CSE305: Programming Languages --- Week 1 Lectures

Important to say right away: **OMWV** --- Our Mileage Will Vary. Here is the UB CSE description:

> CSE305: Examines concepts and constructs found in programming languages. Topics include binding time, strong typing, control and data abstraction, higher-order functions, and polymorphism. The major programming paradigms (procedural, object-oriented, functional, and logic) will be studied and compared. The course will also provide an introduction to syntax and semantics, compilation vs. interpretation, and storage management.

It is possible to "touch" every part of this in 8 weeks, IM(P)HO. The question is which parts to emphasize in depth. Each of the following two poles has many adopters---counting what I know from other universities too:

1. "Functional First and Foremost"---Challenge with new paradigms.
2. "Classic"---really really learn what goes on under-the-hood, first in the paradigms you already know and then crossing over.

I intend to teach "1.5": we will start the functional language (the **OCaml** flavor of ML) early, but it will share mindspace with the procedural-and-OO languages---until the logic language Prolog gets airtime in the last 2-or-so weeks. Currents working against this being a "happy medium" are:

- The perennial textbook by Robert Sebesta, now in its 12th edition, is completely "Classic".
- Moreover, the language Scala---one of whose several motivations was to give a taste of "functional" in years when CSE305 was not required---is still entirely absent from the book.
- CSE305 is being re-required starting next Fall with a blueprint closer to 1.
- Scala is being re-evaluated. It has issues I will show in this course, maybe not enough gain, and IMPHO its gargantuan new version 3.0 is repeating the mistake of Ada95.
- The course was taught in style 1 a year ago, with Sebesta optional.

My intent is to leverage the following to put across the functional ideas:

- Your experience with Python, especially code elements often called "Pythonic".
- Your experience with Scala, at least in CSE250 if not before (transfer students are invited for office-hours flyover of this).
- Affinity with inductive grammars in Sebesta chapter 3. This is the first of several reasons I have made the text **required** again. (Any edition 9 or higher is OK; the 12th edition has one important new section on "Option Types" but I will invoke "fair-use" to make it common. At $50-ish or under, this text is worth owning.)

There will be some bumpiness in the first weeks as I judge how to balance all this. Teaching **OCaml** "all along" may entail *not* introducing a new O-O language such as **C#** but rather talking about deeper features of O-O languages you already---variously---know.

Logistics

A little of everything:

- ***Course webpage*** used for notes, resources, assignments, and "permanent" information.
- ***Piazza*** used for Q&A, "transitory" information (such as if an assignment gets extended or an issue develops) and *some* resources.
- ***TopHat*** used for homework questions, lab questions, and attendance taking both in lectures and labs.
- ***CSE Autograder*** used for submissions of both code assignments (with autograding *hidden* test suites) and pencil-and-paper assignments **in PDF form** (not MS Word or etc.).
- *UBLearns* used only to post grades---usually with lag after allowing time for possible corrections on Autograder.

Relative to last year: (+) more weekly assignments, maybe some bi-weekly; (-) multi-week final project rather than multi-month. All assignments individual except the final project *may be* teams-of-2. (+) **Two** *prelim* exams rather than one midterm; (-) fewer quizzes. No "drop one" policy but a feature that works similarly: at my discretion, and only to a student's advantage without affecting others' grades, I can shift course weights by ± 5%. Most typically, if you do poorly on a prelim but well on the final, that prelim changes from 15% to 10% and *your* final from 35% to 40%.

Weekly recitation/lab attendance is essential. In both lectures and labs we will illustrate features of languages with byte-size examples. The difference is that in labs you will have convenient access to a terminal or your laptop and the focus will be there---with more freedom to phapho---rather than the focus being on my screen. You may need to gain some basic familiarity with Linux/UNIX terminal commands.

Differences **between** languages will be a major topic of this course. Differences between implementations of the **same** language alas occur---even when the implementations have the same version number. This happened with Scala in my CSE250 last year, though mainly (only?) on code that was innately erroneous. The implementations on CSE Autograder are the official ones. Some non-hidden testing will be enabled for checking code.

Recitations do not start until Week 2.

Homework over first weekend (Feb. 4-5): get Sebesta textbook (edition 12 preferred, 9--11 OK) and read chapters 1--2 *as background*. Chapter 2 has lots of chitchat---you won't be tested on it specifically, but the historical development of ideas is good to know. Then start on Chapter 3, which has the course's first technical material.

