Experiences in Teaching Introductory Programming inScheme at University College London

J. A. Campbell

Rapporteur: Marta Koutny

EXPERIENCE IN TEACHING OBJECT-ORIENTED COMPUTING CONCEPTS THROUGH SCHEME

J.A. Campbell

Department of Computer Science,
University College London, Gower Street, London WC1E 6BT

The Scheme language, derived from LISP, was adopted in the Department of Computer Science at University College London in 1986 for the initial teaching of programming to its first-year undergraduates. Part of the material covered deals with object-oriented computing. The general experience (positive) of this teaching during three years of existence of the course is described. Some comments are made about both the advantages and drawbacks of Scheme for teaching object-oriented computing, a topic for which it was not designed.

Background

The introductory programming course at UCL before 1986 for students specialising in computer science used a traditional approach supported by Pascal. In 1986 we changed much of the appearance of the course by choosing to base it on the Scheme language and the MIT Press text "Structure and Interpretation of Computer Programs" by H. Abelson and G. Sussman. The course has now been given for 3 years, and has an audience (including students in Cognitive Science and in a Physics with Electronics and Computing programme) with a wider variety of interests than the one for which it was planned.

On paper the introductory course is given at a rate of 3 lectures per week throughout the academic year. In practice it is divided into 2 parts, with Scheme as the language of the first part and C++ as the language of the second. The original dissatisfaction with Pascal as a teaching language in our case was expressed by enthusiasts for functional programming as a better educational approach. In 1986 no suitable implementation or textbook was available, but Scheme was seen as a suitable non-Pascal-like alternative. (Discussion about the functional or non-functional nature of Scheme, a close relative of LISP, was then short-circuited by my arrival at UCL and expression of a strong taste for presenting a first course in Scheme). At the same time, it was generally agreed that some part of the first-year introduction to programming should include traditional procedural work in a language that would appear in later courses, e.g. on systems software, operating systems and software engineering. As the language that was used already in these courses was C, and as it was also regarded as very desirable to give students a reasonable exposure to the ideas of object-oriented programming (e.g. in connection with modularity and reuse of software, in software engineering), C++ was selected as the language for the second half of the course. Some of the supporters of C++ probably regarded the introduction to object-oriented

ideas in chapter 3.1 of Abelson and Sussman as an unexpected bonus. In the actual teaching it has turned out that this material is an effective overall introduction to object-oriented programming, which dovetails well into the later treatment of the subject in the teaching of C++.

Facilities

The main programming system that is available for students is the MIT implementation "C Scheme", running on a large Pyramid 98X computer. This implementation is robust and able to accommodate peak loads (25 to 28 simultaneous users) without difficulty once the random-access memory available to each user is set at about 1 MB instead of the default value of 4 MB. Even so, users of other software have noticed enough of a slowing-down of their service at peak hours to cause us to move to new arrangements in 1988. In these arrangements, students use the same basic terminals as before, but their jobs are now run on a network of Sun workstations through an assignment mechanism that ensures that no more than 2 Scheme jobs are placed on any one workstation. This has been fully adequate for the course, even though the overall enrolment has grown to 63 and access to the network of Suns is possible from remote locations (e.g. halls of residence) as well as from terminal rooms in the Department. The excellent support of our software and systems group is responsible for the trouble-free service that we have experienced.

In addition to the official provisions, some copies of PC Scheme (a Texas Instruments product) for IBM-compatible microcomputers are in use. The US price is low enough (about $99) to have made individual purchase attractive.

Some comparisons of Scheme with LISP

Scheme is basically LISP with somewhat more natural conventions for writing and processing several of its basic operations, plus the possibility of defining procedures that return procedural values. These higher-order procedures are useful in connection with the concept of message-passing in object-oriented computing, although that is not their main selling-point.

A function (= "procedure" in Scheme: "function" in the Scheme textbook has just its mathematical meaning) to make a copy of a list x in Common LISP is defined via

```
(defun copy (x) (cond ((null x) nil)
    (t (cons (car x) (copy (cdr x)) )) ))   [1]
```

where cons is the basic list-constructor, and car and cdr access respectively the head and the tail of the structure denoted by their argument. The corresponding definition in Scheme, taking advantage of the simplified conditional expression that can be written when only a two-way choice is involved, is

```
(define (copy x) (if (null? x) nil
    (cons (car x) (copy (cdr x)) )))        [2]
```

This example is structurally typical of many definitions in first-course LISP or Scheme. It draws attention to an

amusing feature of programming with this family of languages. The Common LISP definition [1] ends with 6 right brackets. The input conventions of the earliest LISPs would have required 7 for this function. The Scheme definition has 5. All of these numbers are the most likely ones to occur at the ends of simple programming exercises in their respective languages. In the early days of LISP, miscounting of brackets at the ends of function definitions was the most widespread programming error, and the greatest barrier to students' desire to persevere with learning the language. The limited evidence available on teaching Common LISP as a first language suggests that users still complain about right brackets, although much less loudly. Our UCL experience with Scheme is that the miscounting of terminating brackets is just one possible novice error among several, and that right brackets in Scheme do not cause any significant alarm and despondency among students. There is probably a paper on the adaptation of language design to the cognitive make-up of programmers in here somewhere!

