, , 1–37 ()

Definitional Interpreters for Higher-Order Programming Languages *

JOHN C. REYNOLDS **
Systems and Information Science, Syracuse University

Abstract. Higher-order programming languages (i.e., languages in which procedures or labels can occur as values) are usually defined by interpreters that are themselves written in a programming language based on the lambda calculus (i.e., an applicative language such as pure LISP). Examples include McCarthy's definition of LISP, Landin's SECD machine, the Vienna definition of PL/I, Reynolds' definitions of GEDANKEN, and recent unpublished work by L. Morris and C. Wadsworth. Such definitions can be classified according to whether the interpreter contains higher-order functions, and whether the order of application (i.e., call by value versus call by name) in the defined language depends upon the order of application in the defining language. As an example, we consider the definition of a simple applicative programming language by means of an interpreter written in a similar language. Definitions in each of the above classifications are derived from one another by informal but constructive methods. The treatment of imperative features such as jumps and assignment is also discussed.

Keywords: programming language, language definition, interpreter, lambda calculus, applicative language, higher-order function, closure, order of application, continuation, LISP, GEDANKEN, PAL, SECD machine, J-operator, reference.

1. Introduction

An important and frequently used method of defining a programming language is to give an interpreter for the language that is written in a second, hopefully better understood language. (We will call these two languages the *defined* and *defining* languages, respectively.) In this paper, we will describe and classify several varieties of such interpreters, and show how they may be derived from one another by informal but constructive methods. Although our approach to "constructive classification" is original, the paper is basically an attempt to review and systematize previous work in the field, and we have tried to make the presentation accessible to readers who are unfamiliar with this previous work.

(Of course, interpretation can provide an implementation as well as a definition, but there are large practical differences between these usages. Definitional interpreters often achieve clarity by sacrificing all semblance of efficiency.)

We begin by noting some salient characteristics of programming languages themselves. The features of these languages can be divided usefully into two categories: *applicative* features, such as expression evaluation and the definition and application

* Work supported by Rome Air Force Development Center Contract No. 30602-72-C-0281 and ARPA Contract No. DAHC04-72-C-0003. This paper originally appeared in the Proceedings of the ACM National Conference, volume 2, August, 1972, ACM, New York, pages 717–740.
** Current address: Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA. e-mail: John.Reynolds@cs.cmu.edu

of functions, and *imperative* features, such as statement sequencing, labels, jumps, assignment, and procedural side-effects. Most user-oriented languages provide features in both categories. Although machine languages are usually purely imperative, there are few "higher-level" languages that are purely imperative. (IPL/V might be an example.) On the other hand, there is at least one well-known example of a purely applicative language: LISP (i.e., the language defined in McCarthy's original paper [1]; most LISP implementations provide an extended language including imperative features). There are also several more recent, rather theoretical languages (ISWIM [2], PAL [3], and GEDANKEN [4]) that have been designed by starting with an applicative language and adding imperative extensions.

Purely applicative languages are often said to be based on a logical system called the lambda calculus [5, 6], or even to be "syntactically sugared" versions of the lambda calculus. In particular, Landin [7] has shown that such languages can be reduced to the lambda calculus by treating each type of expression as an abbreviation for some expression of the lambda calculus. Indeed, this kind of reducibility could be taken as a precise definition of the notion of "purely applicative." However, as we will see, although an unsugared applicative language is syntactically equivalent to the lambda calculus, there is a subtle semantic difference. Essentially, the semantics of the "real" lambda calculus implies a different "order of application" (i.e., normal-order evaluation) than most applicative programming languages.

A second useful characterization is the notion of a *higher-order* programming language. In analogy with mathematical logic, we will say that a programming language is *higher-order* if procedures or labels can occur as data, i.e., if these entities can be used as arguments to procedures, as results of functions, or as values of assignable variables. A language that is not higher-order will be called *first-order*.

In ALGOL and its various descendents, procedures and labels can be used as procedure arguments, and in more recent languages such as PL/I and ALGOL 68, they may also be used as function results and assignable values, subject to certain "scope" restrictions (which are imposed to preserve a stack discipline for the storage allocation of the representations of functions and labels). However, the unrestricted use of procedures and labels as data is permitted in only a handful of languages which sacrifice efficiency for generality: LISP (in most of its interpretive implementations), ISWIM, PAL, GEDANKEN, and (roughly) POP-2.

With regard to current techniques of language definition, there is a substantial disparity between first-order and higher-order languages. As a result of work by Floyd [8], Manna [9], Hoare [10], and others, most aspects of first-order languages can be defined logically, i.e., one can give an effective method for transforming a program in the defined language into a logical statement of the relation between its inputs and outputs. However, it has not yet been possible to apply this approach to higher-order languages. (Although recent work by Scott [12, 13, 14, 15] and Milner [16] represents a major step in this direction.)

Almost invariably, higher-order languages have been defined by the approach discussed in this paper, i.e., by giving interpreters that are themselves written in a programming language (An apparent exception is definition of ALGOL given by Burstall [17], but this can be characterized as a logical definition of a first-order interpreter for a higher-order language.) Moreover, even when the defined language contains imperative features, the defining language is usually purely applicative (probably because applicative languages are well suited for computations with symbolic expressions). Examples include McCarthy's definition of LISP [1], Landin's SECD machine [7], the Vienna definition of PL/I [18], Reynolds' definitions of GEDANKEN [19], and recent unpublished work by L. Morris [20] and C. Wadsworth.

(There are a few instances of definitional interpreters that fall outside the conceptual framework developed in this paper. A broader review of the field is given by deBakker [21].)

These examples exhibit considerable variety, ranging from very concise and abstract interpreters to much more elaborate and machine-like ones. To achieve a more precise classification, we will introduce two criteria. First, we ask whether the defining language is higher-order, or more precisely, whether any of the functions that comprise the interpreter either accept or produce values that are themselves functions.

The second criterion involves the notion of *order of application*. In designing any language that allows the use of procedures or functions, one must choose between two orders of application which are called (following ALGOL terminology) *call by value* and *call by name*. Even when the language is purely applicative, this choice will affect the meaning of some, but not all, programs that can be written in the language. Remembering that an interpreter is a specific program, we obtain our second criterion: Does the meaning of the interpreter depend upon the order of application chosen for the defining language?

