The UNIVERSITY of VERMONT

Benjamin Fenster
CS-103 (Skalka)
5 September 2003

Notes – Introduction

I. Introduction
  a. Course Materials
    i. Smith & Grant online. Nothing to buy
    ii. Language: OCaml
    iii. Emacs, or any text editor
    iv. Homework submitted using Submit.
  b. Coursework
    i. Regular readings as a recommended lecture supplement
    ii. Homework
      1. Weekly
      2. 50% of final grade
    iii. Late policy: 10 points per day, up to 7 days
    iv. Tests
      1. Other 50% of grade
      2. Midterm (20%)
      3. Final (30%)
    v. Cheating: Conceptual collaboration okay, just don't copy anything.
  c. Suggestions
    i. The course is very systematic: concrete steps
    ii. Don't miss anything.
    iii. Tests are open-notes, so take good notes.
    iv. Don't think of OCaml the same way as C++ or Java

II. Overview
  a. Central Concepts
    i. Computability
      1. How to precisely characterize computing power
      2. Necessary features of programming languages to realize that power.
    ii. Syntax (form), Semantics (meaning)
    iii. Static Analysis (types)
      1. "Things you can tell by looking at code"
      2. Can get certain information about the program just by looking at the code statically.
    iv. Reasoning about programs
  b. Organization
    i. First half: OCaml language
    ii. Second half: Using OCaml to implement an interpreter for a small language

III. Computability
  a. "Computers" (electronic) are just one instance of the broader concept of "computing device."
  b. Functions
    i. Definition: A function, set theoretically, is defined as a set of ordered pairs (x, y) such that if there exists (a, b) in f and (a, b') in f, then b = b'.
    ii. Example: Doubling Function = {(1, 2), (2, 3), … }
    iii. Definition: Given a function f, the domain of f, written dom(f) is $\{x \mid (x, y) \in f\}$. The range of f, written rng(f) is $\{y \mid (x, y) \in f\}$.
    iv. Doesn't necessarily state that f(x) is computable!
  c. Computability
    i. Common notion: A procedure is computable on a given input iff it can be described in a finite manner as a set of definite actions and provides an output in a finite amount of time.
    ii. Any formal definition devised so far conforms to this common notion.
    iii. Note the time constraint. It needs only be finite. Efficiency is not addressed. Thus, an algorithm that takes billions of years is still considered computable.

- d. Turing Machines
  - i. Developed in 1930s by Alan Turing
  - ii. Idealized computing device, doesn't exist.
  - iii. A TM is comprised of…
    - 1. An indefinitely long tape (NOT infinite, just indefinitely long) divided into squares containing either 1 or 0.
    - 2. A read/write shift head (moves left and right) that can see one square at a time.
    - 3. An input card hopper capable of reading instructions from cards.
    - 4. A small region of internal memory containing an internal state ID (arbitrarily large integer)
  - iv. State IDs
    - 1. s1, s2, s3, …
    - 2. At any given moment, a Turing machine will be in a particular configuration
    - 3. Its configuration consists of the value currently under the reading head, together with the current state ID.
    - 4. Start configuration is (s1, n_s) where n_s is the value under the reading head, given the initial tape.
  - v. Programming
    - 1. Input cards of the form $<S_i, n, a, S_f>$ (a quadruple)
      - a. S_i specifies the initial state.
      - b. n specifies the current read/write value
      - c. a specifies the definite action
        - i. Write 1
        - ii. Write 0
        - iii. Shift left
        - iv. Shift right
        - v. Halt
      - d. S_f specifies the result state.
    - 2. Cards are ordered, but are NOT "if..then" instructions in the traditional sense. Whichever card represents the current configuration gets executed.
    - 3. The reason the cards must be ordered is that two cards may contain the same S_i and n, in which case the first card is executed.
    - 4. Clearly very difficult for humans to program.
    - 5. "Computable" means that if the tape is initialized with x, then there exists a stack of cards such that the machine will halt with f(x) on the tape.
  - vi. Non-Termination
    - 1. Consider the card <s1, 0, Write 0, s1>
    - 2. Assuming the tape starts with 0, the program never ends.
    - 3. Much more than just an annoyance to the programmer.
    - 4. Any Turing-Complete language will have this "feature."
    - 5. Fact (Halting Problem): There is no computable procedure for deciding whether an arbitrary TM will halt on a given input.

IV. Programming Languages for Electronic Computers

- a. Instruction set: Basic, primitive operations.
- b. Machine Level languages specify procedures in terms of instruction sets.
- c. Like card stacks for Turing Machines, very difficult and error prone for humans to use.
- d. High Level Languages (HLLs) specify procedures at a much more abstract level: a level that's easy for humans to understand.
- e. Even when using HLLs, we don't change the instruction set of the computer.
  - i. Need a translator of some kind.
  - ii. Compiled Languages

1. Most efficient implementation
2. Compilers are extremely complex.
    a. Take HLL and translate into machine level.
    b. Very complicated.
    c. Most complex software in existence in terms of the amount of theory involved.
3. Interpreted Language
    a. An interpreter of the language written in another HLL
    b. Correctly implements constructs in terms of the other HLL
    c. For example, an interpreter in C++ would determine the meaning of a statement and then execute it by running some C++ code.
4. NB: Any Turing-Complete language can interpret itself (called meta circularity)

ERROR: undefinedfilename
OFFENDING COMMAND: </FONT>

STACK: