Evolution of the Major Programming Languages

Lecture 02

Instructor: C. Pu (Ph.D., Assistant Professor)

puc@marshall.edu

MARSHALL

Zuse's Plankalkül

- Designed in 1945, but not published until 1972.
- Never implemented.
- So few people were familiar with the language, some of its capabilities did not appear in other languages until 15 years after its development.
- Zuse developed a language for expressing computations for the Z4, a project he had begun in 1943 as a proposal for his Ph.D. dissertation.
  - *Plankalkul*, means program calculus.
  - Zuse defined *Plankalkul,* wrote algorithm in the language to solve problems.

Zuse's Plankalkül: Language Review

- The simplest data type in Plankalhul was the single ***bit***
- Integer and floating-point numeric types were built from the bit type
- Plankalkul included ***arrays*** and ***records***
  - The ***records*** could include ***nested records***
- It had an ***iterative*** statement
- It included a ***selection*** statement without ***else*** clause
- One of the most interesting features is the inclusion of mathematical expressions showing the current relationships between program variables
  - These expressions stated what would be true during execution at the points in the code
  - Very similar to the assertions of Java

Zuse's Plankalkül: Language Review

- ***Assertions*** in Java
  - An assertion allows testing the correctness of any assumptions that have been made in the program.
  - While executing assertion, it is believed to be true. If it fails, JVM throws an error named ***AssertionError.***
  - It is mainly used for testing purposes during development.

```
class Test
{
    public static void main( String args[] )
    {
        int value = 15;
        assert value >= 20 : " Underweight";
        System.out.println("value is "+value);
    }
}
```

- Each statement consisted of either two or three lines of code
  - ***1st line***: like the ***statement*** of current language
  - ***2nd line***: optional, contains the ***subscripts*** of the array references in the first line
  - ***3rd line***: contains the ***type*** names for the variables mentioned in the first line
- Example: An assignment statement to assign the expression A[4] + 1 to A[5]

```
  | A + 1 => A
V | 4        5
S | 1.n      1.n
```

V: subscript
S: data type
1.n: an integer of n bits

Minimal Hardware Programming: Pseudocodes

- The computer that became available in the late 1940s and early 1950s were far less usable than those of today.
  - Slow
  - Unreliable
  - Expensive
  - Small memories
  - Difficult to program
- Programming was done in ***machine code***
  - ADD instruction might be specified by the code 14
- What was wrong with using machine code?
  - Poor readability
  - Poor modifiability
  - Expression coding was tedious
  - Machine deficiencies--no indexing or floating point

- Example of Machine code:
  - Statement in high-level language

```
sum := sum + count;
```

  - Machine code

```
0010011010011001
1101010011100101
1100010101001010
0100101101010010
1101011000101100
```

Pseudocodes: Short Code

- Short Code developed by Mauchly in 1949 for BINAC computers, which was one of the first successful stored-program electronic computers.

- Short Code was one of the primary means of programming those machines for several years.

- But, never published
  - Little is known of the original Short Code.

Pseudocodes: Short Code

- Short Code consisted of ***coded*** versions of mathematical expressions that were to be evaluated
- The codes were ***byte-pair*** values, and many equations could be coded in a ***word***
- The following operation codes were included

```
01  -      06  abs value      1n  (n+2)nd power
02  )      07  +              2n  (n+2)nd root
03  =      08  pause          4n  if <= n
04  /      09  (              58  print and tab
```

- The statement X0 = SQRT(ABS(Y0)) would be coded as

```
00 X0 03 20 06 Y0
```

(00: padding to fill the word)

MARSHALL

Pseudocodes: Short Code

- Short Code was not translated to machine code.
- It was implemented with a pure interpreter.
- Short Code interpretation was approximately 50 times slower than machine code.

- One of the primary reasons why the slowness of interpretive systems was tolerated from the late 1940s to the mid-1950s was the lack of floating-point hardware in the available computers
  - All floating-point operations had to be simulated in software, a very time-consuming process
- The announcement of the IBM 704 system, with both indexing and floating-point instructions in hardware, heralded the end of the interpretive era, at least for scientific computation
  - The inclusion of floating-point hardware removed the hiding place for the cost of interpretation

MARSHALL

Design Process of Fortran 0

- Fortran 0 in 1954, not implemented.
- The document stated that
  - Fortran would provide the efficiency of hand-coded programs and the ease of programming of the interpretive pseudocode systems.
  - Fortran would eliminate coding errors and the debugging process.
    - Included little syntax error checking.
  - Environment of development
    - Computers had small memories and were slow and relatively unreliable.
    - The primary use of computers was for scientific computations.
    - There were no existing efficient and effective ways to program computers.
    - Speed of the generated object code was the primary goal of the first Fortran compilers.