These two criteria establish four possible classes of interpreters, each of which contains one or more of the examples cited earlier:

Order-of-application dependence:	Use of higher-order functions: yes	no
yes	direct interpreter for GEDANKEN	McCarthy's definition of LISP
no	Morris-Wadsworth method	SECD machine, Vienna definition

The main goal of this paper is to illustrate and relate these classes of definitional interpreters. In the next section we will introduce a simple applicative language, which we will use as the defining language and also, with several restrictions, as the defined language. Then we will present a simple interpreter that uses higher-order functions and is order-of-application dependent, and we will transform this interpreter into examples of the three remaining classes. Finally, we will consider the problem of adding imperative features to the defined language (while keeping the defining language purely applicative).

2. A Simple Applicative Language

In an applicative language, the meaningful phrases of a program are called *expressions*, the process of executing or interpreting these expressions is called *evaluation*, and the result of evaluating an expression is called a *value*. However, as is evident from a simple arithmetic expression such as $x + y$, different evaluations of the same expression can produce different values, so that the process of evaluation must depend upon something more than just the expression being evaluated. It is evident that this "something more" must specify a value for every variable that might occur in the expression (more precisely, occur free). We will call such a specification an *environment*, and say that it *binds* variables to values.

It is also evident that the evaluation process may involve the creation of new environments from old ones. Suppose $x_1, \ldots, x_n$ are variables, $v_1, \ldots, v_n$ are values, and e and e' are environments. If e' specifies the value v_i for each x_i, and behaves the same way as e for all other variables, then we will say that e' is the *extension* of e that binds the x_i's to the v_i's.

The simplest expressions in our applicative language are *constants* and *variables*. The evaluation of a constant always gives the same value, regardless of the environment. We will not specify the set of constants precisely, but will assume that it contains the integers and the Boolean constants **true** and **false**. The evaluation of a variable simply produces the value that is bound to that variable by the environment. In the programs in this paper we will denote variables by alphanumeric strings, with occasional superscripts and subscripts.

If our language is going to involve functions, then we must have a form of expression whose evaluation will cause the application of function to its arguments. If $r_0, r_1, \ldots, r_n$ are expressions, then $r_0(r_1, \ldots, r_n)$ is an *application expression*, whose *operator* is r_0 and whose *operands* are $r_1, \ldots, r_n$. The evaluation of an application expression in an environment proceeds as follows:

1. The subexpressions $r_0, r_1, \ldots, r_n$ are evaluated in the same environment to obtain values $f,\ a_1, \ldots, a_n$.

2. If f is not a function of n arguments, then an error stop occurs.

3. Otherwise, the function f is *applied* to the arguments $a_1, \ldots, a_n$, and if this application produces a result, then the result is the value of the application expression.

There are several assumptions hiding behind this description that need to be made explicit:

1. A "function of n arguments" is a kind of value that can be subjected to the process of being "applied" to a sequence of n values called "arguments".

2. For some functions and arguments, the process of application may never produce a result, either because the process does not terminate (i.e., it runs on forever), or because it causes an error stop. Similarly, for some expressions and environments, the process of evaluation may never produce a value.

3. In a purely applicative language, the application of the same function to the same sequence of arguments will always have the same effect, i.e., both the result that is produced, and the prior question of whether any result is produced, depend only upon the function and its arguments. Similarly, the evaluation of the same expression in the same environment will always have the same effect.

4. During the evaluation of an application expression, the application process does not begin until after the operator and all of its operands have been evaluated. This is the *call-by-value* order of application mentioned in the introduction. In the alternative order of application, known as *call by name*, the application process would begin as soon as the operator had been evaluated, and each operand would only be evaluated when (and if) the function being applied actually depended upon its value. This distinction will be clarified below.

5. Although we have specified that all of the subexpressions $r_0, \ldots, r_n$ are to be evaluated before the application process begins we have not specified the relative order in which these subexpressions are to be evaluated. In a purely applicative language, this choice has no effect. (A slight exception occurs if the evaluation of one subexpression never terminates while the evaluation of another gives an error stop.) However, the choice will become significant when we start adding imperative features to the defined language. In anticipation of this extension, we will assume that the subexpressions are evaluated successively from left to right.

Next, we must have a form of expression whose evaluation will produce a function. If $x_1, \ldots, x_n$ are variables and r is an expression, then $\lambda(x_1, \ldots, x_n).\ r$ is a *lambda expression*, whose *formal parameters* are $x_1, \ldots, x_n$ and whose body is r. (The parentheses may be omitted if there is only one formal parameter.) The evaluation of a lambda expression with n formal parameters always terminates and always produces a function of n arguments. To describe this function, we must specify what will happen when it is applied to its arguments.

Suppose that f is the function obtained by evaluating $\lambda(x_1, \ldots, x_n).\ r$ in an environment e. Then the application of f to the arguments $a_1, \ldots, a_n$ will cause the evaluation of the body r in the environment that is the extension of e that binds each x_i to the corresponding a_i. If this evaluation produces a value, then the value becomes the result of the application of f.

The key point is that the environment in which the body is evaluated during application is an extension of the earlier environment in which the lambda expression was evaluated (rather than the more recent environment in which the application takes place). As a consequence, if a lambda expression contains global variables (i.e., variables that are not formal parameters), its evaluation in different environments can produce different functions. For example, the lambda expression $\lambda x.\ x + y$ can produce an incrementing function, an identity function (for the integers), or a decrementing function, when evaluated in environments that bind y to the values 1, 0, or -1 respectively.

Nowadays, it is generally accepted that this behavior of lambda expressions and environments is a basic characteristic of a well-designed higher-order language. Its importance is that it permits functional data to depend upon the partial results of a program.

Having introduced application and lambda expressions, we may now clarify the distinction between call by value and call by name. Consider the evaluation of an application expression $r_0(r_1, \ldots, r_n)$ in an environment e_a, and suppose that the value of the operator r_0 is a function f that was originally created by evaluating the lambda expression $\lambda(x_1, \ldots, x_n).\ r_\lambda$ in an environment e_λ. (Possibly this lambda expression is r_0 itself, but more generally r_0 may be a non-lambda expression whose functional value was created earlier in the computation.) When call by value is used, the following steps will occur during the evaluation of the application expression:

1. r_0 is evaluated in the environment e_a to obtain the function value f.
2. $r_1, \ldots, r_n$ are evaluated in the environment e_a to obtain arguments $a_1, \ldots, a_n$.
3. r_λ is evaluated in the extension of e_λ that binds each x_i to the corresponding a_i, to obtain the value of the application expression.

