A critique of Abelson and Sussman
- or -
Why calculating is better than scheming

Received April 1986

Philip Wadler
Programming Research Group
11 Keble Road
Oxford, OX1 3QD

Abelson and Sussman have written an excellent textbook which may start a revolution in the way programming is taught [Abelson and Sussman 1985a, b]. Instead of emphasizing a particular programming language, they emphasize standard engineering techniques as they apply to programming. Still, their textbook is intimately tied to the Scheme dialect of Lisp. I believe that the same approach used in their text, if applied to a language such as KRC or Miranda, would result in an even better introduction to programming as an engineering discipline. My belief has strengthened as my experience in teaching with Scheme and with KRC has increased.

This paper contrasts teaching in Scheme to teaching in KRC and Miranda, particularly with reference to Abelson and Sussman's text. Scheme is a lexically-scoped dialect of Lisp [Steele and Sussman 1978]; languages in a similar style are T [Rees and Adams 1982] and Common Lisp [Steele 1982]. KRC is a functional language in an equational style [Turner 1981]; its successor is Miranda [Turner 1985]; languages in a similar style are SASL [Turner 1976, Richards 1984] LML [Augustsson 1984], and Orwell [Wadler 1984b]. (Only readers who know that KRC stands for "Kent Recursive Calculator" will have understood the title of this paper.)

There are four language features absent in Scheme and present in KRC/Miranda that are important:

1. Pattern-matching.
2. A syntax close to traditional mathematical notation.
3. A static type discipline and user-defined types.
4. Lazy evaluation.

KRC and SASL do not have a type discipline, so point 3 applies only to Miranda,

LML, and Orwell.

This paper should be thought of as a discussion of the relative importance of these language features, rather than just of the relative merits of two different families of languages. However, for convenience this paper will sometimes use the names "Lisp" and "Miranda" to distinguish the two families. (In an earlier version of this paper, the name "functional" was used to characterize the Miranda family, but only point 4 is necessarily connected to the fact that these languages are functional.)

This paper is based largely on my experience over two years with two different courses: a course for undergraduates based on Abelson and Sussman's text, taught by Joe Stoy using Scheme (actually, T modified to look like Scheme); and a course for M.Sc. students in functional programming, taught by Richard Bird using KRC.

This paper is organized as follows. Section 1 discusses data types. Section 2 discusses confusion between program and data. Section 3 discusses programs that manipulate programs. Section 4 discusses lazy evaluation. Section 5 presents conclusions.

1. Data types

1.1. Lists

Many years ago, Peter Landin formulated an excellent way to describe data types [Landin 1966]. Here is a description of lists using that method:

> An *A-List* is
> either *nil*,
> or a *cons*, and has
> a *head*, which is an A
> and a *tail*, which is an A-List

From this type description, one can immediately see the structure of a program that operates on lists. For example, here is a Miranda program to sum a list of numbers:

```
sum []    =  0
sum (x:xs)  =  x + sum xs
```

There are two clauses in this definition, one for each clause (nil and cons) in the

definition of A-List. (In Miranda nil is written `[]`, and a cons with head `x` and tail `xs` is written `x:xs`.)

Here is the same definition in Lisp:

```
(define (sum xs)
  (if (null? xs)
      0
      (+ (car xs) (sum (cdr xs)))))
```

This definition is just plain more cumbersome to read, even though it has essentially the same structure as the functional definition. The primary problem is the lack of pattern-matching. The definition is also harder to read because of the syntax (or, rather, lack of syntax) of Lisp.

Furthermore, the Lisp program obscures the symmetry between the two cases. The nil case is tested for explicitly, and the cons case is assumed otherwise. The symmetry can be recovered by writing:

```
(define (sum xs)
  (cond ((null? xs) 0)
        ((pair? xs) (+ (car xs) (sum (cdr xs))))))
```

This program is perhaps more cumbersome than the preceding one. It is also a bit less efficient, as it may perform two tests instead of one. On the other hand, there are well-known ways to compile pattern-matching efficiently [Augustsson 85].

In Miranda, the type discipline requires that `sum` is only applied to lists of numbers. Since Miranda uses a type inference system the user may give the type of `sum` explicitly, or leave it to be inferred. In other words, a type inference system means that typing (of data) need not involve extra typing (with fingers).

The type discipline is important for two related reasons. First, it is an important tool for *thinking* about functions and function definitions. Second, the language system can ensure that certain kinds of errors do not occur at run-time. Since a large proportion of a beginner's -- or even an experienced programmer's -- errors are type errors, these are important advantages.

