CPS323 Lecture: 50 Years of Programming Language History; Language Paradigms

Last revised 1/5/09

Objectives:

1. To introduce the early history of higher-level programming languages
2. To introduce the major programming language paradigms

Materials:

1. Projectable of Programming Language History Timeline (Zoom in to show)
2. Projectable of "publication ALGOL"

I. Introduction

A. One cannot understand today's programming language spectrum without a good sense of the history of their evolution.

1. After an initial explosion of new languages in the late 1950's and early 1960's, it has become the case that most new languages exhibit significant dependence on one or more predecessors - leading to a family tree phenomenon not unlike that of human languages. Truly novel languages are relatively rare. (One example that comes to mind is PROLOG.)

   PROJECT: Timeline - show roots at left

2. As we intimated in our discussion of the "Tower of Babel", things might be very different today if it were possible to scrap every existing piece of software and start over again - but that clearly isn't an option.

B. Higher-level programming languages emerged in the 1950's.

1. The dominant reality of the time was that computer hardware was VERY expensive, while programmers were (relatively) cheap. (High hardware costs --> few computers)

   a. Thus, EFFICIENCY of programs was a major consideration - both time and space (memories were small.)

   b. All programming was done in machine language in the early 1950's, using all sorts of "tricks" to keep size and time down. Machine language was

      i. Lengthy to write.

      ii. Prone to errors.

      iii. Hard to debug.

      iv. Hard to modify.

      v. Not portable from one machine to another.

      But it was all that was available!

2. In the mid 1950's, assembly languages began to emerge. These reduced most of the problems of machine language coding, but did not eliminate them. (Of course, portability wasn't helped at all.)

3. In 1957, FORTRAN was developed as the first true High Level Language.

   a. For its time, this was a remarkable achievement. No one had ever written an automatic translator like this before, so there was no base of theory to draw on. Moreover, FORTRAN had some features that even today would be hard to handle for a compiler.

   b. A key feature was the ability to use algebraic notation (e.g. X = A + B + C). Indeed, FORTRAN is an acronym for FORmula TRANslator.

   c. FORTRAN encountered a great deal of resistance at first.

      i. It was argued that the code generated by the FORTRAN compiler could not possible be as efficient as that produced by hand by a good assembly-language programmer.

      ii. The developers of FORTRAN were sensitive to this issue, and devoted a great deal of effort to producing a compiler that generated efficient code. In fact, the first FORTRAN compiler (for the IBM 704) was unrivaled by another other for 20 years!

      iii. A major factor in the acceptance of FORTRAN was IBM's sponsorship. They promoted it heavily, and gave FORTRAN compilers away with their machines.

   d. As we saw in the timeline, FORTRAN is a (often distant) ancestor of many of today's programming languages; and the idea of a higher-level language arguably gave rise to some of the other branches of the tree.

4. Two other early languages are worthy of a brief note:

   a. COBOL was developed for the business community, which tends to find algebraic notation unpalatable. (COBOL stands for COmmon Business Oriented Language).

      i. For example, the above algebraic expression might be written in COBOL as ADD A, B, C GIVING X.

      ii. On the other hand, COBOL had (and arguable still has) the most flexible facilities for formatting output, something very important in the business world.

      iii. Though it is one of the oldest languages in use today, and there have been many predictions of its demise, it seems to have a continuing base of use.

         - For example, a 2006 survey of IT managers at 352 companies by Computerworld noted this: "62% of the respondents reported that they actively use Cobol. Of those, three quarters said they use it “a lot” and 58% said they’re using it to develop new applications. Nevertheless, with a few exceptions, companies aren’t enthusiastically expanding their use of Cobol. In the survey, of those who use Cobol, 36% said they are “gradually migrating away” from it, 16% said they will replace it “every chance we get,” and 25% said they’d like to replace Cobol with something else but have found that too difficult or too expensive." [article id 266228].

- A significant concern in some segments of the industry is the need to replace a large number of baby-boomers who use COBOL and who will be retiring soon. IBM, in particular, has taken some proactive steps to encourage education in COBOL.

b. ALGOL - ALGOrithmic Language (1960)

i. Developed by an international committee chaired by Peter Naur (whose name we will see again). Their work was based in part on an older version, the International Algorithmic Language (IAL) also known as ALGOL 58.

ii. Elegance of expression was a major design goal.

- As a result, ALGOL programs are much easier for humans to read than programs written in FORTRAN.

- For many years, to say that a language is "Algol-like" was was tantamount to saying it is beautiful.

- However, to achieve this elegance, symbols were used that were not available on the equipment of the day, though substitutes were provided to accomodate existing equipment.

- For many years, ALGOL was THE language for publication of algorithms n Computer Science journals. A special publication dialect was developed that used bold-face type for reserved words, etc.

  PROJECT: Sample ALGOL published algorithm

iii. ALGOL incorporated three significant innovations, which have influenced many languages since.

- Block structure - giving rise to the notion of local variables

- The use of begin .. end ({ .. } in Java) to group a series of statements so that they can be treated as a unit.

- Recursion (explicitly forbidden in both original FORTRAN and COBOL, because it gives rise to some issues that affect the overall structure of an implentation.

iv. ALGOL is the ancestor of many languages in wide use today, in two major families

- The Pascal family, which includes Pascal and Ada. (For over a decade, Pascal was THE language of CS Education. You will note that your textbook's author frequently refers to it - no doubt reflecting the fact that he has been teaching CS for a while!)

- The C family, which includes C++, Java, C#. This family (through C) makes some significant stylistic changes compared to ALGOL itself and the Pascal family: The use of { } instead of begin .. end, and the use of = for assignment and == for comparison, instead of := and =.