In LISP, the function defun in [1] is reserved for defining functions. In Scheme, the use of define in [2] is not unique: the same procedure is used to attach values to variables, as in

```
(define pi 3.141592)                .          [3]
```

LISP would put setq in place of define in [3], i.e. defining functions/procedures and defining or updating variables are kept separate conceptually. It is a defect of Scheme for teaching, especially on object-oriented computing, that this aspect of LISP is missing.

The simplest example and justification of higher-order procedures in the Scheme textbook is the definition [4]. This procedure computes sums of the form $f(a) + f(a_1) + \ldots + f(b)$, where f is a function represented by term , and where the computation of the next argument (e.g. a_1) from the previous one (e.g. a) in the sequence is carried out by next .

```
(define (sum-series term a b next)
    (if (> a b)  0
      (+ (term a) (sum-series
        term (next a) b next)) ))                [4]
```

Thus, if a procedure (cube x) is defined as (* x x x), a call to (sum-series cube 1 10 1+) causes computation of the sum of the cubes of integers from 1 to 10.

During treatment of the basics of Scheme and of the modern Good Things of education in programming (top-down design, modularity, reuse of software, functional style (almost), referential transparency ...), the first 2 chapters of the textbook introduce examples that have the possibility of teaching some lessons about object-oriented programming. One of them is actually treated as an example about something else, but tends to be noticed by the more wide-awake students as an example that is "special" in a sense for which they have difficulty in finding a phrase. The pleasure that they get when "message-passing" is suggested to fill the gap is very noticeable.

The example was devised to show that cons , car and

cdr are not necessarily fundamental procedures implemented in only one way, which is undefinable in Scheme or LISP. It makes explicit definitions, relying on the existence of the higher-order procedural feature of Scheme.

```
(define (cons x y)
    (define (dispatch m)
      (cond ((eq? m 'car) x)
            ((eq? m 'cdr) y)
            (else (error ..... )) ))
      dispatch)                                    [5]

(define (car z) (z 'car))
(define (cdr z) (z 'cdr))                          [6]
```

Suppose that we set up a test case as follows:

```
(define test (cons 'no 'surrender))
```

Because of [5], test has a procedural value, associated with the 1-argument procedure dispatch . When (cdr test) is evaluated, [6] indicates that test is sent a message that can be read as "your cdr, please", the atom cdr is substituted for m in [5], and the correct answer surrender is obtained.

Objects in Scheme

The standard textbook example here is that of a bank-account object. After refinement of the initial versions of the example, the complete definition is

```
(define (make-account balance)
    (define (withdraw amount)
      (if (>= balance amount)
          (sequence
            (set! balance (- balance amount))
            balance)
          "Insufficient funds"))
    (define (deposit amount)
      (set! balance (+ balance amount))
      balance)
    (define (dispatch m)
      (cond ((eq? m 'withdraw) withdraw)
            ((eq? m 'deposit) deposit)
            (else (error m "not recognised")) ))
    dispatch)                                      [7]
```

Here, make-account is a procedure-valued procedure serving as a general bank-account object, with withdraw and deposit capabilities and with a single local variable, which records a current balance. A new named account which inherits the capabilities and contains its own private "balance" variable can be set up by a suitable call to make-account, e.g.

```
(define fred-student-account (make-account 10)) ,
                                                   [8]
```

which creates an account for fred and awards him a British bank's traditional starting gift of 10 units of currency. Possible future transactions may be

```
((fred-student-account 'deposit) 50)    ,
```

which brings the balance up to 60,

```
((fred-student-account 'withdraw) 20)    ,
```

which would then reduce it to 40,

and

```
((fred-student-account 'withdraw) 100)  ,        [9]
```

which would generate the appropriate message and not change the balance. As in [6], transactions can be read as being initiated by the sending of messages to objects

This presentation has a strong positive effect on the audience. Students remember both the message-passing paradigm and the idea of a secret private history and attitude to the external world (including non-cooperation, as in [9]) and try to use them thereafter. The paradigm obviously has an appeal (to a London audience, at least) that is not yet exploited in textbooks for beginners. Object-oriented questions set as options in examinations are quite heavily favoured, and the structure of material such as in [7] is usually well reproduced.

Specific evidence for the effectiveness of the object-oriented approach is that

- students enjoy the exercise of extending the bank-account example to include passwords (with some designs being quite ingenious, not to say Byzantine);
- there is a high rate of success in answering questions that hint at extension of the "private local variable" idea for their answers, e.g. writing of a procedure that looks up rather than calculates the square root of n if it has been asked to find the square root of the same n previously.