1.2 Proving properties of programs

Say we wish to prove that append is associative. Here is the definition of append (written ++) in Miranda style:

```
[] ++ ys        =  ys                    (1)
(x:xs) ++ ys    =  x:(xs ++ ys)          (2)
```

We wish to prove that for all lists xs, ys, and zs:

```
(xs ++ ys) ++ zs  =  xs ++ (ys ++ zs)
```

The proof is by structural induction on xs [Burstall 69].

Base case. Replace xs by [].

```
([] ++ ys) ++ zs
          =  ys ++ zs              -- unfolding by (1)
          =  [] ++ (ys ++ zs)      -- folding by (1)
```

Inductive case. Replace xs by x:xs.

```
((x:xs) ++ ys) ++ zs
          =  (x:(xs ++ ys)) ++ zs  -- unfolding by (2)
          =  x:((xs ++ ys) ++ zs)  -- unfolding by (2)
          =  x:(xs ++ (ys ++ zs))  -- induction hypothesis
          =  (x:xs) ++ (ys ++ zs)  -- folding by (2)
```

This completes the proof.

Now, here is the definition of append in Lisp:

```
(define (append xs ys)
  (if (null? xs)
      ys
      (cons (car xs) (append (cdr xs) ys))))
```

And here is the equation to be proved:

```
(append (append xs ys) zs)  =  (append xs (append ys zs))
```

It is left as an exercise to the reader to write the proof in terms of Lisp. Attention is drawn to two particular difficulties.

First, in the Miranda-style proof, folding and unfolding can be explained as a simple matter of substituting equals for equals. An equivalent to the unfold operation in Lisp requires expanding the definition and then simplifying. For example, the first step in the base case, corresponding to unfolding by (1), is as follows:

```
(append nil (append ys zs))
        = (if (null? nil)
              (append ys zs)
              (cons (car nil)
                    (append (cdr nil) (append ys zs))))
        = (append ys zs)
```

The folding operation is even more problematic. The lesson here is that pattern-matching notation greatly simplifies the mechanics of the proof.

Second, each step in the Miranda-style proof simply involves a rearrangement of parentheses. Although the terms represented by Lisp are the same, the prefix notation means more mental effort is needed for each step. This effect is strongest for the associative law, but in general any algebraic manipulation is easier in an infix notation; this is one reason such notations have evolved.

For these reasons, writing the proof in full is considerably more difficult in Lisp than Miranda. This is a serious impediment when teaching even simple proof methods to students. I and several of my colleagues, when faced with this problem, decided it was easier to teach our students a Miranda-like notation and then do the proof, rather than try to do the proof directly in Lisp. Teaching a Miranda-like notation first usually can be done quickly and informally, because the notation is quite natural.

Some people may wish to dismiss many of the issues raised in this paper as being "just syntax". It is true that much debate over syntax is of little value. But it is also true that a good choice of notation can greatly aid learning and thought, and a poor choice can hinder it. In particular, pattern-matching seems to aid thought about case analysis, making it easier to construct programs and to prove their properties by structural induction. Also, mathematical notation is easier to manipulate algebraically than Lisp.

1.3. Mobiles

Here is part of exercise 2-27 from Abelson and Sussman:

A binary mobile consists of two branches, a left-branch and a right-branch. Each branch is a rod of a certain length, from which hangs either a weight or another binary mobile. We can represent a binary mobile using compound data by constructing it from two branches (for example, using `list`):

```
(define (make-mobile left right)
  (list left right))
```

A branch is constructed from a length (which must be a number) and a supported-structure, which may be either a number (representing a simple weight) or another mobile:

```
(define (make-branch length structure)
  (list length structure)
```

a. Supply the corresponding selectors `left-branch` and `right-branch`, which return the branches of a mobile, and `branch-length` and `branch-structure`, which return the components of a branch.

b. Using your selectors, define a procedure `total-weight` that returns the total weight of a mobile.

The answer is easy for an experienced programmer to find:

```
(define (left-branch struct) (car struct))
(define (right-branch struct) (cadr struct))
(define (branch-length branch) (car branch))
(define (branch-structure branch) (cadr branch))

(define (total-weight struct)
  (if (atom? struct)
      struct
      (+ (total-weight-branch (left-branch struct))
         (total-weight-branch (right-branch struct)))))

(define (total-weight-branch branch)
  (total-weight (branch-structure branch)))
```

Unfortunately, the answer is not so easy for a novice programmer to find. This is because the question, although it is carefully worded, almost ignores a very important aspect of the data structure -- namely, the base case, the degenerate

mobile (or "structure") consisting of a single weight. Indeed, the question practically misleads the student, because a careful distinction is made between "mobiles" and "structures", and the question asks for a function to find the total weight of a "mobile" rather than a "structure".