Fortran I Overview

- First implemented version of Fortran
- Fortran I included
  - input/output ***formatting***
  - variable names of up to six characters
  - user-defined ***subroutines***
  - the ***If*** selection statement
  - the ***Do loop*** statement
  - ***no data-typing*** statements
    - Variables whose names began with I, J, K, L, M, and N were implicitly integer type
    - All others were implicitly floating-point

- Fortran II compiler was distributed in the spring of 1958
  - It fixed many of the bugs in the Fortran I compilation system
  - It added some significant features to the language
    - The independent compilation of subroutines
    - Without independent compilation, any change in a program required that the entire program be recompiled

Fortran IV

- Fortran IV became one of the most widely used programming languages of its time
  - Evolved over the period 1960 to 1962
  - Standardized as Fortran 66 (ANSI standard)
  - Fortran IV was an improvement over Fortran II
    - ***Explicit type declarations*** for variables
    - Logical ***if*** statement
    - ***Subprogram*** names could be ***parameters***

Fortran 77

- Fortran IV was replaced by Fortran 77, which became the new standard in 1978
- Fortran 77 retained most of the features of Fortran IV, and added
  - Character ***string*** handling
  - Logical ***loop control*** statement
  - ***if-else*** statement

Fortran 90

- Fortran 90 (ANSI, 1992) was dramatically different from Fortran 77
  - ***Dynamic arrays***
  - ***Records***
  - ***Pointers***
  - A ***multiple selection*** statement
  - ***Modules***
  - Subprograms could be ***recursively*** called

- Fortran 95 – relatively minor additions, plus some deletions
- Fortran 2003 – support for ***OOP***
- Fortran 2008 – blocks for local ***scopes***

Functional Programming: Lisp

- The first functional programming language was invented to provide language features for list processing, the need for which grew out of the first applications in the area of artificial intelligence (AI)

- LISP has only two kinds of data structures
  - ***Atoms***
    - Atoms are either symbols, which have the form of identifiers, or numeric literals
  - ***Lists***
    - A sequence of atoms and/or other lists enclosed in parentheses
    - Allow insertions and deletions at any point
- Lists are specified by delimiting their elements with parentheses
  - Simple lists

```
(A B C D)
```

  - Nested list structures are also specified by parentheses

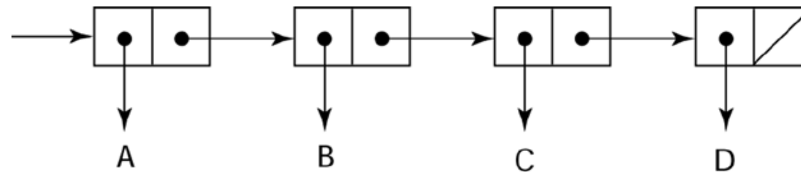
```
(A (B C) D (E (F G)))
```

LISP Overview: Data Structures

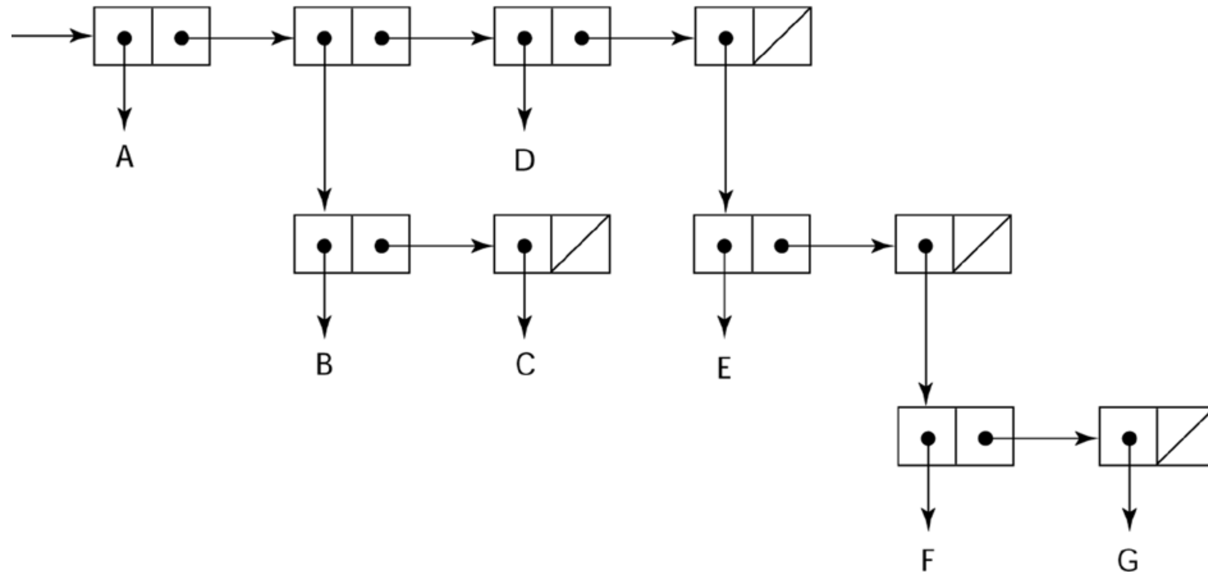
- Internally, lists are stored as single-linked list structures
  - Each node has two pointers and represents a list element
  - A node containing an atom has its first pointer pointing to some representation of the atom, such as its symbol or numeric value, or a pointer to a sublist
  - A node for a sublist element has its first pointer pointing to the first node of the sublist
  - In both cases, the second pointer of a node points to the next element of the list
  - A list is referenced by a pointer to its first element

Representation of Two Lisp Lists

(A B C D)



(A (B C) D (E (F G)))



LISP Overview: Processes

- LISP was designed as a functional programming language
- All computation in a purely functional program is accomplished by applying functions to arguments
- Neither the assignment statements nor the variables that abound in imperative language programs are necessary in functional language programs
- Repetitive processes can be specified with recursive function calls, making iteration (loops) unnecessary

LISP Overview: Syntax

- LISP is very different from the imperative languages
  - because it is a functional programming language and
  - because the appearance of LISP programs is so different from those in languages like Java or C++

- For example
  - The syntax of Java is a complicated mixture of English and algebra
  - LISP's syntax is a model of simplicity
    - Program code and data have exactly the same form: parenthesized lists

```
(A B C D)
```