The Design of a Language for Modular Programs[1]

Richard B. Kieburtz
Oregon Graduate Center

Bengt Nordström
Chalmers Technical University
and
University of Gothenburg

Abstract

Sometimes programming is difficult because of the amount of detail that is relevant to the problem being solved. A suitable language for these problems should aid the programmer in organizing a program as a synthesis of parts. Apple is a language designed for such applications. It is a typed language in which functions are objects, and it provides three complementary ways to modularize programs. These are the definition of *environments*, the use of *functional* abstraction, and the use of *data* abstraction or parameterized, abstract data types.

In this paper we describe the main features of Apple, and explain the motivation for many of the design decisions. Other documents furnish a formal definition, a programmer's manual, and implementation notes.

Keywords: functional programming, abstract data types, static environments, type polymorphism

Oregon Graduate Center Technical Report No. CS/E-82-01, March, 1982

[1]The research reported here has been supported in part by the National Science Foundation, under grant MCS 790417 and by Naturvetenskapliga Forskningsrådet under grant F 3766.

1. Introduction

A programming language provides the basic vehicle for conceptualizing algorithms. We may ask why languages in use today have grown to be so complicated. At least two causes are apparent. In an attempt to make programming languages more expressive, numerous "features" are often added, somewhat promiscuously, to a basic language design. The interaction among these features frequently proves to be much more complex than was anticipated or intended by the designers. For instance, the interaction between the conditional statement and side-effects in Algol 60 was not well understood by the language designers. In the Algol 60 report, the effect of "**if** *a* **then** *b*" when the value of the expression *a* was false was said to be equivalent to the empty statement. This is a nice property but was later changed because of implementation problems when the evaluation of the expression *a* had side-effects.

A second cause of complexity is the desire, lurking in the background of almost every language designer's consciousness, to provide for efficient implementation on conventional computing hardware. This concern often subverts the stated intention of a language design. It is a pitfall difficult to avoid. Languages often have restrictions imposed to secure more efficient execution, such as what kind of values can be assigned, passed as parameters, or returned from a function call. There are also language constructs which explicitly reflect the machine architecture, such as the **goto** statement, pointers, and assignments.

We have set for ourselves the task of proposing a language design based on a consistent set of principles, and upon certain assumptions about its mode of use by programmers. Programs can become complicated in at least two ways. There are some problems whose solutions (i.e., programs) are hard to find because of the difficulty of the problem. We can't, for instance, expect the average programmer to discover a unification algorithm. For such problems it is important to have a programming language with proof rules which are simple to grasp so that it is relatively easy to decide whether the programs are correct. Our language has not been designed specifically for that goal. In that case we would have disallowed some constructs of the language.

Problems can also be difficult to solve because of the *size* of the problem. For instance, it is not a *difficult* problem to write a PL/I - compiler but it is a *big* problem because of the number of concepts involved and the irregular structure of the problem. For such applications, a suitable language should make it easy to compose a solution (program) from its sub-solutions (modules).

In this paper, we discuss a specific set of principles and their ramification, and translate these into design goals. These give much of the rationale for a programming language design. However, the most difficult task of a language designer is to integrate a set of language features into a coherent and unified framework. We also show how we have tried to accomplish this in the design of a new language called Apple [1].

2. Design Goals

The goals we have set for this programming language design are motivated by the need to cope with the intellectual burden of writing medium-sized to large programs (say 10,000 - 100,000 lines of code). We are not primarily

concerned with the particular programming requirements of any single area of application, but rather with the process of algorithm definition and description.

We are very much concerned with the need for a language to be fully defined. Both the syntax and the semantics of the whole language should be formally defined by the language designers. In this way, denotational semantics or some other formal method can be used to guide the design of the language. For instance, at a very early stage we decided not to use the domains *location* and *continuation* which are used to explain machine characteristics like pointers and **gotos**. The first draft of the language has been revised several times because we found irregularities in the language when we formally defined it.

Another design goal we have had is that a language should support the intellectual process of abstraction and should provide for separation of tasks identified in a problem.

2.1. Complete Definition

It is very important that a programming language should be fully defined, for ambiguity on a seemingly minor point has a way of pervading an entire language. There are three categories of definitional failure that we commonly see affecting programming languages:

— **Implicit machine dependence.** It often happens that the definition of operations on the most primitive data types of a language, the characters and the arithmetic types, is left to be specified by the action of a machine on which the language is to be implemented. Insofar as a programming language is conceived as a convenient command language for a particular machine, this approach may seem reasonable. If the language is also intended as a vehicle for the statement of algorithms in more abstract form, then machine-independence of the language definition is an absolute requirement.

— **Failure to specify non-standard uses of operators.** We have all read language specifications that describe the action of an operator under commonly forseen circumstances, then contain a caveat such as "under other circumstances, the effect is undefined". A program written in such a language is well-defined only if all uses of operators conform to cases that have been anticipated by the language designer. Of course, an implementation will often define some result for the invocation of an operator in non-standard circumstances, and tradition dictates that the implementor determines this action to suit his or her convenience or prejudice. A user of such a language is forever plagued by a variety of queer "implementation dependencies" that must be learned and remembered. How much simpler it would be if the language designer had completed the job!

— **Unanticipated ambiguities or omissions in language specification.** The specification of a programming language is a complex business, and it is not surprising that errors or omissions are occasionally made. However, these would occur much less often if the language designer had attempted to give a correct and exhaustive formal semantics for the language. Formal specification forces the designer to pay careful attention to precision of meaning, and it proceeds by systematic analysis by cases. We know from

experience that most specification flaws are discovered by implementators. It is at this stage that the meaning of a language *must* be formalized, albeit in a machine-dependent manner.

If for no other reason than it forces the systematic consideration of all possibilities, the methodology that must be employed in giving a formal semantic specification justifies its use in language design. However, we believe there are other significant reasons to use it as well. A formal semantics furnishes the basis for a system of inference that can be used to verify properties of programs. Formal semantics gives the designer an indication of the complexity of a prototype language that is orthogonal to the indication gotten by programming examples in the language.

