Major Language Paradigms

- Over the next few weeks, we will examine some of the major classes that separate programs:
  - Imperative languages
  - Object-oriented languages
  - Functional languages
  - Logical languages

- We begin with the oldest, most well-developed paradigm, **imperative programming**, which was developed to mirror the architecture of machines themselves.

Two Senses of “Imperative”: A

- The term **imperative programming** is generally used in two ways, one more general, and one more specific…

- In a general sense (used by text, chapter 1), imperative programming is contrasted with declarative programming:
  1. In a declarative language, the programmer says what they want to happen, and the compiler and/or runtime determine how this should occur exactly
  2. In an imperative language, it is up to the programmer to say how to do the various things they want to achieve

- No fully declarative language has ever been developed, and in that sense all programming languages are largely imperative
  - What would it require to develop a fully declarative paradigm?
  - How does the presence of library support, compiler optimizations, operating system and hardware decisions affect this?

Two Senses of “Imperative”: B

- A more specific meaning for imperative programming is the one mirrored by the formal semantics we have been examining (and is the one we will use here going forward):
  - In this model, state of a program consists of variables, with values
  - A program is a sequence of state modifications that converge to a final state—the meaning of the program

- The programming language then provides multiple tools for structuring and organizing these steps
  - Loops, conditionals, procedures/methods, etc.
  - Each can change what it does, based on the current state of the program and its variables
  - Each can also change the current state of those variables by assigning them new values
A Generic Imperative Program

- A program in an imperative language is essentially a simple series of steps:
  1. Declare and assign initial values to some variables
  2. Execute commands to change the values of those variables
  3. Terminate at some point with values in some given condition

FORTRAN Highlights

- First high-level programming language ever implemented
- First compiler developed by IBM for the IBM 704
- Project Leader: John Backus
- Technology-driven design
  - Batch processing, punched cards, small memory, simple IO, GUI's not invented yet

Imperative vs. Functional (C vs. Scala)

```java
def fib(n: Int): Int = n match {
  case 0 => 0
  case 1 => 1
  case j => fib (j - 2) + fib (j - 1)
}
```

Impressive code essentially relies on changing values of variables for calculation, looping, etc.

Purely functional style does not change any variables, simply binds values, executes function calls

Important Imperative Languages

- FORTRAN (J. Backus, IBM, late 50's)
- Pascal (N. Wirth, 70's)
- C (Kernigham & Ritchie, AT&T, late 70's)
- C++ (B. Stroustrup, AT&T, 80's)
- Java (Sun Microsystems, late 90's)
- C# (Microsoft, 00's)
Lexical/Syntactic Structure

- One statement per line
- First 6 columns reserved
- Identifiers no longer than 6 symbols
- Flow control using numeric labels (e.g., goto 10)
- Unstructured programs possible

Hello World in Fortran

```fortran
PROGRAM TINY
  WRITE(UNIT=*, FMT=*) 'Hello, world'
END
```

Basic FORTRAN

- Standard IO
- Default Format

Implicitly Defined Variables
- Type determined by initial letter
  - I-M ~ INTEGER
  - A-H, O-Z FLOAT

Iteration in FORTRAN

```fortran
PROGRAM REDUCE
  WRITE(UNIT=*, FMT=*)'Enter amount, % rate, years'
  READ(UNIT=*, FMT=*) AMOUNT, PCRATE, NYEARS
  RATE = PCRATE / 100.0
  REPAY = RATE * AMOUNT / (1.0 - (1.0+RATE)**(-NYEARS))
  WRITE(UNIT=*, FMT=*)'Annual repayments are ', REPAY
  WRITE(UNIT=*, FMT=*)'End of Year Balance'
  DO 15, IYEAR = 1, NYEARS, 1
    AMOUNT = AMOUNT + (AMOUNT * RATE) - REPAY
    WRITE(UNIT=*, FMT=*)IYEAR, AMOUNT
  15 CONTINUE
END
```