In a language with user-defined types, the first step to solving this problem is to write down an appropriate type declaration. This leads one immediately to perceive the importance of the base case. Here are the appropriate declarations in Miranda:

```
structure  ::=  Weight num  |  Mobile branch branch
branch     ::=  Branch num structure
```

The total weight function can then be written in a straightforward way, using the type declarations as a guide.

```
totalWeight (Weight w)  =  w
totalWeight (Mobile l r)
    =  totalWeightBranch l + totalWeightBranch r

totalWeightBranch (Branch d s)  =  totalWeight s
```

The Miranda program reflects the type structure in a more straightforward way than the Lisp program, and it is also easier to read. Furthermore, the selector functions are not needed at all.

1.4. Data representation and abstract data types

The mobile problem continues as follows:

d. Suppose we change the representation of mobiles so that the constructors are now

```
(define (make-mobile left right)
  (cons left right))

(define (make-branch length structure)
  (cons length structure))
```

How much of your program do you need to change to convert to the new representation?

The answer for Lisp is that we just need to change the selector functions `right-branch` and `branch-structure` to use `cdr` instead of `cadr`. The answer for Miranda is that the question makes no sense, because there are no selector functions. This points out some advantages, and also a disadvantage, of using Miranda to teach issues of data representation.

The first advantage is that for certain data types, namely the *free* data types, Miranda allows one to write programs at a higher level of abstraction, where the choice of representation is not important. A data type is free if two objects of the type are equal if and only if they are constructed in the same way [Burstall and Goguen 1982]. Lists and mobiles are both free data types. For example, lists are free because `x:xs = y:ys` if and only if `x=y` and `xs=ys`. An example of a non-free data type is a set, because `{x} U xs = {y} U ys` does not imply `x=y` and `xs=ys`.

In Lisp there is essentially only one free data type, S-expressions. If the user wants some other free data type -- say, lists or mobiles -- then he or she must choose a representation of that type in terms of S-expressions. As we saw in the mobile example, there may be more than one way to make that choice. In Miranda the user may declare a new free data type directly. There is no need to choose an arbitrary representation. Thus, for the mobile problem above, the question of changing representation is irrelevant, because one can phrase the solution at a higher level of abstraction.

The second advantage is that where choosing a representation is important, Miranda provides a language feature -- abstract data types -- to support seperating use of a type from its choice of representation. For example, Abelson and Sussman discuss several different ways of representing sets. In software engineering, the classical method for abstracting away from an arbitrary choice of representation is the abstract data type. Although they discuss data abstraction at length, Abelson and Sussman do not mention abstract data types *per se*, because Lisp does not contain suitable hiding mechanisms. Languages such as Miranda and LML do support the classical abstract data type mechanism, and so are perhaps better suited for teaching this topic.

The disadvantage is that pattern-matching, which is so useful, cannot be used with abstract data types. Solutions to this problem are on the horizon. One possibility is algebraic types with laws in Miranda, which allow pattern-matching to be used with some non-free types. Another is Views [Wadler 85a], a language feature which allows one to use pattern-matching with any arbitrary representation. Lisp doesn't have this problem, but only because it throws out the baby with the bathwater -- it

doesn't have pattern matching at all.

1.5. A last word on the mobile exercise

Finally, a minor point. The mobile exercise above is not really a good model for teaching students about change of representation. The problem is that although the representation of mobiles and branches is hidden by the selector functions, the representation of a single weight is not. This is not a problem with Lisp, as it is easy to add the necessary constructors and selectors:

```
(define (make-weight weight) weight)
(define (weight? struct) (atom? struct))
(define (weight struct) struct)
```

The modified definition of `total-weight` is then:

```
(define (total-weight struct)
  (if (weight? struct)
      (weight struct)
      (+ (total-weight-branch (left-branch struct))
         (total-weight-branch (right-branch struct)))))
```

Perhaps this was an oversight on Abelson and Sussman's part, or perhaps they did not wish to complicate the problem further. The same problem is unlikely to arise in a language with user-defined data types, because, as we have seen, these lead one to a solution that treats weights in a properly abstract way.

2. Confusion between program and data

An important feature of Lisp is that program and data have the same representation, namely S-expressions, and that a special form, "quote", is available to turn programs into data. This makes possible a convenient style for writing programs that manipulate programs, such as interpreters. It also makes Lisp an easy language to extend.