When call by name is used, the same expressions are evaluated in the same environments. But the evaluations of the operands $r_1, \ldots, r_n$ will occur at a later time and may occur a different number of times. Specifically, instead of being evaluated before step (3), each operand r_i is repeatedly evaluated during step (3), each time that its value a_i is actually used (as a function to be applied, a Boolean value determining a branch, or an argument of a primitive operation).

At first sight, since the evaluation of the same expression in the same environment always produces the same effect, it would appear that the result of a program in a purely applicative language should be unaffected by changing the order of application (although it is evident that the repeated evaluation of operands occurring with call by name can be grossly inefficient). But this overlooks the possibility that "repeatedly" may mean "never". During step (3) of the evaluation of $r_0(r_1, \ldots, r_n)$, it may happen that certain arguments a_i are never used, so that the corresponding operands r_i will never be evaluated under call by name. Now suppose that the evaluation of one of these r_i never terminates (or gives an error stop). Then the evaluation of the original application expression will terminate under call by name but not call by value. In brief, changing the order of application can affect the value of an application expression when the function being applied is independent of some of its arguments and the corresponding operands are nonterminating.

(In ALGOL the distinction between call by value and call by name also involves a change in "coercion conventions". However, this change is irrelevant in the absence of assignment.)

In the defined language, we will consider only the use of call by value, but in the defining language we will consider both orders of application. In particular, we will inquire whether the above-described situation occurs in our interpreters, so that changing the order of application in the defining language can affect the meaning of the defined language.

We now introduce some additional kinds of expressions. If r_p, r_c and r_a are expressions, then **if** r_p **then** r_c **else** r_a is a *simple conditional expression*, whose *premiss* is r_p, whose *conclusion* is r_c, and whose *alternative* is r_a. The evaluation of a conditional expression in an environment e begins with the evaluation of its premiss r_p in the same environment. Then, depending upon whether the value of the premiss is **true** or **false**, the value of the conditional expression is obtained by evaluating either the conclusion r_c or the alternative r_a in e. Any other value of the premiss causes an error stop.

It is also convenient to use a LISP-like notation for "multiple" conditional expressions. If $r_{p1}, \ldots, r_{pn}$ and $r_{c1}, \ldots, r_{cn}$ are expressions, then

$$(r_{p1} \rightarrow r_{c1}, r_{p2}, \rightarrow r_{c2}, \ldots, r_{pn} \rightarrow r_{cn})$$

is a *multiple conditional expression*, with the same meaning as the following sequence of simple conditional expressions:

$$\textbf{if } r_{p1} \textbf{ then } r_{c1} \textbf{ else if } r_{p2} \textbf{ then } r_{c2} \textbf{ else} \cdots \textbf{ if } r_{pn} \textbf{ then } r_{cn} \textbf{ else error}.$$

Next, we introduce a form of expression (due to Landin [7]) that is analogous to the block in ALGOL. If $x_1, \ldots, x_n$ are variables, and $r_1, \ldots, r_n$ and r_b are expressions, then

$$\textbf{let } x_1 = r_1 \textbf{ and} \cdots \textbf{and } x_n = r_n \textbf{ in } r_b$$

is a **let** *expression*, whose *declared variables* are $x_1, \ldots, x_n$, whose *declaring expressions* are $r_1, \ldots, r_n$, and whose *body* is r_b. (We will call each pair $x_i = r_i$ a *declaration*.) The evaluation of a **let** expression in an environment e begins with the evaluation of its declaring expressions r_i in the same environment. Then the value of the **let** expression is obtained by evaluating its body r_b in the environment that is the extension of e that binds each declared variable x_i to the value of the corresponding declaring expression r_i.

It should be noted that the extended environment only affects the evaluation of the body, not the declaring expressions. For example, in an environment that binds x to 4, the value of **let** $x = x + 1$ **and** $y = x - 1$ **in** $x \times y$ is 15. As a consequence, **let** expressions cannot be used (at least directly) to define recursive functions. One might expect, for instance, that

$$\textbf{let } f = \lambda x.\ \textbf{if } x = 0 \textbf{ then } 1 \textbf{ else } x \times f(x - 1) \textbf{ in} \cdots$$

would create an extended environment in which f was bound to a recursive function (for computing the factorial). But in fact, the occurrence of f inside the declaring expression will not "feel" the binding of f to the value of the declaring expression, so that the resulting function will not call itself recursively.

To overcome this problem, we introduce a second kind of block-like expression. If $x_1, \ldots, x_n$ are variables, $\ell_1, \ldots, \ell_n$ are lambda expressions, and r_b is an expression, then

$$\textbf{letrec } x_1 = \ell_1 \textbf{ and} \cdots \textbf{and } x_n = \ell_n \textbf{ in } r_b$$

is a *recursive* **let** *expression*, whose *declared variables* are $x_1, \ldots, x_n$, whose *declaring expressions* are $\ell_1, \ldots, \ell_n$, and whose *body* is r_b. The value of a recursive **let** expression in an environment e is obtained by evaluating its body in an environment e' which satisfies the following property: e' is the extension of e that binds each declared variable x_i to the function obtained by evaluating the corresponding declaring lambda expression ℓ_i in the environment e'.

There is a circularity in the property "e' is the ... in the environment e'" that is characteristic of recursion, and that prevents this property from being an explicit definition of e'. To be rigorous, we would have to show that there actually exists an environment that satisfies this property, and also deal with the possibility that this environment might not be unique. The mathematical techniques needed to achieve this rigor are beyond the scope of this paper [22, 12, 13, 14, 15]. However, we will eventually derive an interpreter that defines recursive **let** expressions more explicitly.

(It is possible to generalize recursive **let** expressions by allowing arbitrary declaring expressions. We have chosen not to do so, since the generalization would considerably complicate some of the definitional interpreters, and is not unique.)