Some drawbacks

The example [7] contains many of the desirable features of an object-oriented definition, even though Scheme was not planned for such a purpose. This positive outcome can be credited to the fact that Scheme was designed to embody (and even enforce) aspects of programming practice that are understood widely to be good. This appreciation was not so widespread (particularly for teaching) around 1974-75, when the present form of Scheme was being evolved.

[7] also indicates some sources of confusion for students. One of these has been mentioned already, together with [3]: the use of `define` for two different purposes In [7] the local variable is updated with `set!` , but some students continue to use `define` and are puzzled when they are told that this is not good behaviour. These students would prefer the origin of the confusion (the dual use of `define`) to be illegal in the language Some students say this as soon as the dual use is shown to be possible, i.e. well before any hint of object-oriented computing appears in the course.

[7] demonstrates a further drawback from the points of view of both teaching and aesthetics. This is that, even when `define` is limited to the definition of procedures it still has two rather different interpretations. The first is the expected one, which is shown clearly in [2]

and [4]-[6]. The second occurs in [7] when any procedure except make-account is defined. In the first case, a defined procedure is later accessible everywhere. In the second, when the object-oriented paradigm is taken seriously, capabilities like deposit and withdraw are intended to be local to their parent object and irrelevant outside it, but students infer quite reasonably that syntactic context alone is not enough to distinguish the two in general, and comment that there ought to be two separate "define"s (one for local and one for global use, so that the programmer can always show clearly what the intention is in marginal cases - and at least one student has called [5] and [6] a marginal case).

A final drawback for general teaching of object-oriented computing is that Scheme allows nothing more (and hence nothing more flexible) than the inheritance mechanism used in [8]. Therefore there is not even as much scope for discussion of inheritance as a unifying idea as there would be if one taught with the help of a classical simulation language from the 1960s, such as Simula. This would have been a problem if object-oriented teaching had had to continue in Scheme, but (as indicated above) this has not been the pattern followed at UCL.

General consequences

For students, the main effect of the change from Pascal to Scheme has been a much earlier introduction to a modern range of concerns about (and models for) computing in practice, with a minimum of diversion of attention through a need to think about the details of the scaffolding (i.e. the language in which the practice is given). These concerns certainly include the essentials of object-oriented computing, which are built on in the continuation of the course via the use of C++. It is fair to say that C++ is full of scaffolding, especially because the first books available as reading material were not textbooks for beginners. However, the grounding given in the essentials of computing through Scheme has meant that the sudden change of environment to C++ has not been intimidating for the students.

A consequence of the increase in ground that the new first programming course has been able to cover is that the students are better prepared to deal with information in later courses that have come to rely on it. This is so for the later courses in systems software, software engineering and artificial intelligence (disguised as "expert systems"). In addition, the audience trained on Scheme has forced the pace in a new second-year course entitled Programming Paradigms (involving concurrency, programming with logic, and constraint-based programming). A new result visible for the first time in 1988-89 is that veterans of the Scheme course have chosen some interesting and demanding final-year projects, e.g. on exchange of knowledge between different representations, which build directly on their experience of Scheme.

For the academic staff, one observed result has been happier lecturers in the second- and third-year courses just mentioned. Apart from courses, it is possible to

say that some projects have more of a research flavour than would have been the case before 1986. In at least 2 examples, completion of the projects (which use educational foundations from the Scheme course) should generate material immediately for publishable papers.

A success story is possibly a little suspicious unless it hints at some lack of success along with its positive results. If so, then Scheme is above suspicion here. The pressure in favour of a first course in a functional programming language at UCL has only been dormant, not abolished, since 1986. It has been revived lately because a textbook and an apparently student-proof implementation of one functional language are now available. This revival is almost independent of the merits or demerits of Scheme, and entirely independently of the merits or demerits of teaching first-year undergraduates about object-oriented computing. But many good things, like the pre-war Estonian Republic, are incidental casualties of vast historical movements. In that particular case, its merits are being rediscovered 48 years later under the influence of perestroika. It is reasonable to suppose that, if Scheme disappears temporarily from the UCL map, a corresponding educational perestroika may arrive after an order of magnitude less time.

DISCUSSION

After the talk Professor Campbell was asked for further details about the way in which the assessment of students is carried out, and if there is any additional support to the lectures. The speaker replied that the majority of assessment is by a traditional examination method, and that there is a tendency to increase the proportion of (interesting) programming exercises in the examination process. As the most important additional support Professor Campbell mentioned quite frequent tutorial meetings.

In response to Dr. Wolczko's question, Professor Campbell explained that his Department runs a course on pure object oriented language (Smalltalk) and shortly described its organisation.

Dr. Schäffert asked whether the lectures cover continuations, and how the students react to this. Professor Campbell responded that continuations surface in the context of a subsequent C + + course rather than in the teaching of Scheme, and that the general reaction of students was positive.