# CLASSES AS SYNTACTIC SUGAR*

by

Mitchell Wand

Computer Science Department

Indiana University

Bloomington, Indiana 47401

TECHNICAL REPORT NO. 81

CLASSES AS SYNTACTIC SUGAR

MITCHELL WAND

DECEMBER, 1978

* This material is based upon work supported by the National Science Foundation under Grant MCS75-06678 A01.

Abstract: The notion of a class, introduced in SIMULA, has become a key concept in data structuring. This note describes a simple interpretation of classes as syntactic sugar in SCHEME, an extended lambda calculus due to Steele and Sussman.

Key Words and Phrases: Classes, clusters, data abstractions, lambda-calculus, LISP, SCHEME, SIMULA

CR Categories: 4.20, 4.22, 4.12, 5.20

This material is based upon work supported by the National Science Foundation under Grant MCS75-06678 A01.

Author's address: Computer Science Department, Indiana University, Bloomington, IN 47401

## 1. Introduction

The notion of a *class*, introduced in SIMULA 67 [1], has become a key concept in data structuring [2, 5, 11]. Interpretation of classes in the lambda calculus have been given by Sandewall [8], Steele and Sussman [10, *inter alia*], and Reynolds [7]. In this note we will give a particularly simple interpretation of classes* as syntactic sugar in SCHEME, an extended lambda calculus due to Steele and Sussman [9].

SCHEME is a LISP-like language with static scoping and full FUNARGS. The following samples give a flavor of the syntax:

(LET $((v_1\ e_1)\ldots(v_n\ e_n))$ exp)

is equivalent to

((LAMBDA $(v_1\ldots v_n)$ exp) $e_1\ldots e_n$)

and

(LABELS $((v_1\ e_1)\ldots(v_n\ e_n))$ exp)

is equivalent to ISWIM's [4]

*letrec* $v_1=e_1,\ldots,\ v_n=e_n$ *in* exp

where the $e_i$ are lambda expressions. Here, as below, key words appear in upper case and metavariables appear in italics. Conditionals may be written as (IF pred then-part else-part). An association-list, as in LISP [6] may be searched by RASSOC, defined as follows:

```
(DEFINE RASSOC (LAMBDA (KEY LIST)
    (IF (NULL LIST) NIL
        (IF (EQ KEY (CAAR LIST))  (CAR LIST)
            (RASSOC KEY (CDR LIST))  )  )  ))
```

The semantics of the language allow the recursive call to be implemented iteratively.

---

*except for *resume* and *detach*, which have not been adopted in the literature on data types.

## 2. Code and Commentary

A possible concrete syntax for classes is as follows:

```
(CLASS
  ((loc_1 val_1)...(loc_n val_n))
  ((procname_1 λ-exp_1)
     :
   (procname_m λ-exp_m)))
```

An instance of the class defined in this manner should have n local variables, named $loc_1,\ldots, loc_n$ and initialized to $val_1,\ldots, val_n$ respectively. Associated with the class should be m class procedures, named $procname_1,\ldots, procname_m$, with procedure bodies $\lambda\text{-}exp_1,\ldots, \lambda\text{-}exp_m$. These procedures may refer to the locals and to each other (possibly recursively), but, in keeping with current thinking, the locals should be accessible to the user of a class instance only through the class procedures. To achieve this, the definition is expanded as follows:

```
(LET ((loc_1 val_1)...(loc_n val_n))
  (LABELS ((procname_1 λ-exp_1)...(procname_m λ-exp_m))
    (LET
      ((D (LIST (CONS (QUOTE procname_1) procname_1)
                :
                (CONS (QUOTE procname_m) procname_m))))
      (LAMBDA (Z) (RASSOC Z D)) )))
```

If X is an instance of a class with a procedure named P, a call written conventionally as $X.P(t_1,\ldots,t_n)$ is expanded as

```
((X (QUOTE P)) t_1...t_n)
```

Execution of the code for (CLASS locals procs) proceeds as follows:

(i) an environment is created in which the locals are bound to the appropriate values.

(ii) an environment is created in which the procedure names are recursively bound to the bodies, referencing the variables of (i) as non-locals.

(iii) an explicit association list (called D) is created in which each procedure name is associated with its corresponding closure (created in (ii)).

(iv) a function is created which takes as its argument a procedure name and returns the corresponding procedure.

This function is returned and is the class instance. Thus the procedure call above retrieves procedure P of class instance X, and invokes that procedure with arguments $t_1,\ldots,t_n$. Since fresh closures are created for each class instance, this procedure works on X's local variables.