To maintain generality, we have avoided specifying the set of data that can occur as the result of expression evaluation (beyond asserting that this set should contain functions and the Boolean values **true** and **false**). However, it is evident that our language must contain basic (i.e., built-in) operations and tests for manipulating this data. For example, if integers are to occur as data, we will need at least an incrementing operation and a test for integer equality. More likely, we will want all of the usual arithmetic operations and tests. If some form of structured data is to be used, we will need operations for constructing and analyzing the structures, and tests for classifying them.

Regardless of the specific nature of the data, there are three ways to introduce basic operations and tests into our applicative language:

1. We may introduce constants denoting the basic functions (whose application will perform the basic operations and tests).

2. We may introduce *predefined variables* denoting the basic functions. These variables differ from constants in that the programmer can redefine them with his own declarations. They are specified by introducing an *initial environment*, to be used for the evaluation of the entire program, that binds the predefined variables to their functional values.

3. We may introduce special expressions whose evaluation will perform the basic operations and tests. Since this approach is used in most programming languages (and in mathematical notation), we will frequently use the common forms of arithmetic and Boolean expressions without explanation.

3. The Defined Language

Although our defining language will use all of the features described in the previous section, along with appropriate basic operations and tests, the defined language will be considerably more limited, in order to avoid complications that would be out of place in an introductory paper. Specifically:

1. Functions will be limited to a single argument. Thus all applicative expressions will have a single operand, and all lambda expressions will have a single formal parameter.

2. Only call by value will be used.

3. Only simple conditional expressions will be used.

4. Nonrecursive **let** expressions will be excluded.

5. All recursive **let** expressions will contain a single declaration.

6. Values will be integers, booleans, and functions. The only basic operations and tests will be functions for incrementing integers and for testing integer equality, denoted by the predefined variables *succ* and *equal*, respectively.

The reader may accept an assurance that these limitations will eliminate a variety of tedious complications without evading any intellectually significant problems. Indeed, with slight exceptions, the eliminated features can be regarded as syntactic sugar, i.e., they can be defined as abbreviations for expressions in the restricted language [7, 4].

4. Abstract Syntax

We now turn our attention to the defining language. To permit the writing of interpreters, the values used in the defining language must include expressions of the defined language. At first sight, this suggests that we should use character strings as values denoting expressions, but this approach would enmesh us in questions of grammar and parsing that are beyond the scope of this paper. (An excellent review of these matters is contained in Reference [23].)

Instead, we use the approach of *abstract syntax*, originally suggested by McCarthy [24]. In this approach, it is assumed that programs are “really” abstract, hierarchically structured data objects, and that the character strings that one actually reads into the computer are simply representations of these abstract objects (in the same sense that digit strings are representations of integers). Thus the problems of grammar and parsing can be set aside as “input editing”. (Of course, this does not eliminate these problems, but it separates them clearly from semantic considerations. See, for example, Wozencraft and Evans [25].)

We are left with two closely related problems: how to define sets of abstract expressions (and other structured data to be used by the interpreters), and how to define the basic functions for constructing, analyzing, and classifying these objects. Both problems are solved by introducing three forms of *abstract-syntax equations*. (A more elaborate defined language would require a more complex treatment of abstract syntax, as given in Reference [18], for example.) Within these equations, upper-case letter strings denote sets, and lower-case letter strings denote basic functions.

Let $S_0, S_1, \ldots, S_n$ be upper-case letter strings and $a_1, \ldots, a_n$ be lowercase letter strings. Then a *record equation* of the form

$$S_0 = [a_1 : S_1, \ldots, a_n : S_n]$$

implies that:

1. S_0 is a set, disjoint from any other set defined by a record equation, whose members are records with n fields in which the value of the ith field belongs to the set S_i. (Mathematically, S_0 is a disjoint set in one-to-one correspondence with the Cartesian product $S_1 \times \cdots \times S_n$.)

2. Each a_i (is a predefined variable which) denotes the *selector* function that accepts a member of S_0 and produces its ith field value.

3. Let s_0 be the string obtained from S_0 by lowering the case of each character. Then s_0*?* denotes the *classifier* function that tests whether its argument belong to S_0, and *mk-*s_0 denotes the constructor function of n arguments (belonging to the sets $S_1, \ldots, S_n$) that creates a record in S_0 from its field values.

For example, the record equation

$$\mathrm{APPL} = [\mathit{opr}\text{: } \mathrm{EXP},\ \mathit{opnd}\text{: } \mathrm{EXP}]$$

implies that an application expression (i.e., a member of APPL) is a two-field record whose field values are both expressions (i.e., members of EXP). It also implies that *opr* and *opnd* are selector functions that produce the first and second field values of an application expression, that *appl?* is a classifier function that tests whether a value is an application expression, and that *mk-appl* is a two-argument constructor function that constructs an application expression from its field values. It is evident that if r_1 and r_2 are expressions,

$$\begin{aligned} \mathit{opr}\big(\mathit{mk\text{-}appl}(r_1, r_2)\big) &= r_1 \\ \mathit{opnd}\big(\mathit{mk\text{-}appl}(r_1, r_2)\big) &= r_2, \end{aligned}$$

and if $\mathit{appl?}(r)$ is **true**,

$$\mathit{mk\text{-}appl}\big(\mathit{opr}(r), \mathit{opnd}(r)\big) = r.$$

The remaining forms of abstract syntax equations are the *union equation*:

$$S_0 = S_1 \cup \cdots \cup S_n,$$

which implies that S_0 is the union of sets $S_1, \ldots, S_n$, and the *function equation*:

$$S_0 = S_1, \ldots, S_n \rightarrow S_r,$$

which implies that S_0 is the set of n-argument functions that accept arguments in $S_1, \ldots, S_n$ and produce results in S_r. (More precisely, S_0 is the set of n-argument functions f with the property that if f is applied to arguments in the sets $S_1, \ldots, S_n$, and if f terminates without an error stop, then the result of f belongs to S_r.)

