by Swiss professor Niklaus Wirth
• Designed for teaching "structured programming"
• Small and simple
• Had a strong influence on subsequent high-level languages Ada, ML, Modula
ML

- Statically typed, general-purpose programming language
  - “Meta-Language” of the LCF theorem proving system
- Type safe, with formal semantics
- Compiled language, but intended for interactive use
- Combination of Lisp and Algol-like features
  - Expression-oriented
  - Higher-order functions
  - Garbage collection
  - Abstract data types
  - Module system
  - Exceptions
- Used in printed textbook as example language

Robin Milner, ACM Turing-Award for ML, LCF Theorem Prover, …
Towards OO programming

- 1980s: Object-oriented programming
  - Important innovation for software development
    - Encapsulation and inheritance
    - Dynamic binding
  - The concept of a “class” is based on the notion of an “abstract data type” (ADT) in Simula 67, a language for discrete event simulation that has class-like types but no inheritance
Smalltalk-80

class name          Avex
superclass          Object
instance variable names  intlist

"Class methods"
"Create an instance"
new
^ super new

"Instance methods"
"Initialize"
initialize
  intlist <- Array new: 0

"Add int to list"
add: n | oldintlist |
  oldintlist <- intlist.
  intlist <- Array new: intlist size + 1.
  intlist <- replaceFrom: 1 to: intlist size with: oldintlist.
  ^ intlist at: intlist size put: n

"Calculate average"
average | sum |
sum <- 0.
1 to: intlist size do:
  [:index | sum <- sum + intlist at: index].
^ sum // intlist size

"Filter greater than average"
filtergreater: n | oldintlist i |
oldintlist <- intlist.
i <- 1.
1 to: oldintlist size do:
  [:index | (oldintlist at: index) > n
    ifTrue: [oldintlist at: i put: (oldintlist at: index)]]
intlist <- Array new: oldintlist size.
intlist replaceFrom: 1 to: oldintlist size with: oldintlist

• Developed by XEROX PARC: first IDE with windows-based graphical user interfaces (GUIs)
• The first full implementation of an object-oriented language
• Example run:

av <- Avex new
av initialize
av add: 1
1
av add: 2
2
av add: 3
3
av filtergreater: av average
av at: 1
3
with TEXT_IO;
use TEXT_IO;

procedure AVEX is
  package INT_IO is new INTEGER_IO (INTEGER);
  use INT_IO;
  type INT_LIST_TYPE is array (1..99) of INTEGER;
  INT_LIST : INT_LIST_TYPE;
  LIST_LEN, SUM, AVERAGE : INTEGER;
begin
  SUM := 0;
  -- read the length of the input list
  GET (LIST_LEN);
  if (LIST_LEN > 0) and (LIST_LEN < 100) then
    -- read the input into an array
    for COUNTER := 1 .. LIST_LEN loop
      GET (INT_LIST(COUNTER));
      SUM := SUM + INT_LIST(COUNTER);
    end loop;
    -- compute the average
    AVERAGE := SUM / LIST_LEN;
    -- write the input values > average
    for counter := 1 .. LIST_LEN loop
      if (INT_LIST(COUNTER) > AVERAGE) then
        PUT (INT_LIST(COUNTER));
        NEW_LINE;
      end if;
    end loop;
  else
    PUT_LINE ("Error in input list length");
  end if;
end AVEX;

• Originally intended to be the standard language for all software commissioned by the US Department of Defense
• Very large
• Elaborate support for packages, exception handling, generic program units, concurrency
• Ada 95 is a revision developed under government contract by a team at Intermetrics, Inc.
  – Adds objects, shared-memory synchronization, and several other features
Haskell

• The leading purely functional language, based on Miranda
• Similar to ML: general-purpose, strongly typed, higher-order, functional, curried functions, supports type inference, static polymorphic typing, pattern matching, interactive and compiled use, modules
• Different from ML: lazy evaluation, purely functional core, rapidly evolving type system, list comprehensions, monadic I/O, and layout (indentation)-based syntactic grouping

```
sum []     = 0
sum (a:x) = a + sum x

avex []   = []
avex (a:x) = [n | n <- a:x, n > sum (a:x) / length (a:x)]
```
```cpp
main()
{
    std::vector<int> intlist;
    int listlen;
    /* read the length of the list */
    std::cin >> listlen;
    if (listlen > 0 && listlen < 100)
    {
        int sum = 0;
        /* read the input into an STL vector */
        for (int counter = 0; counter < listlen; counter++)
        {
            int value;
            std::cin >> value;
            intlist.push_back(value);
            sum += value;
        }
        /* compute the average */
        int average = sum / listlen;
        /* write the input values > average */
        for (std::vector<int>::const_iterator it = intlist.begin(); it != intlist.end(); ++it)
        {
            if ((*it) > average)
                std::cout << (*it) << std::endl;
        }
    }
    else
        std::cerr << "Error in input list length" << std::endl;
}
```