On the other hand, it also makes it easy for a new student to become confused about the relationship between program and data. This section describes several such confusions, which I have seen in many students during tutorial sessions. Further, even when it comes to writing programs that manipulate programs, although Lisp

has some advantages, so does Miranda. This is discussed in the following section. My conclusion is that the disadvantages of having program and data in the same form outweigh the advantages, especially for beginning students.

2.1. Lisp lists are not self-quoting

In Lisp, numbers as data are self-quoting, whereas lists are not. For example, to include the number 3 as a datum in a program one just writes 3, whereas to include the list (1 2 3) as a datum one must write (quote (1 2 3)) (which is often abbreviated as '(1 2 3)).

The difference between (1 2 3) and (quote (1 2 3)) is subtle, and it inevitably confuses students. In particular, it plays havoc with the substitution model of evaluation. For example, one can use the substitution model to explain the evaluation of (* (+ 3 4) 6) as follows:

```
(* (+ 3 4) 6)   --->   (* 7 6)   --->   42
```

All three steps of this derivation ((* (+ 3 4) 6), (* 7 6), 42) are themselves legal Lisp expressions.

Now, consider using the substitution model to explain the evaluation of the term (list (list 1 2) nil):

```
(list (list 1 2) nil)
    ---> (list (1 2) nil)
    ---> (list (1 2) ())
    ---> ((1 2) ())
```

The intermediate steps are no longer legal Lisp expressions. One must keep track of which parts of the expression have been evaluated, and which have not. One could get around this by writing:

```
(list (list 1 2) nil)
    ---> (list '(1 2) nil)
    ---> (list '(1 2) '())
    ---> '((1 2) ())
```

But I find this tricky to explain.

In Miranda, on the other hand, one just writes `[[1,2], []]`. There isn't any evaluation to explain! When there is evaluation, it can be explained by the substitution model:

```
[7*6, 5*9]   --->  [42, 5*9]   --->  [42, 54]
```

Each step of the derivation is a legal Miranda expression.

The point of this is not that evaluation of `(list (list 1 2) nil)` cannot be explained. Of course it can. But it takes much more work to explain it than to explain the Miranda expression `[[1,2], []]`. In this case, perhaps one can afford the extra effort. But the problem is greatly compounded when one must explain this sort of thing in the middle of some other derivation. I have encountered this sort of problem many times.

2.2. Further confusion with quote

Here is exercise 2-30 from Abelson and Sussman:

> Eva Lu Ator types to the interpreter the expression
>
> ```
> (car ''abracadabra)
> ```
>
> To her surprise, the interpreter prints back quote. Explain. What would be printed in response to
>
> ```
> (cdddr '(this list contains '(a quote)))
> ```

The answer to the first part is that `(car ''abracadabra)` is equivalent to `(car (quote (quote abracadabra)))`, and so one has the following evaluation:

```
(car (quote (quote abracadabra)))
    --->  (car (quote abracadabra)
    --->  quote
```

Here the need to keep track of what has and has not been evaluated is unavoidable.

The answer to the second part is that the Lisp input system transforms the given expression to:

```
(cdddr (quote (this list contains (quote (a quote)))))
```

and this evaluates to `((quote (a quote)))`.

All this is quite obscure. Indeed, the equivalence of `'a` and `(quote a)` is only explained in a footnote. But if one is to fully understand Lisp, one must understand this sort of thing. In languages without quote, this sort of problem simply does not arise.

2.3. Evaluating too little or too much

Because program and data have the same form in Lisp, the form alone does not tell one whether one is dealing with program or data. As pointed out above, one has to remember this information when applying the substitution model. It is easy for students to forget this information, and evaluate either too little or too much.

What is the value of this expression?

```
(car (quote (a b)))
```

The right answer is, of course, `a`. However, I have seen students give the answer `quote`, which results from evaluating too little. This error is particularly common after they have done the abracadabra exercise above.

I have also seen students give the answer "the value of the variable `a`", which results from evaluating too much. That is, the student first evaluates far enough to get the right answer, `a`, and then evaluates one step more, returning whatever value the variable `a` is bound to. Students can make this error even when no variable named `a` has been mentioned in connection with the problem.

Again, this sort of problem does not arise in a language without quote.

2.4. Other confusions with lists

There are two other confusions I have seen in students with regard to the list data type. These confusions are inherent to the list data type itself, and appear when teaching either Lisp or Miranda. However, I believe the problems are harder to rectify in Lisp, because the student is already suffering from a confusion between

data and program.

The first problem is that students inevitably confuse a list containing just one element `x` with the value `x` itself. The confusion is compounded in Lisp because this unit list is written either `(list x)` or `(x)`, depending on whether it has been evaluated yet. In Miranda one always writes `[x]`. The concrete syntax of Miranda also helps a bit: students are used to dropping parenthesis, so that `(x)` becomes `x`, whereas they are a little less likely to convert `[x]` to `x`. Also, in a typed language like Miranda this problem becomes much easier to explain, because if `x` has type `t` then `[x]` has type list of `t`; and errors will be caught by the type-checker.

The second problem is that students get confused between `cons` and `list`. Again, this problem is increased in Lisp, because `(cons x y)` and `(list x y)` look so similar, whereas `x:y` and `[x,y]` look rather different. And again, the problem is easier to explain and detect in a typed language.

2.5. Syntax

Finally, it is hard not to say something about the famous Lisp S-expression syntax. There are strong advantages to the Lisp syntax. It is easy to learn, and it gives the students a good appreciation of the underlying abstract structure.

On the other hand, as seen above, Lisp programs often have much more sheer bulk than the corresponding Miranda programs. Also, as noted above, S-expression notation hinders reasoning with algebraic properties, such as associativity. Perhaps most important, the unfamiliarity of Lisp syntax can be a real stumbling block to beginning students.

I remember giving a small group of students a demonstration, attempting to convince them of the great power and sheer fun of using Lisp. After explaining that `3 + 4` was typed as `(+ 3 4)`, I went on to type some larger expressions. One of these was `((+ 3 4) = (+ 5 2))`, which naturally caused the interpreter to complain. I quickly figured out why, but I had lost much ground in trying to convince the students how "natural" the S-expression syntax was. If I make such mistakes as an experienced Lisp programmer, I wonder how much trouble they cause beginning students?

3. Programs that manipulate programs

Lisp is famous for the ease with which one can construct programs that manipulate programs, such as interpreters, compilers, and program-transformation systems. However, Miranda also has advantages, complementary to those of Lisp, for constructing such programs. This section compares Lisp and Miranda styles for writing such programs.

3.1. A simple interpreter in Miranda and Lisp

As a simple example of an interpreter, let's consider an evaluator for terms in the lambda calculus. There are three kinds of terms: variables, lambda abstractions, and applications. In addition, there is one more kind of term, a closure, that will be used internally by the evaluator.

The evaluator consists of two mutually recursive functions. A call `(eval e t)` evaluates the (non-closure) term t_0 in environment `e`. A call `(apply t0 t1)` evaluates the application of term t_1 (which must be a closure) to term `t1`. Miranda and Lisp versions of the evaluator are shown in figures 1 and 2. The data structures used in these programs are explained in more detail below.

Clearly, it is possible to write program manipulating programs in Miranda as well as Lisp, even though in Miranda there is no mechanism analogous to quote. Lisp and Miranda have complementary strengths when writing such programs, related to their different treatment of data types.

3.2. Representing programs with free data types

In Miranda, the data type for terms is described as follows:

```
term  ::=   Var var
          | Lambda var term
          | Apply term term
          | Closure env var term

env   ==  [(var, term)]
var   ==  [char]
```

The term `Closure e v t` represents the closure in environment `e` of a lambda term with bound variable `v` and body `t`. The last two lines say that an environment is represented by a list of (variable, term) pairs, and that a variable name is

represented by a list of characters.

In this representation, the term, say,

```
(λx.(x x)) (λx.(x x))
```

would be represented by

```
(Apply (Lambda "x" (Apply (Var "x") (Var "x")))
       (Lambda "x" (Apply (Var "x") (Var "x")))) .
```

The advantage of Miranda is that the type declaration for terms is concise and informative, and pattern-matching makes the program easier to write and read. The disadvantage is that the notation for programs-as-data (like the term above) is cumbersome. In short, manipulating the data is easy, but writing the data to be manipulated is hard.

Experience in teaching with KRC and Orwell has shown that, although cumbersome, the notation above is usable in practice for small to medium sized examples. Often one can lessen the problems by introducing a few extra definitions to make the data easier to enter. For example, one might define

```
lambda (var v) t  =  Lambda v t
app t0 t1  =  Apply t0 t1
x  =  Var "x"
```

and then write

```
app (lambda x (app x x)) (lambda x (app x x))
```

which is tolerable, if less than elegant.

A better approach would be to write parsers and unparsers, to convert between a convenient notation for reading and writing programs-as-data and a convenient notation for manipulating them. This clearly requires more work, but also yields more benefit. Parsers and unparsers are interesting subjects in their own right, and are important in most practical systems that treat programs as data. One interesting approach to writing parsers in a functional language is discussed in [Wadler 1985c].