We may now use these forms of abstract syntax equations to define the principal set of data used by our interpreters, i.e., the set EXP of expressions of the defined language:

$$\begin{aligned}
&\mathrm{EXP} = \mathrm{CONST} \cup \mathrm{VAR} \cup \mathrm{APPL} \cup \mathrm{LAMBDA} \cup \mathrm{COND} \cup \mathrm{LETREC}\\
&\mathrm{APPL} = [\mathit{opr}\colon \mathrm{EXP},\ \mathit{opnd}\colon \mathrm{EXP}]\\
&\mathrm{LAMBDA} = [\mathit{fp}\colon \mathrm{VAR},\ \mathit{body}\colon \mathrm{EXP}]\\
&\mathrm{COND} = [\mathit{prem}\colon \mathrm{EXP},\ \mathit{conc}\colon \mathrm{EXP},\ \mathit{altr}\colon \mathrm{EXP}]\\
&\mathrm{LETREC} = [\mathit{dvar}\colon \mathrm{VAR},\ \mathit{dexp}\colon \mathrm{LAMBDA},\ \mathit{body}\colon \mathrm{EXP}].
\end{aligned}$$

A cumbersome but fairly accurate translation into English is that an expression (i.e., a member of EXP) is one of the following:

1. A constant (a member of CONST),
2. A variable (a member of VAR),
3. An application expression (a member of APPL), which consists of an expression called its operator (selected by the basic function *opr*) and an expression called its operand (selected by *opnd*),
4. A lambda expression (a member of LAMBDA), which consists of a variable called its formal parameter (selected by *fp*) and an expression called its body (selected by *body*),
5. A conditional expression (a member of COND), which consists of an expression called its premiss (selected by *prem*) and an expression called its conclusion (selected by *conc*) and an expression called its alternative (selected by *altr*),
6. A recursive **let** expression (a member of LETREC), which consists of a variable called its declared variable (selected by *dvar*), a lambda expression called its declaring expression (selected by *dexp*), and an expression called its body (selected by *body*).

We have purposely left the sets CONST and VAR unspecified. For CONST, we will assume only that there is a basic function *const?* which tests whether its argument is a constant, and a basic function *evcon* which maps each constant into the value that it denotes. For VAR, we will assume that there is a basic function *var?* which tests whether its argument is a variable, that variables can be tested for equality (of the variables themselves, not their values), and that two particular variables are denoted by the quoted strings "succ" and "equal".

We must also define the abstract syntax of two other data sets that will be used by our interpreter. The first is the set VAL of values of the defined language:

$$\begin{aligned}
&\mathrm{VAL} = \mathrm{INTEGER} \cup \mathrm{BOOLEAN} \cup \mathrm{FUNVAL}\\
&\mathrm{FUNVAL} = \mathrm{VAL} \to \mathrm{VAL}.
\end{aligned}$$

One must be careful not to confuse values in the defined and defining languages. Strictly speaking, VAL is a subset of the values of the defining language whose members *represent* the values of the defined language. However, since the variety of values provided in the defining language is richer than in the defined language, we have been able to represent each defined-language value by the same defining-language value. In our later interpreters this situation will change, and it will become more evident that VAL is a set of value representations.

Finally, we must define the set ENV of environments. Since the purpose of an environment is to specify the value that is bound to each variable, the simplest approach is to assume that an environment is a function from variables to values, i.e.,

$$\mathrm{ENV} = \mathrm{VAR} \to \mathrm{VAL}.$$

Within the various interpreters that we will present, each variable will range over some set defined by abstract syntax equations. For clarity, we will use different variables for different sets, as summarized in the following table:

Variable	Range	Variable	Range
r	EXP	$e\ e'$	ENV
$x\ z$	VAR	$c\ c'$	CONT
ℓ	LAMBDA	$m\ m'\ m''$	MEM
$a\ b$	VAL	rf	REF
f	FUNVAL	n	INTEGER

(The sets CONT, MEM, and REF will be defined later.)

5. A Meta-Circular Interpreter

Our first interpreter is a straightforward transcription of the informal language definition we have already given. Its central component is a function *eval* that produces the value of an expression r in a environment e:

$$eval = \lambda(r, e). \qquad \text{I.1}$$
$$\big(const?(r) \to evcon(r), \qquad \text{I.2}$$
$$var?(r) \to e(r), \qquad \text{I.3}$$
$$appl?(r) \to \big(eval(opr(r), e)\big)\big(eval(opnd(r), e)\big), \qquad \text{I.4}$$
$$lambda?(r) \to evlambda(r, e), \qquad \text{I.5}$$
$$cond?(r) \to \mathbf{if}\ eval(prem(r), e) \qquad \text{I.6}$$
$$\mathbf{then}\ eval(conc(r), e)\ \mathbf{else}\ eval(altr(r), e), \qquad \text{I.7}$$
$$letrec?(r) \to \mathbf{letrec}\ e' = \qquad \text{I.8}$$
$$\lambda x.\ \mathbf{if}\ x = dvar(r)\ \mathbf{then}\ evlambda(dexp(r), e')\ \mathbf{else}\ e(x) \qquad \text{I.9}$$
$$\mathbf{in}\ eval(body(r), e')\big) \qquad \text{I.10}$$
$$evlambda = \lambda(\ell, e).\ \underline{\lambda a.\ eval\big(body(\ell), ext(fp(\ell), a, e)\big)} \qquad \text{I.11}$$
$$ext = \lambda(z, a, e).\ \lambda x.\ \mathbf{if}\ x = z\ \mathbf{then}\ a\ \mathbf{else}\ e(x). \qquad \text{I.12}$$

The subsidiary function *evlambda* produces the value of a lambda expression ℓ in an environment e. (We have extracted it as a separate function since it is called from two places, in lines I.5 and I.9.) The subsidiary function *ext* produces the extension of an environment e that binds the variable z to the value a. It should be noted that, in the evaluation of a recursive **let** expression (lines I.8 to I.10), the circularity in the definition of the extended environment e' is handled by making e' a recursive function. (However, it is a rather unusual recursive function which, instead of calling itself, calls another function *evlambda*, to which it provides itself as an argument.)

The function *eval* does not define the meaning of the predefined variables. For this purpose, we introduce the "main" function *interpret*, which causes a complete program r to be evaluated in an initial environment *initenv* that maps each predefined variable into the corresponding basic function:

$$interpret = \lambda r.\ eval(r, initenv) \tag{I.13}$$

$$initenv = \lambda x.\ \big(x = \text{“succ”} \rightarrow \lambda a.\ succ(a), \ \cdots \tag{I.14}$$

$$\cdots\ x = \text{“equal”} \rightarrow \lambda a.\ \lambda b.\ equal(a, b)\big). \tag{I.15}$$

In the last line we have used a trick called Currying (after the logician H. Curry) to solve the problem of introducing a binary operation into a language where all functions must accept a single argument. (The referee comments that although "Currying" is tastier, "Schönfinkeling" might be more accurate.) In the defined language, *equal* is a function which accepts a single argument a and returns another function, which in turn accepts a single argument b and returns **true** or **false** depending upon whether $a = b$. Thus in the defined language, one would write $(equal(a))(b)$ instead of $equal(a, b)$.