2.2. Simplicity of Design and of Use

A language design is simple if its semantic rules are relatively few in number and easy to state and to comprehend. A language is simple to use if its operations are powerful and well-matched with one another and to the data types. A simple but powerful language displays a high degree of orthogonality, meaning that the rules for composing operations and types are quite generally applicable, with few exceptions.

In addition, the syntax of a language should closely reflect its semantics. One reason that side effects are found so troublesome is that the syntax of a procedure call or a function application does not alert the reader of a program text to the semantic action that it may signify.

The choice of a rich and complementary set of data types is a task that has engaged the imagination of many language designers. However, one can never anticipate all of the types that might be found useful for programming applications. The best that can be done seems to be to provide in the language a few basic data types, and a rich set of type constructors from which new types can be composed. (The array and record templates of Pascal are such examples.) In Apple, a **class** definition is allowed to have parameters, and each abstract data type defined by a programmer also becomes a new type constructor.

It is a great convenience if a language defines a standard notation for values in each of its types, so that values can be writen in a uniform way, whether within the text of a program or for input in symbolic format. There is a problem in denoting values of abstract data types, however. If a value is denoted by a value of the class's representation type, then the denotation fails to hide the representation. Alternatively, a representation-hiding denotation might use only expressions of the free algebra generated by the operators of the abstract data type. However, this fails to provide a unique denotation for each value of the type. In Apple, we have not provided a denotation for values of an abstract type.

A language is also much easier to use if values of variables of compound types can be input or output without explicitly programming an iterative routine to traverse all components. In general, a good language should not force the use of temporary variables for bookkeeping, in order to compose operations.

This last stated principle also applies to the assignments of values. An exchange of values, for instance, should be a simple operation. The traditional statement sequence required to perform an exchange looks like:

```
t := x;  x := y;  y := t
```

in which t is a temporary variable, required as a place-holder. How much simpler is the operation if multiple asignment is used:

$$(x, y) := (y, x)$$

and there is less opportunity to make an error in programming the exchange in this way. Multiple assignment appeared as a feature of the programming language CPL [2], but has been largely ignored in more recent languages. Its advantages, particularly with respect to selective updating of an array, have been commented upon in [7].

One of the most complex aspects of programming in conventional, statement-oriented languages involves the use of multiple, alias names for a variable. We believe this practice to be totally unnecessary if three measures are adopted:

--- Variables are not passed as parameters to procedures or functions;

--- Explicit pointers are eliminated from the programming language;

--- Selective updating of an indexed variable (array) is not defined in the language.

Since each of these three practices has widespread use in programming with current languages, their abolition must be accompanied by some suggestion of alternatives.

Variables are passed as parameters in conventional language notations for either of two reasons:

--- to secure efficiency, by saving the cost of copying large data structures;

--- to allow the return of multiple result values by a procedure or function.

An alternative, adopted in Apple, is that all parameters have the status of constants within the body of a function definition. Thus it becomes an implementation decision whether or not to pass an actual parameter by copying it, or to use indirection. This decision cannot affect the semantics of a call, since alias naming is impossible. Also, a function can return a value of any type that can be defined, including a cartesian product type. This provides for the return of multiple values. Multiple assignment allows the several component values of a cartesian product type to be assigned to several variables, of compatible types. For example, a call such as

(Numerator, Denominator) := ReduceFraction(Numerator, Denominator)

is syntactically much more transparent than is the traditional procedure call in which the two variables are passed (by reference) as parameters.

Pointers have been used in low-level programming to allow easy implementation of storage management routines, and to pass access to storage blocks from one program component to another. They are used in higher-level programming to simulate recursive data types. We are not attempting to address the needs of low-level programming with Apple; there are many languages which cater for these needs. Recursive type definition is explicitly supported in Apple, obviating this particular need for pointers.

A secondary benefit of banning explicit pointers from a language is that simpler storage administration strategies, based on reference counting, are possible when the data structures utilizing heap storage are known to consist only of instances of recursively-defined types.

Selective updating refers to an assignment made to a component of a variable of a compund type, of which record types and array types are typical instances. In a selective update of a record, the field that is to be modified is specified by an identifier; its identity is determined statically and is apparent to a reader of a program. This is ordinarily not a potential source of confusion. On the other hand, in a selective update of an array variable, the component to be modified is specified by an expression whose value may be determined dynamically. The reader of a program may or may not be able to infer correctly which component or components of an array are modified by a particular selective update assignment.

The specific point of view taken in Apple is that an array is just one possible representation of a function. Since functions are legitimate objects of computation in Apple, function-valued variables are also possible. There is no type-constructor for arrays in Apple, only a type-constructor for function types. Apple provides an operator to be used instead of selective updating, when a programmer wants to specify a new function as an incremental modification of another function (see Sec. 6.0).

2.3. A typed language

Data types are introduced into a programming language in order to categorize the values which are computed. Such categorization is important when designing a program and also as part of its documentation. There is by now enough experience with the use of typed languages (such as Algol-68, Pascal, Mesa, Euclid, and ML [16, 17, 6, 4, 5]) to justify accepting the discipline typing imposes. Although a program is more verbose in a typed language than in one without types (except for languages such as ML where types are deduced [14]), the additional declarative text may make it easier for a reader to understand the program.

Many typed languages have an "escape" mechanism which makes it possible to change the type of an expression. With the possible exception of a language for low-level systems implementation (which is not one of our goals), it is neither necessary nor desirable to have such an escape mechanism. In the data abstraction language CLU [13], we have been shown how to restrict a representational view of a typed object to the module that implements that class of objects, and to distinguish between the use of an object and its representation. In this way it is possible to preserve the protection against programming errors that is afforded by strong typing, yet provide the programmer with the flexibility to do both high-level and low-level programming.

