Report of the DUGC committee on CS 152 content and positioning

Abhiram Ranade, Parag Chaudhuri, Pushpak Bhattacharya

April 24, 2015

1 Mandate and background

The mandate of the committee was to recommend whether (a) the content of the course CS 152, Abstractions and paradigms for programming, needs to change, (b) its position in the curriculum should be exchanged with CS 213, Data structures and algorithms, (c) whether there is need to ensure that the content of CS 152 does not change from year to year, (d) how the decision is affected by the title "Departmental introductory course" given to the curricular provision under which this course runs.

The above questions were raised in DUGC by the student member Navin Chandak and were discussed substantially in DUGC. These discussions formed the background to the deliberations of the committee. The discussions provided some detail (but not comprehensively) about how the course has run over the years. There is also a description of the course in the Academic Section documents; however this is very sketchy:

> *Review of the program development process, Issues in program design, Structured programming, Data and control abstractions, Programming with assertions. Reasoning about programs and proving correctness of programs. Ideas behind imperative, applicative, object oriented and logic programming paradigms such as typing, expressions, pure functions, recursion, higher order functions, encapsulation, inheritance, goal satisfaction, backtracking, unification. Some of the ideas behind the implementation of the paradigms. Course to be centered around problems and applications that demonstrate the main themes.*

In addition, the committee also considered the anecdotal remarks made very commonly in faculty and student discussions about how our students' skills in programming need improvement.

2 Preliminary remarks

The intent behind CS 152, in our opinion, is to improve the programming skills of students. The study of different paradigms is useful in this regard, but it might be noted that the syllabus specifies the study of "Ideas behind imperative, applicative, objected oriented and logic programming", rather than the study of the paradigms themselves.

The more interesting and difficult ideas in programming need to be motivated by appropriate programming problems. And indeed, the syllabus suggests "Course to be centered around problems and applications that demonstrate the main themes".

We believe that over the years different offerings of the course have stayed close to the spirit (to the extent we have information about them). There also have been some difficulties, for example, logic programming has been found somewhat difficult to teach. This is partly because of shortage of time, and also because perhaps ideas such as unification, cuts are somewhat involved. On the other hand, ideas such as backtracking and constraint satisfaction are very accessible.

In the largest number of offerings, the main language used in the course has been Scheme. Scheme supports the imperative, functional, and object oriented paradigms reasonably well. However, there is some concern whether students end up viewing the ideas learned in the course as "ideas good for Scheme programming", rather than "ideas that ought be used no matter what language you use".

2.1 On CS 213

CS 213 is a prerequisite for almost every course in the CSE curriculum, and could thus be considered the "spirit of CSE", and thus a course better suited to be the "Departmental Introductory Course".

However, it is to be noted that the course is quite loaded with theoretical/algorithmic topics. It has well established content with specific data structures to study (e.g. balanced search trees, heaps), specific algorithms

(e.g. sorting and searching, breadth first and depth first search, shortest paths, minimum spanning trees). There is a good deal of algorithmic difficulty, and the spirit of the course is "theory", even if there is a programming lab associated with it. Even in the programming lab, the emphasis has often been on inventing data structures, even somewhat clever ones.

2.2 Structure and Interpretation of Computer Programs

SICP, by Abelson and Sussman, has cast a long shadow on CS 152. SICP is a heady brew that is purported to be an introduction to programming. It is possible to read the book like a novel you cannot put down, and Sussman's lectures on MIT OCW from 1986 (!) are thrilling and give goosebumps. But if we want to be enlightened rather than just enthralled, it is useful to analyze the content and figure out just what exactly makes SICP appealing.

The first important idea is of course the model: substitution as a model of computation. This is very attractive, since we are supposedly used to it from Mathematics. Substitution is elegant, but by no means easy: it takes a certain amount of mind bending to understand the nested lambda expressions. It does have some direct tangible benefits: lack of assignments makes proving correctness easier; in general it can be said that programs are easier to understand if all variables are assigned a value just once. Several of these properties were perhaps first noted in the functional programming literature; however, many, if not all, are important and useful in imperative programming.

The situation is more complex when we consider assignment statements. The assignment statement of Scheme is quite different from the statement in C++. The semantics of the latter is trivial to understand as compared to the former. In fact, with assignments, the Scheme execution model becomes decidedly different from C++. When functions are returned from a call, they can refer to variables created in the call – which would really be dangling references in C++. So in addition to the syntax, the computational model is quite different.

SICP emphasizes many programming methodological ideas. One important idea is that of creating embedded languages – like the language created for manipulating mathematical formulae. Another idea is that of creating generic (type independent) data structures/control abstractions. This was

impossible in C, but is reasonably convenient in C++ and similar languages. However, the way this is accomplished in C++ is quite different from the way it happens in Scheme.

Finally, SICP also takes programming to a very rich set of applications. These would certainly be difficult to implement in C; but can be implemented reasonably well in C++/Java.

The above analysis of SICP does not explain the cult following (well deserved) that SICP has come to acquire. We believe there is a psychological dimension to it. This concerns how SICP introduces programming.

