Basic Components of Programming Languages

Steven Zeil

Aug. 11, 2003

Contents

Basic Components of Programming Languages

1. History
2. Classification
3. Translation

1 History

Themes:

- The medium affects the message.
- Budgeting for the lifetime of the software.
- Coping with complexity.

Most early commercial computers used separate media for instructions and data.

- punched cards for data
- patch boards for instructions

Only "iteration" was in moving from card to card, doing the same thing to each card.

---

stored-program computers held the instructions in the machines internal memory. (ENIAC, 1946)

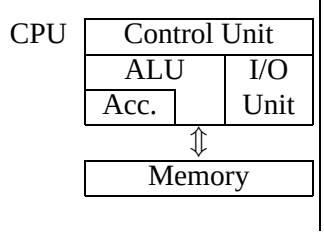
- more flexible control
- (later) programs could be manipulated as data

1.1 Lower-Level Languages

machine code is the native language of the processor.

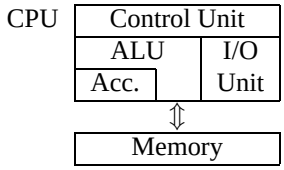
- binary numbers
- low-level operations

Classic "single accumulator machine" (Von Neuman, 1947):



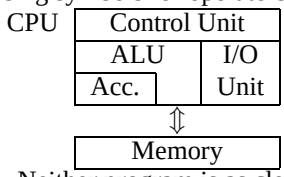
Data:	numbers only
Arith Op:	`+ - * /`, etc.
Assign.:	`A` ↔ `M[addr]`
Control:	`goto addr`
	`goto M[addr]`
	`if A = 0`
	`  goto Addr`

A program might look like



op	**address**
0 0 0 1	0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 1	0 0 0 0 1 0 0 0 0 0 0 0
0 0 1 0	0 0 0 0 0 0 0 0 1 0 0 0

Assembler language is a 1-to-1 encoding of machine instructions using symbols for operators and addresses.




```
LDA X
ADD Y
STA Z
```

Neither program is as clear as `Z := X + Y`

Assembler was a breakthrough because it used the computer to help develop the program.

- Machine code and assembler are termed **lower-level languages** (LLL's) because their operations are defined by the hardware.
- **Higher-level languages** (HLL's) would choose their operations to suit the application program.

1.2 Higher Level Languages

Early proposals for HLL's were controversial.

- Critics said they would yield unacceptably slow code.
- Proponents argued that a significant reduction in coding and debugging time would result.

1.2.1 FORTRAN

- *FOR*mula *TRAN*slator, 1955-57, John Backus
- Dominated by concern for run-time speed
- Intended for use in numerical applications.

FORTRAN did indeed produce slower code than an expert Assembler programmer, but not as bad as feared.

Major payoffs in

readability: algorithms could be communicated

development time: compile times offset by reduced programming time

education: easier to train new programmers

portability: programs could be run (with minor mods.) on many machines

---

1.2.2 ALGOL

"universal" *ALGO*rithmic *L*anguage, 1958. Goals:

- close to standard mathematics notation
- useful for the description of algorithms
- compilable into machine code
- portable across computer architectures

Algol had limited success in Europe, little in U.S., but was very influential.

- spawned class of "ALGOL-like" languages.
- Backus introduced use of formal language theory (Backus-Naur form) to describe ALGOL syntax.
- Many developments in compiler theory (e.g., stack-oriented evaluation)

1.2.3 LISP

John McCarthy et al. developed the *LIS*t *P*rocessor language (1959) was intended symbolic processing.

- inspired by FLPL, Fortran List Processing Language
- features recursion, functional programming
- common data structure for data and programs
- dominated the field of Artificial Intelligence

1.2.4 COBOL

Grace Hopper led group at Univac, under DOD support, to develop *CO*mmon *B*usiness *O*riented *L*anguage, 1960.

Goal:

- program in "natural" English,

Example,

```
ADD X TO Y GIVING Z.
```

- managers could read programs
- though programmers would still need specialized training to write in COBOL

1.3 The Explosion

By the early 1970s, the U.S. Dept. of Defense estimated that over 450 different programming languages were in use on defense projects.

Why do so many languages even exist?

- specialization
- new concepts (e.g., user defined data types, ADTs)
- advances in hardware (e.g., time-sharing, parallel machines)
- advances in compiler techniques
- inertia — old languages never die

Changing Economic Factors

- Hardware got cheaper, programmers got more expensive
  - run-time efficiency becomes less important
  - error-checking and prevention become more important
  - code re-use is a major payoff

Changing Economic Factors

- Hardware got cheaper, programmers got more expensive
- Software lifetimes got longer
  - "Maintenence" of software became a major factor.
  - Readability becomes more important.

Changing Economic Factors

- Hardware got cheaper, programmers got more expensive
- Software lifetimes got longer
- Software Projects got *(orders of magnitude)* larger
  - Old tools and techniques did not always scale up well.

---

2 Classification

1. Paradigms
2. "Generations"

2.1 Programming Paradigms

A **paradigm** is a basic model of how something gets done.