Academic Integrity

All submissions to Autograder require you to affirm that this is your own work in accordance with University regulations. That had better be true. In the past I've caught and given automatic Fs for cheating on the final project in this course. Departmental statement here. The University recently produced a revised big page that is very well written.

There are some legitimate gray areas. The nature of using multiple languages, not all covered in the textbook, means that online resources come into play. I will collect and "bless" some of them, but that will not exclude others. What most people forget to do is **cite** an outside source.

I will say more in detail upon giving the first written assignment. Your first lab work will go over certain points of academic integrity while giving some first experience with **OCaml**.

Flyover of Some Course Topics

The progression, reflected in the text's structure, will be from "Low Level" to "High Level":

I. Lexical. This means having to do with symbols, or **tokens** made from symbols, but not even necessarily fully into statement-level syntax yet. Some more-or-less familiar examples:

- End statements with semocolons ; or use end-of-line (EOL)? (The latter convention often needs an "anti-EOL" character such as '\' to continue a statement across lines.)
- Is indentation more than cosmetic?
- Are capital letters different from lowercase---in ways that are more than cosmetic? (**OCaml** differs from **Standard ML** on this point.)
- Which of =, ==, `:=` is used for assignment and which for equality testing?
- Are enclosing parens `( ... )` needed on arguments to functions, even when there is just one argument?
- Are parens needed around **tuples**, which are fixed-size lists allowed to mix different kinds of elements? Python and OCaml: *no*.
- The term **list** usually refers to lists that can be extended or cut down, and that are **homogeneous**, meaning elements have the same (base) type. Those usually use enclosing square brackets `[ ... ]`. But in another difference from Standard ML *that is going to drive me nuts*, **OCaml** uses semicolon `;` not comma `,` to separate list elements.

Many lexical details are "incidental", beside the point of the *operations* they build. But some govern operations. Whether *nonce* or *nub*, they are most in-your-face while programming, hence tend to get outsized emphasis.

II. Statement-Level Syntax and Semantics.

Here is one issue among a bunch we will encounter first in the course.

- (When) Is a statement a **Statement** or an **Expression**? Note:
  - a **statement** carries out an action that generally **changes the state** of execution as the program is run.
  - an expression calculates and yields a **value** (which may be a compound object) ...
  - ...but the expression may carry out actions "on the side".

Such **side-effects** are a boon and bane. They not only can cause bugs but can impede optimizing the executed code. This is a central issue between the procedural and functional paradigms.

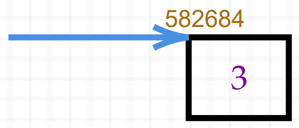
You may think the relationship is simply part-to-whole: expressions are part of statements. This certainly looks true with a simple assignment statement: `<variable> = <expression>;` Storing a (usually) new value in the variable (usually) changes the program state. But the value need not go away. In fact, in particular if you've programmed in C or C++ (and definitely if you've used Scala), the assignment "statement" is actually syntactically classed as an expression that yields the assigned value. A chain of "statements" can be treated as a sequence whose last line yierlds the whole value---this is why functions in Scala can skip the keyword `return`.

- Java and other procedural languages dial this back and have a more rigid distinction between statement and expression.
- Functional languages go the other way: they make expressions and values primary.

Statements and expressions and other "program categories" are defined first via the **syntax** of the language, much of which can expressed by a **BNF grammar**, the topic of Sebesta chapter 3.

III. Storage Objects: Values Versus References

This little kind of diagram will go a long way:



A Rough Analogy:

- Value = you.
- Reference = where you live. Or your handle, like @HamlinIsland
- Pointer = directions to where you live.

Pointer and Reference are often conflated (one can say the reference address is the value of the pointer), but the distinction between both and the *value* will be a major subject of this course. (One can also have pointers-to-pointer, `**p` in C/C++, which is like directions for looking up the directions for where you live.)

The layout of compound objects (including lists and arrays), "deep" versus "surface" items, and the degree of protection against mutation (e.g. via "`const`" or "`final`" or "`val`") will also be a major topic. This feeds into the next theme.

IV. Type Systems

The original purpose of having types was to help the compiler target machine hardware features. Then the type system helped catch more errors *at compile time* rather than have them flare up at runtime. Then---notably with COBOL **records** in 1958---the type system started becoming a vehicle for modeling compound data objects that are conceptual, not just numerical. This blossomed into ideas of *modules*, *extensible modules*, and full *object-oriented programming*.

