CS 131 Programming Languages

December 3, 2009

Languages Timeline

Fortran became very popular very fast because it generated very good assembly code, surprisingly better than any other language of the time. Because it was an IBM and commercial product, Fortran has encountered competition: competitors started to write their own compilers. How does GE know what to implement? They had to read IBM's documentation for the Fortran compiler (which wasn't very good), and then reverse-engineer the compiler. When in doubt, GE folks ran the IBM compiler and looked at what happened. Therefore the downside of Fortran was its poor definition, and the fact that IBM held a monopoly on it.

Europeans didn't like IBM's monopoly a bit, and they created *Algol* in 1960, coming up with a formal BNF syntax for the language, something Fortran didn't have. Algol was essentially Fortran with a few extra goodies (call by reference, recursion). There's one other language of the 1950's that's still in use – *LISP*, which came out in 1959 and was initially intended for AI research at MIT. LISP was the first language to implement *recursion* and *S-expressions*: LISP program can easily be represented as LISP data. In modern days we call it *reflection* – the ability of software system to look at its own data structures.

While research and education fields had their Fortran, Algol, and LISP, business field was left out until COBOL was created in 1960. The idea was to create a syntax that is readable not just by programmers but by their managers as well. That idea failed – the grammar didn't end up readable enough to be useful for non-programmers. However, COBOL did succeed in that it introduced *records* and *structures* as was generally a success in business field. By the end of 1960's and 1970's, COBOL and Fortran were dominant.

IBM decided to design a language that had features of all the successful languages at the time, and named it PL/I (programming language one), coming out with initial release in 1964 and with a good release in 1968. It is known to be a first example of a *kitchen sink language*: it had everything. As a result of that, it took quite some time to develop a compiler for PL/I. Another result was some unanticipated problems when features collided. For example, take PL/I expression `10 + 1/2`. `1/2` is represented as `0.50000` and adding `10` to it overflows because of a 6-decimal-point limit. At some point, PL/I nearly eclipsed Fortran and COBOL.

Two more efforts of the 1960's should be mentioned. One is *BASIC* programming language, introduced in 1964. BASIC introduced the KISS principle (*keep it simple, stupid*). Also, *APL* (a programming language) came out in 1961, introducing a lot of fancy built-in operators. APL is the language that had to be used with its own keyboard because of the multitude of operators. Yet another language introduced in 1964 was *SNOBOL*, which introduced regular expressions, patterns, and iterators.

European committee saw Algol lose to the other languages and split in their decision on how to implement its next version. The "winners" got to design *Algol68* while the "losers" came up with *Pascal*. While Algol68 introduced orthogonality, Pascal won its place in history by focusing on structured programming, a revolution of programming style at the time.

In 1968, BCPL was introduced as a practical replacement to an assembler while still being portable enough to run on a variety of architectures. It had only one data type, the integer, which could be used in a variety of ways. In 1972, C came out, which was essentially a combination of ideas from BCPL, PL/I, and Algol68 and included types as part of its grammar. The major reason C succeeded is because of the Unix operating system, which was written in it. Unix was a first example of a portable operating system written in a high-level programming language.

A language called *Simula67* was designed to be a simulation language, by that introducing objects, methods, and classes. A function had to be called, whose frame became an object. When a function returned, it returned in a special way that kept the frame around, which could then be reused.

C++ first came out in 1980 and could be essentially thought of as C with classes.

Objective C came out some time before C++ and tries to be C with a relatively small number of changes. It borrows some ideas from *Smalltalk*, which first came out in 1976 and is more known for its 1980 version. The idea of Smalltalk is to have a big, integrated development environment, a contradiction to the Unix philosophy where the goal is to break problem down in small pieces.

Prolog and *ML* were produced in roughly 1977 and were somewhat a replacement for LISP. ML can be thought of as LISP with types (introducing type inference) while Prolog started a new family of languages. *Haskell* can be thought of a variant of ML that introduced *lazy evaluation*.

Perl came from the C side and Unix side. Associated with Unix were two popular language types: the shell (*sh*) programming language and *Awk*, a string processing language. Guy who designed Perl hated the fact that programmers had to learn multiple languages in order to achieve a task. The goal was to supplement functionality provided by C, shell, and Awk, and stands for *practical extraction and report language*. *Python* was inspired by Perl by a negative example. Python is the only language that doesn't use BNF and instead relies on indentation.

Algol spun off many other languages which were not mentioned. One of them was *JOVIAL*, which somehow became the primary programming language of the USAF. In 1983 DoD finally came up with a replacement for it with a language designed in France called *Ada*, which borrowed somewhat from Pascal and which was designed by a committee supervised by DoD.

Java came out under influence from C++ and Smalltalk in 1995 and C# in 2002 introduced *kitchen sink runtime*. *Javascript* came out at about the same time as Java; its strongest ancestor is probably Awk.

Ruby came out about 10 years ago, and is a first major language programming language not designed in Europe or the US (it was designed in Japan). Ruby borrows ideas from Smalltalk while introducing scripting, and had a negative influence from Perl and Python. *Ruby-on-Rails* uses the fact that Ruby is a nice meta-language, a flexible way to create other languages.

Features of Programming Languages

Error handling

- Throwing exceptions (Python, Java).
- Returning special values (IEEE-754 floating point).
- Compile-time checking – find a class of errors that is easy to detect at compile time and insist on fixing them.
- Preconditions – write down caller's responsibility to ensure for every method. In Scheme, calling `(car x)` implies a caller's responsibility that `x` is a pair.
- Undefined behavior – core dump, crash, etc.

Semantics of Programming Languages

What does a program mean?

- *Syntax*: BNF
- *Static semantics*: What can you deduce about what the program means without running it? Type checking, keeping track of identifiers that have been declared vs. used.
    - *Attribute grammars* act as BNF except that associated with every terminal and non-terminal is a bunch of attributes that place restrictions on its members.
- *Dynamic semantics*: We would like a formal system to specify dynamic semantics of a programming language.
    - Method A – *Operational Semantics*: Let's write an interpreter for the language N, using language M, then run the program to it and see what it does. The first formal definition for LISP was defined using LISP itself, by original LISP author (e.g., N = M).
    - Method B – *Axiomatic Semantics*: Let's write a series of axioms and inference rules for the language. If you want to prove something about a program (a certain property), you reason using axioms and inference rule and prove it that way.
        - From Webber 23.4, here's some sample axioms that are used:
            - $\langle k, C\rangle \rightarrow k$, k is a constant.
            - $\langle v, C\rangle \rightarrow \mathrm{lookup}(v, C)$, v is a variable
            - $\langle E1 + E2, C\rangle \rightarrow \mathrm{if}\ (\langle E_1, C\rangle \rightarrow v_1, \langle E_2, C\rangle \rightarrow v_2) \rightarrow v_1 + v_2$.
            - $\langle \mathrm{let}\ x = E_1\ \mathrm{in}\ E_2, C\rangle \rightarrow \mathrm{if}\ (\langle E_1, C\rangle \rightarrow v_1, \langle E_2, (x, v_1)::C\rangle \rightarrow v_2) \rightarrow v_2$
            - $\langle \mathrm{fun}\ x \rightarrow E, C\rangle \rightarrow \lambda(x, E, C)$
                - $\langle E_1\ E_2, C\rangle \rightarrow \mathrm{if}\ (\langle E_1, C\rangle \rightarrow \lambda(X, E', C'), \langle E_2, C\rangle \rightarrow v_2, \langle E', (x, v_2)::C'\rangle \rightarrow v_3) \rightarrow v_3$
    - Method C - *Denotational Semantics*: describes semantics of a program as a function.