Apple has the traditional types found in many other programming languages.

$\{a_1, a_2, \ldots, a_n\}$	enumeration
float	the rationals
integer	integers
record $i_1:t_1, \ldots, i_n:t_n$ **end**	record

$t_1 * \ldots * t_n$	cartesian product
t -> t'	function
set of t	powerset
$i_1 \# t_1 + \ldots + i_n \# t_n$	disjoint union

There are two type forming operations which are new. The first is the recursive type which is used to create lists, trees and other inductive structures. The second is the *restriction* {x:T|b(x)} where T is a type, x is a variable free in b(x) and b(x) is a boolean expression under the assumption that the type of x is T. An expression *e* is of type {x:T|b(x)} if b(*e*) is true and *e* has type T. For instance, the subrange a..b in Pascal corresponds to the restricted type

{x : integer | (a<=x) **and** (x>=b)}

We can for instance give the following type to a sorting algorithm:

sort : Int_list -> {x : Int_list | sorted(x)}

where "sorted" names a boolean function yielding true if and only if its argument is a sorted list. It would perhaps be more interesting to allow the range type of a function to depend on the value of its argument in order to write more precise specifications as types. This would, however, require another foundation for the language.

3. Modularity

In order to make any real progress with the management of complexity, it is necessary to be able to subdivide programs into parts which we will call *modules*. Ideally, the meaning of each program module should be understandable independently of other modules. But since the action of a large program occurs as an interaction among several modules, this ideal is not totally achievable. Instead, we can require that the meaning of a module is to be understood from the details of its own representation, and from the abstract meanings of any other modules it uses. The meaning of one module must not depend upon the representation of modules that it uses, nor upon knowledge of how other modules make use of it.

In Apple there are three different kinds of modules: functions, classes and environments. The first two are examples of *abstractions*.

Informally, abstraction means the ability to understand what an algorithm does without knowing explicitly how it does it. An abstract characterization of an object specifies its external properties but does not specify its internal representation. Abstraction gives us the power to generalize algorithms.

One abstraction technique commonly used in mathematics is *functional abstraction*. An unspecified object is allowed to appear as an operand in an expression, and thereby, the expression defines a function. In programming, when a name is given to denote an object to be used in this way, we say that it is a *formal parameter* of a function. If functions themselves are considered to be objects (as they are, for instance, in Church's lambda-calculus [3]) then functions can be abstracted, and the abstraction process can be carried on to as high a level as is desired.

Functions should be understood by their input-output behavior. For a given argument a, the value of f(a) should always be the same. When this is the case, it is possible to understand the function itself, without being forced to know the "state" in which the function is computed. We call these kinds of functions pure functions. Other functions are not abstractions. One of the most successful operating systems in use today (UNIX) relies heavily on the function concept as the primary means of modularizing the system. Our old friend functional composition $f \circ g$ has reappeared in a disguised form as "pipes", written g|f. This has been considered to be a very powerful tool. Its "discovery" would have occurred much faster if programming languages were equipped with pure functions treated as objects.

In programming, we also make use of another form of abstraction that is of little or no concern in mathematics. Programs constantly deal with the *representation* of information. Often the specific representation chosen bears no direct relationship to the application for which a program is intended; it may be more strongly influenced by the programming language in which an algorithm is coded, or the architecture of a machine for which the program is targeted.

Programming language designers have become intensely interested in the concept that they call *abstract data types*, by which is meant the abstraction of a class of objects exhibiting common modes of behavior. Thus an abstract data type is characterized by a set of operators defining the external behavior of objects of the class. The actual representation of objects of the class is a level of detail that can remain hidden from the users of these objects. We therefore call this form of abstraction *data abstraction*.

By using data abstraction, a programmer is free to focus attention on data objects closely related to the domain of an application [15]. It is usually much easier to conceive a suitable algorithm if the details of representation can be ignored initially. Subsequently, the programmer may wish to pay careful attention to the representation of one or more object types, in order to improve the efficiency of a useful algorithm. Use of abstract data types allows these two levels of programming activity, algorithm conception and performance improvement, to be separated.

In Simula 67 [4] we find the origin of a genuine facility for abstracting a class of objects. In a Simula class, the representation of an object is given by declaring a data base for any instance, along with the operators that act upon it. Multiple instances of a class can be declared, and an operator of the class is syntactically bound to the instance upon which it acts. Simula was designed as a language for simulation, and the class notation was developed to allow abstraction of the objects that are to be simulated. We have since come to understand that most statement-oriented programs can be usefully conceived as simulations of the behaviour of abstract objects, and that the class notion is therefore more fundamental to this style of programming than even its inventors might have anticipated.

A disadvantage in using the Simula class as originally defined to represent abstract data types is its failure to hide or protect the representation of an object. A user of the class was allowed to define *ad hoc*, non-standard operations on an object by directly manipulating its representation. A change of the implementation of a data type then may force a change in the programs which use the abstract data type.

Most of the problems with the Simula class definition as a means of incorporating an abstraction facility have been dealt with in newer languages, such as Alphard [19] and CLU [13]. Abstract data types in these languages do not export

their representations, but only a set of operators. A class definition consists of the following parts:

- a heading which names the class and gives a list of its formal parameters. These may be types and constants to be used in the class definition.
- a defines list which gives names and type signatures to the constants that are to be exported by the class definition. For each type derived from the class (i.e., a type constructed from a class by giving actual parameters to the class in a type declaration), a set of these constants will be included in the environment in which the class-derived type is defined.
- a representation part, which declares a local environment in which implementations are given to the constants named in the defines list. This local environment must include:
  - declaration of a type named *carrier*, in terms of which each class-derived type is represented.
  - declarations of the constants named and typed in the defines list.

The representation part may also include declarations of other constants and types.