- In OOP, the type system embodies the **Is-A** relationship via class hierarchies and compatibility. It also regulates the **Has-A** relationship.
- Functional languages provide a whole different dimension of **type constructors** which kind-of work in the opposite direction to inheritance.

The latter resemble BNF grammar rules, which is part of how and why I'm timing **OCaml** simultaneously with week 3 of the course. But right away let's consider a wider question:

- How much "semantic weight" should the type system bear? That is, how much of the behavior and meaning of the program should be expressed via the mechanism of types?

The answer from Mathematical Logic is that **all** of it can be done via types. The acronym TLC doesn't stand for "tender loving care" but rather the **Typed Lambda Calculus**. This is covered in CSE505, and I just co-wrote an article on a professor who brought TLC down to undergrad programming level.

[End of first lecture was here.]

But we will keep this more grounded. Let me illustrate how the type system can govern other aspects of programming by going into the famous **Square Is-A Rectangle?** example (in Java):

```
class Rectangle {
   double x;
   double y;
   Rectangle(double gx, double gy) { x = gx; y = gy; }
   void lengthen(double morex) { x += morex; }
   ...
}
class Square extends Rectangle {
   Square(double side) { super(side,side); }
   ...
}
```

We've made Square Is-A Rectangle by the type-compatibility rule of inheritance, but is that really legit? Consider these lines of code:

```
      Square s = new Square(3.0);
      Rectangle r = s;   //not a fresh copy of s, but just another reference to s
      r.lengthen(2.0);
```

This is all legal, but our original square has become a 5×3 rectangle. This means that elsewhere in the code we can never really bank on a `Square` staying a square. The buzzword is that this code is **type-unsafe**. Most authorities say that deriving `Square` from `Rectangle` this way is not a proper use of inheritance.

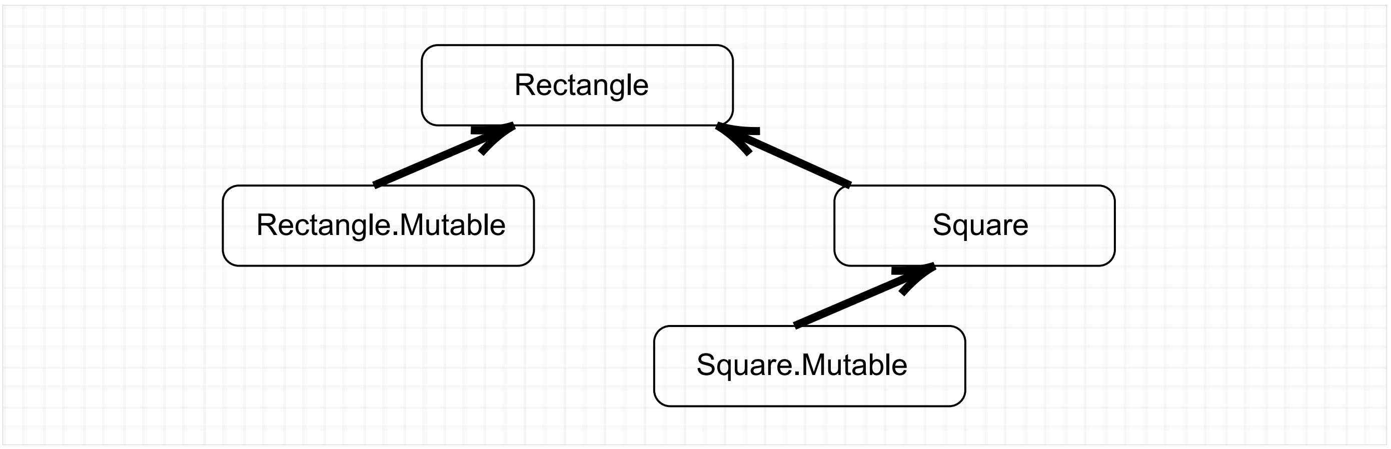
My Own Pet Idea (not in any textbook, probably known to others...)

In "Platonic" terms (actually as systematized by Aristotle), a square is-a rectangle. That's because they conceived mathematical forms as **immutable**. There is no problem if you make Rectangle a **value class** by not allowing **mutator** methods like `lengthen`. Older languages do not have support for value classes, but instead prevent changes at certain points by prefixing keywords like `const` or `final` or using pass-by-value in method calls.