3.3. Representing programs with almost abstract syntax

In Lisp, there are various choices for how one may represent terms. One common choice would be:

```
v                       -- a variable
(lambda (v) t)          -- a lambda abstraction
(t0 t1)                 -- an application
(closure e v t)         -- a closure
```

In this representation, the term above would be written:

```
'((lambda (x) (x x)) (lambda (x) (x x)))
```

which is far less cumbersome than the corresponding Miranda expression.

This representation is typically called an *abstract syntax*, but a more appropriate name might be *almost abstract syntax*. A true abstract syntax would be:

```
(var v)                 -- a variable
(lambda v t)            -- a lambda abstraction
(apply t0 t1)           -- an application
(closure e v t)         -- a closure
```

And now one would write:

```
'(apply (lambda x (apply (var x) (var x))
        (lambda x (apply (var x) (var x)))
```

which is as cumbersome as Miranda.

The key idea behind almost abstract syntax is that if convenient notation is provided for a few common kinds of data (in this case, variables and applications), then full abstract syntax for everything else is tolerable. Representations based on the "almost abstract syntax" principle are common in Lisp. For example, the representation of mobiles discussed in section 1 uses this principle, where a special notation is provided for weights, but branches and mobiles use fully abstract notation.

Manipulating programs-as-data seems easier in Miranda, but entering the data itself is easier in Lisp. To a large extent, this seems to be because of the "almost abstract syntax" principle, rather than because programs and data have the same form in

Lisp. One wonders if this principle could be added to a Miranda-style language in some way, perhaps by adding a special notation for some data items?

Figure 1: *Lambda term evaluator in Miranda*

```
|| data types

term  ::=    Var var
           | Lambda var term
           | Apply term term
           | Closure env var term

env  ==  [(var,term)]
var  ==  [char]


|| evaluate and apply

eval e (Var v)  =  lookup e v
eval e (Lambda v t)  =  Closure e v t
eval e (Apply t0 t1)  =  apply (eval e t0) (eval e t1)

apply (Closure e v t0) t1  =  eval (extend e v t1) t0


|| environment manipulation

lookup ((v0,t):e) v1  =  t,             if (v0 = v1)
                      =  lookup e v1,   otherwise

extend e v t  =  (v,t):e
empty  =  []
```

Figure 2: *Lambda term evaluator in Lisp*

```
;; evaluate term t in environment e

(define (eval e t)
  (cond ((variable? t)
         (lookup e (variable-name t)))
        ((lambda? t)
         (make-closure e (lambda-var t) (lambda-body t)))
        ((apply? t)
         (apply (eval e (apply-operator t))
                (eval e (apply-operand t))))))


;; apply term t0 to term t1

(define (apply t0 t1)
  (cond ((closure? t0)
         (eval (extend (closure-env t0) (closure-var t0) t1)
               (closure-body t0)))))


;; environment manipulation

(define (lookup v e)
  (cond ((pair? e)
         (if (eq? v (caar e)) (cadr e) (lookup v (cdr e))))))

(define (extend e v t) (cons (cons v t) e))
(define empty nil)


;; create and access terms

(define (make-var v) v)
(define (variable? t) (atom? t))
(define (variable-name t) t)

(define (make-lambda v t) (list 'lambda (list v) t))
(define (lambda? t) (and (not (atom? t)) (eq? (car t) 'lambda)))
(define (lambda-var t) (cadr t))
```

```
(define (lambda-body t) (caddr t))

(define (make-apply t0 t1) (list t0 t1))
(define (apply? t)
  (and (not (atom? t)) (not (eq? (car t) 'lambda))))
(define (apply-operator t) (car t))
(define (apply-operand t) (cadr t))

(define (make-closure e v t) (list 'closure e v t))
(define (closure? c)) (and (not (atom? c)) (eq? (car c) 'closure))
(define (closure-env c) (cadr c))
(define (closure-var c) (caddr c))
(define (closure-body c) (cadddr c))
```

4. Lazy evaluation

4.1. Streams in Lisp

A great deal of the power of Miranda derives from the use of lazy evaluation. Some arguments in favour of lazy evaluation are contained in [Turner 82, Hughes 85, Wadler 85c].

Abelson and Sussman recognize the importance of lazy evaluation and include a limited version of it, under the name of streams. Their section on streams teaches most of the important methods of programming with lazy evaluation. However, as usual, Lisp is rather more cumbersome than Miranda. For example, to find the sum of the squares of the odd numbers from 1 to 100 one writes

```
sum [i*i | i <- [1..100]; odd i]
```

in Miranda, and

```
(sum-stream
  (collect (* i i)
           ((i (enumerate-interval 1 100)))
           (odd i)))
```

in Lisp.