5. With the emergence of FORTRAN and other higher-level languages, there were both gains and losses.

   a. Gains:

      i. Ease of programming: programming is no longer a "magical art" for the few, and overall time to produce a given program is greatly reduced.

      ii. Programs are less prone to error.

      iii. Ease of debugging.

      iv. Ease of modifying.

      v. A major gain is portability. At least in theory, a higher-level language program developed on one machine could be moved to another with few or no changes.

   b. Losses:

      i. Some things that the hardware can do now become more difficult to do - e.g. accessing hardware IO devices directly. This is fine for application programs, but poses a problem when writing system software - hence some languages (e.g. C) incorporate features to facilitate this.

      ii. In any given language, some programming techniques are easier to use than others. Thus, the language tends to shape the programmer's style.

      Example: COBOL makes it very easy to produce nicely formatted output (e.g. printing currency as $1,323,074.99), but forbids recursion. Java is the exact opposite. This can affect what programmers using those languages try and don't try to do.

C. Interestingly, while programming languages have evolved a great deal over the last 50 years, and hardware has become many orders of magnitude faster than the machines of the 1950's, the basic model of computation of digital computers is essentially the same today as it was then.

1. As your recall - or will learn - in CS311, a computer system consists of three major component parts:

   a. A memory system - historically, using just one technology, but today a hierarchy of technologies of varying speed and capacity.

      The memory system stores two basic kinds of information: data and executable code. Data is ultimately represented by numbers. Code is in the form of basic machine instructions, each on the order of 16-64 bits or so long.

   b. A CPU - historically, just one, but often today multiple processor cores or full processors.

      One key component of the CPU is a program counter, which holds the location in memory of the next machine instruction to be executed. The CPU carries out a basic cycle called the fetch-execute cycle:

- Fetch an instruction (sequence of bits) from the location in memory pointed to by the program counter.
- Update the program counter to refer to the instruction just after the one fetched.
- Execute the instruction just fetched by performing a single, primitive operation.

c. An input-output system.

2. The primitive operations performed by the hardware are relatively few in number:

a. Copy individual pieces of data (e.g. a single number) within a subsystem, or between subsystems. (Copying a piece of data to or from the input-output subsystem results in bringing a value (e.g. a single character) into the system from an input device or outputting a value (e.g. a single character) on an output device.)

b. Perform simple arithmetic and comparison operations on one or two individual pieces of data (e.g. add, subtract, multiply, divide, or compare two numbers).

c. Alter the value in the program counter, which changes the order of execution.

3. Early higher-level languages (like FORTRAN) tended to closely mirror the structure of the underlying hardware. Newer languages have constructs which are easier for humans to use; but these still need to be translated into primitive operations of the sort the hardware can handle. Increasingly, the focus has been on ease of use by humans, rather than on mirroring the basic structure of the hardware.

II. Language Paradigms

A. In the history of programming languages, several major programming language paradigms have emerged.

1. Peter Wegner has defined a paradigm as a "pattern of thought for problem solving".

2. The paradigm of the programming language(s) one uses tends to shape the way a person actually _thinks_ about solving a problem. That is, paradigm differences are not simply differences of features and style - they are differences of thought-process. One of your challenges in this course will be to not only learn about language features, but also to learn to think about problems in different ways.

3. One can always use a programming language in a way that differs from its paradigm - but this is often a misuse. (One can use a BMW to haul garbage - but ...)

B. Most writers agree on four major paradigms, though many would add one or more others.

1. The imperative or procedural paradigm - of which FORTRAN and COBOL are early representatives. (More modern representatives include C and BASIC.)

a. The imperative paradigm is closest to the actual structure of a computer. Indeed, the earliest imperative languages made heavy use of constructs like GOTO which directly mirror primitive hardware operations.

b. Much of the theoretical material in the course will be covered in the context of the imperative paradigm.

i. Historically, this is where many concepts were developed.

ii. Other paradigms typically implement these notions in similar ways.

2. The object-oriented paradigm - of which Smalltalk is the early representative. (More modern representatives include Java, C#, and many others.)

a. The object-oriented paradigm has many similarities to the imperative paradigm, and, indeed, grew out of developments in the imperative world such as data abstraction.

b. Programmers who have initially learned an imperative language and then learn OO often struggle with a mental process known as "the OO paradigm shift". Indeed, it is not uncommon for students who have learned an imperative language in high school to find courses like CPS112 and CPS211 very difficult, and to "speak Java with a thick Basic accent" :-).

3. The functional paradigm - of which LISP is an early representative. (More modern representatives include Haskell and ML).

The functional paradigm is _very_ different from the two paradigms we have just discussed - but since the best way to learn it is in the context of learning an actual functional language, we will defer discussion of it until we get to LISP.

4. The logic or declarative paradigm - of which PROLOG is the primary example. (PROLOG is an acronym for PROgramming in LOGic).

Again, the logic paradigm is _very_ different from the other paradigms, and we will defer discussion of it until we get to PROLOG.

5. Some programming languages are multi-paradigm hybrids - i.e. they can be used in the manner of more than one paradigm. For example, C++ and Ada are both imperative and object-oriented. Ruby supports imperative, object-oriented, and functional style programming.

A danger with these languages is that, if one knows a particular paradigm, one will tend to use the hybrid language in the way one is familiar with, neglecting the features that belong to other paradigms.

C. Most of your experience has been with the object-oriented paradigm. For this reason, we will say relatively little about OO in this course. Hopefully, by the end of the course, you will have become familiar and comfortable with the other paradigms as well.