Introduction to Computer Science: A Mathematical Perspective

Prabhakar Ragde

1 Description

The connections between mathematics and computer science are varied and deep. This review course for the 2015-2016 program at AIMS Senegal will introduce students to foundational ideas in computer science and their relationship to foundational ideas in mathematics through the use of a functional programming language (Racket) designed for education. No prior experience with programming is required, though students with exposure to popular programming languages will also benefit from this alternate approach.

The course is based on material developed for an online course aimed at teachers of mathematics enrolled in the Master of Mathematics for Teaching program at the University of Waterloo, which in turn was based on courses required of first-year undergraduate students in the Faculty of Mathematics.

2 Learning Objectives

Upon successful completion of this course, students should be able to write a properly-formatted and well-documented program of up to a hundred lines of Racket code to solve a problem or carry out a task, given only a clear and concise statement of the problem or task. They should be able to use elementary data structures such as lists and trees in such programs, and to make use of higher-order functions to improve readability and efficiency. They should be able to justify correctness and efficiency of small Racket programs by reference to a formal model of Racket computation, and be able to bring a computational perspective to bear on their study of mathematics.

3 Syllabus

The course is divided into thematic modules that vary in length.

Module 1: Introduction to Racket. The DrRacket programming environment. Values. Expressions using built-in functions. The substitution model for evaluation. Constant definitions. Function definitions. Documenting a Racket function. Conditional expressions.

Module 2: Natural numbers. Grouping values using structures. A recursive definition of a natural number. Representing numbers using structures. Elementary arithmetic using the unary representation. Recursive computation and its relationship to mathematical induction. Different kinds of recursion. Proving properties of functions on natural numbers.

Module 3: The design recipe. A process for designing and documenting Racket programs. Boolean values and conditional expressions. Testing Racket programs.

Module 4: Syntax and semantics. Precise definitions of the grammar of Racket and the computation model (the substitution model).

Module 5: Lists. Representing sequences using structures. Functions consuming and producing lists. Efficiency of computation. Representations of sets.

Module 6: Functional Abstraction. Functions as values. Functions that consume and produce functions. Higher-order functions. Local definitions. Lexical scope.

Module 7: Efficient Representations. The binary definition of a natural number. Elementary arithmetic using the binary definition. Mutual recursion. Representing integers.

Module 8: Trees. Efficient representation of sequences. Binary search trees. Expression trees. The limits of computation.

4 Assessment

Student grades will be based on fifteen to twenty programming exercises. Each exercise will have a concise description and a relatively concise solution (ten to fifty lines of code, excluding test expressions), though arriving at the solution may take some thought and some experimentation.

5 Resources

Racket is a free download and works on multiple platforms (Unix, Mac, Windows, Linux). We have used it in first-year CS at the University of Waterloo since 2004. For this course, we will be using the current version, Racket 6.3.

There is no appropriate textbook for the course. Lecture slides (PDF) will be provided electronically in advance. Additional materials may be provided as PDF documents or Web pages.

6 References

`http://www.racket-lang.org` : the main Racket site.

`http://www.programbydesign.org` : explains the educational philosophy that motivated Racket.

`http://www.bootstrapworld.org` : using Racket to teach algebra to middle school students.

`http://www.htdp.org` : the original textbook (2001-2003) using what are now the Racket teaching languages. Obsolete, but a major inspiration for this course.

`http://www.ccs.neu.edu/home/matthias/HtDP2e/` a new edition (2016) in progress.
Uses images and animation.

`http://www.realmofracket.com` : a textbook using full Racket and games.

7 Motivation

This section does not form part of the syllabus. It was originally drafted to provide deeper background for those who needed to grant approval for the course, but it may be of use for students wishing to learn more about the course in advance, or decide whether to take the course for credit. It explains why the use of Racket will not only result in a course that is best positioned to draw connections between mathematics and computer science, but will be significantly different from introductions using more conventional languages.

The syntax of the Racket language is quite simple, featuring a small number of powerful constructs. In this respect, it resembles a Swiss army knife rather than a big, cluttered toolbox. We can spend less time on language details and more time on problem solving. We can also give a mathematical model of what it means to run a program that is just an application of secondary-school mathematics.

