CS 131 Programming Languages (Programming Notations)

September 24, 2009

Paul Eggert, eggert@cs.ucla.edu OH Boelter Hall 4532J, Mon. 12:50-1:50pm, Wed. 11am-12pm.

Jerry Weng, jtweng@cs.ucla.edu OH unknown

Official hours: lecture, 4 hours, discussion: 2 hours, outside study: 6 hours.

Real hours: lecture, 3.5 hours, discussion: 1.8 hours, study: 20 hours.

Grading: homework, 40%, midterm, 20%, final, 40%. Must get a passing grade on the homework to pass the course. Midterm is on October 27, Tuesday.

Rules: Exams are open book and notes, closed computer. 7 homework assignments, one is worth double points (the project). Lateness penalty is $2^{(d-1)}$% (1% if 1 day late, 2% if 2 days late, 4% if 3 days late, etc). For the homework, submit code that works and is robust.

Course Topics:

> *Theory*: language design, syntax, semantics, functions, names, types, control, scripting, objects, exceptions, concurrency.
>
> *Practice*: OCaml, Java, Prolog, Scheme, Python, one more language.

Be good at 3 things: learning a language, evaluating languages, designing languages.

Categories of Languages

1. *Imperative* – like a cookbook recipe, consists of instructions, meant to be executed sequentially. Focuses on commands/statements.
2. *Functional* – define a set of mathematical functions such that if you evaluate one of the functions, the program will exhibit the desired behavior. Focuses on function definitions and calls. There are no commands/statements.
3. *Logic* – assertions/predicates about the real world.

Languages could be further categorized by being object-oriented and non-object-oriented, scripting languages versus compiled languages, etc.

Functional Languages

Motivations:

1. Clarity: mathematical functions are very well understood entities. The idea is to base the programming notation on the mathematical notation and to escape the need to follow the evaluation order in statements.
2. Performance: compilers can generate better code if they're not constrained to execute statements in sequence. There's more opportunity for the complier to generate multicore code.

Function: in mathematics, mapping from a domain to a range (D->R). *Functional form*: a function which arguments are other functions. For example, the operator ○ is used to create composite functions: $(f \circ g)(x) = f(g(x))$. Evaluation order is controlled by recursion; there's no such thing as iteration. There are no variables in a sense that there is no assignment statements. Every evaluation of an identifier or an expression yields the same value (*referential transparency*).

The book uses ML and we're using OCaml as the functional language. There are a few differences in syntax.

OCaml	ML
&&	and also
'c'	#"c"
[3;5;7]	[3,5,7]
let x = 3+4	val x = 3+4
3.0 +. 4.0	3.0 + 4.0

OCaml syntax examples:

```
# 3,19 ;;
-: 'int*int'=(3,19)
# [3;19];;
-: 'int list = [3;19]
```

Some OCaml notes:

- A compiled language and has compile-time checking (like Java, C)
- You don't need to write the data types (Python, Lisp)
- It has garbage collector (Java)
- Good support for higher-order functions.

Word Count Problem

M.D. McIlroy came up with a problem on word search: given an input of ASCII text, output the count of every word (matching [A-Za-z]+) in descending order of occurrence.

D. E. Knuth (Stanford) wrote *TeX* (in Pascal), a program for formatting books. He didn't like the way Pascal forced one to a rigid code structure so came up with *literate programming*. The syntax of his code could be used to generate Pascal code and TeX document alike. Knuth wanted to publish his work but didn't think the code was easy to read and understand. McIlroy suggested to Knuth to solve his problem using his own concept of *literate programming*. Knuth published his paper and McIlroy published his commentary in CACL.

McIlroy's idea: solve the problem with no data structures. His solution: first write a little program (program A) that will generate a file containing every word occurring in the input, in the same order (1 word per line). Next, another program (program B) sorts the file alphabetically, case significant. Next, another program (program C) compares adjacent lines in the file and counts the duplicate words (that are right next to each other since the file is sorted), putting the resulting number before the word. Finally, another program sorts the resulting file numerically, which results in the desired output. McIlroy made a shell script to create his solution:

```
(tr –cs 'A-Za-z' '[\n]*' | sort | uniq –c | sort –nr) < input.txt > output.txt
```

Knuth's approach is very different and is much longer – many pages of source code.

Bottom line: *choice of notation is the key*. There's no single programming language that suits every kind of problem.

Reading assignment: read chapters with "ML" in the title: 1-3,5,7.