Unlike a standard programming introduction, which is almost entirely about arithmetic, SICP dispenses with arithmetic relatively quickly and rises briskly to representation of algebraic expressions. Very quickly the book also moves up to calculus, taking derivatives of algebraic expressions. To people who have grown up with calculators, arithmetic is for children; algebra and calculus signals to them that something higher is being discussed. Even for arithmetic, the examples picked are not about calculation of tax or changing from Centigrade to Fahrenheit, but are about more interesting problems: finding square roots, calculating trigonometric functions. Most introductory programming books attempt to go easy on the reader: they take up computations which the reader is sure to know. Indeed, one pedagogical theme in introductory programming is to encourage the reader to express in a program what he or she knows intuitively. In contrast, the Babylonian square root algorithm presented in SICP is mysterious; the reader is stunned by its mathematical appeal as much as its recursive expression. It is as if the author is saying, I want to tell you something worth your time, not baby stuff.

To conclude this analysis: many of the good things of SICP are independent of the programming language. This includes the wealth of applications considered, the exhortation to rise above writing programs to creating embedded languages, the many ideas about programming style. On the other hand, there are substantial deep and superficial differences between Scheme and C++ that students will have difficulty in transferring what is learnt in Scheme into general programming practice.

3 Recommendations

The committee has the following recommendations, addressing the above mentioned issues.

3.1 Recommended content

The broad goal of the course is: greater mastery of programming motivated by a rich range of programming problems drawn from diverse applications.

Here are some examples of specific programming style and techniques to study:

- Avoiding use of global variables, applicative programming, higher order functions, side effect free programming.
- Substantial practice with recursion, including structural recursion.
- Backtracking as a paradigm.
- Class design, inheritance and class heirarchies. Polymorphism.
- Design of embedded languages and general purpose libraries, in addition to individual programs.
- Reasoning about correctness. Invariants, preconditions, post conditions.

The programming problems should involve irregular and symbolic data, e.g. mathematical expressions. The course should address how to represent and manipulate such data. The focus should be on representation and correctness, and not algorithm discovery or analysis. Problems requiring some kind of an interpreter/embedded language should also be considred, so that the state needs to be maintained and the set of abstract operations on the state must be understood. Simulators, or even a library for implementing a small-scale relational database would be good examples. Another example is the design of a finite domain solver. Hierarchical composition of objects (e.g., a circuit component is composed of other circuit components) must be explored. Problems chosen must expose advanced programming techniques, their need and usage.

It should be noted that CS 152 is expected to be very rigorous in its treatment of programming. Arguing correctness is a significant part of it. However, its focus is not efficiency.

In many ways, this content is similar to the old content. However, it is suggested that logic programming topics such as unification be not covered. Such topics can be covered in later courses, e.g. Logic for CS, or Artificial Intelligence. In light of this recommendation, the name of the course could be changed to "Advanced Programming".

3.2 Language

It is recommended that the language used in CS 152 be C++. Note that the student has already learned C++ in CS 101, and will use C++ (or similar languages) in many other courses. Thus the advanced programming ideas learned in CS 152 will be assimilated better if the language is C++.

It could be argued that some programs are better written in other languages, whereas C++ introduces some syntactic clutter. This is true, but given that students already know C++, it may not produce too much cognitive load. Also, if you consider maintainability, readability, the additional verbiage as required by C++ might well be inevitable in other languages too.[1]

If some instructors strongly want to use other languages, it should be allowed (partly to increase the pool of willing instructors and allow experimentation), but they should note the above discussion. Thus they should talk about programming style and skill in a somewhat language independent manner.

3.3 Placement with respect to CS 213

It is recommended that the current positions of the two courses be retained, i.e. CS 152 in semester 2, and CS 213 in semester 3.

It is felt that CS 152 is closer in spirit to CS 101, with its emphasis on programming, rather than on algorithm discovery and its focus on representation and correctness, rather than on clever algorithmic techniques, their discovery and analysis. CS152 broadens the students' vistas, and improves

[1]It should be pointed out that languages such as Scheme were considered very elegant for writing programs 30-40 years ago. They still are, as compared to a language such as C. However modern languages such as C++ and Java have made great progress over C.

the maturity and programming ability. These aspects will serve the student well in taking CS 213 later.

3.4 Prerequisite structure

In principle, CS 152 could be considered to be a prerequisite for CS 213; however, this will prevent branch change students from taking CS 213, because CS 152 would be done in semester 2 before they arrive into the department. Since all students do CS 152 by semester 4 anyway, we could ignore this issue. Or alternatively, we could designate CS 152 as a prequisite for subsequent courses which are heavy on programming. For example, the courses on compilers or artificial intelligence.

3.5 Departmental introductory course

Because of its breadth, it is felt that perhaps CS 152 as recommended here will be a good departmental introductory course. It can, after all, have applications drawn from artificial intelligence, parsing, data bases, circuit design, and simulations (e.g. deadlocks).

This is not to deny that CS 213 is a more important course. However, perhaps CS 152 is better suited for the first year.

3.6 A pragmatic consideration

Pragmatically, note that CS 152 also tries to fill gaps left, if any, from the student's training in CS101 and bring everybody up to a level of programming proficiency that makes them well equipped to handle subsequent courses.

3.7 Consistency across offerings

Consistency across offerings is desirable, but some experimentation is also desirable to keep improving the course.

It is hoped that this document clarifies the spirit of the course and will thus serve as a guide in this respect.