Computation in Racket revolves around evaluations of expressions. Consider a mathematical expression such as $(1 + 2) \times (3 + 4)$. We can use substitution to simplify this, first replacing $(1 + 2)$ with 3 to obtain $3 \times (3 + 4)$, then replacing $(3 + 4)$ with 7 to obtain 3×7, and finally replacing 3×7 with 21. This substitution model is the basis of the computational model for Racket.

The preceding example used the binary infix operators $+$ and $\times$. We also use prefix notation in mathematics, for example with negation ($-x$), and trigonometric functions ($\sin x$). Prefix notation is also used for functions that we define. As an example, we may define $f(x, y) = x^2 + y^2$, and use this function in the expression $f(3, 4)$.

The evaluation of $f(3, 4)$ also uses substitution. The values 3 and 4 are substituted for x and y, respectively, in the expression $x^2 + y^2$ given in the definition of f. After that, evaluation proceeds as already described.

Racket unifies the treatment of operators and functions by using only prefix notation. The name of the function is moved inside the parentheses, and spaces are used to separate arguments, instead of commas. So $f(3,4)$ becomes `(f 3 4)`, and $3+4$ becomes `(+ 3 4)`. Here is the Racket definition of f (given a better name in the program), followed by an example of its use.

```
(define (sum-of-squares x y)
  (+ (* x x) (* y y)))

(sum-of-squares 3 4)
```

A Racket program is a sequence of definitions and expressions. When the program is run, the expressions are simplified down to values, using substitution. This view of computation, in which the program itself is simplified, is different from the view taken by most other programming languages. When programming in Java, C++, Python, or MATLAB, typically the program is viewed as fixed, and something else is changing during the computation (values stored in variables, or memory).

Here is the simplification of the above expression as it will be described to students. The double right arrow `=>` represents one substitution step.

```
(define (sum-of-squares x y)
  (+ (* x x) (* y y)))

(sum-of-squares 3 4)
=> (+ (* 3 3) (* 4 4))
=> (+ 9 (* 4 4))
=> (+ 9 16)
=> 25
```

We introduce very few additional programming features.

- conditional expressions (`if` and `cond`);
- structures to group values (akin to tuples in mathematics);
- a way to create functions during the computation (`lambda`);
- a way to create definitions with restricted (`local`) scope.

For each feature, as it is introduced, we extend the substitution model to explain it. Thus the student always has a strong mental model of what should happen when a program is run.

The programming environment DrRacket offers further advantages. When opened, it presents to the user a window divided into two panes. The top pane is the Definitions window, holding the program. When the program is run, the values of its expressions are printed in the bottom pane, the Interactions window. The programmer can then type in additional expressions interactively and have them evaluated one at a time. This completely eliminates the need to discuss input and output. DrRacket also provides a Stepper which does interactive step-by-step simplification using the substitution model.

Racket is actually a family of languages, and DrRacket provides nested subsets of the full Racket language, known as the teaching languages. These restrict the set of language constructs available to the beginner, avoiding situations where they accidentally write something with a meaning unknown to them, and permitting more informative error messages. At the same time, DrRacket is powerful enough to be used for graduate-level research in computer science and mathematics, and by experienced programmers in industrial settings.

The teaching languages provide a lightweight testing framework through the use of `check-expect` expressions, each of which consists of a Racket expression and an expected value. The tests thus set up are run at the end of evaluation, allowing `check-expect` expressions to be used for both examples and testing. Here is the sum of squares example documented in the style that will be expected of students (a semicolon starts a comment):

```
; sum-of-squares: Number Number -> Number
; produces the sum of the squares of x and y
; example:
  (check-expect (sum-of squares 3 4) 5)

(define (sum-of-squares x y) (+ (* x x) (* y y)))

; tests
(check-expect (sum-of-squares 4 5) 41)
(check-expect (sum-of-squares 5 12) 169)
```

This approach ensures that the overhead of creating, running, and properly testing programs is low, allowing students to focus on the essential problem-solving elements of any assigned task.