We can't resist the temptation to show the canonical example of an abstract data type in Apple:

```
class Stack of type T       // T is a parameter of the class
   == def Push : carrier*T -> carrier,
         Pop  : carrier -> carrier,
         Top  : carrier -> T,
         IsEmpty: carrier -> bool
      rep
         type carrier : record
                           Store : integer -> T,
                           Index : integer
                        end
                        init [Store == function(x) return initial(T) end,
                              Index == 0]

         const
            Push == function (S,x)
                           return let P : integer == succ S.Index in
                                       [Index == P, Store == S.Store|P:Index]
                                       // a|i:s updates the mapping a on index i with s
                                       endlet
                              end,

            Pop == function (S)
                           return let P : integer == S.index in
                                       if P >= 0 then
                                       [Store == S.Store, Index == pred P]
                                       else
                                       S
                                       endif
                                       endlet
                           end,
```

```
Top == function (S)
            return S.Store(S.Index)
            end,

IsEmpty == function (S)
            return S.Index = -1
            end
endclass
```

As mentioned earlier, a class may have parameters which can be types and constants. The constant parameters normally include functions for the type parameters, for otherwise, there would be no operations applicable to an object of a parameter type within the class definition.

For example, the following class heading is parameterized by a type and a function for that type:

```
class Storage_Class of type T const Equiv : T*T -> Bool
```

To form a type from this class, suitable parameters are substituted for the formal parameters, as in

```
type IntegerCell : Storage_Class [type T : integer,
                                  const Equiv : Integer*Integer -> Bool ==
                                     function (x)
                                        return x!1 = x!2
                                        end
                                  ]
```

In this declaration of the new type IntegerCell, the type parameter T of Storage_Class has been bound to the type *integer*, and the value parameter Equiv to a function defined by an equality relation (the notation $x!i$ is used to select the ith component of an expression x, of a cartesian product type.)

4. Generic Types

In the declaration of a class, or abstract data type, we may decide to utilize a formal parameter to represent a type. The parameter is often referred to as a *generic* type, since it may be replaced by various argument types, in those declarations in which particular types are derived from the class. This ability to perform abstraction on types gives to a lanaguage a new dimension of expressiveness, but also raises a non-trivial problem.

In order to make type abstraction useful, the operators of generic parameter types must be available for use in the representation of the parameterized type defined by the class. Within the representation part of the class, it should be possible to declare instances of the parameter type, to refer to constants of the type, and to invoke its operators. Herein lies the difficulty. There is no uniform set of operator or constant names applicable to all types. The operators are type-specific. This is in fact the whole point underlying strong data-typing.

It must be possible to refer by a common symbol to operators having differing representations, and applicable to different types. There are two approaches to achieving the desired non-specificity of naming.

One approach, called *overloading* or more precisely, the definition of polymorphic operators, allows an operator symbol to denote operators that apply to distinct types. An overloaded operator symbol carries its multiple meanings throughout all contexts in which the various types to which it applies

are defined. Most languages make use of overloaded operator symbols to some degree. Examples are "=" to represent a polymorphic boolean expression, denoting equality, and "+" to represent addition on any arithmetic type. The point is, that language-defined overloadings usually do represent operators having uniform algebraic or operational properties, so that the intuition of the reader of a program is not confounded by the multiple meanings of overloaded symbols.

When overloading is placed in the hands of the programmer, no such assurance can be given. If the operator symbol "=" is allowed to be overloaded by a programmer, the language may only impose the restriction that "=" applies to a pair of objects of the same type, and produces a Boolean result. It cannot assure that the user-defined function that gets bound to this symbol will define an equivalence relation. Although years of mathematical training will have taught the reader of a program to associate "=" with equivalence, he/she will have to fight against making the intuitive assumption that "=" always is an equivalence relation.

An alternative to overloading is to use formal parameter names for the operators of generic types. A formal parameter name is restricted to the textual scope of the class in which it is defined. It is a programmer-defined name, and so carries with it no implication of special algebraic properties. The use of parameterization to obtain names for generic operators seems to us vastly preferable to the strategy of programmer-defined overloading of language-defined operator symbols.

5. Static Environments

The meanings of names that are used in a programming language are declared relative to an *environment* that provides data needed to interpret definitions. Usually today, because languages don't support multiple environments, they are implemented as separately compiled files.

An environment is said to be *static* if once established, its data remain invariant throughout its lifetime. Otherwide an environment is *dynamic*. We believe it is important to our ability to understand programs that all declarations are interpreted in static environments.

Thus, in our concept, a static environment does not incorporate variables, and no definition (of a type, constant, class, or function) given relative to a static environment can depend upon the value of a variable. This principle prevents the importation of "global" variables into definitions.

Therefore, an *environment* in Apple is a collection of definitions of types, constants (including functions) and type constructors, or classes. An environment has no state. An environment can be named in an environment declaration:

```
env display
    const erase: ascii == ^H,
    class screen == ...
endenv
```

A named environment is a self-contained unit. The declarations it contains use only those definitions given within it, or which have been imported.

In order to import some definitions from another environment, a **ref**erence declaration can be given, referring to the name of an environment and to the names of the types, constants, classes or environments that are to be imported.

```
ref foo (type tnew|tfoo      // declares a new name for imported type
                             //   tfoo in the current environment
         const c, cc         // these constants are imported without renaming
         env display
        )
```

There is a universal environment that is presumed always to be included. It includes such types as *integer*, *float*, *Boolean*, *ascii*, and a set of primitive type constructors as well. These predefined types have machine independent definition. A programmer who wants to impose a machine-dependent restriction upon the integer values used by a program might declare within the local environment a type *Int*, defining it to be a subtype of *integer*, and specifying the values that can be represented in the registers of the intended target machine. Since this restriction would be explicitly stated as part of a program, machine-independence is not sacrificed. The program can be translated by a compiler for any target machine, although the efficiency of the translated program may be better on the particular machine for which it was originally intended.

Programmer-defined type constructors, or abstract data types, are defined by a class notation, following Simula 67 nomenclature. We have made the class definition a unit of modularization as well. A class definition may explicitly include one or more named environments to be incorporated into its internal environment.