This code is similar in its effect to the implementation of classes in SMALLTALK [3] and in its use of closures to Sandewall's code [8]. Our code extends Sandewall's by allowing multiple class procedures; this is the primary source of the complexity in the code. We differ from SIMULA and most implementations of classes by making a class definition an expression, which can appear anywhere in a program. For example:

```
(DEFINE CELL (LAMBDA (X)
  (CLASS (( CONTENTS X ))
    (( CONT (LAMBDA () CONTENTS ))
     (UPD (LAMBDA (VAL) (ASETQ CONTENTS VAL )))))))
```

A cell may then be created as:

```
(LET ((Z (CELL 3)))
  (BLOCK
    (PRINT ((Z (QUOTE CONT))))
    ((Z (QUOTE UPD)) 4)
    (PRINT ((Z (QUOTE CONT)))) ))
```

which creates a cell initialized to 3, calls it Z, and changes its contents to 4, printing out 3 and 4.

Note that class parameters are introduced by creating a function

```
(LAMBDA (class-parameters) (CLASS locals procs))
```

which may be bound to a variable name of any lexical scope, and that procedure names (since they are quoted) need not be declared at all. Instances of the class, however, may be passed outside this scope; this makes the code more general than any special naming scheme. Steele and Sussman's transcription [10] requires the procedure names to be declared wherever a class instance is used; this prevents classes from sharing procedure names, a necessity for concatenated classes [1]. Unlike Reynolds [7], we also avoid major transformations in the program structure; imperative features are entirely optional.

At the expense of additional syntax, a variety of additional features could be added to the framework. Some functions could be hidden by suitable editing of the association list D Direct access to some of the locals could be added similarly. Concatenated classes could be done by specifying one of the locals as

an instance of the base class and changing the function returned so that if the desired procedure is not found locally, it is passed along to the base instance, i.e.,

```
(LET ((the-basis (base-class base-class-params)) ... )
  (LABELS (...)
    (LET ((D...))
      (LAMBDA (Z)
        (IF (IS-PRESENT Z D) (RASSOC Z D)
          (the-basis Z ))))))
```

Since the locals are initialized, no class body is usually needed; one could easily be added if desired.

## 3. Conclusion

Operationally, classes are just syntactic sugar--their operational semantics requires no new concepts. We believe the significance of the notion of classes is as a syntactic structuring device. Structuring devices such as strong typing or good loop structures play an important role in ease of program writing and debugging, as shown by PASCAL, by turning run-time errors into compile-time errors. The significance of classes, we believe, is as a structuring device which allows better checking at compile time and verification time.

REFERENCES

1. Dahl, O. J., and Hoare, C.A.R. "Hierarchical Program Structures" in Dahl, O.J., Dijkstra, E.W., and Hoare, C.A.R. Structured Programming. Academic Press, London, 1972, pp. 175-220.

2. Hoare, C.A.R. Proving correctness of data representations. Acta Informatica 1 (1972), 271-281.

3. Ingalls, D. The smalltalk-76 programming system. Conf. Rec. 5th Ann. ACM Symp. on Principles of Programming Languages (1978), 9-16.

4. Landin, P.J. The next 700 programming languages. Comm. ACM 9 (1966), 157-166.

5. Liskov, B., and Zilles, S. Specification techniques for data abstractions. IEEE Trans. on Software Eng., SE-1 (1975), 7-19.

6. McCarthy, John, et. al. LISP 1.5 Programmer's Manual. MIT Press, Cambridge, Massachusetts, 1965.

7. Reynolds, J.C. Syntactic control of interference. Conf. Rec. 5th ACM Symp. on Principles of Programming Languages (1978), 39-46.

8. Sandewall, E. A proposed solution to the FUNARG problem. Report No. 29, Department of Computer Sciences, Uppsala University, November, 1970.

9. Steele, G.L., and Sussman, G.J. The revised report on SCHEME. Mass. Inst. of Tech. Artif. Intell. Memo No. 452, Cambridge, MA, January, 1978.

10. Steele, G.L., and Sussman, G.J. The art of the interpreter or, the modularity complex, (parts zero, one, and two). Mass. Inst. of Techn. Artif. Intell. Memo No. 453, Cambridge, MA, May, 1978.

11. Wulf, W., London, R., and Shaw, M. Abstraction and verification in Alphard: defining and specifying iterative and generators. Comm. ACM 20 (1977), 553-564.