My pet idea 15+ years ago was to make "value class" the default---as in functional languages and "kind-of" in Scala---and make the mutable version an **inner subclass** called **`.Mutable`**. Now using the more-efficient Scala syntax for simultaneous declaration-and-construction, we can fix up the Rectangle-Square relationship as follows:

```
class Rectangle(Double x, Double y)  {
   subclass Mutable(Double.Mutable x, Double.Mutable y) extends Rectangle(x,y) {
      void lengthen(double morex) { x += morex; }
   }
   ...
}
class Square(Double side) extends Rectangle(side, side) {
   ... //accessor-methods go here...
   ... //Hmmm....what if we try to make a Square.Mutable class too?
}
```

The Booch-type diagram for this relationship is now:



This is **type-safe** because there is no compatibility arrow from `Square` to `Rectangle.Mutable`, i.e. no way to assign a square to a Rectangle variable that is mutable. You can't even do that with a `Square.Mutable` object.

This idea co-opts the type system to make immutable objects the default but allow mutability in a derived type. Well, functional languages say we can get even closer to heaven by not having direct mutability to begin with...

This idea isn't technically perfect---indeed, there's an issue with the parameters in my `Rectangle.Mutable` constructor. You won't be tested on it and I don't even know a language that implements it directly. But it does serve to illustrate some purposes and realities of this course:

- We are still at a stage where unresolved issues are accessible to incoming undergraduates.
- There is no perfect programming language---not yet, never?

V. Preserving Data Integrity

Keywords like `const` in C/C++, `final` in Java, and `val` in Scala are supposed to protect data items against changes. But do they? We ask especially about this for fields of classes. The key distinction to bear in mind is between

- Mutable or immutable ***fields***, versus
- Mutable or immutable ***data***.

In Java, `String` and `StringBuffer` are two classes that give strings, but the former are immutable. If you do

```
String st = "Hello";
```

then you can't do `st.setCharAt(0,'J');` to make the variable `st` give "Jello". But you can do this if you started with `StringBuffer st = "Hello";` instead. Java now has a newer `StringBuilder` library option, and that is what Scala uses. Scala even allows it to use its round-parentheses notation for arrays to change characters, so the following is legal:

```
val sb = new StringBuilder("Hello");
sb(0) = 'J';
println(sb);      //prints "Jello"
```

So the use of "val" was no guarantee of data integrity. It only prevents `sb` from being **reassigned** to be a reference to a different object.

Does **OCaml** do better? It doesn't allow declaring mutable `var`iables directly at all. (You can emulate them via explicit references, and we may see how this feature of the older Standard ML was re-implemented using the OCaml object mechanism.) But it does allow mutable fields of objects. The keyword `val` by itself works like in Scala, but `val mutable` works like `var` in Scala. Again, this is only in class/object fields, not just anywhere, and to reassign a mutable field you have to use `<-` not `=` or `:=`.

I've written test code using OCaml `string` versus the `Buffer` module which is (IMPHO) a little more barebones than `StringBuffer`. *Long story short*, the answer is **no**. Fields that are merely marked immutable do not protect the data within. The upshot is:

To build truly immutable *data*, you have to construct *immutable objects* from the ground up.

Programming language support for *that* will be a major topic featuring OCaml, where it will be (should be!) less fussy than the manner prescribed in Scala.

VI. Other Topics

The other topics listed in the description will of course be covered, and some others besides: what "declarative" means, **tail recursion**, **referential transparency**, the stages of compilation, interpreters, **generic polymorphism**, and more. The following topics are most "on the bubble" however---to be covered as time and interest allow:

- **Dynamic scoping**.
- **Exceptions** (which are often (mis-)used to simulate dynamic scoping).
- **Concurrency**, i.e., **multithreading**. (This was on-the-bubble last year too. I may go just as far as exemplifying how immutable data matters for thread safety.)
- **Scripting languages** (I did cover Ruby in 2007, before Rails became prominent, and Sebesta still mentions Ruby).

- **Availability of Good Libraries**. This may actually be the most enduringly important factor, but it's less clearly a design issue of the programming language itself.