(Each of our interpreters will consist of a sequence of function declarations. We will assume that these are implicitly embedded in a recursive **let** expression whose body is $interpret(R)$, where R is the program to be interpreted.)

We have coined the word "meta-circular" to indicate the basic character of this interpreter: It defines each feature of the defined language by using the corresponding feature of the defining language. For example, when *eval* is applied to an application expression (lambda expression, conditional expression, recursive **let** expression) of the defined language, it evaluates an application expression (lambda expression, conditional expression, recursive **let** expression) in the defining language. Similarly, the initial environment defines the basic functions of the defined language in terms of the same functions in the defining language.

In one sense, this situation is not undesirable. For the reader who already has a thorough and correct understanding of the defining language, a meta-circular definition will provide a concise and complete description of the defined language. (Of course this is a rather vacuous accomplishment when the defined language is a subset of the defining language.) The problem is that any misunderstandings about the defining language are likely to be carried over to the defined language intact. For example, if we were to assume that in the defining language, the function *succ* decreases an integer by one, or that a conditional expression gives the same result when the value of its premiss is non-Boolean as when it is **false**, the above interpreter would lead us to the same assumptions about the defined language.

These particular difficulties are easily overcome; we could define functions such as *succ* in terms of elementary mathematics, and we could insert explicit tests for erroneous values. But there are three objections to meta-circularity that are much more serious:

1. The meta-circular interpreter does not shed much light on the nature of higher-order functions. For this purpose, we would prefer an interpreter of a higher-order defined language that was written in a first-order defining language.

2. Changing the order of application used in the defining language induces a similar change in the defined language. To see this, suppose that *eval* is applied to an application expression $r_0(r_1)$ of the defined language. Then the result of *eval* will be obtained by evaluating the application expression (line I.4)

   $$\big(eval(r_0, e)\big)\big(eval(r_1, e)\big)$$

   in the defining language. If call by value is used in the defining language, then $eval(r_1, e)$ will be evaluated before the functional value of $eval(r_0, e)$ is applied. But evaluating $eval(r_1, e)$ interprets the evaluation of r_1, and applying the value of $eval(r_0, e)$ interprets the application of the value of r_0. Thus in terms of the defined language, r_1 will be evaluated before the value of r_0 is applied, i.e., call by value will be used in the defined language.

   On the other hand, if call by name is used in the defining language, then the application of the functional value of $eval(r_0, e)$ will begin as soon as $eval(r_0, e)$ has been evaluated, and the operand $eval(r_1, e)$ will only be evaluated when and if the function being applied depends upon its value. In terms of the defined language, the application of the value of r_0 will begin as soon as r_0 has been evaluated, and the operand r_1 will only be evaluated when and if the function being applied depends upon its value, i.e., call by name will be used in the defined language.

3. Suppose we wish to extend the defined language by introducing the imperative features of labels and jumps (including jumps out of blocks). As far as is known, it is impossible to extend the meta-circular definition straightforwardly to accommodate these features (without introducing similar features into the defining language).

In the following sections we will develop transformations of the meta-circular interpreter that will meet the first two of these objections. Then we will find that the transformation designed to meet the second objection also meets the third.

It should be emphasized that, although these transformations are motivated by their application to interpreters, they are actually applicable to any program written in the defining language, and their validity depends entirely upon the properties of the defining language.

6. Elimination of Higher-Order Functions

Our first task is to modify the meta-circular interpreter so that none of the functions that comprise this interpreter accept arguments or produce results that are functions. An examination of the abstract syntax shows that this goal will be met if we can replace the two sets FUNVAL and ENV by sets of values that are not functions. Specifically, the new members of these sets will be records that *represent* functions.

We first consider the set FUNVAL. Since the new members of this set are to be records rather than functions, we can no longer apply these members directly to arguments. Instead we will introduce a new function *apply* that will "interpret" the new members of FUNVAL. Specifically, if f_{new} is a record in FUNVAL that represents a function f_{old} and if a is any member of VAL, then $apply(f_{new}, a)$ will produce the same result as $f_{old}(a)$. Assuming for the moment that we will be able to define *apply*, we must replace each application of a member of FUNVAL (to an argument a) by an application of *apply* (to the member of FUNVAL and the argument a). In fact, the only such application occurs in line I.4, which must become

$$appl?(r) \rightarrow apply\big(eval(opr(r), e), eval(opnd(r), e)\big). \qquad \text{I.4}'$$

To decide upon the form of the new members of FUNVAL, we recall that whenever a function is obtained by evaluating a lambda expression, the function will be determined by two items of information: (1) the lambda expression itself, and (2) the values that were bound to the global variables of the lambda expression at the time of its evaluation. It is evident that these items of information will be sufficient to represent the function. This suggests that the new set FUNVAL should be a union of disjoint sets of records, one set for each lambda expression whose value belonged to the old FUNVAL, and that the fields of each record should contain values of the global variables of the corresponding lambda expression.

In fact, the meta-circular interpreter contains four lambda expressions (indicated by solid underlining) that produce members of FUNVAL. The following table gives their locations and global variables, and the equations defining the new sets of records that will represent their values. (The connotations of the set and selector names we have chosen will become apparent when we discuss the role of these entities in the interpretation of the defined language.)