6. The imperative parts of Apple

Apple provides imperative language syntax for use in composing function bodies. Although it is not necessary to take advantage of the ability to write statements, this facility has been provided for two reasons. Many persons engaged in the practice of programming are experienced in thinking about programs as sequences of statements. This seems to help them in decomposing a seemingly large task into smaller ones, and we would not want to deny it to programmers who have come to rely upon it. A second reason is that we ourselves are forced into thinking in terms of statements in order to describe a command language for a computer system in which objects such as files exist over a period of time. We should like to be able to use a fragment of Apple as this command language.

When a variable is declared, it is always given an initial value. An initial value can be explicitly designated by appending an assignment clause to a declaration, but in case it is not, Apple provides a standard, default initial value for every type. There is no such thing as an uninitialized variable in an Apple program.

A particularly troublesome feature of imperative languages since Fortran is the use of assignment to selectively update a component of a so-called indexed variable, or array. This is the feature that has prevented the acceptance of multiple assignment in programming languages, since selective updating may require data-dependent evaluation of the target of an assignment. Thus a multiple assignment such as

$$(a[i],a[j]) := (x,y)$$

is potentially ambiguous, in case the values of i and j are equal. Selective updating confuses the concept of assignment to a variable with the representation of values in the cells of a computer's memory.

A high-level view of assignment is that it replaces the value of a variable by a new value of the same type. Thus assignment to a function-typed variable defines a new value of the mapping that it represents. When one wishes to define

a new mapping as an incremental modification of an existing value, the operator defined in [8], which we call "supersedes" is appropriate. Adopting this notation, the assignment that would be made by selective updating in Pascal, for instance,

$$x[i] := e$$

is written in Apple as an assignment to the variable x,

$$x := x|i:e$$

The expression on the right of the assignment is read as "the mapping x except where applied to argument value i, when it produces e". The notation is much clearer, and since the target of the assignment is an entire variable, its use does not lead to undetectable ambiguity in the definition of multiple assignment. As to efficiency of implementation, a simple live-dead analysis of the Apple assignment statement given above leads to the same implementation as in Pascal.

6.1. The Recursive Statement

To secure execution other than that controlled by bounded iteration, we have provided an explicitly iterative control called the *recursive statement* [9]. Such a statement makes it possible to invocate a bracketed statement list that has been given a name. Within the bracketed statement-list, there can appear a component statement

repeat <statement-identifier>

where the scope of a statement-identifier is precisely the statement list enclosed by the brackets to which it is attached.

The initial invoction of a recursive statement occurs when it is encountered in the course of sequential execution, as is the case for any other statement. In a recursive invocation, the continuation of the statement following the repeat statement is remembered, and following termination of a recursive invocation, this continuation is executed.

The recursive statement permits repetitive execution of a bracketed statement list, and thus allows the successive updating of variables each time a controlled statement is repeated. This would be awkward to program if it were necessary to define recursive functions to perform such computations. Thus in a programming language that uses imperative statements and assignments to variables, yet has no procedures, a control structure such as the recursive statement seems essential. In fact, the recursive statement is simpler, yet more general than is the variety of iterative control statements in more common use.

The accustomed repetitive statements of structured programming can be emulated by the recursive statement, used in combination with a conditional statement, and without introducing any additional Boolean variables.

traditional statement (Pascal syntax)	recursive equivalent (Apple syntax)
while B **do** S	while:: **if** B **then** S; **repeat** while **endif** ::while
repeat S **until** B	until:: S; **if not** B **then repeat** until **else skip** **endif** ::until

while B **do**
begin
S;
if B' **then goto** L
else S'
end
L:
	while_exit::
	if B **then**
	S;
	if B' **then skip**
	else S'; **repeat** while_exit
	endif
	endif
	::while_exit

(The last of these control structures is the "loop n-and-one-half times" construct, which is not primitive in Pascal.)

The recursive statement has a simple denotational semantics description, and we believe it furnishes a natural way to program iteration.

6.2. Translation from imperative to applicative Apple

In spite of the fact that a function body can contain a sequence of imperative statements, Apple is a functional language. This is because variables are only local to a function; there are no global variables. Therefore, all functions are "pure". Evaluation of an applicative expression always produces the same result, regardless of its context or the order of evaluation used.

The reader may be more easily convinced of the claim that Apple is a functional language if we show a simple construction which produces purely applicative Apple from any Apple program that contains imperative statements. The construction applies to the function constructors of an Apple program, and consists of two phases. First, the variables that are declared within a function constructor are consolidated into a single variable as follows.

A list of declarations such as

$$\textbf{var}\ x_1 : t_1 == e_1, \ldots x_k : t_k == e_k$$

is translated into

$$\textbf{var}\ x : \textbf{record}\ x_1 : t_1, \ldots x_k : t_k\ \textbf{end} == [x_1 : e_1, \ldots x_k : e_k]$$

Each occurrence of a variable x_i within an expression is replaced by $x.x_i$, selecting the corresponding field of the record-typed variable x. Each assignment statement $x_i := e$ is replaced by

$$x := x|x_i : e$$

in which the right-side expression gives the value of x, except in the field named by x_i, where it takes the value of e. Each multiple assignment,

$$(x_{i_1},\ldots x_{i_m}) := (e_1,\ldots e_m)$$

where $1 \le i_j \le k$, for $1 \le j \le m$, is replaced by

$$x := x \,|\, x_{i_1} : e_1 \ldots \,|\, x_{i_m} : e_m$$

By this translation, each function constructor that contains declarations of variables is replaced by a semantically equivalent function constructor which contains only a single variable.

After consolidating its variables, each function constructor which contains imperative statements is of the form

function (a) **var** x : t == v **do** S **return** e **end**

where t is a record type-expression.