[If time allows, a little Philosophy: Two related ideas that are good to keep in mind are "Intension versus Extension" and "Sense versus Reference". *Intension* generally means the aspect of intentions. In programming, it is what you mean by a code element, whereas its *extension* is what you actually get when the code is executed. Jeffrey D. Ullman opened his first lecture in my sophomore programming course with the warning that:

If you ask a computer, "Make me a milkshake," it will go "Zzzzap!!---You're a milkshake!"

A program mismatch of intension and extension is, by definition, a *bug*.

Regarding the sense-reference distinction, here is my personal example. Note above that I named the `Rectangle` field names `x` and `y` but named the constructor parameters `gx` and `gy` with the `g` standing for "given". I didn't have to do this in Java: I could have used the same names `x` and `y` and written `this.x = x;` and `this.y = y;` in the constructor body. But prefixing or suffixing role or type information to variable names remains a fairly common practice. Now the example:

> "This is a habit of my sister-in-law's former boss's boss's ex-boyfriend, a real space cadet who dumped her after 15 years to suddenly marry a Swedish heiress half his age."

The *sense* is that the habit is dubious. Before we look up the *reference*, a further question: Who does the word "her" refer to? To my sister-in-law? *Ummm...* Answers here and here. I also use this as a way of vividly reminding people not to submit MS Word `.docx` files to *CSE Autograder*.]

Rest Of Lecture 2: What Makes Up a Programming Language? (cf. text chapters 1--2)

[To try to make this lecture look more like presenting slides than reading from text, I've inserted horizontal dividers.]

First: **What is a Programming Language?**

Definition (a): A PL = a compiler or interpreter (??) E.g. with C++ there are

- Gnu C++ (`g++`), ~~Sun~~ C++ (`CC`), CFRONT, `clang++` ...

Different compilers $\neq$ different languages!

ANSI C differs noticeably from “traditional” C. Are *they* different languages?

~~Try "`man gcc`" at the Unix prompt and~~ look at all the switches that can be combined in various ways. Does this mean we have hundreds of different PLs?

How about language versions?

- Python 2.7, Python 2.8, ..., Python 3.4, ... Python 3.11.1
- Scala 2.13.8 (well, there were multiple incompatible flavors of that), **Scala 3.0**...

Definition (b): A PL is a specification of properties that must be met by a compiler, in order for it to be called a compiler for that language.

This will include specification of *allowed options*, *deviations*, and *extensions*.

This leads to **Language Standards**, as often specified by "**Language Reference Manuals**."

Examples:
ISO Standard Pascal (ISO; Jensen and Wirth)
ANSI C (ANSI; Kernighan and Ritchie)
Common Lisp (Guy L. Steele)
Ada L.R.M. (~~U.S. DoD~~ **Gnu---integrated with their C/C++ system!**)
The Definition of Standard ML (Robin Milner)
...SML-NJ ...ML'97... ML2000... **OCaml**...
Java: still a closed standard owned by ~~Sun~~ Oracle Corp. (Besides the defiant challenge by Microsoft, many developers ~~are~~ were unhappy with Sun's stewardship of Java.)
ANSI Standard C++ (first 1998, then updates every 3-4 years...)

(ISO = International Standards Organization,
ANSI = American National Standards Institute.)

LRMs are now housed on **Official Language Websites**:

- Python: python.org
- Java: https://www.java.com/ (maintained by Oracle Corp.)
- C++: https://isocpp.org/ But note https://cplusplus.com/ and http://cppreference.com/ (Note this Quora forum question.)
- Ada was absorbed by Gnu C/C++: gcc.gnu.org
- Scala: scala-lang.org
- OCaml: https://ocaml.org/ But the official repository is https://github.com/ocaml/ocaml.org
- JavaScript: https://developer.mozilla.org/en-US/docs/Web/JavaScript more than javascript.com
  The former references this and this for the official specification.

- Prolog: Official ISO standard at https://www.iso.org/standard/21413.html and https://www.iso.org/standard/20775.html But mainline websites are https://www.swi-prolog.org/ and http://www.gprolog.org/ (GNU Prolog).

Creation of a Language

PL Definer: Can be a single person:
Ken Thompson "B"
Kernighan & Ritchie "C" NOTE: a User's Manual
Robin Milner "ML"
or a corporate team:
Ada design competition of 1976-79 had Blue, Red, Green, & Yellow design teams (Green won.)
Java! (Sun Microsystems) (acquired by Oracle Corp.).