It is particularly annoying that two very similar types -- lists and streams -- must be

treated differently. Thus, one needs sum to find the sum of a list of numbers, and sum-stream to find the sum of a stream of numbers.

A more subtle – and therefore more serious – problem arises in the interaction between streams and the applicative order of evaluation used by Lisp. For example, the following theorem is quite useful:

```
map f (xs ++ ys)  =  map f xs ++ map f ys
```

(map f xs applies f to each element of xs, and xs ++ ys appends xs to ys).

Unfortunately, this theorem is not true in Lisp! For example, evaluating

```
(head
  (map sqrt
    (append-stream (enumerate-interval 7 42)
                   (enumerate-interval -42 -7))))
```

returns the square root of 7, whereas evaluating

```
(head
  (append-stream
    (map sqrt (enumerate-interval 7 42))
    (map sqrt (enumerate-interval -42 -7))))
```

reports a run-time error while trying to find the square root of -42. The problem is that append-stream, like all functions in Lisp, must evaluate all of its arguments.

(This particular problem would go away if streams were redesigned to delay evaluation of the head as well as delaying evaluation of the tail. However, the theorem would still not be true, as can be seen by replacing (enumerate-interval -42 -7) by (bottom), where evaluation of (bottom) enters an infinite loop.)

Obviously, problems like this are damaging if one is trying to present programming as a discipline subject to mathematical analysis. They can also lead to subtle bugs (for example, see Abelson and Sussman exercise 3-54 and the associated discussion).

Abelson and Sussman recognize this problem, and their discussion of streams includes an explanation of why streams are much better suited to a language with normal-order (lazy) evaluation. They go on to explain that they chose not to adopt

normal-order evaluation because it would make assignment impossibly difficult to use. Their choice allows the student to be exposed to two important methods of program construction, assignment and streams; but as a result, streams cannot be shown in their best form.

I would argue that the value of lazy evaluation outweighs the value of being able to teach assignment in the first course. Indeed, I believe there is great value in delaying the introduction of assignment until after the first course. (Abelson and Sussman agree that assignment should not be introduced too early, delaying its introduction until half-way through the book.)

4.2. Special forms aren't needed under lazy evaluation

Here is exercise 1-4 from Abelson and Sussman:

> Alyssa P. Hacker doesn't see why `if` needs to be provided as a special form. "Why can't I just define it as an ordinary procedure in terms of `cond`?" she asks. Alyssa's friend Eva Lu Ator claims this can indeed be done, and she defines a new version of `if` as follows:
>
> ```
> (define (new-if predicate then-clause else-clause)
>   (cond (predicate then-clause)
>         (else else-clause)))
> ```
>
> ... Delighted, Alyssa uses `new-if` to rewrite the square root program:
>
> ```
> (define (sqrt-iter guess x)
>   (new-if (good-enough? guess x)
>           guess
>           (sqrt-iter (improve guess x)
>                      x)))
> ```
>
> What happens when Alyssa attempts to use this to compute square roots? Explain.

The answer, of course, is that `sqrt-iter` gets into an infinite loop, because `new-if` always evaluates *all* of its arguments, whereas the special form `(if e1 e2 e3)` only evaluates `e2` if `e1` is nil, or `e3` otherwise.

Only the very brightest students get this question correct, and it takes a fair amount

of effort to explain to some of the others just what a "special form" is and why it is needed.

In a functional language with lazy evaluation this problem does not arise. One can define a function `newIf`:

```
newIf true  x y = x
newIf false x y = y
```

and `newIf e1 e2 e3` evaluates only `e2` or `e3` depending on the value of `e1`. There is no need for special forms. It follows that the substitution model of evaluation is more uniform and easier to explain.

Again, the problem is not that special forms cannot be explained. The point is that lazy evaluation avoids a complication that appears early on in teaching Lisp.

5. Conclusions

Abelson and Sussman's text provides an excellent introduction to programming as an engineering discipline. My experience suggests that languages such as KRC and Miranda are a significantly better vehicle for this task than Lisp.

Section 1 showed how pattern-matching and user-defined types offer an improved approach to data types. Section 2 showed that having program and data in the same form leads to many confusions in Lisp that do not occur in KRC or Miranda. It is sometimes claimed that these confusions are necessary, in order to support programs that manipulate programs. Section 3 showed that this is not quite true, in that the Lisp and Miranda approaches offer complementary advantages. Section 4 showed that streams are easier to treat in a language with lazy evaluation, and some problems with special forms do not arise; the cost of this is leaving discussion of assignment to a later course.

