SE 2A04 Fall 2000

02. Software Fundamentals

Instructor: W. M. Farmer

Revised: 6 October 2000

Some Starting Questions

- What is a program?
- What is a software product?
- What does it mean for a software product to be correct?

Program as Sequence of Instructions

- A program is most often viewed a **sequence of instructions for a machine**
- A **machine language program** is a sequence of instructions for a physical machine
  - Usually represented as a sequence of 0s and 1s
  - Not intelligible to humans
- A **high-level language program** can be viewed as a sequence of instructions for a high-level abstract machine
  - Ultimately executed on a physical machine via **interpretation** or **compilation**
- As a sequence of instructions, a program is **data** for the machine, interpreter, or compiler

Program as Input/Output Rule

- A program can be viewed as **a rule that associates a set I of inputs with a set O of outputs**
  - A rule can be represented as a relation $R \subseteq I \times O$
- An input/output rule is **total** if every input $i \in I$ is associated with some output $o \in O$
- An input/output rule is **deterministic** if every input is associated with at most one output
  - A deterministic rule can be represented as a function $F : I \to O$
- An input/output rule is usually implemented by an **algorithm**

Program as Abstract Machine

- A program can be viewed as a **description of an abstract machine** composed of:
  - A process control mechanism
  - Input and output mechanisms
  - Memory holding the **state** of the machine
- Two important abstract machine models:
  - Turing machine
  - Finite state machine

Turing Machines (Alan Turing, 1936)

A **(deterministic) Turing machine** M consists of the following components:

1. A two-way infinite **tape** composed of cells
2. A **tape head** that can read a cell and then (a) write a symbol in it, (b) move left, or (c) move right
3. A **finite state control mechanism**
4. Fixed finite set $\{s_1, \ldots, s_m\}$ of **tape symbols**
5. Fixed finite set $\{q_1, \ldots, q_n\}$ of **states**
6. A **transition relation** consisting of a finite set T of quadruples of the form (q_i, s_j, s_k, q_l), (q_i, s_j, L, q_l), or (q_i, s_j, R, q_l) such that (a) $1 \leq i, l \leq n$, (b) $0 \leq j, k \leq m$, and (c) for all (q_i, s_j), there is at most one $(q_i, s_j, \alpha, \beta) \in T$

Finite State Machines

A **finite state machine** M consists of the following components:

1. A fixed finite set S of **states**
2. A fixed (possibly infinite) set I of **inputs**
3. A fixed (possibly infinite) set O of **outputs**
4. A **next state** relation ns $\subseteq S \times I \times S$
5. An **output** relation out $\subseteq S \times I \times O$

M is **deterministic** if the relations are functions, i.e., ns : $S \times I \to S$ and out : $S \times I \to O$

Program as Expression

- A program can be viewed as an **expression of a formal language**
- As an expression, a program describes:
  - Its value (**declarative semantics**)
  - How its value is computed (**procedural semantics**)
- A language is **referentially transparent** if "equals can be substituted for equals" without changing the values of expressions
  - Languages not referentially transparent languages are poorly behaved
  - A programming language with assignment (e.g., C) is usually not referentially transparent with respect to the notion of equality provided by the language itself

Program as Proof

- A program can be viewed as a **constructive proof of a existential statement**
  - The existential statement is effectively a specification of the program
  - By virtue of being "constructive", the proof is executable
  - Programming is reduced to proving
- This view of programs is intellectually attractive but has severe practical limitations
  - Verification of correctness comes for free
  - Programs are well understood and well documented
  - Since programs are constructed as proofs, they tend to be highly inefficient

What is a Program: Summary

- Program as an instruction for a machine
  - An understanding of a program requires an understanding of the machine
- Program as an input/output rule
  - Good if the state of the program is not relevant
- Program as an abstract machine
  - Good if the state of the program is relevant
- Program as an expression in a formal language
  - Good if the language is well behaved
- Program as a constructive proof of an existential formula
  - Very impractical with today's technology

Some Basic Definitions

- A **software module** is a collection of programs that work together as a unit.
- A **software product** or **software system** is a system of programs, data, and documentation. A software product is often composed of modules.
- A **computer system** is a computational system consisting of both hardware and software products.
- An **embedded system** is a computer system that is part of a physical product.

Ways of Classifying Programs

- Sequential vs. concurrent
- Terminating vs. nonterminating
- Active vs. reactive
- Application vs. control

Software Specifications

- A **software specification** describes what a software product is required to do
- Serve three purposes:
  - Blueprint for developing the product
  - Document that describes the product
  - Basis for verifying the correctness of the product
- There is usually a range of specifications associated with a software product
  - At one end is a **requirements specification** which specifies the required properties of a product
  - At the other end is a **product description** which describes the actual properties of a product

Software Implementations

- An **implementation** of a software specification is
  - Another software specification or
  - A software product

  that **satisfies** the requirements of the specification
- A specification answers the question **What is the program required do?**, while an implementation answers the question **How does the program meet its requirements?**
- The implementation is often a **refinement** of the specification
  - Includes details which are not in the specification
  - Says how the requirements are satisfied

Programming Languages

- Programming languages are intended to facilitate the writing of programs but not necessarily the designing of programs
- There are many kinds of programming languages
  - Imperative (Examples: C, Pascal, Basic, Fortran)
  - Object-oriented (Examples: Smalltalk, C++, Java)
  - Higher-order languages (Examples: Lisp, Scheme, ML)
  - Functional (Examples: Haskell)
  - Logical (Examples: Prolog)
- The design of a program should be tied to a specific programming language as little as possible

Types

- Types are used in several ways in logic and programming
- In logic, a type is a **syntactic object** that denotes a set of abstract elements
  - Types are used to control the formation of expressions
  - Types are used to classify expressions by value
- In programming, a type is a **set of data values**
  - Types are used to classify values (**latent types**)
  - Types are used to classify variables (**manifest types**)
  - Types are used to control the formation of expressions and to classify expressions by value

Variables

- The meaning of "variable" is different in logic, programming, and control theory
- In logic, a variable is a **symbol** that denotes some (often unspecified) element
- In programming, a variable is a **memory location** with the following attributes:
  - **Identifier**: Name of the variable
  - **Value**: Data value stored in the location
  - **Type**: Type of data that can be stored
- In control theory, a variable is a **changing value** that is a component of the **state** of a system
  - A **monitored variable** is a variable the system can observe but not change
  - A **controlled variable** is a variable the system can both observe and change

Components of a Powerful Language

1. Primitive expressions
2. Means of combination
   - Compound expressions are built from simpler ones via constructors
   - The expressions denote combinations of objects
3. Means of abstraction
   - Compound expressions are built from simpler ones via constructors
   - The expressions denote new objects

Taken from Abelson, Sussman, and Sussman, *Structure and Interpretation of Computer Programs* (see references)

Example: Imperative Programming Language

- Primitive expressions: assignment statements
- Means of combination:
  - Composition
  - Conditional selection
  - Iteration
- Means of abstraction:
  - Type definitions
  - Variable and constant definitions
  - Procedure and function definitions

Example: Lambda Notation

- Lambda notation (or something equivalent) is used in many languages to express ideas about functions
  - Lambda Calculus (a model of computability)
  - Simple Type Theory (a higher-order predicate logic)
- Primitive expressions: variable and constant symbols for denoting primitive functions and individuals
- Means of combination: function application $f(a)$
- Means of abstraction: function abstraction $\lambda x \,.\, s$
- Alpha conversion: $\lambda x \,.\, s = \lambda y \,.\, s[x \mapsto y]$
- Beta conversion: $(\lambda x \,.\, s)t = s[x \mapsto t]$