Human element: standards committees do not always agree on what the standard should be. E.g.,
ISO Pascal Level 0 requires all arrays to be of fixed size.
ISO Pascal Level 1 allows variable-sized arrays as parameters to procedures.

Standards evolve.

What is a Programming Language? (cont'd)

Definition (c): A programming language is a vehicle for human-computer communication. (Also human-human or computer-computer communication.)

Low-Level PLs tend toward speaking the machine's language.

High-Level PLs speak toward our ways of organizing concepts (Structured Programming, OOP), if not yet in human language. (discuss COBOL).
Most modern PLs embody a particular philosophy of programming.

1. "Programmers (should) know what they're doing. Give them maximum access to the system, maximum flexibilty, maximum efficiency of well-crafted code." — C, C++.

2. "Keep programmers honest. Make programs as simple and clear as possible. Make bug-prone constructs impossible—programmers are all too fallible." — Pascal, Modula-2.

3. = 2. with “Be All That You Can Be” in place of simplicity. — **Ada95**. **Scala 3.0**.

4. “Simplify as much as possible, without sacrificing what you want to express. I Am The System—Lean On Me.” (+ “It’s the Library, Stupid!”) — Java.

5. Functional style: program toward what the spec says you need to do with data; don't leave it lying around and subject to changes that are hard to track. — ML, Lisp—especially “pure Lisp” or Scheme.

6. Declarative style: programmer says what to do rather than how.” -- Prolog.

Back to Language Standards

Standards typically include:

1) “**Positive Precepts**” — features that any compliant compiler must implement. The “normal” part of the
language.

2) “**Negative Precepts**” — constructs that the compiler must report as errors.

3) “**Positive Options**” — choices a compiler may make,
(e.g. , MAX_INT = 215-1 or 231-1 or 263-1), and:

Language extensions forseen and
allowed by the standards committee.

4) “**Negative Options**” — e.g. implementer is allowed to give a warning message rather than refuse to compile.

The chief goal of all such efforts at standardization is:

Portability.

A formal-sounding definition:

A PL offers **portability** if for any two compliant compilers C1 and C2, for “most” programs P, and for “most” data inputs x, C1 (P (x)) = C2 (P (x)).

BUT...humans being humans...we get in practice:

5) "**Omitted options**" — such as:
"Holes": stuff that "should" have been there but was left out. (Later committes may try to fill in gaps.)
"Trapdoors" (more an issue for individual compilers and software packages)
"Undocumented features."

6) "**Deviations**" — a) unintended: "compiler bugs."
b) intended: changes in language constructs or results thereof.
Example: Should -13 mod 7 equal +1, -6, or undefined?

7) **Non-Standard Extensions**

Evolution of Standards:

- Add useful features.
- Maintain upward compatibility (?) so old programs (incl. "legacy code") still run.
- Eliminate obsolete features (?).
- **Go Va Banque**. Did Ada95 kill the original Ada83? How about Scala 3.0 right now?

What is a Program?

Two world views:	PL paradigm:
1) Set of commands to computer	**Imperative**
2) Formal description of a process	**Functional, Declarative**

("Imperative" comes from Latin imperator, emperor, commander. "Precept" has a similar origin.)

1) Ada, C, Pascal, FORTRAN, COBOL, ALGOL,...

2) Prolog, Lisp (mostly), ML, Haskell,...

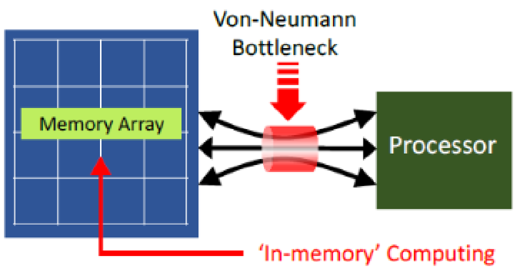
Historically, view 2) is the original one (unless you count Babbage in the mid-1800s). Beginning in the late 1800s, many mathematicians and philosophers sought formal definitions of “process” and “computation”:

C.S. Peirce (USA), David Hilbert, Gottlob Frege (Germany), Axel Thue (Norway) (1890–1930).

Kurt Gödel (Austria), Alan Turing (UK), Alonzo Church, Emil Post (USA), et al. (1930s).

View 1) took over in the 1940s when John von Neumann (USA; earlier Hungary and Germany) proposed to build real computers using the

von Neumann architecture.