Location	Global Variables	New Record Equation
I.11	ℓ e	CLOSR = [*lam*: LAMBDA, *en*: ENV]
I.14	none	SC = []
I.15 (outer)	none	EQ1 = []
I.15 (inner)	a	EQ2 = [*arg1*: VAL]

Thus the new set FUNVAL will be

$$\text{FUNVAL} = \text{CLOSR} \cup \text{SC} \cup \text{EQ1} \cup \text{EQ2},$$

and the overall structure of *apply* will be:

$$\begin{aligned}&\mathit{apply} = \lambda(f, a).\\ &\quad\big(\mathit{closr?}(f) \to \cdots,\\ &\quad\mathit{sc?}(f) \to \cdots,\\ &\quad\mathit{eq1?}(f) \to \cdots,\\ &\quad\mathit{eq2?}(f) \to \cdots\big).\end{aligned}$$

Our remaining task is to replace each of the four solidly underlined lambda expressions by appropriate record-creation operations, and to insert expressions in the branches of *apply* that will interpret the corresponding records. The lambda expression in line I.11 must be replaced by an expression that creates a CLOSR-record containing the value of the global variables ℓ and e:

$$\mathit{evlambda} = \lambda(\ell, e).\ \mathit{mk\text{-}closr}(\ell, e). \qquad \text{I.11}'$$

Now $\mathit{apply}(f, a)$ must produce the result of applying the function represented by f to the argument a. When f is a CLOSR-record, this result may be obtained by evaluating the body

$$\mathit{eval}\big(\mathit{body}(\ell), \mathit{ext}(\mathit{fp}(\ell), a, e)\big)$$

of the replaced lambda expression in an appropriate environment. This environment must bind the formal parameter a of the replaced lambda expression to the value of a and must bind the global variables ℓ and e of the lambda expression to the same value as the environment in which the CLOSR-record f was created. Since the latter values are stored in the fields of f, we have:

$$\begin{aligned}&\mathit{apply} = \lambda(f, a).\\ &\quad\big(\mathit{closr?}(f) \to \mathbf{let}\ a = a\ \mathbf{and}\ \ell = \mathit{lam}(f)\ \mathbf{and}\ e = \mathit{en}(f)\\ &\qquad\mathbf{in}\ \mathit{eval}\big(\mathit{body}(\ell), \mathit{ext}(\mathit{fp}(\ell), a, e)\big),\\ &\quad\ldots\big).\end{aligned}$$

(In this particular case, but not in general, the declaration $a = a$ is unnecessary, since the formal parameter of the replaced lambda expression and the second formal parameter of *apply* are the same variable. From now on, we will omit such vacuous declarations.)

A similar treatment (somewhat simplified since there are no global variables) of the lambda expression in I.14 and the outer lambda expression in I.15 gives:

$$\mathit{initenv} = \lambda x.\ \big(x = \text{“succ”} \to \mathit{mk\text{-}sc}(),\ \cdots \qquad \text{I.14}'$$

$$\cdots\ x = \text{“equal”} \to \mathit{mk\text{-}eq1}()\big) \qquad \text{I.15}'$$

and

$$
\begin{aligned}
&apply = \lambda(f,a).\\
&\quad \big(closr?(f) \to \mathbf{let}\ \ell = lam(f)\ \mathbf{and}\ e = en(f)\\
&\quad\quad \mathbf{in}\ eval\big(body(\ell), ext(fp(\ell), a, e)\big),\\
&\quad sc?(f) \to succ(a),\\
&\quad eq1?(f) \to \underline{\lambda b.\ equal(a,b)},\\
&\quad eq2?(f) \to \cdots\big).
\end{aligned}
$$

Finally, we must replace the lambda expression that originally occurred as the inner expression in I.15. Although we have already moved this expression into the body of *apply* (since it was the body of a previously replaced lambda expression), the same basic treatment can be applied to the new occurrence, giving:

$$
\begin{aligned}
&apply = \lambda(f,a).\\
&\quad \big(closr?(f) \to \mathbf{let}\ \ell = lam(f)\ \mathbf{and}\ e = en(f)\\
&\quad\quad \mathbf{in}\ eval\big(body(\ell), ext(fp(\ell), a, e)\big),\\
&\quad sc?(f) \to succ(a),\\
&\quad eq1?(f) \to mk\text{-}eq2(a),\\
&\quad eq2?(f) \to \mathbf{let}\ b = a\ \mathbf{and}\ a = arg1(f)\ \mathbf{in}\ equal(a,b)\big).
\end{aligned}
$$

(Note that the declaration relating formal parameters is not vacuous in this case.)

The entire transformation that converts FUNVAL from a set of functions to a set of records has been informally justified by appealing to an understanding of the defining language, without regard to the meaning or use of the particular program being transformed. But now it is illuminating to examine the different kinds of records in FUNVAL in terms of their role in the interpretation of the defined language. The records in the set CLOSR represent functional values that are produced by evaluating the lambda expressions occurring in the defined language programs. They are equivalent to the objects called *FUNARG triplets* in LISP and *closures* in the work of Landin [7]. The unique records in the one-element sets SC and EQ1 represent the basic functions *succ* and *equal*. Finally, the records in EQ2 represent the functions that are created by applying *equal* to one argument.

A similar transformation can be used to "defunctionalize" the set ENV of environments. To interpret the new members of ENV, we will introduce a function *get*, with the property that if e_{new} represents an environment e_{old} and x is a member of VAR, then $get(e_{new}, x) = e_{old}(x)$. Applications of *get* must be inserted at the three points (in lines I.3, I.9, and I.12) in the interpreter where environments are applied to variables:

$$var?(r) \to get(e,r), \qquad \text{I.3}'$$

$$\vdots$$

$$\underline{\lambda x.\ \mathbf{if}\ x = dvar(r)\ \mathbf{then}\ evlambda(dexp(r), e')\ \mathbf{else}\ get(e,x)} \qquad \text{I.9}'$$

$$\vdots$$

$$ext = \lambda(z,a,e).\ \underline{\lambda x.\ \mathbf{if}\ x = z\ \mathbf{then}\ a\ \mathbf{else}\ get(e,x)}. \qquad \text{I.12}'$$

Next, there are three lambda expressions that produce environments; they are indicated by broken underlining which we have carefully preserved during the previous transformations. The following table gives their locations and global variables, and the equations defining the new sets of records that will represent their values:

Location	Global Variables	New Record Equation
I.14′-15′	none	INIT = []
I.12′	z a e	SIMP = [*bvar*: VAR, *bval*: VAL, *old*: ENV]
I.9′	r e e'	REC = [*letx*: LETREC, *old*: ENV, *new*: ENV]