The second phase of the construction applies to the statement body of each such function constructor a function

Ψ : statements → expressions

The original function constructor is then replaced by the semantically equivalent constructor

function (a) **return let** x : t == Ψ[S](v) **in** e **endlet end**

We see that the crucial part of the construction lies in the definition of Ψ, which is defined by cases on the statement forms as follows:

1) $\Psi[\, S_1;S_2 \,] = \Psi[\, S_2 \,] \,@\, \Psi[\, S_1 \,]$
2) $\Psi[$ **skip** $]$ = **function** (x) **return** x **end**
3) $\Psi[\, x := e \,]$ = **function** (x) **return** e **end**
4) $\Psi[$ **if** e **then** R **else** S **fi** $]$ =
   function (x) **return if** e **then** $\Psi[\, R \,](x)$ **else** $\Psi[\, S \,](x)$ **endif end**
5) $\Psi[$ **repeat** L $] = L$
6) $\Psi[\, L:: S ::L \,]$ =
   let L : t == **function** (p) **return** $\Psi[\, S \,](p)$ **end**
   in L **endlet**
7) $\Psi[$ **unioncase** e **of** $i_1 \# a_1 : t_1$ => S_1 || ... **endcase** $]$ =
   function (x) **return**
   unioncase e **of** $i_1 \# a_1 : t_1$ => $\Psi[\, S_1 \,](x)$ || ... **endcase**
   end

In which "@" is the operator symbol that denotes function composition in Apple. The resulting function constructor is seen to be expressed in purely applicative notation.

The basis of the translation is that every assignment can be translated into a function from values of the type t into values of the same type, since the evaluation of expressions has no side effect. Note that in (6) we rely upon the fact that a **let** expression in Apple may be recursively defined, i.e. we use a common notation for both the *let* and *letrec* expressions of ISWIM [20]. We have omitted to give a translation for the **for** statement of Apple, as it can be defined in terms of a recursive statement if a new identifier is also declared [1].

7. Recursive Data Types

For many applications of non-numeric programming, some part of the problem domain can be modelled by trees and linear lists. These classes of acyclic graphs are of sufficient importance to warrant their inclusion, providing type

constructors for their representation. Each recursive type can be given as the solution of a type equation.

In many programming languages, pointer or reference variables have been used to allow the presentation of recursively defined data structures. But when pointers are used, one does not declare a recursive type, but rather the type of a node with which data structures of the intended type can be represented. For example, to utilize a binary tree as a data structure, one might define in a language such as Pascal

```
type  Ptr = ^Node;
      Node = record
                Info : Infotype;
                Left, Right : Ptr
             end
```

The trouble with such definitions is that in using them, a program in execution may create data structures that are not binary trees, and in fact, may not even have acyclic graphs [10, 11].

Contrast the preceeding example with a declaration of a binary tree type in Apple:

```
type  nulltype : {null},
      treetype : tree rec  emptytree # nulltype +
                 mktree # (Infotype * tree * tree)
```

The definition tells us that a value of type tree is either the value null (the unique value of type nulltype), or is constructed from a triple containing one value of Infotype and two values of type tree. This kind of inductive definition of data types is very similar to the definition of natural numbers in mathematics, which would have the following appearance in Apple:

```
type Nat: Nat rec zero # nulltype + successor # Nat
```

It is of course not possible to create cyclic graphs with this mechanism.

The programmer is liberated from the need to conceptualize the "deferencing" operation on pointers. It isn't necessary to write program statements to perform storage alloction or deallocation and the language provides a standard denotation for values of type tree. For instance, the following expressions denote trees:

```
emptytree # null

mktree # (3
              mktree # (4, emptytree#null, emptytree#null),
              mktree # (5, emptytree#null, emptytree#null))
```

The language-defined default value for initialization of this type would be emptytree#null. We shall say more later about how a default initial value is determined.

8. How Control Structures Complement Data Types

Perhaps the most important consideration in the design of a strongly typed imperative language should be that the control structure complements the data types, for it is in this way that a programmer is given the power to express algorithms elegantly. We have already discussed the notion of abstract data types

and their realization by classes, and the generalizations of assignment. Next we shall review the predefined types and type constructors and the control structures of Apple.

8.1. Union types and the Unioncase Statement

The unioncase statement of Apple discriminates among the component types of a union, associating a statement list with each. Selection is made on the basis of the component type of the current value of a selector expression of the union type. Similar type-discriminating statements are found in Algol 68 [16] and in Euclid [12].

One anticipated use of union types in Apple will be to distinguish data values that are undefined, or otherwise regarded as exceptional, and to provide the programmer with an opportunity to deal with such special cases. For example, consider the problem familiar from Pascal, in which a variable of a subrange type is repeatedly incremented and tested:

```
(* Pascal *)
var X : 1..N;
begin
    X := 1;
    repeat
        <statement list>;
        X := succ(X)
    until X > N
end
```

In order that the <statement list> can be executed for each value of X in the range of its type, the final incrementation of X must take it out of its range, which is an "undefined" operation in Pascal. Thus the program segment given above would be incorrect, in spite of the fact that its "exceptional condition" is anticipated and is intended to signify termination of the iteration.

Let us illustrate the use of union types to cope with this situation. Suppose we declare a type

```
type Range :  1..N,
     OfloType  :  {Oflo}
```

and define a successor function appropriate to these types:

```
const
   Successor_in_Range :
            Range -> (Inrange#Range + OutofRange#Oflotype) ==
               function (x)
                  return if x < N then
                              Inrange#succ(x)
                         else
                              OutofRange#Oflo
                         endif
                  end
```

In order to refer to the case selector expression within the controlled statement list corresponding to each case, it is renamed, following the appropriate

unioncase label. This name then designates a constant which has the value of the case selector expression, but whose type is the particular component of the untion type that is designated by the unioncase label. Now the desired iteration control can be written without difficulty:

```
// Apple
var x : Inrange#Range + OutofRange#Oflotype == Inrange#1
loop::
    unioncase x of
        Inrange# y : Range do <statement list>;
                             x := successor_in_range(y);
                             repeat loop
      || OutofRange# y : Ovflotype do skip
    endcase
::loop
```

As a second example, consider the abstract data type Stack, as defined in the example of Section 3. An operator of this class is the function Top, which when applied to an instance of a Stack type, is to return the value at the top of the stack. If a stack is empty then Top applied to that stack should not produce a value of the element type T, but a value distinguishable as exceptional. Rather than using a sentinel value of type T for this purpose, suppose we were to redefine the type of the operator Top to be:

```
def Top : carrier -> (Normal#T + Exceptional#Ovflotype)
```

Then, given the additional declarations

```
type IntStack : Stack [type T : Integer]

var LeftStack : IntStack
```

we can express the application Top LeftStack. This expression does not have type Integer, however; its type is the union

(Normal#Integer + Exceptional#Ovflotype).