(picture source) We still base on this architecture today, but while memory and dataflow have grown many times over, individual processors have not.

The major issue in the “Software Crisis”:
the conceptual distance between:
- **What We Want** (sometimes called **semantic intent**, same as what I mean by **intension**), and
- **What Actually Goes On In The Machine.**

(Read Sebesta ch. 2 with this in mind.)

Human:	**Machine:**
What We Want	**What Goes On**

Meanwhile, a main impetus in PL development and soft-ware engineering: to lessen the conceptual distance between:

- **Human intention and design, and**
- **What Actually Gets Programmed.**

How the PL world views influence this issue:

1) **Imperative**: Move from Specification to Program.

"The Standard Software Development Cycle":

Prototype -> Debug -> Test ->
Refine -> Debug -> Beta-Test ...

2a) **Functional, OOP**: Specification developed hand-in-hand with Program

2b) **Declarative**: The Specification IS the Program.

(Not quite true in Prolog.)

Major Innovations in PLs

- Freedom from direct programming of hardware (e.g. relay flipping) — machine language.
- Freedom from dependence on particular hardware — Assemblers (1st Gen. Langs.)
- Abstraction of program structure — (2 GLs) (FORTRAN, BASIC, ALGOL-60, parts of C)
- Full abstraction of data objects and processes — (3GLs) (COBOL, ALGOL-60, Lisp, ALGOL-68, C, Pascal) The most pivotal innovation. (Note on how this differs from the next point, which really is late-1970s.)
- Abstract Data Types, Modules —(3GLs) (Modula-2, Ada, Oberon) (more so than C's .h, .c files.)
- "Relationships among types" of objects and functions, polymorphism — (still 3rd Gen Langs.) C++, ML, Object Lisp (CLOS), Ada83 (somewhat)
- Object Oriented Progrmming — Simula (1967!); Smalltalk, C++, CLOS, Eiffel, Ada95, Java (Still considered 3rd generation languages!)
- "Macro Environments" that make useful routines available "without the need to program" (Hah!) — Lotus 1-2-3, dBase, HTML, SQL, HyperCard, "scripting languages" that come with PC or Mac software.... (Are these properly "4GLs"?)
- "Development Environments" as a layer above PLs, for RAD ("Rapid Application Design").(Hah!)

— NextStep for micros, APSE for Ada, Microsoft OLE and Tools, Apple's new Carbon API... (Metrowerks CodeWarrior Pro has a layer that simultaneously sits atop Pascal, C, C++, and Java, and the Java layer sits atop Sun's SDK and JavaBeans! And it is still not truly RAD!)

These, along with Fuzzy Logic systems, Natural Language Processing, and other capabilities may yet realize the hopes for Fifth Generation Langs that were raised in the 1980s. (The Japanese now admit that this experiment, based largely on Prolog, had only minor successes.)

Compilation and Interpretation

Java Interpreter   "Just-In-Time" Compiler

Interpretation is usually slower—a 10x slowdown over equivalent C++ code for current Java-ware. But JIT compilers promise to come well within a factor of 2. This may make Java into a "Universal Platform": OS, GUI, all (which is why Microsoft wants to crush this aspect of it).
Now if only Sun can get all the BUGS out of Java:

Write Once,
Run Anywhere,...
Debug Everywhere.

End of Ch. 1-2 Notes, except "Compilation" to come.

[If time allows, discuss some other terms:

Orthogonality: This is a technical term based on a mathematical analogy with the x,y,z dimensions in real space being orthogonal—and you can combine x,y,z values in any way to get a point (x,y,z). A simpler explanation is:

Orthogonality is when you can combine features of a language in natural ways without having lots of special-case restrictions to worry about. E.g. in the old Ada:

Procedure parameters could be **in** (read-only), **out** (used to mean write-only), or **in out** (read/write). Then objects *of any type* could be passed as parameters using these rules. This includes arrays. Those who have had Scala have seen ways in which arrays can be treated as value objects.

In C, however, there is the exception that arrays cannot be passed by value—since an array is "really" a pointer, it always gets passed as a modifiable object. This lack of orthogonality sometimes leads to errors by C programmers who write e.g.

```
void promote(Employee e, int[] payScale) {
  e.title = ‘‘Associate’’;
  payScale[e.IDnumber] += 5000;
}
```

forgetting that the `struct e` is passed by value and won’t be modified whereas the array will be...]