Programming language paradigms represent fundamendally different views of how to program:

1. Computation Model
2. Organization
3. Parallelism

2.1.1 Computation Models

How are basic sequences of computations expressed?

The **computation model** describes how a single algorithm or function is constructed.

- Imperative
- Functional
- Logic

Imperative Programming

Imperative programming is characterized by the concepts

- assignment
- control flow

---

assignment: The value of a variable is temporary and can be changed by the programmer at at any time.

```
x := x + 1;
```

---

control flow: Statements are executed in a prescribed order.

At any moment in time during a program execution, we can point to a statement (or part of a statement) that is the current "point of execution".

```
void normalize (double x[], int N)
{
  sum = 0;
  for (i = 0; i < N; ++i)
    sum += x[i];
  avg = sum / N;
  for (i = 0; i < N; ++i)
    x[i] = x[i] / avg;
}
```

Functional Programming

In **functional programming**, entire program is viewed as a single function.

- Instead of control flow, we have a weaker notion of functional dependence.
- Variables are "shorthand" for constants — once established, values are never changed.
- Functions often operate on other functions.

```
fun normalize x N =
  let fun avg (x N) =
        (reduce + x) / N
  and fun dividebyAvg (z) =
        z / avg(x N)
  in
    map dividebyAvg x;
```

Logic Programming

Expresses programs as a series of *facts* and *inference rules* for deriving new facts from old ones.

- No sense of control flow
- variables are immutable, as in functional programming

An array Y is a normalized form of an array X if all of its elements are formed by dividing the corresponding element of X by the average of X.

"if all of its elements are formed by dividing the corresponding element of X by the average of X"

This property holds if

- both arrays are empty, *or*
- the first item in Y equals the first item in X divided by the average, *and*
  - the property holds for the remaining portions of Y and X.

```
normalized(Y, X) :-
      isAverageOf(Z, X),
      dividedBy(Y, X, Z).

dividedBy([], [], Z).
dividedBy([Y1|Y], [X1|X], Z) :-
      Y1 is X1 / Z,
      dividedBy(Y, X, Z).
```

```
isAverageOf(Z, X) :-
      isSumOf(S, X),
      isLengthOf(N, X),
      Z is S / N.

isSumOf(0, []).
isSumOf(S, [X1|X]) :-
      isSumOf(Z, X),
      S is Z + X1.
```

```
isLengthOf(0, []).
isLengthOf(S, [X1|X]) :-
      isLengthOf(Z, X),
      S is Z + 1.
```

Recall our classification by paradigm:

1. Computation Model
   (a) Imperative
   (b) Functional
   (c) Logic

2. Organization
3. Parallelism

2.1.2 Organization

How are large programs organized from individual language constructs?

Subroutines

Modules

Object-Based

Subroutines Expressions and statements can be grouped together in functions and/or procedures.

Available in all HLL's, beginning with FORTRAN.

Modules

Object-Based

Subroutines

Modules Related functions and procedures can be grouped into modules.

Modula (197?), Modula 2 (1980), Ada (1983)

Object-Based

Subroutines

Modules

Object-Based Functions and procedures are grouped together with the data they share in common.

Simula (1969), Smalltalk (1980), C++ (1983), CLOS (1988), Modula 3 (1989), Java (1992)

2.1.3 Parallelism

As individual processors get cheaper, one way to increase speed is to devote more processors to the same problem.

- Sequential
- Concurrent

Sequential

Most languages offer no support for parallelism.

- Parallel processing can only be achieved via machine-specific function libraries.
- The resulting code is seldom portable.
- May be problems with optimization, other libraries, etc.

Concurrent

A few languages have constructs that support parallelism directly:

- Concurrent Pascal, Ada, Modula 3, Java

2.2 Generations

An older classification scheme,

- 1st Generation
- 2nd Generation
- 3rd Generation
- 4th Generation

2.2.1 1st Generation 1954-58

- Fortran I, ALGOL 58
  - mathematical expressions

2.2.2 2nd Generation 1959-61

- Fortran II
  - subroutines, separate compilation
- ALGOL 60
  - block structure, data types
- COBOL
  - data description, file handling
- LISP
  - pointers, lists, interpretation

2.2.3 3rd Generation

1962–1970 Early 3rd generation

- Simula 67
  - modules and classes
- ALGOL 68, Pascal (1971)
  - user defined data types, nesting, while/repeat loops, strong typing
- C (1972)
  - systems programming, weak typing, efficiency

3rd Generation (cont.)

1970–1990 Late 3rd generation

- Modula 2
  - Pascal plus modules
  - strong typing
- Ada
  - modules, generics (templates), concurrency
  - strong typing

2.2.4 4th Generation

"Software Engineering" languages tailored to

- particular application domains
- particular CASE environments

By their nature, none of these is particularly wide-spread.

The classification by generations has fallen out of use because

- There is little agreement on what, if anything, constitutes the 4th generation.
- The term "4th generation" has been largely co-opted by advertisers.
- The generation scheme does not take into account object-oriented langauges, and never reflected developments in functional and logic programming.