In order to use the value of Top Leftstack in a statement, it must usually be projected onto its component types in a **unioncase** prefix. For instance, we could write

```
unioncase Top LeftStack of
    Normal # x : Integer do <normal statement list>
  || Exceptional # x : Ovflotype do <exception handling statements>
endcase
```

There are several aspects of this example that are noteworthy. First, the "exception" is associated with data contamination, in this case with the value at the top of the stack. Second, because the designer of this particular abstract data type has made provision for the possibility of its malfunction, the user of it *must* also be prepared to deal with an exceptional value; the type-checking mechanism will not allow it to be ignored! We believe that this allows a much better solution to the problem of possibly abnormal data vlaues than is provided when the result of a computation is said to be "undefined" when abnormal data values occur. Furthermore, the language supports and enforces the consideration of abnormal values, insofar as their occurrence is anticipated by a programmer.

8.2. Conditional statements and type Boolean

The control structure provided for use with Boolean values is the conditional statement,

if <Boolean expr> **then** <statement>
[**elseif** <Boolean expr> **then** <statement>]*
else <statement>
endif

(in which the brackets denote optional phrases, and the asterisk following a closing bracket denotes repetition).

8.3. Powersets and Bounded Iteration

Apple permits the definition of powerset types. A powerset type can be defined over enumerations, integers and restrictions of these. The operators on powerset types are a membership predicate, equality and inclusion predicates, and the functions of union, intersection and relative complement.

A value of a powerset type can also be denoted by explicit construction. The set constructor of Apple shows the use of a characteristic predicate to define the members of a set, and is thus considerably more powerful than its analog in Pascal. The basic syntax of a set constructor is:

$$\{x : T \mid e\}$$

where x is a variable of type T and may occur in the boolean expression e.

The stroke, "|", is read "such that", and the membership of the set consists of all values of the type T such that the Boolean expression evaluates *true*, when the value is substituted in place of the bound variable.

For convenience, another form of the set constructor, analogous to that of Pascal, is also allowed. In the second form, members of a set are indicated by expressions denoting individual values or intervals.

In statement-oriented programming, it is often necessary to repeat the execution of a fixed program component for each value in a prescribed set. Since Apple provides a powerful mechanism for defining sets by the use of characteristic predicates, it is natural that this mechanism should be further utilized in specifying iteration.

The syntax of the iteration statement is:

for x : t [**incr**] e **from** p **do** S **endfor**

where x is a constant of type t bound in the statement list S to successive elements of the set e, starting with the element p. The keyword **incr** (or **decr**) expresses that the iteration starts with p and then continues with elements following (preceding) p according to the order defined upon values of type t.

9. Streams

Apple programs interface to a computing system environment through objects called streams. A stream type is derived from a class which has its

definition in the universal environment of Apple. The class signature is:

```
class Stream of type T
    def nullstream : carrier,
        put : carrier * T -> carrier,
        get : carrier -> carrier * T,
        eos : carrier -> Bool
endclass
```

The representation of this class is not specified, and may be implementation dependent. However, its operators are required to obey the following four equations:

1) eos nullstream = true
2) eos put(S,x) = false
3) get nullstream = (nullstream, initial(T))
4) get put(S,x) = **if** eos S **then** (nullstream, x)
   else (put((get S)!1,x), (get S)!2)

The use of streams provides a convenient interface between a computing system and a program compiled from an Apple function constructor taking a stream object as its argument and delivering another stream as its result. The argument for such a function can be implemented as a sequential file open for reading, and the result by another file open for writing. The command to evaluate a function application and save its result is analogous to an assignment statement

output := f input

The evaluation rule can be lazy evaluation [18], since application of f to its argument produces no side effect.

Function applications can also be composed, as in the assignment

output := f g h input

in which f, g, and h are all functions acting on stream-typed objects. In an implementation in which a composite function application is evaluated lazily, stream objects may be used by the implementation to buffer the result of one application, which in turn forms the argument (or input) of the function symbol to its left. In the command language of the UNIX operating system, which provides for asynchronous (although not lazy) evaluation of composite applications, the stream-typed objects are called *pipes* and are denoted by a stroke "|". Typical UNIX command language syntax for an application such as that given above would be the pipeline expression

h input | g | f >output

One cannot help noticing that UNIX notation would be more uniform if the application of a function to an argument were written in postfix notation.

10. Conclusions

The programming process becomes difficult for us when we try to consider too many distinct, abstract concepts that interact with one another. Because of their interaction, it often seems inescapably necessary to us to deal with such a group of concepts simultaneously. It is a well-known fact among psychologists that the number of independent ideas that a human can cope with at one time is between five and nine, just a few more than the number of objects a good juggler can keep in the air.

A programming language may or may not be able to help with this difficulty, but if it can, it will do so in two ways. It will reduce the amount of irrelevant detail concerned with a specific computational model that tends to clutter one's mind when composing and analyzing an algorithm. And it will aid in formulating and composing abstractions, so that details of representation can be buried. A language should allow and encourage us to partition a program syntactically into modules which individually contain few enough variables, statements and expressions that thier complexity is manageable. These goals have provided the rationale for the design of Apple; nearly every other facet of the language has been shaped to serve them.

