Higher-Order and Symbolic Computation, 13, 11–49, 2000
© 2000 Kluwer Academic Publishers. Manufactured in The Netherlands.

Fundamental Concepts in Programming Languages

CHRISTOPHER STRACHEY
Reader in Computation at Oxford University, Programming Research Group, 45 Banbury Road, Oxford, UK

Abstract. This paper forms the substance of a course of lectures given at the International Summer School in Computer Programming at Copenhagen in August, 1967. The lectures were originally given from notes and the paper was written after the course was finished. In spite of this, and only partly because of the shortage of time, the paper still retains many of the shortcomings of a lecture course. The chief of these are an uncertainty of aim—it is never quite clear what sort of audience there will be for such lectures—and an associated switching from formal to informal modes of presentation which may well be less acceptable in print than it is natural in the lecture room. For these (and other) faults, I apologise to the reader.

There are numerous references throughout the course to CPL [1–3]. This is a programming language which has been under development since 1962 at Cambridge and London and Oxford. It has served as a vehicle for research into both programming languages and the design of compilers. Partial implementations exist at Cambridge and London. The language is still evolving so that there is no definitive manual available yet. We hope to reach another resting point in its evolution quite soon and to produce a compiler and reference manuals for this version. The compiler will probably be written in such a way that it is relatively easy to transfer it to another machine, and in the first instance we hope to establish it on three or four machines more or less at the same time.

The lack of a precise formulation for CPL should not cause much difficulty in this course, as we are primarily concerned with the ideas and concepts involved rather than with their precise representation in a programming language.

Keywords: programming languages, semantics, foundations of computing, CPL, L-values, R-values, parameter passing, variable binding, functions as data, parametric polymorphism, ad hoc polymorphism, binding mechanisms, type completeness

1. Preliminaries

1.1. Introduction

Any discussion on the foundations of computing runs into severe problems right at the start. The difficulty is that although we all use words such as ‘name’, ‘value’, ‘program’, ‘expression’ or ‘command’ which we think we understand, it often turns out on closer investigation that in point of fact we all mean different things by these words, so that communication is at best precarious. These misunderstandings arise in at least two ways. The first is straightforwardly incorrect or muddled thinking. An investigation of the meanings of these basic terms is undoubtedly an exercise in mathematical logic and neither to the taste nor within the field of competence of many people who work on programming languages. As a result the practice and development of programming languages has outrun our ability to fit them into a secure mathematical framework so that they have to be described in ad hoc ways. Because these start from various points they often use conflicting and sometimes also inconsistent interpretations of the same basic terms.

A second and more subtle reason for misunderstandings is the existence of profound differences in philosophical outlook between mathematicians. This is not the place to discuss this issue at length, nor am I the right person to do it. I have found, however, that these differences affect both the motivation and the methodology of any investigation like this to such an extent as to make it virtually incomprehensible without some preliminary warning. In the rest of the section, therefore, I shall try to outline my position and describe the way in which I think the mathematical problems of programming languages should be tackled. Readers who are not interested can safely skip to Section 2.

1.2. Philosophical considerations

The important philosophical difference is between those mathematicians who will not allow the existence of an object until they have a construction rule for it, and those who admit the existence of a wider range of objects including some for which there are no construction rules. (The precise definition of these terms is of no importance here as the difference is really one of psychological approach and survives any minor tinkering.) This may not seem to be a very large difference, but it does lead to a completely different outlook and approach to the methods of attacking the problems of programming languages.

The advantages of rigour lie, not surprisingly, almost wholly with those who require construction rules. Owing to the care they take not to introduce undefined terms, the better examples of the work of this school are models of exact mathematical reasoning. Unfortunately, but also not surprisingly, their emphasis on construction rules leads them to an intense concern for the way in which things are written—i.e., for their representation, generally as strings of symbols on paper—and this in turn seems to lead to a preoccupation with the problems of syntax. By now the connection with programming languages as we know them has become tenuous, and it generally becomes more so as they get deeper into syntactical questions. Faced with the situation as it exists today, where there is a generally known method of describing a certain class of grammars (known as BNF or context-free), the first instinct of these mathematicians seems to be to investigate the limits of BNF—what can you express in BNF even at the cost of very cumbersome and artificial constructions? This may be a question of some mathematical interest (whatever that means), but it has very little relevance to programming languages where it is more important to discover better methods of describing the syntax than BNF (which is already both inconvenient and inadequate for ALGOL) than it is to examine the possible limits of what we already know to be an unsatisfactory technique.

This is probably an unfair criticism, for, as will become clear later, I am not only temperamentally a Platonist and prone to talking about abstracts if I think they throw light on a discussion, but I also regard syntactical problems as essentially irrelevant to programming languages at their present stage of development. In a rough and ready sort of way it seems to me fair to think of the semantics as being what we want to say and the syntax as how we have to say it. In these terms the urgent task in programming languages is to explore the field of semantic possibilities. When we have discovered the main outlines and the principal peaks we can set about devising a suitably neat and satisfactory notation for them, and this is the moment for syntactic questions.

But first we must try to get a better understanding of the processes of computing and their description in programming languages. In computing we have what I believe to be a new field of mathematics which is at least as important as that opened up by the discovery (or should it be invention?) of calculus. We are still intellectually at the stage that calculus was at when it was called the 'Method of Fluxions' and everyone was arguing about how big a differential was. We need to develop our insight into computing processes and to recognise and isolate the central concepts—things analogous to the concepts of continuity and convergence in analysis. To do this we must become familiar with them and give them names even before we are really satisfied that we have described them precisely. If we attempt to formalise our ideas before we have really sorted out the important concepts the result, though possibly rigorous, is of very little value—indeed it may well do more harm than good by making it harder to discover the really important concepts. Our motto should be 'No axiomatisation without insight'.

However, it is equally important to avoid the opposite of perpetual vagueness. My own view is that the best way to do this in a rapidly developing field such as computing, is to be extremely careful in our choice of terms for new concepts. If we use words such as 'name', 'address', 'value' or 'set' which already have meanings with complicated associations and overtones either in ordinary usage or in mathematics, we run into the danger that these associations or overtones may influence us unconsciously to misuse our new terms—either in context or meaning. For this reason I think we should try to give a new concept a neutral name at any rate to start with. The number of new concepts required may ultimately be quite large, but most of these will be constructs which can be defined with considerable precision in terms of a much smaller number of more basic ones. This intermediate form of definition should always be made as precise as possible although the rigorous description of the basic concepts in terms of more elementary ideas may not yet be available. Who when defining the eigenvalues of a matrix is concerned with tracing the definition back to Peano's axioms?

Not very much of this will show up in the rest of this course. The reason for this is partly that it is easier, with the aid of hindsight, to preach than to practice what you preach. In part, however, the reason is that my aim is not to give an historical account of how we reached the present position but to try to convey what the position is. For this reason I have often preferred a somewhat informal approach even when mere formality would in fact have been easy.

2. Basic concepts

2.1. Assignment commands

One of the characteristic features of computers is that they have a store into which it is possible to put information and from which it can subsequently be recovered. Furthermore the act of inserting an item into the store erases whatever was in that particular area of the store before—in other words the process is one of overwriting. This leads to the assignment command which is a prominent feature of most programming languages.

The simplest forms of assignments such as

```
x := 3
x := y + 1
x := x + 1
```

lend themselves to very simple explications. ‘Set x equal to 3’, ‘Set x to be the value of y plus 1’ or ‘Add one to x’. But this simplicity is deceptive; the examples are themselves special cases of a more general form and the first explications which come to mind will not generalise satisfactorily. This situation crops up over and over again in the exploration of a new field; it is important to resist the temptation to start with a confusingly simple example.

The following assignment commands show this danger.

```
                 i := a > b → j,k            (See note 1)
              A[i] := A[a > b → j,k]
A[a > b → j, k] := A[i]
   a > b → j, k := i                       (See note 2)
```

All these commands are legal in CPL (and all but the last, apart from minor syntactic alterations, in ALGOL also). They show an increasing complexity of the expressions written on the left of the assignment. We are tempted to write them all in the general form

ε_1 := ε_2

where ε_1 and ε_2 stand for expressions, and to try as an explication something like ‘evaluate the two expressions and then do the assignment’. But this clearly will not do, as the meaning of an expression (and a name or identifier is only a simple case of an expression) on the left of an assignment is clearly different from its meaning on the right. Roughly speaking an expression on the left stands for an ‘address’ and one on the right for a ‘value’ which will be stored there. We shall therefore accept this view and say that there are two values associated with an expression or identifier. In order to avoid the overtones which go with the word ‘address’ we shall give these two values the neutral names: *L*-value for the address-like object appropriate on the left of an assignment, and *R*-value for the contents-like object appropriate for the right.

2.2. L-values and R-values

An *L*-value represents an area of the store of the computer. We call this a *location* rather than an address in order to avoid confusion with the normal store-addressing mechanism of the computer. There is no reason why a location should be exactly one machine-word in size—the objects discussed in programming languages may be, like complex or multiple precision numbers, more than one word long, or, like characters, less. Some locations are addressable (in which case their numerical machine address may be a good representation) but some are not. Before we can decide what sort of representation a general, non-addressable location should have, we should consider what properties we require of it.

The two essential features of a location are that it has a content—i.e. an associated *R*-value—and that it is in general possible to change this content by a suitable updating operation. These two operations are sufficient to characterise a general location which are consequently sometimes known as 'Load-Update Pairs' or LUPs. They will be discussed again in Section 4.1.

2.3. Definitions

In CPL a programmer can introduce a new quantity and give it a value by an initialised definition such as

```
let  p = 3.5
```

(In ALGOL this would be done by **real** `p; p := 3.5;`). This introduces a new use of the name `p` (ALGOL uses the term 'identifier' instead of name), and the best way of looking at this is that the activation of the definition causes a new location not previously used to be set up as the *L*-value of `p` and that the *R*-value 3.5 is then assigned to this location.

The relationship between a name and its *L*-value cannot be altered by assignment, and it is this fact which makes the *L*-value important. However in both ALGOL and CPL one name can have several different *L*-values in different parts of the program. It is the concept of scope (sometimes called lexicographical scope) which is controlled by the block structure which allows us to determine at any point which *L*-value is relevant.

In CPL, but not in ALGOL, it is also possible to have several names with the same *L*-value. This is done by using a special form of definition:

```
let  q ≃ p
```

which has the effect of giving the name of the same L-value as `p` (which must already exist). This feature is generally used when the right side of the definition is a more complicated expression than a simple name. Thus if `M` is a matrix, the definition

```
let  x ≃ M[2,2]
```

gives `x` the same *L*-value as one of the elements of the matrix. It is then said to be sharing with `M[2,2]`, and an assignment to `x` will have the same effect as one to `M[2,2]`.

It is worth noting that the expression on the right of this form of definition is evaluated in the *L*-mode to get an *L*-value at the time the definition is obeyed. It is this *L*-value which is associated with `x`. Thus if we have

```
let i = 2
let x ≃ M[i,i]
    i := 3
```

the *L*-value of `x` will remain that of `M[2,2]`.

`M[i,i]` is an example of an anonymous quantity i.e., an expression rather than a simple name—which has both an *L*-value and an *R*-value. There are other expressions, such as

`a+b`, which only have R-values. In both cases the expression has no name as such although it does have either one value or two.

2.4. Names

It is important to be clear about this as a good deal of confusion can be caused by differing uses of the terms. ALGOL 60 uses ‘identifier’ where we have used ‘name’, and reserves the word ‘name’ for a wholly different use concerned with the mode of calling parameters for a procedure. (See Section 3.4.3.) ALGOL X, on the other hand, appears likely to use the word ‘name’ to mean approximately what we should call an L-value, (and hence something which is a location or generalised address). The term *reference* is also used by several languages to mean (again approximately) an L-value.

It seems to me wiser not to make a distinction between the meaning of ‘name’ and that of ‘identifier’ and I shall use them interchangeably. The important feature of a name is that it has no internal structure at any rate in the context in which we are using it as a name. Names are thus atomic objects and the only thing we know about them is that given two names it is always possible to determine whether they are equal (i.e., the same name) or not.

2.5. Numerals

We use the word ‘number’ for the abstract object and ‘numeral’ for its written representation. Thus 24 and XXIV are two different numerals representing the same number. There is often some confusion about the status of numerals in programming languages. One view commonly expressed is that numerals are the ‘names of numbers’ which presumably means that every distinguishable numeral has an appropriate R-value associated with it. This seems to me an artificial point of view and one which falls foul of Occam’s razor by unnecessarily multiplying the number of entities (in this case names). This is because it overlooks the important fact that numerals in general do have an internal structure and are therefore not atomic in the sense that we said names were in the last section.

An interpretation more in keeping with our general approach is to regard numerals as R-value expressions written according to special rules. Thus for example the numeral 253 is a syntactic variant for the expression

$$2 \times 10^2 + 5 \times 10 + 3$$

while the CPL constant **8** 253 is a variant of

$$2 \times 8^2 + 5 \times 8 + 3$$

Local rules for special forms of expression can be regarded as a sort of ‘micro-syntax’ and form an important feature of programming languages. The micro-syntax is frequently used in a preliminary ‘pre-processing’ or ‘lexical’ pass of compilers to deal with the recognition of names, numerals, strings, basic symbols (e.g. boldface words in ALGOL) and similar

objects which are represented in the input stream by strings of symbols in spite of being atomic inside the language.

With this interpretation the only numerals which are also names are the single digits and these are, of course, constants with the appropriate R-value.

2.6. Conceptual model

It is sometimes helpful to have a picture showing the relationships between the various objects in the programming language, their representations in the store of a computer and the abstract objects to which they correspond. Figure 1 is an attempt to portray the conceptual model which is being used in this course.

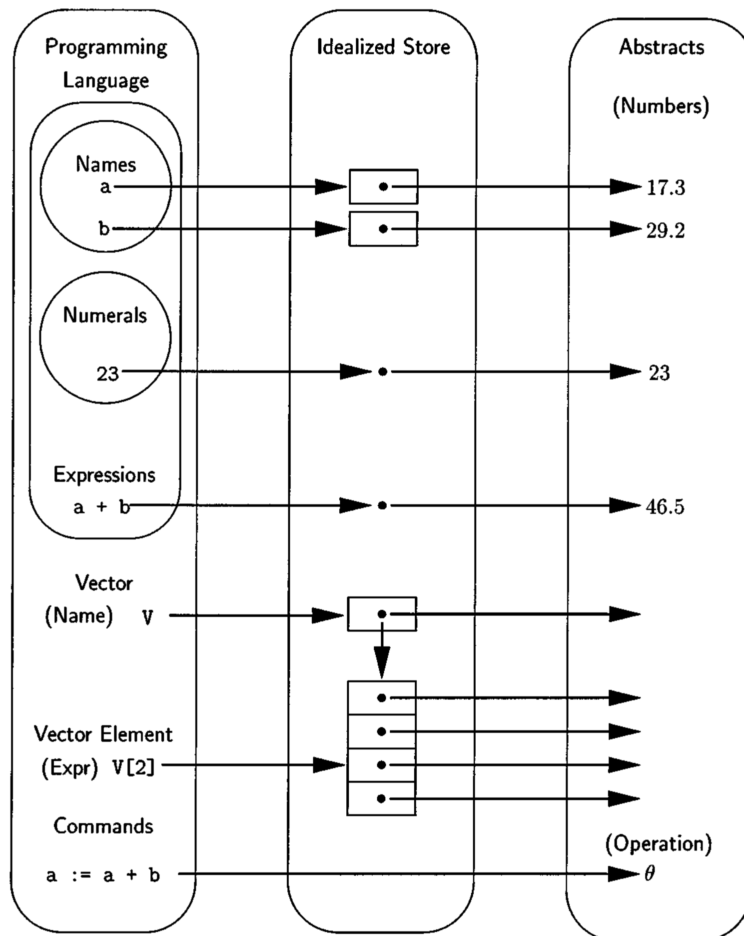


Figure 1. The conceptual model.

On the left are some of the components of the programming language. Many of these correspond to either an L-value or an R-value and the correspondence is indicated by an arrow terminating on the value concerned. Both L-values and R-values are in the idealised store, a location being represented by a box and its contents by a dot inside it. R-values without corresponding L-values are represented by dots without boxes, and R-values which are themselves locations (as, for example, that of a vector) are given arrows which terminate on another box in the idealised store.

R-values which correspond to numbers are given arrows which terminate in the right hand part of the diagram which represents the abstract objects with which the program deals.

The bottom section of the diagram, which is concerned with vectors and vector elements will be more easily understood after reading the section on compound data structures. (Section 3.7.)

3. Conceptual constructs

3.1. Expressions and commands

All the first and simplest programming language—by which I mean machine codes and assembly languages—consist of strings of commands. When obeyed, each of these causes the computer to perform some elementary operation such as subtraction, and the more elaborate results are obtained by using long sequences of commands.

In the rest of mathematics, however, there are generally no commands as such. Expressions using brackets, either written or implied, are used to build up complicated results. When talking about these expressions we use descriptive phrases such as 'the sum of x and y' or possibly 'the result of adding x to y' but never the imperative 'add x to y'.

As programming languages developed and became more powerful they came under pressure to allow ordinary mathematical expressions as well as the elementary commands. It is, after all, much more convenient to write as in CPL, `x := a(b+c)+d` than the more elementary

```
CLA b
ADD c
MPY a
ADD d
STO x
```

and also, almost equally important, much easier to follow.

To a large extent it is true that the increase in power of programming languages has corresponded to the increase in the size and complexity of the right hand sides of their assignment commands for this is the situation in which expressions are most valuable. In almost all programming languages, however, commands are still used and it is their inclusion which makes these languages quite different from the rest of mathematics.

There is a danger of confusion between the properties of expressions, not all of which are familiar, and the additional features introduced by commands, and in particular those

introduced by the assignment command. In order to avoid this as far as possible, the next section will be concerned with the properties of expressions in the absence of commands.

3.2. Expressions and evaluation

3.2.1. Values. The characteristic feature of an expression is that it has a *value*. We have seen that in general in a programming language, an expression may have two values—an L-value and an R-value. In this section, however, we are considering expressions in the absence of assignments and in these circumstances L-values are not required. Like the rest of mathematics, we shall be concerned only with R-values.

One of the most useful properties of expressions is that called by Quine [4] *referential transparency*. In essence this means that if we wish to find the value of an expression which contains a sub-expression, the only thing we need to know about the sub-expression is its value. Any other features of the sub-expression, such as its internal structure, the number and nature of its components, the order in which they are evaluated or the colour of the ink in which they are written, are irrelevant to the value of the main expression.

We are quite familiar with this property of expressions in ordinary mathematics and often make use of it unconsciously. Thus we expect the expressions

$$sin(6) \quad sin(1+5) \quad sin(30/5)$$

to have the same value. Note, however, that we cannot replace the symbol string $1+5$ by the symbol 6 in all circumstances as, for example $21+52$ is not equal to 262. The equivalence only applies to complete expressions or sub-expressions and assumes that these have been identified by a suitable syntactic analysis.

3.2.2. Environments. In order to find the value of an expression it is necessary to know the value of its components. Thus to find the value of $a+5+b/a$ we need to know the values of a and b. Thus we speak of evaluating an expression in an environment (or sometimes relative to an environment) which provides the values of components.

One way in which such an environment can be provided is by a *where-clause*.

Thus

$$a+3/a \text{ where } a=2+3/7$$
$$a+b-3/a \text{ where } a=b+2/b$$

have a self evident meaning. An alternative syntactic form which has the same effect is the initialised definition:

$$\text{let } a=2+3/7 \;\ldots\; a+3/a$$
$$\text{let } a=b+2/b \;\ldots\; a+b-3/a$$

Another way of writing these is to use λ-expressions:

$$(\lambda a.\, a+3/a)(2+3/7)$$
$$(\lambda a.\, a+b-3/a)(b+2/b)$$

All three methods are exactly equivalent and are, in fact, merely syntactic variants whose choice is a matter of taste. In each the letter a is singled out and given a value and is known as the *bound variable*. The letter b in the second expression is not bound and its value still has to be found from the environment in which the expression is to be evaluated. Variables of this sort are known as *free variables.*

3.2.3. Applicative structure. Another important feature of expressions is that it is possible to write them in such a way as to demonstrate an *applicative structure*—i.e., as an operator applied to one or more operands. One way to do this is to write the operator in front of its operand or list of operands enclosed in parentheses. Thus

$a + b$ corresponds to $+(a, b)$
$a + 3/a$ corresponds to $+(a, /(3, a))$

In this scheme a λ-expression can occur as an operator provided it is enclosed in parentheses. Thus the expression

$a + a/3$ where $a = 2 + 3/7$

can be written to show its full applicative structure as

$$\{\lambda a. + (a, /(3, a))\}(+(2, /(3, 7))).$$

Expressions written in this way with deeply nesting brackets are very difficult to read. Their importance lies only in emphasising the uniformity of applicative structure from which they are built up. In normal use the more conventional syntactic forms which are familiar and easier to read are much to be preferred—providing that we keep the underlying applicative structure at the back of our minds.

In the examples so far given all the operators have been either a λ-expression or a single symbol, while the operands have been either single symbols or sub-expressions. There is, in fact, no reason why the operator should not also be an expression. Thus for example if we use D for the differentiating operator, $D(sin) = cos$ so that $\{D(sin)\}(\times(3, a))$ is an expression with a compound operator whose value would be $cos(3a)$. Note that this is not the same as the expression $\frac{d}{dx} sin(3x)$ for $x = a$ which would be written $(D(\lambda x.sin(x(3, x))))(a)$.

3.2.4. Evaluation. We thus have a distinction between *evaluating* an operator and *applying* it to its operands. Evaluating the compound operator $D(sin)$ produces the result (or value) *cos* and can be performed quite independently of the process of applying this to the operands. Furthermore it is evident that we need to evaluate both the operator and the operands before we can apply the first to the second. This leads to the general rule for evaluating compound expressions in the operator-operand form viz:

1. Evaluate the operator and the operand(s) in any order.
2. After this has been done, apply the operator to the operand(s).

The interesting thing about this rule is that it specifies a partial ordering of the operations needed to evaluate an expression. Thus for example when evaluating

$$(a + b)(c + d/e)$$

both the additions must be performed before the multiplication, and the division before the second addition but the sequence of the first addition and the division is not specified. This partial ordering is a characteristic of algorithms which is not yet adequately reflected in most programming languages. In ALGOL, for example, not only is the sequence of commands fully specified, but the left to right rule specifies precisely the order of the operations. Although this has the advantage of precision in that the effect of any program is exactly defined, it makes it impossible for the programmer to specify indifference about sequencing or to indicate a partial ordering. The result is that he has to make a large number of logically unnecessary decisions, some of which may have unpredictable effects on the efficiency of his program (though not on its outcome).

There is a device originated by Schönfinkel [5], for reducing operators with several operands to the successive application of single operand operators. Thus, for example, instead of $+(2, p)$ where the operator $+$ takes two arguments we introduce another adding operator say $+'$ which takes a single argument such that $+'(2)$ is itself a function which adds 2 to its argument. Thus $(+'(2))(p) = +(2, p) = 2 + p$. In order to avoid a large number of brackets we make a further rule of association to the left and write $+'\ 2p$ in place of $((+'\ 2)p)$ or $(+'\ (2))(p)$. This convention is used from time to time in the rest of this paper. Initially, it may cause some difficulty as the concept of functions which produce functions as results is a somewhat unfamiliar one and the strict rule of association to the left difficult to get used to. But the effort is well worth while in terms of the simpler and more transparent formulae which result.

It might be thought that the remarks about partial ordering would no longer apply to monadic operators, but in fact this makes no difference. There is still the choice of evaluating the operator or the operand first and this allows all the freedom which was possible with several operands. Thus, for example, if p and q are sub-expressions, the evaluation of $p + q$ (or $+(p, q)$) implies nothing about the sequence of evaluation of p and q although both must be evaluated before the operator $+$ can be applied. In Schönfinkel's form this is $(+'\ p)q$ and we have the choice of evaluating $(+'\ p)$ and q in any sequence. The evaluation of $+'\ p$ involves the evaluation of $+'$ and p in either order so that once more there is no restriction on the order of evaluation of the components of the original expression.

3.2.5. Conditional expressions. There is one important form of expression which appears to break the applicative expression evaluation rule. A conditional expression such as

```
(x = 0) → 0,1/x
```

(in ALGOL this would be written `if x = 0 then 0 else 1/x`) cannot be treated as an ordinary function of three arguments. The difficulty is that it may not be possible to evaluate both arms of the condition—in this case when `x = 0` the second arm becomes undefined.

Various devices can be used to convert this to a true applicative form, and in essence all have the effect of delaying the evaluation of the arms until after the condition has been decided. Thus suppose that *If* is a function of a Boolean argument whose result is the selector *First* or *Second* so that *If*(*True*) = *First* and *If*(*False*) = *Second*, the naive interpretation of the conditional expression given above as

$$\{If(x = 0)\}(0, 1/x)$$

is wrong because it implies the evaluation of both members of the list $(0, 1/x)$ before applying the operator $\{If(x = 0)\}$. However the expression

$$[\{If(x = 0)\}(\{\lambda a.\ 0\}, \{\lambda a.\ 1/x\})]a$$

will have the desired effect as the selector function $If\ (x = 0)$ is now applied to the list $(\{\lambda a.\ 0\}, \{\lambda a.\ 1/x\})$ whose members are λ-expressions and these can be evaluated (but not applied) without danger. After the selection has been made the result is applied to a and provided a has been chosen not to conflict with other identifiers in the expression, this produces the required effect.

Recursive (self referential) functions do not require commands or loops for their definition, although to be effective they do need conditional expressions. For various reasons, of which the principal one is lack of time, they will not be discussed in this course.

3.3. Commands and sequencing

3.3.1. Variables. One important characteristic of mathematics is our habit of using names for things. Curiously enough mathematicians tend to call these things 'variables' although their most important property is precisely that they do not vary. We tend to assume automatically that the symbol x in an expression such as $3x^2 + 2x + 17$ stands for the same thing (or has the same value) on each occasion it occurs. This is the most important consequence of referential transparency and it is only in virtue of this property that we can use the where-clauses or λ-expressions described in the last section.

The introduction of the assignment command alters all this, and if we confine ourselves to the R-values of conventional mathematics we are faced with the problem of variables which actually vary, so that their value may not be the same on two occasions and we can no longer even be sure that the Boolean expression $x = x$ has the value *True*. Referential transparency has been destroyed, and without it we have lost most of our familiar mathematical tools—for how much of mathematics can survive the loss of identity?

If we consider L-values as well as R-values, however, we can preserve referential transparency as far as L-values are concerned. This is because L-values, being generalised addresses, are not altered by assignment commands. Thus the command `x := x+1` leaves the address of the cell representing `x` (L-value of `x`) unchanged although it does alter the contents of this cell (R-value of `x`). So if we agree that the values concerned are all L-values, we can continue to use where-clauses and λ-expressions for describing parts of a program which include assignments.

The cost of doing this is considerable. We are obliged to consider carefully the relationship between L and R-values and to revise all our operations which previously took R-value operands so that they take L-values. I think these problems are inevitable and although much of the work remains to be done, I feel hopeful that when completed it will not seem so formidable as it does at present, and that it will bring clarification to many areas of programming language study which are very obscure today. In particular the problems of side effects will, I hope, become more amenable.

In the rest of this section I shall outline informally a way in which this problem can be attacked. It amounts to a proposal for a method in which to formalise the semantics of a programming language. The relation of this proposal to others with the same aim will be discussed later. (Section 4.3.)

3.3.2. The abstract store. Our conceptual model of the computing process includes an abstract store which contains both L-values and R-values. The important feature of this abstract store is that at any moment it specifies the relationship between L-values and the corresponding R-values. We shall always use the symbol σ to stand for this mapping from L-values onto R-values. Thus if α is an L-value and β the corresponding R-value we shall write (remembering the conventions discussed in the last section)

$$\beta = \sigma\alpha.$$

The effect of an assignment command is to change the contents of the store of the machine. Thus it alters the relationship between L-values and R-values and so changes σ. We can therefore regard assignment as an operator on σ which produces a fresh σ. If we update the L-value α (whose original R-value in σ was β) by a fresh R-value β' to produce a new store σ', we want the R-value of α in σ' to be β', while the R-value of all other L-values remain unaltered. This can be expressed by the equation

$$(U(\alpha, \beta'))\sigma = \sigma' \text{ where } \sigma' x = (x = \alpha) \rightarrow \beta', \sigma x.$$

Thus U is a function which takes two arguments (an L-value and an R-value) and produces as a result an operator which transforms σ into σ' as defined.

The arguments of U are L-values and R-values and we need some way of getting these from the expressions written in the program. Both the L-value and the R-value of an expression such as `V[i+3]` depend on the R-value of `i` and hence on the store. Thus both must involve σ and if ε stands for a written expression in the programming language we shall write $\mathcal{L}\ \varepsilon\ \sigma$ and $\mathcal{R}\ \varepsilon\ \sigma$ for its L-value and R-value respectively.

Both $\mathcal{L}$ and $\mathcal{R}$ are to be regarded as functions which operate on segments of text of the programming language. The question of how those segments are isolated can be regarded as a matter of syntactic analysis and forms no part of our present discussion.

These functions show an application to Schönfinkel's device which is of more than merely notational convenience. The function $\mathcal{R}$, for example, shows that its result depends on both ε and σ, so it might be thought natural to write it as $\mathcal{R}(\varepsilon, \sigma)$. However by writing $\mathcal{R}\ \varepsilon\ \sigma$ and remembering that by our convention of association to the left this means $(\mathcal{R}\ \varepsilon)\sigma$ it becomes natural to consider the application of $\mathcal{R}$ to ε separately and before the application

of $\mathcal{R}\ \varepsilon$ to σ. These two phases correspond in a very convenient way to the processes of compilation, which involves manipulation of the text of the program, and execution which involves using the store of the computer. Thus the notation allows us to distinguish clearly between compile-time and execution-time processes. This isolation of the effect of σ is a characteristic of the method of semantic description described here.

It is sometimes convenient to use the contents function C defined by $C\ \alpha\ \sigma = \sigma\ \alpha$. Then if

$$\alpha = \mathcal{L}\ \varepsilon\ \sigma$$
$$\beta = \mathcal{R}\ \varepsilon\ \sigma$$

we have $\beta = C\ \alpha\ \sigma = \sigma\ \alpha$. After updating α by β', we have

$$\sigma' = U(\alpha, \beta')\sigma$$

and

$$C\ \alpha\ \sigma' = \beta'.$$

3.3.3. Commands. Commands can be considered as functions which transform σ. Thus the assignment

$$\varepsilon_1 \ := \ \varepsilon_2$$

has the effect of producing a store

$$\sigma' = U(\alpha_1, \beta_2)\sigma$$

where

$$\alpha_1 = \mathcal{L}\ \varepsilon_1\ \sigma$$

and

$$\beta_2 = \mathcal{R}\ \varepsilon_2\ \sigma$$

so that

$$\sigma' = U(\mathcal{L}\ \varepsilon_1\ \sigma, \mathcal{R}\ \varepsilon_2\ \sigma)\sigma$$

and if θ is the function on σ which is equivalent to the original command we have

$$\sigma' = \theta\sigma$$

where

$$\theta = \lambda\ \sigma.\ U(\mathcal{L}\ \varepsilon_1\ \sigma, \mathcal{R}\ \varepsilon_2\ \sigma)\sigma$$

Sequences of commands imply the successive application of sequences of θ's. Thus, for example, if γ_1, γ_2, γ_3 are commands and θ_1, θ_2, θ_3 the equivalent functions on σ, the command sequence (or compound command)

$$\gamma_1\,;\gamma_2\,;\gamma_3\,;$$

applied to a store σ will produce a store

$$\begin{aligned}\sigma' &= \theta_3(\theta_2(\theta_1\ \sigma))\\ &= (\theta_3 \cdot \theta_2 \cdot \theta_1)\sigma\end{aligned}$$

where $f \cdot g$ is the function product of f and g.

Conditional commands now take a form similar to that of conditional expressions. Thus the command

$$\begin{aligned}&\texttt{Test}\ \varepsilon_1\ \texttt{If so do}\ \ \gamma_1\\ &\qquad\quad\ \texttt{If not do}\ \ \gamma_2\end{aligned}$$

corresponds to the operator

$$\lambda\sigma.\ \mathit{If}(\mathcal{R}\ \varepsilon_1\ \sigma)(\theta_1, \theta_2)\sigma$$

where θ_1 and θ_2 correspond to γ_1 and γ_2.

Conditional expressions can also be treated more naturally. The dummy argument introduced in the last section to delay evaluation can be taken to be σ with considerable advantages in transparency. Thus

$$\mathcal{R}(\varepsilon_1 \rightarrow \varepsilon_2, \varepsilon_3)\sigma = \mathit{If}(\mathcal{R}\ \varepsilon_1\ \sigma)(\mathcal{R}\ \varepsilon_2, \mathcal{R}\ \varepsilon_3)\sigma$$

and

$$\mathcal{L}(\varepsilon_1 \rightarrow \varepsilon_2, \varepsilon_3)\sigma = \mathit{If}(\mathcal{R}\ \varepsilon_1\ \sigma)(\mathcal{L}\ \varepsilon_2, \mathcal{L}\ \varepsilon_3)\sigma$$

Informally $\mathcal{R}\,\varepsilon_2$ and $\mathcal{L}\,\varepsilon_2$ correspond to the compiled program for evaluating ε_2 in the R-mode or L-mode respectively. The selector $\mathit{If}(\mathcal{R}\ \varepsilon_1\ \sigma)$ chooses between these at execution time on the basis of the R-value of ε_1 while the final application to σ corresponds to running the chosen piece of program.

If we consider commands as being functions operating on σ, loops and cycles are merely recursive functions also operating on σ. There is, however, no time to go further into these in this course.

An interesting feature of this approach to the semantics of programming languages is that all concept of sequencing appears to have vanished. It is, in fact, replaced by the partially ordered sequence of functional applications which is specified by λ-expressions.

In the remaining sections we shall revert to a slightly less formal approach, and try to isolate some important 'high level' concepts in programming languages.

3.4. Definition of functions and routines

3.4.1. Functional abstractions. In order to combine programs hierarchically we need the process of functional abstraction. That is to say that we need to be able to form functions from expressions such as

```
let  f[x] = 5x²+ 3x + 2/x³
```

This could be thought of as defining `f` to be a function and giving it an initial value. Thus the form of definition given above is merely a syntactic variant of the standard form of definition (which has the quantity defined alone on the left side)

```
let  f = λx. 5x²+ 3x + 2/x³
```

This form makes it clear that it is `f` which is being defined and that `x` is a bound or dummy variable and could be replaced by any other non-clashing name without altering the value given to `f`.

3.4.2. Parameter calling modes. When the function is used (or called or applied) we write `f[`ε`]` where ε can be an expression. If we are using a referentially transparent language all we require to know about the expression ε in order to evaluate `f[`ε`]` is its value. There are, however, two sorts of value, so we have to decide whether to supply the R-value or the L-value of ε to the function `f`. Either is possible, so that it becomes a part of the definition of the function to specify for each of its bound variables (also called its formal parameters) whether it requires an R-value or an L-value. These alternatives will also be known as calling a parameter by *value* (R-value) or *reference* (L-value).

Existing programming languages show a curious diversity in their modes of calling parameters. FORTRAN calls all its parameters by reference and has a special rule for providing R-value expressions such as $a+b$ with a temporary L-value. ALGOL 60, on the other hand, has two modes of calling parameters (specified by the programmer): *value* and *name*. The ALGOL call by *value* corresponds to call by R-value as above; the call by name,[3] however, is quite different (and more complex). Only if the actual parameter (i.e., the expression ε above) is a simple variable is the effect the same as a call by reference. This incompatibility in their methods of calling parameters makes it difficult to combine the two languages in a single program.

3.4.3. Modes of free variables. The obscurity which surrounds the modes of calling the bound variables becomes much worse when we come to consider the free variables of a function. Let us consider for a moment the very simple function

```
f[x] = x + a
```

where `a` is a free variable which is defined in the surrounding program. When `f` is defined we want in some way to incorporate `a` into its definition, and the question is do we use its

R-value or its *L*-value? The difference is illustrated in the following pair of CPL programs. (In CPL a function definition using = takes its free variables by *R*-value and one using ≡ takes them by *L*-value.)

Free variable by *R*-value	Free variable by *L*-value
`let a = 3`	`let a = 3`
`let f[x] = x + a`	`let f[x] ≡ x + a`
`... (f[5] = 8)...`	`...(f[5] = 8)...`
`a := 10`	`a := 10`
`... (f[5] = 8)...`	`...(f[5] = 15)...`

The expressions in parentheses are all Booleans with the value **true**.

Thus the first example freezes the current *R*-value of `a` into the definition of `f` so that it is unaffected by any future alterations (by assignment) to `a`, while the second does not. It is important to realize, however, that even the second example freezes something (i.e., the *L*-value of `a`) into the definition of `f`. Consider the example

```
let  a = 3
let  f[x] ≡ x + a
      ...(f[5] = 8),(a = 3) ...
      § let a = 100
       ...(f[5] = 8),(a = 100) ...
           a := 10
       ...(f[5] = 8),(a = 10) ...
       ...........§
      ...(f[5] = 8),(a = 3) ...
```

Here there is an inner block enclosed in the statement brackets § .... § (which corresponds to **begin** and **end** in ALGOL), and inside this an entirely fresh `a` has been defined. This forms a hole in the scope of the original `a` in which it continues to exist but becomes inaccessible to the programmer. However as its *L*-value was incorporated in the definition of `f`, it is the original `a` which is used to find `f[5]`. Note that assignments to `a` in the inner block affect only the second `a` and so do not alter `f`.

It is possible to imagine a third method of treating free variables (though there is nothing corresponding for bound variables) in which the locally current meaning of the variables is used, so that in the example above the second and third occurrences of `f[5]` would have the values `105` and `15` respectively. I believe that things very close to this exist in LISP2 and are known as fluid variables. The objection to this scheme is that it appears to destroy referential transparency irrevocably without any apparent compensating advantages.

In CPL the facilities for specifying the mode of the free variables are considerably coarser than the corresponding facilities for bound variables. In the case of bound variables the mode has to be specified explicitly or by default for each variable separately. For the free variables, however, it is only possible to make a single specification which covers all the free variables, so that they must all be treated alike. The first method is more flexible and provides greater power for the programmer, but is also more onerous (although good

default conventions can help to reduce the burden); the second is much simpler to use but sometimes does not allow a fine enough control. Decisions between methods of this sort are bound to be compromises reflecting the individual taste of the language designer and are always open to objection on grounds of convenience. It is no part of a discussion on the fundamental concepts of programming languages to make this sort of choice—it should rest content with pointing out the possibilities.

A crude but convenient method of specification, such as CPL uses for the mode of the free variables of a function, becomes more acceptable if there exists an alternative method by which the finer distinctions can be made, although at the cost of syntactic inelegance. Such a method exists in CPL and involves using an analogue to the *own* variables in ALGOL 60 proposed by Landin [6].

3.4.4. Own variables. The idea behind *own* variables is to allow some private or secret information which is in some way protected from outside interference. The details were never very clearly expressed in ALGOL and at least two rival interpretations sprang up, neither being particularly satisfactory. The reason for this was that *owns* were associated with blocks whereas, as Landin pointed out, the natural association is with a procedure body. (In this case of functions this corresponds to the expression on the right side of the function definition.)

The purpose is to allow a variable to preserve its value from one application of a function to the next—say to produce a pseudo-random number or to count the number of times the function is applied. This is not possible with ordinary local variables defined within the body of the function as all locals are redefined afresh on each application of the function. It would be possible to preserve information in a non-local variable—i.e., one whose scope included both the function definition and all its applications, but it would not then be protected and would be accessible from the whole of this part of the program. What we need is a way of limiting the scope of a variable to be the definition only. In CPL we indicate this by using the word in to connect the definition of the own variable (which is usually an initialised one) with the function definitions it qualifies.

In order to clarify this point programs using each of the three possible scopes (non-local, own and local) are written below in three ways viz. Normal CPL, CPL mixed with λ-expressions to make the function definition in its standard form, and finally in pure λ-expressions. The differences in the scope rules become of importance only when there is a clash of names, so in each of these examples one or both of the names `a` and `x` are used twice. In order to make it easy to determine which is which, a prime has been added to one of them. However, the scope rules imply that if all the primes were omitted the program would be unaltered.

1. Non-local variable

CPL

```
let  a' = 6
let  x' = 10
let  a  = 3/x'
let  f[x] ≡ x + a
     ....  f[a] ....
```

Mixed
```
let  a' = 6
let  x' = 10
let  a  = 3/x'
let  f ≡ λx. x + a
     ....  f[a] ....
```
Pure λ
```
{λa'.{λx'.{λa.{λf.f a}[λx.x + a]}[3/x']}10}6
```

2. Own variable

CPL
```
let  a' = 6
let  x' = 10
let  a  = 3/x'
in  f[x] ≡ x + a
 ....  f[a'] ....
```
Mixed
```
let  a' = 6
let  x' = 10
let  f  ≡ {λa.λx.x + a}[3/x']
 ....  f[a'] ....
```
Pure λ
```
{λa'.{λx'.{λf.f a'}[{λa.λx.x + a}[3/x']]}10}6
```

3. Local variable

CPL
```
let  a' = 6
let  x' = 10
let  f[x] ≡ (x + a where a = 3/x)
     ....  f[a'] ....
```
Mixed
```
let  a' = 6
let  x' = 10
let  f  ≡ λx.{λa.x + a}[3/x]
     ....  f[a'] ....
```
Pure λ
```
{λa'.{λx'.{λf.f a'}[λx.{λa.x + a}[3/x]]}10}6
```

We can now return to the question of controlling the mode of calling the free variables of a function. Suppose we want to define `f[x]` to be `ax + b + c` and use the R-value of `a` and `b` but the L-value of `c`. A CPL program which achieves this effect is

```
let a' = a and b' = b
in f[x] ≡ a'x + b' + c
  ....
```

(Again the primes may be omitted without altering the effect.)

The form of definition causes the L-values of `a'`, `b'` and `c` to be used, while the definition of the variables `a'` and `b'` ensures that these are given fresh L-values which are initialised to the R-values of `a` and `b`. As they are own variables, they are protected from any subsequent assignments to `a` and `b`.

3.4.5. Functions and routines. We have so far discussed the process of functional abstraction as applied to expressions. The result is called a *function* and when applied to suitable arguments it produces a value. Thus a function can be regarded as a complicated sort of expression. The same process of abstraction can be applied to a command (or sequence of commands), and the result is know in CPL as a *routine*. The application of a routine to a suitable set of arguments is a complicated command, so that although it affects the store of the computer, it produces no value as a result.

Functions and routines are as different in their nature as expressions and commands. It is unfortunate, therefore, that most programming languages manage to confuse them very successfully. The trouble comes from the fact that it is possible to write a function which also alters the store, so that it has the effect of a function and a routine. Such functions are sometimes said to have side effects and their uncontrolled use can lead to great obscurity in the program. There is no generally agreed way of controlling or avoiding the side effects of functions, and most programming languages make no attempt to deal with the problem at all—indeed their confusion between routines and functions adds to the difficulties.

The problem arises because we naturally expect referential transparency of R-values in expressions, particularly those on the right of assignment commands. This is, I think, a very reasonable expectation as without this property, the value of the expression is much harder to determine, so that the whole program is much more obscure. The formal conditions on expressions which have to be satisfied in order to produce this R-value referential transparency still need to be investigated. However in special cases the question is usually easy to decide and I suggest that as a matter of good programming practice it should always be done. Any departure of R-value referential transparency in a R-value context should either be eliminated by decomposing the expression into several commands and simpler expressions, or, if this turns out to be difficult, the subject of a comment.

3.4.6. Constants and variables. There is another approach to the problem of side effects which is somewhat simpler to apply, though it does not get round all the difficulties. This is, in effect, to turn the problem inside out and instead of trying to specify functions and expressions which have no side effect to specify objects which are immune from any possible side effect of others. There are two chief forms which this protection can take which can roughly be described as hiding and freezing. Their inaccessibility (by reason of the scope rules) makes them safe from alteration except from inside the body of the function or routine they qualify. We shall be concerned in this section and the next with different forms of protection by freezing.

The characteristic thing about variables is that their R-values can be altered by an assignment command. If we are looking for an object which is frozen, or invariant, an obvious possibility is to forbid assignments to it. This makes it what in CPL we call a *constant*. It has an L-value and R-value in the ordinary way, but applying the update function to it either has no effect or produces an error message. Constancy is thus an attribute of an L-value, and is, moreover, an invariant attribute. Thus when we create a new L-value, and in particular when we define a new quantity, we must decide whether it is a constant or a variable.

As with many other attributes, it is convenient in a practical programming language to have a default convention—if the attribute is not given explicitly some conventional value is assumed. The choice of these default conventions is largely a matter of taste and judgement,

but it is an important one as they can affect profoundly both the convenience of the language and the number of slips made by programmers. In the case of constancy, it is reasonable that the ordinary quantities, such as numbers and strings, should be variable. It is only rather rarely that we want to protect a numerical constant such as `Pi` from interference. Functions and routines, on the other hand, are generally considered to be constants. We tend to give them familiar or mnemonic names such a `CubeRt` or `LCM` and we would rightly feel confused by an assignment such as `CubeRt := SqRt`. Routines and functions are therefore given the default attribute of being a constant.

3.4.7. Fixed and free. The constancy or otherwise of a function has no connection with the mode in which it uses its free variables. If we write a definition in its standard form such as

```
let f ≡ λx. x + a
```

we see that this has the effect of initialising `f` with a λ-expression. The constancy of `f` merely means that we are not allowed to assign to it. The mode of its free variables (indicated by ≡) is a property of the λ-expression.

Functions which call their free variables by reference (L-value) are liable to alteration by assignments to their free variables. This can occur either inside or outside the function body, and indeed, even if the function itself is a constant. Furthermore they cease to have a meaning if they are removed from an environment in which their free variables exist. (In ALGOL this would be outside the block in which their free variables were declared.) Such functions are called *free* functions.

The converse of a free function is a *fixed* function. This is defined as a function which either has no free variables, or if it has, whose free variables are all both constant and fixed. The crucial feature of a fixed function is that it is independent of its environment and is always the same function. It can therefore be taken out of the computer (e.g., by being compiled separately) and reinserted again without altering its effect.

Note that fixity is a property of the λ-expression—i.e., a property of the R-value, while constancy is a property of the L-value. Numbers, for example, are always fixed as are all 'atomic' R-values (i.e., ones which cannot be decomposed into smaller parts). It is only in composite objects that the distinction between fixed and free has any meaning. If such an object is fixed, it remains possible to get at its component parts, but not to alter them. Thus, for example, a fixed vector is a look-up table whose entries will not (cannot) be altered, while a free vector is the ordinary sort of vector in which any element may be changed if necessary.

3.4.8. Segmentation. A fixed routine or function is precisely the sort of object which can be compiled separately. We can make use of this to allow the segmentation of programs and their subsequent assembly even when they do communicate with each other through free variables. The method is logically rather similar to the FORTRAN Common variables.

Suppose `R[x]` is a routine which uses `a`, `b`, and `c` by reference as free variables. We can define a function `R'[a,b,c]` which has as formal parameters all the free variables of `R` and

whose result is the routine R[x]. Then R' will have no free variables and will thus be a fixed function which can be compiled separately.

The following CPL program shows how this can be done:

```
§ let  R'[ref a,b,c] = value of
       § let  R[x] be
            § ... a,b,c ...
                (body of R) §
          result is  R §

WriteFixedFunction [R']

finish  §
```

The command WriteFixedFunction [R'] is assumed to output its argument in some form of relocatable binary or otherwise so that it can be read in later by the function ReadFixedFunction.

If we now wish to use R in an environment where its free variables are to be p, q and r and its name is to be S we can write

```
§ let  p,q,r = ... (Setting up the environment)
  let  S' = ReadFixedFunction
  let  S  = S'[p,q,r]

     .... S[u] ....    §
```

In this way S' becomes the same function as R' and the call S'[p,q,r], which use the L-values of p, q and r, produces S which is the original routine R but with p, q and r as its free variables instead of a, b and c.

One advantage of this way of looking at segmentation is that it becomes a part of the ordinary programming language instead of a special ad hoc device. An unfamiliar feature will be its use of a function R' which has as its result another function or routine. This is discussed in more detail in the next section.

3.5. Functions and routines as data items.

3.5.1. First and second class objects. In ALGOL a real number may appear in an expression or be assigned to a variable, and either may appear as an actual parameter in a procedure call. A procedure, on the other hand, may only appear in another procedure call either as the operator (the most common case) or as one of the actual parameters. There are no other expressions involving procedures or whose results are procedures. Thus in a sense procedures in ALGOL are second class citizens—they always have to appear in person and can never be represented by a variable or expression (except in the case of a formal parameter), while we can write (in ALGOL still)

```
(if  x > 1 then a else b) + 6
```

when a and b are reals, we cannot correctly write

```
(if  x > 1 then sin else cos)(x)
```

nor can we write a type procedure (ALGOL's nearest approach to a function) with a result which is itself a procedure.

Historically this second class status of procedures in ALGOL is probably a consequence of the view of functions taken by many mathematicians: that they are constants whose name one can always recognise. This second class view of functions is demonstrated by the remarkable fact that ordinary mathematics lacks a systematic notation for functions. The following example is given by Curry [7, p. 81].

Suppose P is an operator (called by some a 'functional') which operates on functions. The result of applying P to a function $f(x)$ is often written $P[f(x)]$. What then does $P[f(x+1)]$ mean? There are two possible meanings (a) we form $g(x) = f(x+1)$ and the result is $P[g(x)]$ or (b) we form $h(x) = P[f(x)]$ and the result is $h(x+1)$. In many cases these are the same but not always. Let

$$P[f(x)] = \begin{cases} \dfrac{f(x) - f(0)}{x} & \text{for } x \neq 0 \\ f'(x) & \text{for } x = 0 \end{cases}$$

Then if $f(x) = x^2$

$$P[g(x)] = P[x^2 + 2x + 1] = x + 2$$

while

$$h(x) = P[f(x)] = x$$

so that $h(x+1) = x + 1$.

This sort of confusion is, of course, avoided by using λ-expressions or by treating functions as first class objects. Thus, for example, we should prefer to write $(P[f])[x]$ in place of $P[f(x)]$ above (or, using the association rule $P[f][x]$ or even $P\ f\ x$). The two alternatives which were confused would then become

$$P\ g\ x \quad \text{where } g\ x = f(x+1)$$

and $P\ f\ (x+1)$.

The first of these could also be written $P(\lambda x.\ f(x+1))x$.

I have spent some time on this discussion in spite of its apparently trivial nature, because I found, both from personal experience and from talking to others, that it is remarkably difficult to stop looking on functions as second class objects. This is particularly unfortunate as many of the more interesting developments of programming and programming languages come from the unrestricted use of functions, and in particular of functions which have functions as a result. As usual with new or unfamiliar ways of looking at things, it is harder for the teachers to change their habits of thought than it is for their pupils to follow them. The

difficulty is considerably greater in the case of practical programmers for whom an abstract concept such as a function has little reality until they can clothe it with a representation and so understand what it is that they are dealing with.

3.5.2. Representation of functions. If we want to make it possible to assign functions we must be clear what are their L-values and R-values. The L-value is simple—it is the location where the R-value is stored—but the R-value is a little more complicated. When a function is applied it is the R-value which is used, so that at least sufficient information must be included in the R-value of a function to allow it to be applied. The application of a function to its arguments involves the evaluation of its defining expression after supplying the values of its bound variables from the argument list. To do this it is necessary to provide an environment which supplies the values of the free variables of the function.

Thus the R-value of a function contains two parts—a rule for evaluating the expression, and an environment which supplies its free variables. An R-value of this sort will be called a *closure*. There is no problem in representing the rule in a closure, as the address of a piece of program (i.e., a subroutine entry point) is sufficient. The most straightforward way of representing the environment part is by a pointer to a *Free Variable List* (FVL) which has an entry for each free variable of the function. This list is formed when the function is initially defined (more precisely when the λ-expression which is the function is evaluated, usually during a function definition) and at this time either the R-value or the L-value of each of the free variables is copied into the FVL. The choice of R- or L-value is determined by the mode in which the function uses its free variables. Thus in CPL functions defined by $=$ have R-values in their FVL while functions defined by $\equiv$ have L-values. Own variables of the kind discussed in the previous section can also be conveniently accommodated in the FVL.

The concept of a closure as the R-value of a function makes it easier to understand operations such as passing a function as a parameter, assigning to a variable of type function, or producing a function as the value of an expression or result of another function application. In each case the value concerned, which is passed on or assigned, is a closure consisting of a pair of addresses.

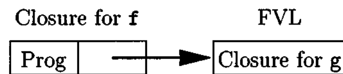
It is important to note that a function closure does not *contain* all the information associated with the function, it merely *gives access* (or points) to it, and that as the R-value of a function is a closure, the same applies to it. This is in sharp distinction to the case of data items such as *reals* or *integers* whose R-value is in some sense atomic or indivisible and contains all the information about the data items.

This situation, where some of the information is in the FVL rather than the R-value, is quite common and occurs not only with functions and routines, but also with labels, arrays and all forms of compound data structure. In these cases it is meaningful to ask if the information which is in the FVL or accessible through it is alterable or whether it cannot be changed at all, and this property provides the distinction between free and fixed objects.

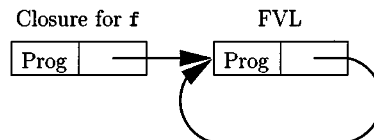
A function which has been defined recursively so that the expression representing it includes at least one mention of its own name, can also be represented rather simply by making use of closures. Suppose, for example, we take the non-recursive function

```
let  f[x]= (x=0) → 1,x*g[x-1]
```

This has a single free variable, the function g which is taken by R-value. Thus the closure for `f` would take the form



If we now identify g with `f`, so that the function becomes the recursively defined factorial, all we need to do is to ensure that the FVL contains the closure for `f`, Thus it will take the form



so that the FVL, which now contains a copy of the closure for `f`, in fact points to itself. It is a characteristic feature of recursively defined functions of all sorts that they have some sort of a closed loop in their representation.

3.6. Types and polymorphism

3.6.1. Types. Most programming languages deal with more than one sort of object—for example with integers and floating point numbers and labels and procedures. We shall call each of these a different *type* and spend a little time examining the concept of type and trying to clarify it.

A possible starting point is the remark in the CPL Working Papers [3] that "The Type of an object determines its representation and constrains the range of abstract object it may be used to represent. Both the representation and the range may be implementation dependent". This is true, but not particularly helpful. In fact the two factors mentioned—representation and range—have very different effects. The most important feature of a representation is the space it occupies and it is perfectly possible to ignore types completely as far as representation and storage is concerned if all types occupy the same size of storage. This is in fact the position of most assembly languages and machine code—the only differences of type encountered are those of storage size.

In more sophisticated programming languages, however, we use the type to tell us what sort of object we are dealing with (i.e., to restrict its range to one sort of object). We also expect the compiling system to check that we have not made silly mistakes (such as multiplying two labels) and to interpret correctly ambiguous symbols (such as `+`) which mean different things according to the types of their operands. We call ambiguous operators of this sort *polymorphic* as they have several forms depending on their arguments.

The problem of dealing with polymorphic operators is complicated by the fact that the range of types sometimes overlap. Thus for example `3` may be an *integer* or a *real* and it may be necessary to change it from one type to the other. The functions which perform this operation are known as *transfer functions* and may either be used explicitly by the programmer, or, in some systems, inserted automatically by the compiling system.

3.6.2. Manifest and latent. It is natural to ask whether type is an attribute of an L-value or of an R-value—of a location or of its content. The answer to this question turns out to be a matter of language design, and the choice affects the amount of work, which can be done when a program is compiled as opposed to that which must be postponed until it is run.

In CPL the type is a property of an expression and hence an attribute of both its L-value and its R-value. Moreover L-values are invariant under assignment and this invariance includes their type. This means that the type of any particular written expression is determined solely by its position in the program. This in turn determines from their scopes which definitions govern the variables of the expression, and hence give their types. An additional rule states that the type of the result of a polymorphic operator must be determinable from a knowledge of the types of its operands without knowing their values. Thus we must be able to find the type of `a + b` without knowing the value of either `a` or `b` provided only that we know both their types.[4]

The result of these rules is that the type of every expression can be determined at compile time so that the appropriate code can be produced both for performing the operations and for storing the results.

We call attributes which can be determined at compile time in this way *manifest*; attributes that can only be determined by running the program are known as *latent*. The distinction between manifest and latent properties is not very clear cut and depends to a certain extent on questions of taste. Do we, for example, take the value of `2 + 3` to be manifest or latent? There may well be a useful and precise definition—on the other hand there may not. In either case at present we are less interested in the demarkation problem than in properties which are clearly on one side or other of the boundary.

3.6.3. Dynamic type determination. The decision in CPL to make types a manifest property of expressions was a deliberate one of language design. The opposite extreme is also worth examining. We now decide that types are to be attributes of R-values only and that any type of R-value may be assigned to any L-value. We can settle difficulties about storage by requiring that all types occupy the same storage space, but how do we ensure that the correct operations are performed for polymorphic operators? Assembly languages and other 'simple' languages merely forbid polymorphism. An alternative, which has interesting features, is to carry around with each R-value an indication of its type. Polymorphic operators will then be able to test this dynamically (either by hardware or program) and choose the appropriate version.

This scheme of dynamic type determination may seem to involve a great deal of extra work at run time, and it is true that in most existing computers it would slow down programs considerably. However the design of central processing units is not immutable and logical hardware of the sort required to do a limited form of type determination is relatively cheap. We should not reject a system which is logically satisfactory merely because today's computers are unsuitable for it. If we can prove a sufficient advantage for it machines with the necessary hardware will ultimately appear even if this is rather complicated; the introduction of floating-point arithmetic units is one case when this has already happened.

3.6.4. Polymorphism. The difficulties of dealing with polymorphic operators are not removed by treating types dynamically (i.e., making them latent). The problems of choosing

the correct version of the operator and inserting transfer functions if required remain more or less the same. The chief difference in treating types as manifest is that this information has to be made available to the compiler. The desire to do this leads to an examination of the various forms of polymorphism. There seem to be two main classes, which can be called ad hoc polymorphism and parametric polymorphism.

In ad hoc polymorphism there is no single systematic way of determining the type of the result from the type of the arguments. There may be several rules of limited extent which reduce the number of cases, but these are themselves ad hoc both in scope and content. All the ordinary arithmetic operators and functions come into this category. It seems, moreover, that the automatic insertion of transfer functions by the compiling system is limited to this class.

Parametric polymorphism is more regular and may be illustrated by an example. Suppose `f` is a function whose argument is of type α and whose results is of β (so that the type of `f` might be written $\alpha \Rightarrow \beta$), and that `L` is a list whose elements are all of type α (so that the type of `L` is α **`list`**). We can imagine a function, say `Map`, which applies `f` in turn to each member of `L` and makes a list of the results. Thus `Map[f,L]` will produce a β **`list`**. We would like `Map` to work on all types of list provided f was a suitable function, so that `Map` would have to be polymorphic. However its polymorphism is of a particularly simple parametric type which could be written

$$(\alpha \Rightarrow \beta, \alpha\ \mathbf{list}) \Rightarrow \beta\ \mathbf{list}$$

where α and β stand for any types.

Polymorphism of both classes presents a considerable challenge to the language designer, but it is not one which we shall take up here.

3.6.5. Types of functions. The type of a function includes both the types and modes of calling of its parameters and the types of its results. That is to say, in more mathematical terminology, that it includes the domain and the range of the function. Although this seems a reasonable and logical requirement, it makes it necessary to introduce the parametric polymorphism discussed above as without it functions such as `Map` have to be redefined almost every time they are used.

Some programming languages allow functions with a variable number of arguments; those are particularly popular for input and output. They will be known as *variadic* functions, and can be regarded as an extreme form of polymorphic function.[5]

A question of greater interest is whether a polymorphic function is a first class object in the sense of Section 3.5.1. If it is, we need to know what type it is. This must clearly include in some way the types of all its possible versions. Thus the type of a polymorphic function includes or specifies in some way the nature of its polymorphism. If, as in CPL, the types are manifest, all this information must be available to the compiler. Although this is not impossible, it causes a considerable increase in the complexity of the compiler and exerts a strong pressure either to forbid programmers to define new polymorphic functions or even to reduce all polymorphic functions to second class status. A decision on these points has not yet been taken for CPL.

3.7. *Compound data structures*

3.7.1. List processing. While programming was confined to problems of numerical analysis the need for general forms of data structure was so small that it was often ignored. For this reason ALGOL, which is primarily a language for numerical problems, contains no structure other than arrays. COBOL, being concerned with commercial data processing, was inevitably concerned with larger and more complicated structures. Unfortunately, however, the combined effect of the business man's fear of mathematics and the mathematician's contempt for business ensured that this fact had no influence on the development of general programming languages.

It was not until mathematicians began using computers for non-numerical purposes—initially in problems connected with artificial intelligence—that any general forms of compound data structure for programming languages began to be discussed. Both IPL V and LISP used data structures built up from lists and soon a number of other 'List Processing' languages were devised.

The characteristic feature of all these languages is that they are designed to manipulate more or less elaborate structures, which are built up from large numbers of components drawn from a very limited number of types. In LISP, for instance, there are only two sorts of object, an *atom* and a *cons-word* which is a doublet. The crucial feature is that each member of a doublet can itself be either an atom or another cons-word. Structures are built up by joining together a number of cons-words and atoms.

This scheme of building up complex structures from numbers of similar and much simpler elements has a great deal to recommend it. In some sense, moreover, the doublet of LISP is the simplest possible component from which to construct a structure and it is certainly possible to represent any other structure in terms of doublets. However from the practical point of view, not only for economy of implementation but also for convenience in use, the logically simplest representation is not always the best.

The later list processing languages attempted to remedy this by proposing other forms of basic building block with more useful properties, while still, of course, retaining the main plan of using many relatively simple components to form a complex structure. The resulting languages were generally very much more convenient for some classes of problems (particularly those they had been designed for) and much less suitable (possibly on grounds of efficiency) for others. They all, however, had an ad hoc look about them and arguments about their relative merits seemed somewhat unreal.

In about 1965 or 1966 interest began to turn to more general schemes for compound data structures which allowed the programmer to specify his own building blocks in some very general manner rather than having to make do with those provided by the language designer. Several such schemes are now around and in spite of being to a large extent developed independently they have a great deal in common—at least as far as the structures described in the next section as nodes are concerned. In order to illustrate these ideas, I shall outline the scheme which will probably be incorporated in CPL.

3.7.2. Nodes and elements. The building blocks from which structures are formed are known as *nodes*. Nodes may be of many types and the definition of a new node is in fact

the definition of a new programmer-defined type in the sense of section 3.6. A node may be defined to consist of one or more components; both the number and the type of each component is fixed by the definition of the node. A component may be of any basic or programmer-defined type (such as a node), or may be an *element*. This represents a data object of one of a limited number of types; the actual type of object being represented is determined dynamically. An element definition also forms a new programmer-defined type in the sense of Section 3.6 and it also specifies which particular data types it may represent.

Both node and element definitions are definitions of new types, but at the same time they are used to form certain basic functions which can be used to operate on and construct individual objects of these types. Compound data structures may be built up from individuals of these types by using these functions.

The following example shows the node and element definitions which allow the lists of LISP to beformed.

```
node Cons is LispList : Car
       with  Cons : Cdr
element LispList  is Atom
                  or Cons
node Atom is string PrintName
         with Cons : PropertyList
```

These definitions introduce three new types: `Cons` and `Atom`, which are nodes, and `LispList` which is an element. They also define the basic selector and constructor functions which operate on them. These functions have the following effect.

If `x` is an object of type `Cons`, it has two components associated with it; the first, which is of manifest type `LispList` is obtained by applying the appropriate *selector function* `Car` to `x`, thus `Car[x]` is the first component of `x` and is of type LispList. The second component of `x` is `Cdr[x]` and is an object of type `Cons`.

If `p` is an object of type `LispList` and `q` is an object of type `Cons`, we can form a fresh node of type `Cons` whose first component is `p` and whose second component is `q` by using the *constructor function* `Cons[p,q]` which always has the same name as the node type. Thus we have the basic identities

```
Car[Cons[p,q]]= p
Cdr[Cons[p,q]]= q
```

In an exactly similar way the definition of the node `Atom` will also define the two selector functions `PrintName` and `PropertyList` and the constructor function `Atom`.

The number of components of a node is not limited to two—any non-zero number is allowed. There is also the possibility that any component may be the special object `NIL`.This can be tested for by the system predicate `Null`. Thus, for example, if end of a list is indicated by a `NIL` second component, we can test for this by the predicate `Null[Cdr[x]]`.

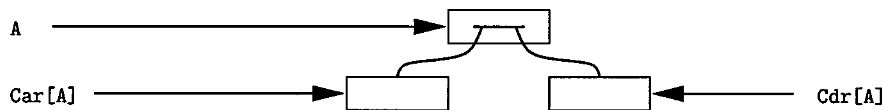
There is also a constructor function associated with an element type. Thus, for example if `n` is an atom, `LispList[n]` is an object of type `LispList` dynamically marked as being an atom and being in fact the atom `n`. There are two general system functions which apply

to elements, both are concerned with finding their dynamically current type. The function `Type[p]` where `p` is a `LispList` will have the result either `Atom` or `Cons` according to the current type of `p`. In a similar way the system predicate `Is[Atom,p]` will have the value **`true`** if `p` is dynamically of type `Atom`.

These definitions give the basic building block of LISP using the same names with the exception of `Atom`. In Lisp `Atom[p]` is the predicate which would be written here as `Is[Atom,p]`. We use the function `Atom` to construct a new atom from a `PrintName` and a `PropertyList`.

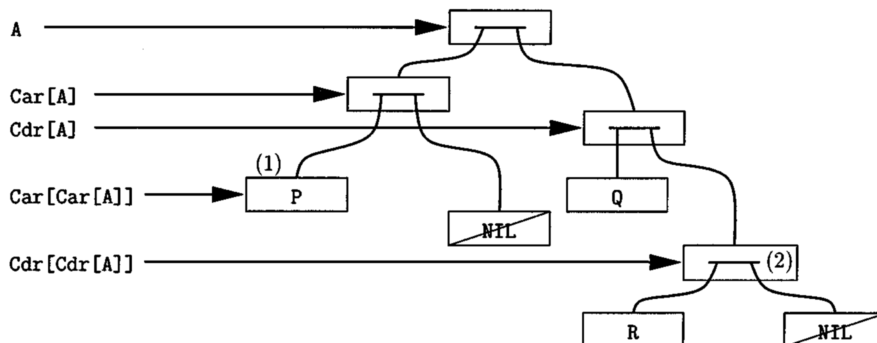
3.7.3. Assignments. In order to understand assignments in compound data structure we need to know what are the L- and R-values of nodes and their components.

Let us suppose that `A` is a *variable* of type `Cons`—i.e., that `A` is a named quantity to which we propose to assign objects of type `Cons`. The L-value of `A` presents no problems; like any other L-value it is the location where the R-value of `A` is stored. The R-value of `A` must give access to the two components of `A` (`Car[A]` and `Cdr[A]`)—i.e., it must give their L-values or locations. Thus, we have the diagram:



The L-values or locations are represented by boxes. The R-value of `A` is represented by the 'puppet strings' which lead from the inside of the L-value of `A` to the L-values of its components. One can think of the 'shape' of the box as representing its type and hence specifying the kind of occupant which may be put there.

Using this sort of diagram, it is now simple to determine the effect of an assignment. Consider the structure



The effect of obeying the assignment command

```
Car[Car[A]] := Cdr[Cdr[A]]
```

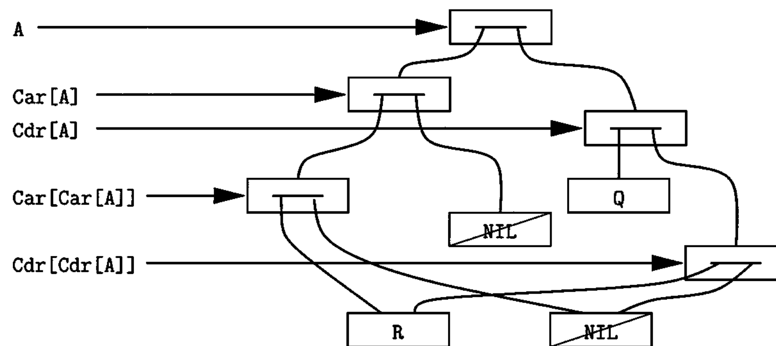
can be determined by the following steps

(1) Find the L-value of `[Car[Car[A]]`.
This is the box marked (1).
(2) Find the R-value of `[Cdr[Cdr[A]]`.
This is the puppet string marked (2).

(1) and (2) may be carried out in either order as neither actually alters the structure.

(3) Replace the contents of (1) by a copy of (2).

The resulting structure is as follows



Notice that this assignment has changed the pattern of sharing in the structure so that now `Car[Car[Car[A]]]` and `Car[Cdr[Cdr[A]]]` actually share the same L-value (and hence also the same R-value). This is because the assignment statements only take a copy of the R-value of its right hand side, not a copy of all the information associated with it. In this respect, structures are similar to functions whose FVL is not copied on assignment.

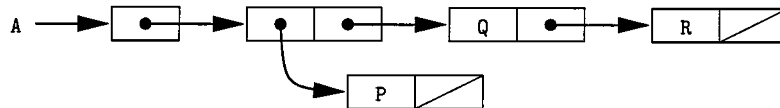
Thus, as with functions, the R-value of a compound data structure gives access to all the information in the structure but does not contain it all. So that the distinction between fixed and free applies as much to structures as it does to functions.

3.7.4. Implementation. The discussion of R- and L-values of nodes has so far been quite general. I have indicated what information must be available, but in spite of giving diagrams I have not specified in any way how it should be represented. I do not propose to go into problems of implementation in any detail—in any case many of them are very machine dependent—but an outline of a possible scheme may help to clarify the concepts.

Suppose we have a machine with a word length which is a few bits longer than a single address. The R-value of a node will then be an address pointing to a small block of consecutive words, one for each component, containing the R-values of the components. An element requires for its R-value an address (e.g., the R-value of a node) and a marker to say which of the various possibilities is its dynamically current type. (There should be an escape mechanism in case there are too few bits available for the marker.) The allocation

and control of storage for these nodes presents certain difficulties. A great deal of work has been done on this problem and workable systems have been devised. Unfortunately there is no time to discuss these here.

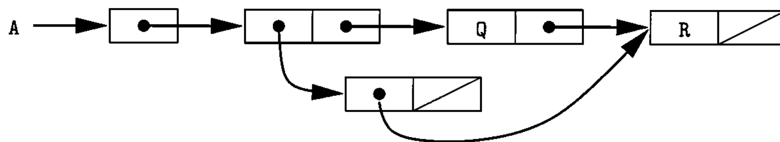
If we use an implementation of this sort for our example in the last section, we shall find that nodes of type Cons will fill two consecutive words. The 'puppet string' R-values can be replaced by the address of the first of these, so that we can redraw our diagram as



After the assignment

```
Car[Car[A]] := Cdr[Cdr[A]]
```

this becomes



3.7.5. Programming example. The following example shows the use of a recursively defined routine which has a structure as a parameter and calls it by reference (L-value). A tree sort takes place in two phases. During the first the items to be sorted are supplied in sequence as arguments to the routine AddtoTree. The effect is to build up a tree structure with an item and two branches at each node. The following node definitions define the necessary components.

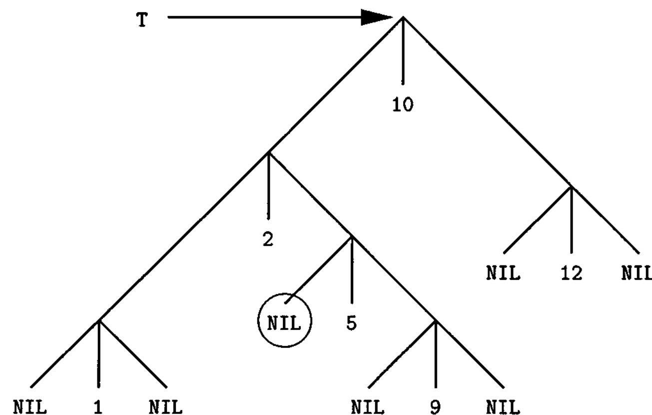
```
node  Knot is  Knot    : Pre
         with  Knot    : Suc
         with  Data    : Item

node  Data is  integer   Key
         with  Body    : Rest
```

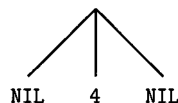
Here the key on which the sort is to be performed is an integer and the rest of the information is of type Body. The routine for the first phase is

```
rec AddtoTree [ref Knot : x, value Data : n] is
    § Test  Null[x]
        If so do x := Knot[NIL,NIL,n]
        If not do  AddtoTree[((Key[n] < Key[Item[x]]) → Pre[x],
                                                           Suc[x]),n]
        return  §
```

The effect of this is to build up a tree where all the items accessible from the Pre (predecessor) branch of a Knot precede (i.e., have smaller keys) than the item at the Knot itself, and this in turn precedes all those which are accessible from the Suc (successor) branch.



The effect of AddtoTree(T,N) where N is a data-item whose Key is 4 would be to replace the circled NIL node by the node



where the central branch marked 4 stands for the entire data-item N.

The second phase of a tree sort forms a singularly elegant example of the use of a recursively defined routine. Its purpose is effectively to traverse the tree from left to right printing out the data-items at each Knot. The way the tree has been built up ensures that the items will be in ascending order of Keys.

We suppose that we have a routine PrintBody which will print information in a data-item in the required format. The following routine will then print out the entire tree.

```
rec PrintTree[Knot:x] is

 § Unless Null[x] do
     § PrintTree[Pre[x]]
       PrintBody[Rest[Item[x]]]
       PrintTree[Suc[x]] §
  return  §
```

3.7.6. Pointers. There is no reason why an R-value should not represent (or be) a location; such objects are known as *pointers*. Suppose X is a *real* variable with an L-value α. Then if P is an object whose R-value is α, we say the type of P is **real pointer** and that P

'points to' X. Notice that the type of a pointer includes the type of the thing it points to, so that pointers form an example of parametric type. (Arrays form another.) We could, for example, have another pointer Q which pointed to P; in this case Q would be of type **real pointer pointer**.

There are two basic (polymorphic) functions associated with pointers:

Follow[P] (also written ↓ P in CPL) calls its argument by *R*-value and produces as a result the *L*-value of the object pointed to. This is, apart from changes of representation, the same as its argument. Thus we have

L-value of Follow[P] = P

R-value of Follow[P] = Contents of P

The function Pointer[X] calls its argument by *L*-value and produces as a result an *R*-value which is a pointer to X.

```
Follow[Pointer[X]]
```

has the same *L*-value as X.

We can assign either to P or to Follow[P], but as their types are not the same we must be careful to distinguish which we mean.

```
P := Follow[Y]
```

will move the pointer P

```
↓P := ↓P + 2
```

will add 2 to the number P points to.

Pointers are useful for operating on structures and often allow the use of loops instead of recursive functions. (Whether this is an advantage or not may be a matter for discussion. With current machines and compilers loops are generally faster than recursion, but the program is sometimes harder to follow.) The following routine has the same effect as the first routine in the previous section. (It is not nearly so easy to turn the other recursive routine into a loop, although it can be done.)

```
AddtoTree' [ref Knot : x, value Data : n] is
 § let p = Pointer[x]
   until Null[↓p] do
      p := (Key[n] < Key[Item[↓p]]) → a Pointer[Pre[↓p]],
                                        Pointer[Suc[↓p]]
   ↓p := Knot[NIL,NIL,n]
   return §
```

3.7.7. Other forms of structure. Vectors and arrays are reasonably well understood. They are parametric types so that the type of an array includes its dimensionality (the number of

its dimensions but not their size) and also the type of its elements. Thus unlike in nodes, all the elements of an array have to be of the same type, though their number may vary dynamically. It is convenient, though perhaps not really necessary, to regard an n-array (i.e., one with n dimensions) as a vector whose elements are $(n - 1)$-arrays.

We can then regard the R-value of a vector as something rather similar to that of a node in that it gives access (or points to) the elements rather than containing them. Thus the assignment of a vector does not involve copying its elements.