Loop control code consists of two separate statements.
- Second is labeled (15), allowing control to jump to that location at any time
- Easy to construct unstructured (and confusing) programs

Enter amount, % rate, years
- 2000, 9.5, 5
- Annual repayments are 520.8728
- End of Year Balance
  - 1 1669.127
  - 2 1306.822
  - 3 910.0968
  - 4 475.6832
  - 5 2.9800416E-04
FORTRAN ARRAYS

Subroutines are analogous to void functions

All parameters are passed by reference

```fortran
SUBROUTINE MEANSD(X, NPTS, AVG, SD)
  INTEGER NPTS
  REAL X(NPTS), AVG, SD
  SUM = 0.0
  SUMSQ = 0.0
  DO 15, I = 1, NPTS
    SUM = SUM + X(I)
    SUMSQ = SUMSQ + X(I)**2
  15 CONTINUE
  AVG = SUM / NPTS
  SD = SQRT(SUMSQ - NPTS * AVG)/(NPTS - 1)
END
```

Expressions and Assignment

- The fundamental core of imperative programming:
  \[ \text{target} = \text{expression} \]

- Copy semantics: an expression is evaluated to a \textit{value}; this value is then copied over to the target

- Distinct from reference semantics, used by OO languages: expression evaluated to an \textit{object}; a \textit{pointer} to object then copied over to the target

Computational Completeness

- A language with the following basic imperative operations and control structures is Turing complete (i.e., it can be used to compute anything computable by any universal Turing machine):
  1. Integer variables, values, and operations
  2. Assignment statements
  3. Conditional statements
  4. GOTO operations (for a more structured programming style, replace these with while loops)

Side Effects

- A side effect is any permanent state change caused by execution of a language construct
  - Most often discussed in connection with functions, but can be seen in many language constructs
  - Some noticeable effect on the state
  - Assignments are the ultimate example of side effects: they obviously affect the state by changing the value of a variable directly

- Purely functional/logic/dataflow languages forbid side effects
  - These languages are called single-assignment languages
  - Each assignment is essentially permanent: a variable cannot change unless it is re-declared and replaced (all effectively final)
Side Effects

- Several languages outlaw side effects for functions
  - Easier to prove things about programs
  - Easier to optimize
  - Easier to understand (often)
- Example: languages Euclid and Turing: all side-effects of non-void methods (designated as functions) are banned
  - Every such method always returns the same thing
- But side effects can be useful in practice
  - Consider Math.random() in Java

Side Effects in Expressions

- Side effects are a particular problem if they affect state in other parts of an expression in which function is called, as in this example:

```java
int x = 3;
int funA() { x = x + 1; return x; }
int funB() { x = x * 3; return x; }
int y = funA() + funB();
int z = funB() + funA();
```

- On the one hand, it’s nice not to have to specify an order in which the operands of the + here must be evaluated, because it makes it easier to optimize code
  - However, this makes the outcome of a program like this one harder to explain and predict
- Fortran allowed side effects, but tried to limit them in practice:
  - Programmers aren’t supposed to write functions that change other parts of an expression containing a call of that function
  - Unfortunately, compilers can’t check this completely, and most don’t try

Imperative Programming in Scala

- In many ways, Scala behaves like a functional language, with restriction on side-effects, and single-interpretation bindings of val identifiers
  - However, it is not a purely functional language, and contains a number of constructs allowing for programming in imperative style, along with object-oriented features
  - Some Scala constructs are mutable already (especially ideas brought over from Java)

Next Week

- **Topic**: Programming Paradigms: Object Oriented Languages (Text, chapter 10)
- **Meet**: usual schedule
- **Homework 04**: due Friday, 26 April, 5:00 PM
- **Office Hours**: Wing 210
  - Monday: 9:00 AM – 10:30 AM
  - Tuesday: 3:00 PM – 4:00 PM
  - Wednesday: 9:00 AM – 10:30 AM
  - Thursday: 2:00 PM – 3:00 PM
  - Friday: 9:00 AM – 10:30 AM