We have found it useful to distinguish clearly between the environment provided to a program component, and the state of execution of that same program component, when in execution. Environments are static, and therefore the complexity of an environment is of much less concern to us than is the complexity of a program state space. A human can cope with a complex program environment by employing textual lookup of details if it is necessary to refresh his/her memory. But a program state is dynamic, and in order to understand an imperative program one must be able to conceptualize the transformations of the state space that are brought about by executing the statements of a program.

It is the variables of a program which contribute components to its state, and the key to controlling complexity is to limit the number of variables whose values can change in any individual context. The design of Apple provides two ways to control the proliferation of variables. One way is by restricting scope; program modules are never allowed to exchange variables. A variable exists only within the body of a function constructor. For communication between modules, only values are transmitted.

A second way in which needless complexity is avoided is to treat variables declared to be of an abstract data type in the same way as other variables. Assignment to such a variable is whole-value assignment, and the operators on abstract data types are programmed functions that are functional upon only the arguments to which they are explicitly applied. The language-defined operators on predefined types have exactly the same properties. Earlier languages supporting the concept of abstract data types have not always followed this principle. In CLU, for instance, the operators on an abstract data type may include procedures as well as functions, and these may import global variables. In consequence, the complete functionality of a CLU operator is not always apparent from the local context in which it is applied.

The construction (or analysis) of a program in Apple has essentially two aspects. One is the definition of an environment, including the specification of useful abstractions. The other is the composition of representations for each of the abstractions that have been defined, in the form of a representation type and a set of functions to/from this type. The two modes of thinking required for these activities are practiced alternatively and interactively by a human programmer. The programming language has been structured to take this into account.

The syntax of Apple should not be considered as frozen in its initial definition, although we hope that its semantics will be. In particular, declarative typing adds verbosity to a programming language. One of the attractive aspects of the syntax of an untyped applicative language is that one isn't required to think very much about syntax when writing an expression. It may be that textual clutter could be reduced without sacrificing clarity of meaning if Apple adopted a more permissive policy with regard to type declaration. Declarations of the types of constants and variables might be made optional, allowing types to

be computed in a static analysis of a program, as in [14]. Such a policy has been adopted in the typed functional language ML [5], and subsequently in HOPE [21].

11. Acknowledgement

We wish to thank Bruce Russell, Gary Lindstrom, Åke Wikström, and Kent Petersson for reading and commenting on early versions of this manuscript. Special thanks are due to Gene Rollins for suggesting the imperative-to-applicative Apple translation given in Section 6.1.

12. References

[1] Kieburtz, R. B. and Nordström, B. "The formal definition of Apple", Oregon Graduate Center, Technical Report CS/E-82-02 (1982).

[2] Barron, D. W., et. al. "The main features of CPL." **Comp. J.** 6 (1963), 134-143.

[3] Church, A. **The Calculii of Lambda-Conversion.** Princeton Univ. Press, Princeton, 1971.

[4] Dahl, O-J., Myhraug, B., and Nygaard, K. **Simula 67 Common Base Language.** Norwegian Computing Center, Oslo, 1968.

[5] Gordon, M.J.C., Milner, R., Morris, L., Newey, M., and Wadsworth, C., "A meta-language for interactive proof in LCF" Proc. 5th ACM POPL, Tucson, Arizona (1978), 119-130.

[6] Geschke, C. M., Morris, J. H. Jr. and Satterhwaite, E. H. "Early experience with Mesa." **Comm. ACM 20**, 8 (August 1977), 540-552.

[7] Gries, D. "The multiple assignment statement." **IEEE Trans. on Software Engr. TSE-4**, 2 (1978), 89-93.

[8] Hehner, E. C. R. "On removing the machine from the language". **Acta Informatica 10**, 3 (1978), 229-243.

[9] Hehner, E. C. R. "do considered od: A contribution to the programming calculus". **Acta Informatica 11**, 4 (1979), 287-304.

[10] Hoare, C. A. R. "Recursive data structures." **Inter. J. of Computer and Infor. Sci. 4**, 2 (1975), 105-132.

[11] Kieburtz, R. B. "Programming without pointer variables." **ACM Sigplan Notices 8**, 2, Proceedings of ACM Conference on Data, (1976), 95-107.

[12] Mitchell, J.G., Popek, G. J. "Revised report on the programming language Euclid". Tech. Rept. CSL 78-2, Xerox PARC, 1978.

[13] Liskov, B. H., et. al. "Abstraction mechanisms in CLU." **Comm. ACM 20**, 8 (August 1977), 564-576.

[14] Milner, R. "A theory of Type Polymorphism in Programming", **JCSS 17** 3 (Dec. 1978), 348-375.

[15] Nordström, B. "Programming with Abstract Data Types, Some Examples." ACM'78 Proceedings, Dec. 4-6, Washington, D.C. pp. 646-654.

[16] van Wijngarten, A., et. al. "Revised report on the algorithmic language Algol 68." **Acta Informatica 5**, 1-3 (1975), 1-236.

[17] Wirth, N. "The programming language Pascal." **Acta Informatica 1**, 1 (1971), 36-63.

[18] Henderson, P. and Morris, J.H., Jr., "A lazy evaluator", Proc. 3rd ACM POPL, Atlanta, Ga., (1976), 95-103.

[19] Wulf, W. A., London, R. L. and Shaw, M. "An introduction to the constructon and verificaton of Alphard programs." **IEEE Trans. on Software Engr. SE-2**, 4 (December 1976), 253-365.

[20] Landin, P.J., "The next 700 programming languages", **CACM 9** 3 (March, 1966), 157-166.

[21] Burstall, R.M., MacQueen, D.B., and Sannella, D.T., "HOPE: an experimental applicative language", Proc. 1980 LISP Conf., Stanford, Calif. (Aug. 1980), 136-143.