Some readers may object that languages like KRC or Miranda are not "credible" for teaching, because they are not used in the real world. It is true that a first course in KRC or Miranda is not sufficient to prepare students for the real world; and the same is also true of a first course in Lisp. The purpose of the first course is to teach basic principles and develop good habits of thought. In this paper I have explained why I believe languages like KRC and Miranda are good vehicles for this. Later courses should apply these principles to an imperative language, such as Pascal or

Modula; and perhaps to other languages, for data bases or for distributed computing, as well. Then the student is prepared to program in Fortran or Cobol, if need be, and to agitate for the introduction of Pascal, Lisp, or Miranda where they are appropriate.

I embarked upon teaching Lisp with the attitude that the differences from KRC and Miranda would be, at most, a small annoyance. The basic concepts were the same, and I did not feel that the syntax or idiosyncracies of Lisp would be a major barrier. Experience has convinced me otherwise. Although each difficulty by itself is minor, the cumulative effect is significant.

Abelson and Sussman are to be thanked for pointing the way to a new approach to teaching programming. I look forward to other teachers following that path, and I am particularly eager to see a new generation of textbooks – written using languages in the style of KRC and Miranda.

Acknowledgements

Hal Abelson and Gerry Sussman kindly made many detailed and perceptive comments on an earlier draft of this paper. Comments from Richard Bird and Joe Stoy were also very helpful in writing this paper.

This work was performed while on a research fellowship sponsored by ICL.

References

[Abelson and Sussman 1985a] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press and McGraw Hill, 1985.

[Abelson and Sussman 1985b] Harold Abelson and Gerald Jay Sussman. Computation: an introduction to engineering design. Massachusetts Institute of Technology, 1985.

[Augustsson 1984] Lennart Augustsson. A compiler for Lazy ML. *ACM Symposium on Lisp and Functional Programming*, Austin, Texas, August 1984.

[Burstall 1969] R. M. Burstall. Proving properties of programs by structural

induction. *The Computer Journal*, 12(1), February 1969.

[Burstall and Goguen 1982] R. M. Burstall and J. A. Goguen. Algebras, theories and freeness: an introduction for computer scientists. Internal report CSR-101-82, Department of Computer Science, Edinburgh University, February 1982.

[Fairbairn 1982] Jon Fairbairn. Ponder and its type system. Technical report 31, Cambridge University Computer Laboratory, 1982.

[Hughes 1985] R. J. M. Hughes. Why functional programming matters. Programming Methodology Group Memo PMG-40, Chalmers Tekniska Hogskola, Goteborg, Sweden, 1985.

[Landin 1966] Peter Landin. The next 700 programming languages. *Communications of the ACM*, 9(3), March 1966, 157-166.

[Rees and Adams 1982] J. A. Rees and N. I. Adams. T: a dialect of Lisp, or Lambda: the ultimate software tool. *ACM Symposium on Lisp and Functional Programming*, Pittsburgh, Pennsylvania, August 1982.

[Richards 1984] Hamilton Richards, Jr. An overview of ARC SASL. *SIGPLAN Notices*, 19(10), October 1984, 40-45.

[Steele and Sussman 1978] Guy Lewis Steele, Jr. and Gerald Jay Sussman. The revised report on Scheme, a dialect of Lisp. MIT Artificial Intelligence Laboratory Memo 452, January 1978.

[Steele 1982] Guy Lewis Steele, Jr. An overview of Common Lisp. *ACM Symposium on Lisp and Functional Programming*, Pittsburgh, Pennsylvania, August 1982.

[Turner 1976] David Turner. SASL language manual. Computer Laboratory, University of Kent, Canterbury, 1976 (revised, 1979).

[Turner 1981] David Turner. Recursion equations as a programming language. In Darlington, Henderson, and Turner (editors), *Functional Programming and Its Applications*. Cambridge University Press, 1981.

[Turner 1985] David Turner. Miranda: A non-strict functional language with polymorphic types. *Symposium on Functional Programming Languages and Computer Architecture*, Nancy, France, September 1985.

[Wadler 1985a] Philip Wadler. Views: A way for elegant definitions and efficient representations to coexist. Workshop on implementing functional languages, Aspenas, Sweden, January 1985.

[Wadler 1985b] Philip Wadler. An introduction to Orwell. Internal report, Programming Research Group, Oxford University, April 1985.

[Wadler 1985c] Philip Wadler. How to replace failure by a list of successes: Exception handling, backtracking and pattern matching in lazy functional languages. *Symposium on Functional Programming Languages and Computer Architecture*, Nancy, France, September 1985.

—