- The most successful of several object-oriented successors of C
- Evolved from C and Simula 67
- Large and complex, partly because it supports both procedural and object-oriented programming
import java.io;
class Avex
{
    public static void main(String args[])
        throws IOException
    {
        DataInputStream in = new DataInputStream(System.in);
        int listlen, counter, sum = 0, average;
        int [] intlist = int[100];
        // read the length of the list
        listlen = Integer.parseInt(in.readLine());
        if (listlen > 0 && listlen < 100)
        {
            // read the input into an array
            for (counter = 0; counter < listlen; counter++)
            {
                intlist[counter] =
                Integer.valueOf(in.readLine()).intValue();
                sum += intlist[counter];
            }
            // compute the average
            average = sum / listlen;
            // write the input values > average
            for (counter = 0; counter < listlen; counter++)
            {
                if (intlist[counter] > average)
                {
                    System.out.println(intlist[counter] + "\n");
                }
            }
        }
        else
        {
            System.out.println("Error in input length\n");
        }
    }
}

- Developed by Sun Microsystems
- Based on C++, but significantly simplified
- Supports only object-oriented programming
- Safe language (e.g. no pointers but references, strongly typed, and implicit garbage collection)
- Portable and machine-independent with Java virtual machine (JVM)
Other Notable Languages

• C#
  – Similar to Java, but platform dependent (MS .NET)
  – Common Language Runtime (CLR) manages objects that can be shared among the different languages in .NET

• Simula 67
  – Based on Algol 60
  – Primarily designed for discrete-event simulation
  – Introduced concept of coroutines and the class concept for data abstraction

• APL
  – Intended for interactive use ("throw-away" programming)
  – Highly expressive functional language makes programs short, but hard to read

• Scripting languages
  – Perl, Python, Ruby, …
Why are There so Many Programming Languages?

• **Evolution**
  – Design considerations: What is a good or bad programming construct?
  – Early 70s: structured programming in which goto-based control flow was replaced by high-level constructs (e.g. while loops and case statements)
  – Late 80s: nested block structure gave way to object-oriented structures

• **Special Purposes**
  – Many languages were designed for a specific problem domain, e.g:
    • Scientific applications
    • Business applications
    • Artificial intelligence
    • Systems programming
    • Internet programming

• **Personal Preference**
  – The strength and variety of personal preference makes it unlikely that anyone will ever develop a universally accepted programming language
Most Research Languages

The quick death
Successful Research Languages

The slow death
C++, Java, Perl, Ruby

The complete absence of death

Practitioners

Geeks

Threshold of immortality
Committee languages

The slow death
What Makes a Programming Language Successful?

• **Expressive Power**
  – Theoretically, all languages are equally powerful (Turing complete)
  – Language features have a huge impact on the programmer's ability to read, write, maintain, and analyze programs
  – Abstraction facilities enhance expressive power

• **Ease of Use for Novice**
  – Low learning curve and often interpreted, e.g. Basic and Logo

• **Ease of Implementation**
  – Runs on virtually everything, e.g. Basic, Pascal, and Java

• **Open Source**
  • Freely available, e.g. Java

• **Excellent Compilers and Tools**
  – Fortran has extremely good compilers
  – Supporting tools to help the programmer manage very large projects

• **Economics, Patronage, and Inertia**
  – Powerful sponsor: Cobol, PL/I, Ada
  – Some languages remain widely used long after "better" alternatives
Classification of Programming Languages
## Classification of Programming Languages

<table>
<thead>
<tr>
<th>Declarative</th>
<th>Functional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implicit solution</td>
<td>(Lisp, Scheme, ML, Haskell)</td>
</tr>
<tr>
<td>&quot;What the computer should do&quot;</td>
<td>Logical</td>
</tr>
<tr>
<td></td>
<td>(Prolog)</td>
</tr>
<tr>
<td></td>
<td>Dataflow</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>Imperative</td>
<td>Procedural</td>
</tr>
<tr>
<td>Explicit solution</td>
<td>&quot;von Neumann&quot; (Fortran, C)</td>
</tr>
<tr>
<td>&quot;How the computer should do it&quot;</td>
<td>Object-oriented</td>
</tr>
<tr>
<td></td>
<td>(Smalltalk, C++, Java)</td>
</tr>
</tbody>
</table>
Contrasting Examples

**Procedural (C):**

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

**Logical (Prolog):**

```prolog
gcd(A, A, A).
gcd(A, B, G) :- A > B, N is A-B, gcd(N, B, G).
gcd(A, B, G) :- A < B, N is B-A, gcd(A, N, G).
```

**Functional (Haskell):**

```haskell
gcd a b
| a == b = a
| a > b = gcd (a-b) b
| a < b = gcd a (b-a)
```