Clearly if this is the case we need a system function `Copy` (or possibly `CopyVector`) which does produce a fresh copy.

There are many other possible parametric structure types which are less well understood. The following list is certainly incomplete.

List An ordered sequence of objects all of the same type. The number is dynamically variable.

Ntuple A fixed (manifest) number of objects all of the same type. This has many advantages for the implementer.

Set In the mathematical sense. An unordered collection of objects all of which are of the same type but different from each other. Operations on sets have been proposed for some languages. The lack of ordering presents considerable difficulty.

Bag or Coll This is a new sort of collection for which there is, as yet, no generally accepted name. It consists of an unordered collection of objects all of which are of the same type and differs from a set in that repetitions are allowed. (The name bag is derived from probability problems concerned with balls of various colours in a bag.) A bag is frequently the collection over which an iteration is required—e.g., when averaging.

There are also structures such as *rings* which cannot be 'syntactically' defined in the manner of nodes. They will probably have to be defined in terms of the primitive functions which operate on them or produce them.

It is easy enough to include any selection of these in a programming language, but the result would seem rather arbitrary. We still lack a convincing way of describing those and any other extensions to the sort of structures that a programmer may want to use.

4. Miscellaneous topics

In this section we take up a few points whose detailed discussion would have been out of place before.

4.1. Load-Update Pairs

A general L-value (location) has two important features: There is a function which gives the corresponding R-value (contents) and another which will update this. If the location is not simply addressable, it can therefore be represented by a structure with two components—a `Load` part and an `Update` part; these two can generally share a common FVL. Such an

L-value is known as a Load-Update Pair (LUP). We can now represent any location of type α by an element (in the sense of Section 3.7.2)

```
element α Location is α Address
                   or α LUP

node α LUP is α Function[ ] : Load
              with Routine [α: *] : Update
```

Note that these are parametrically polymorphic definitions. There is also a constraint on the components of a LUP that if X is an α LUP and y is of type α

```
y = value of  § Update[x][y]
              result is   Load[X] §
```

LUPs are of considerable practical value even when using machine code. A uniform system which tests a general location to see if it is addressable or not (in which case it is a LUP)—say by testing a single bit—can then use the appropriate machine instruction (e.g. CDA or STO) or apply the appropriate part of the LUP. This allows all parts of the machine to be treated in a uniform manner as if they were all addressable. In particular index registers, which may need loading by special instruction, can then be used much more freely.

Another interesting example of the use of a LUP is in dealing with the registers which set up the peripheral equipment. In some machines these registers can be set but not read by the hardware. Supervisory programs are therefore forced to keep a copy of their settings in normal store, and it is quite easy to fail to keep these two in step. If the L-value of the offending register is a LUP, and it is always referred to by this, the Update part can be made to change both the register and its copy, while the Load part reads from the copy.

The importance of this use of LUPs is that it reduces the number of ad hoc features of the machine and allows much greater uniformity by treatment. This in turn makes it easier for programmers at the machine code level to avoid oversights and other errors and, possibly more important, makes it easier to write the software programs dealing with these parts of the machine in a high level language and to compile them.

The disadvantage in current machines is that, roughly speaking, every indirect reference requires an extra test to see if the location is addressable. Although this may be unacceptable for reasons of space or time (a point of view which requires the support of much more convincing reasons than have yet been given), it would be a relatively insignificant extra complication to build a trap into the hardware for this test. It is the job of people investigating the fundamental concepts of programming to isolate the features such as this whose incorporation in the hardware of machine would allow or encourage the simplification of its software.

4.2. Macrogenerators

Throughout this course, I have adopted the point of view that programming languages are dealing with abstract objects (such as numbers or functions) and that the details of the way

in which we represent these are of relatively secondary importance. It will not have escaped many readers that in the computing world, and even more so in the world of mathematicians today, this is an unfashionable if not heretical point of view. A much more conventional view is that a program is a symbol string (with the strong implication that it is nothing more), a programming language the set of rules for writing down local strings, and mathematics in general a set of rules for manipulating strings.

The outcome of this attitude is a macrogenerator whose function is to manipulate or generate symbol strings in programming languages without any regard to their semantic content. Typically such a macrogenerator produces 'code' in some language which is already implemented on the machine and whose detailed representation must be familiar to anyone writing further more definitions. It will be used to extend the power of the base language, although generally at the expense of syntactic convenience and often transparency, by adding new macrocommands.

This process should be compared with that of functional abstraction and the definition of functions and routines. Both aim to extend the power of the language by introducing now operations. Both put a rather severe limit on the syntactic freedom with which the extensions can be made.

The difference lies in the fact that macrogenerators deal with the symbols which represent the variables, values and other objects of concern to a program so that all their manipulation is performed before the final compiling. In other words all macrogeneration is manifest. Function and routine definitions on the other hand are concerned with the values themselves, not with the symbols which represent them and thus, in the first instance are dynamic (or latent) rather than manifest.

The distinction is blurred by the fact that the boundary between manifest and latent is not very clear cut, and also by the fact that it is possible by ingenuity and at the expense of clarity to do by a macrogenerator almost everything that can be done by a function definition and *vice versa*. However the fact that it is possible to push a pea up a mountain with your nose does not mean that this is a sensible way of getting it there. Each of these techniques of language extension should be used in its proper place.

Macrogeneration seems to be particularly valuable when a semantic extension of the language is required. If this is one which was not contemplated by the language designer the only alternative to trickery with macros is to rewrite the compiler—in effect to design a new language. This has normally been the situation with machine code and assembly languages and also to a large extend with operating systems. The best way to avoid spending all your time fighting the system (or language) is to use a macrogenerator and build up your own.

However with a more sophisticated language the need for a macrogenerator diminishes, and it is a fact that ALGOL systems on the whole use macrogenerators very rarely. It is, I believe, a proper aim for programming language designers to try to make the use of macrogenerators wholly unnecessary.

4.3. Formal semantics

Section 3.3 gives an outline of a possible method for formalising the semantics of programming languages. It is a development of an earlier proposal [8], but it is far from complete and cannot yet be regarded as adequate.

There are at present (Oct. 1967) only three examples of the formal description of the semantics of a real programming language, as opposed to those which deal with emasculated versions of languages with all the difficulties removed. These are the following:

(i) Landin's reduction of ALGOL to λ-expressions with the addition of assignments and jumps. This requires a special form of evaluating mechanism (which is, of course, a notional computer) to deal with the otherwise non-applicative parts of the language. The method is described in [6] and given in full in [9].
(ii) de Bakker [10] has published a formalisation of most of ALGOL based on an extension of Markov algorithms. This is an extreme example of treating the language as a symbol string. It requires no special machine except, of course, the symbol string manipulator.
(iii) A team at the IBM Laboratories in Vienna have published [12, 13] a description of PL/I which is based on an earlier evaluating mechanism for pure λ-expressions suggested by Landin [11] and the concept of a state vector for a machine suggested by McCarthy [14]. This method requires a special 'PL/I machine' whose properties and transition function are described. The whole description is very long and complex and it is hard to determine how much of this complexity is due to the method of semantic description and how much to the amorphous nature of PL/I.

The method suggested in Section 3.3 has more in common with the approach of Landin or the IBM team than it has with de Bakker's. It differs, however, in that the ultimative machine required (and all methods of describing semantics come to a machine ultimately) is in no way specialised. Its only requirement is that it should be able to evaluate pure λ-expressions. It achieves this result by explicitly bringing in the store of the computer in an abstract form, an operation which brings with it the unexpected bonus of being able to distinguish explicitly between manifest and latent properties. However until the whole of a real language has been described in these terms, it must remain as a proposal for a method, rather than a method to be recommended.

Notes

1. This is the CPL notation for a conditional expression which is similar to that used by LISP. In ALGOL the equivalent would be **if** `a > b` **then** `j` **else** `k`.
2. The ALGOL equivalent of this would have to be **if** `a > b` **then** `j := i` **else** `k := i`.
3. ALGOL 60 **call by name** Let `f` be an ALGOL procedure which calls a formal parameter `x` by name. Then a call for `f` with an actual parameter expression ε will have the same effect as forming a parameterless procedure $\lambda\,().\varepsilon$ and supplying this by value to a procedure `f*` which is derived from `f` by replacing every written occurrence of `x` in the body of `f` by `x()`. The notation $\lambda().\varepsilon$ denotes a parameterless procedure whose body is ε while `x()` denotes its application (to a null parameter list).
4. The only elementary operator to which this rule does not already apply is exponentiation. Thus, for example, if `a` and `b` are both integers a^b will be an integer if $b \geq 0$ and a real if $b < 0$. If `a` and `b` are reals, the type of a^b depends on the sign of `a` as well as that of `b`. In CPL this leads to a definition of $a \uparrow b$ which differs slightly in its domain from a^b.
5. By analogy with monadic, dyadic and polyadic for functions with one, two and many arguments. Functions with no arguments will be known as *anadic*. Unfortunately there appears to be no suitable Greek prefix meaning variable.

References

1. Barron, D.W., Buxton, J.N., Hartley, D.F., Nixon, E., and Strachey, C. The main features of CPL. *Comp. J.* **6** (1963) 134–143.
2. Buxton, J.N., Gray, J.C., and Park, D. CPL elementary programming manual, Edition II. Technical Report, Cambridge, 1966.
3. Strachey, C. (Ed.). CPL working papers. Technical Report, London and Cambridge Universities, 1966.
4. Quine, W.V. *Word and Object*. New York Technology Press and Wiley, 1960.
5. Schönfinkel, M. Über die Bausteine der mathematischen Logik. *Math. Ann.* **92** (1924) 305–316.
6. Landin, F.J. A formal description of ALGOL 60. In *Formal Language Description Languages for Computer Programming*, T.B. Steel (Ed.). North Holland Publishing Company, Amsterdam, 1966, pp. 266–294.
7. Curry, H.B. and Feys, R. *Combinatory Logic*, Vol. 1, North Holland Publishing Company, Amsterdam, 1958.
8. Strachey, C. Towards a formal semantics. In *Formal Language Description Languages for Computer Programming* T.B. Steel (Ed.). North Holland Publishing Company, Amsterdam, 1966, pp. 198–216.
9. Landin, P.J. A correspondence between ALGOL 60 and Church's Lambda notation. *Comm. ACM* **8** (1965) 89–101, 158–165.
10. de Bakker, J.W. *Mathematical Centre Tracts 16: Formal Definition of Programming Languages*. Mathematisch Centrum, Amsterdam, 1967.
11. Landin, P.J. The Mechanical Evaluation of Expressions. *Comp. J*. **6** (1964) 308–320.
12. PL/I—Definition Group of the Vienna Laboratory. Formal definition of PL/I. IBM Technical Report TR 25.071, 1966.
13. Alber, K. Syntactical description of PL/I text and its translation into abstract normal form. IBM Technical Report TR 25.074, 1967.
14. McCarthy, J. Problems in the theory of computation. In *Proc. IFIP Congress 1965*, Vol. 1, W.A. Kalenich (Ed.). Spartan Books, Washington, 1965, pp. 219–222.