CPS323 Lecture: Course Introduction Revised 1/5/09

Materials: Projectable of "Tower of Babel"

Handouts: Syllabus

I. Roll

II. A First Look at the Nature of this Course

A. This course is concerned with higher-level languages

1. Their design

a. Goals that have motivated language designers.

b. Criteria for evaluation a language design

2. Their description

a. Syntax - rules for forming a valid program in the language

b. Semantics - what does a particular program mean?

3. Their implementation

a. How higher-level constructs are represented and carried out at run-time.

b. A VERY quick introduction to the translation process used by compilers.

4. Current issues in language design - where are programming languages heading?

B. The present reality is that the programming language landscape resembles the Tower of Babel

TRANSPARENCY

1. This is from a book on the history of programming languages published in 1969 (hence the absence of many familiar names.) But the same situation prevails today.

2. The diversity of languages is likely to continue to increase.

3. Why are there so many programming languages? ASK

a. One reason is the DIVERSITY OF APPLICATIONS for which software must be developed. We need here to consider two terms: "general" and "universal"

i. A language is GENERAL if it can be used to express ANY algorithm that can possibly be formulated.

- An important "article of faith" for computer scientists is Church's Thesis: any algorithm that can be formulated can be expressed as a program for an abstract model of computation known as a Turing machine. (This is very plausible, though it cannot be proved.)

- It can be shown that - subject to limits of finite memory - any conventional digital computer be programmed to realize any algorithm that can be realized by a Turing machine - hence any algorithm.

- For most programming languages, it can also be shown (subject again to finite memory limits) that any algorithm that can be realized by a Turing machine can be expressed as a program in that language.

- Thus, if Church's thesis is correct, ANY algorithm can be realized by any modern digital computer, using almost any programming language - memory limits permitting.

(That is - almost any programming language is GENERAL. The exception would be some specialized languages created to formulate problems in highly specialized application domains.)

ii. However, a given language typically can be used to express some kinds of algorithms more easily than others. The needs of different problem domains have called forth much of the diversity of languages. One writer identified five major application domains, each of which makes different demands on a language:

- Scientific applications

- Business data processing applications

- Text processing applications

- Artificial intelligence applications

- Systems programming applications (writing of operating systems, compilers etc; development of embedded systems)

To this, we could add a sixth domain today: the whole world of web-based systems.

Several well known languages are clearly identified with a particular domain. ASK CLASS FOR AN EXAMPLE OF A LANGUAGE CLEARLY IDENTIFIED WITH EACH OF THE ABOVE, INCLUDING WEB.

iii. There have been several attempts to formulate a UNIVERSAL programming language, which would be equally useful for all major domains. Such attempts have, thus far, largely failed.

- ASK for an example of a language which had this as its goal.

- "Universal" languages fail for two reasons: They tend to be large and complex because of all their features - hence hard to master; and they still fail to be truly universal.

- We might compare this phenomenon to the failure of attempts to formulate a universal HUMAN language - e.g. Esperanto

iv. Since it is relatively easy to develop a general language but very hard to develop a universal one, we tend to have a plethora of languages that are better for some tasks than others. (Hence the need for a course like this, to become familiar with a spectrum of languages.)

v. In addition, there will always be specialized problem domains for which the best solution is to develop a specialized language for use in that domain. For example, a language known as APT was developed in the 1960's - and came into wide use - specifically for writing programs to control automatic machine tools.

b. Another reason for the diversity of languages is related to SOFTWARE ENGINEERING.

i. Much of the history of computing has been characterized by hardware growing in power while dropping in cost at a dramatic rate. As a result, larger and larger problems are being computerized. These call for larger and larger programs.

ii. However, it is well known that doubling the size of a program much more than doubles its complexity. The whole sub-discipline of software engineering has emerged as an attempt to find ways to solve large problems correctly, on time, and on budget.

iii. Many newer languages incorporate features that facilitate good software engineering - particularly features that support cleanly breaking large programs into well-defined, independent modules. Progress in this area continues to call for better language tools. (Consider how difficult it would be to write a 100,000 line program in a language like minimal standard BASIC!)

c. A third reason for the diversity of languages is INERTIA.

i. There are many factors which operate to prevent older languages from being replaced by newer ones, even when the newer ones are clearly superior.

- If a language has already been used to develop a large base of software, then it is usually not feasible to translate this software into the newer language. Thus, maintenance and enhancement of the existing software base must still be done in the older language.

- Often times, organizations make a large investment to support programming in their chosen language: manuals, debugging aids, software that helps in the writing of software, libraries of subroutines etc. A major language change can result in that investment being lost.

- Once an individual becomes comfortable with a particular language, it is natural for that individual to be reticent to change to a new one.

ii. Thus, to gain a wide usage a new language either needs to have something about it that attracts a following, or it needs a powerful "sponsor".

- e.g. a major reason for the early success of FORTRAN and COBOL was sponsorship by IBM - at that time the biggest player in the computer industry.

- e.g. C succeeded in large measure because it facilitated writing software that is PORTABLE across different computer systems.

- e.g. Ada succeeded because it was sponsored by the world's largest software purchaser - the US department of defense.

- e.g. arguably a key factor in the current success of C# is its sponsorship by Microsoft.

But, even in these cases, while newer language may supplant older ones for new software development, the older ones continue in wide use.

d. A first look at languages we will use.

i. This course presumes - and therefore will not explicitly cover - familiarity with Java and C++. We will, however, frequently use these languages to illustrate various ideas.

ii. Among languages we will explicitly study, several go back to the very early years of higher-level languages, some are "middle-aged", and others are very new:

- Older languages: FORTRAN, LISP - representative of two very different approaches to expressing a computation.

  As we will see in our look at programming language history next class, each of these has been the ancestor of a whole family of programming languages. (There are several other languages like this we won't look at due to time, including COBOL and Smalltalk - plus one in the next group)

- Middle-aged languages:

  - Ada - still widely used both by the DOD and in life-critical software engineering applications such as aircraft

  - PROLOG - representative of yet another very different approach to expressing a computation

- Newer languages: Python, Ruby

III. Syllabus