Thus the new set of environment representations is:

$$\text{ENV} = \text{INIT} \cup \text{SIMP} \cup \text{REC}.$$

Replacement of the three environment-producing lambda expressions gives:

$$\mathit{letrec?}(r) \rightarrow \mathbf{letrec}\; e' = \mathit{mk\text{-}rec}(r, e, e') \;\cdots \qquad \text{I.8-9}''$$

$$\vdots$$

$$\mathit{ext} = \lambda(z, a, e).\; \mathit{mk\text{-}simp}(z, a, e) \qquad \text{I.12}''$$

$$\vdots$$

$$\mathit{initenv} = \mathit{mk\text{-}init}(), \qquad \text{I.14}''\text{-15}''$$

and the environment-interpreting function is:

$$
\begin{aligned}
&\mathit{get} = \lambda(e, x).\\
&\quad \big(\mathit{init?}(e) \rightarrow \big(x = \text{“succ”} \rightarrow \mathit{mk\text{-}sc}(), x = \text{“equal”} \rightarrow \mathit{mk\text{-}eq1}()\big),\\
&\quad \mathit{simp?}(e) \rightarrow \mathbf{let}\; z = \mathit{bvar}(e) \;\mathbf{and}\; a = \mathit{bval}(e) \;\mathbf{and}\; e = \mathit{old}(e)\\
&\qquad \mathbf{in\ if}\; x = z \;\mathbf{then}\; a \;\mathbf{else}\; \mathit{get}(e, x),\\
&\quad \mathit{rec?}(e) \rightarrow \mathbf{let}\; r = \mathit{letx}(e) \;\mathbf{and}\; e = \mathit{old}(e) \;\mathbf{and}\; e' = \mathit{new}(e)\\
&\qquad \mathbf{in\ if}\; x = \mathit{dvar}(r) \;\mathbf{then}\; \mathit{evlambda}(\mathit{dexp}(r), e') \;\mathbf{else}\; \mathit{get}(e, x)\big).
\end{aligned}
$$

But now we are faced with a new problem. By eliminating the lambda expression in I.9′, we have created a recursive **let** expression

$$\mathbf{letrec}\; e' = \mathit{mk\text{-}rec}(r, e, e') \;\cdots$$

that violates the structure of the defining language, since its declaring subexpression is no longer a lambda expression. However, there is still an obvious intuitive interpretation of this illicit construction: it binds e' to a “cyclic” record, whose last field is (a pointer to) the record itself.

If we accept this interpretation, then whenever e is a member of REC, we will have $\mathit{new}(e) = e$. This allows us to replace the only occurrence of $\mathit{new}(e)$ by e, so that the penultimate line of *get* becomes:

$$\mathit{rec?}(e) \rightarrow \mathbf{let}\; r = \mathit{letx}(e) \;\mathbf{and}\; e = \mathit{old}(e) \;\mathbf{and}\; e' = e \;\cdots.$$

But now our program no longer contains any references to the cyclic *new* fields, so that these fields can be deleted from the records in REC. Thus the record equation for REC is reduced to:

$$\text{REC} = [\textit{letx}\colon \text{LETREC}, \textit{old}\colon \text{ENV}],$$

and the offending recursive **let** expression becomes:

$$\textit{letrec?}(r) \rightarrow \textbf{let } e' = \textit{mk-rec}(r, e) \ \cdots . \qquad \text{I.8}'\text{-}9'''$$

At this point, once we have collected the bits and pieces produced by the various transformations, we will have obtained an interpreter that no longer contains any higher-order functions. However, it is convenient to make a few simplications:

1. **let** expressions can be eliminated by substituting the declaring expressions for each occurrence of the corresponding declared variables in the body.

2. Line I.11$'$ can be eliminated by replacing occurrences of *evlambda* by *mk-closr*.

3. Line I.12$''$ can be eliminated by replacing occurrences of *ext* by *mk-simp*.

4. Lines I.14$''$-15$''$ can be eliminated by replacing occurrences of *initenv* by *mk-init*().

Thus we obtain our second interpreter:

$$
\begin{array}{ll}
\text{FUNVAL} = \text{CLOSR} \cup \text{SC} \cup \text{EQ1} \cup \text{EQ2} & \\
\text{CLOSR} = [\textit{lam}\colon \text{LAMBDA}, \textit{en}\colon \text{ENV}] & \\
\text{SC} = [\,] & \\
\text{EQ1} = [\,] & \\
\text{EQ2} = [\textit{arg1}\colon \text{VAL}] & \\
\text{ENV} = \text{INIT} \cup \text{SIMP} \cup \text{REC} & \\
\text{INIT} = [\,] & \\
\text{SIMP} = [\textit{bvar}\colon \text{VAR}, \textit{bval}\colon \text{VAL}, \textit{old}\colon \text{ENV}] & \\
\text{REC} = [\textit{letx}\colon \text{LETREC}, \textit{old}\colon \text{ENV}] & \\
\textit{interpret} = \lambda r.\ \textit{eval}\big(r, \textit{mk-init}()\big) & \text{II.1} \\
\textit{eval} = \lambda(r, e). & \text{II.2} \\
\quad \big(\textit{const?}(r) \rightarrow \textit{evcon}(r), & \text{II.3} \\
\quad \textit{var?}(r) \rightarrow \textit{get}(e, r), & \text{II.4} \\
\quad \textit{appl?}(r) \rightarrow \textit{apply}\big(\textit{eval}(\textit{opr}(r), e), \textit{eval}(\textit{opnd}(r), e)\big), & \text{II.5} \\
\quad \textit{lambda?}(r) \rightarrow \textit{mk-closr}(r, e), & \text{II.6} \\
\quad \textit{cond?}(r) \rightarrow \textbf{if } \textit{eval}(\textit{prem}(r), e) & \text{II.7} \\
\qquad \textbf{then } \textit{eval}(\textit{conc}(r), e) \textbf{ else } \textit{eval}(\textit{altr}(r), e), & \text{II.8} \\
\quad \textit{letrec?}(r) \rightarrow \textit{eval}(\textit{body}(r), \textit{mk-rec}(r, e))\big) & \text{II.9}
\end{array}
$$

$apply = \lambda(f, a).$ II.10

$\big(closr?(f) \rightarrow$ II.11

$eval\big(body(lam(f)), mk\text{-}simp(fp(lam(f)), a, en(f))\big),$ II.12

$sc?(f) \rightarrow succ(a),$ II.13

$eq1?(f) \rightarrow mk\text{-}eq2(a),$ II.14

$eq2?(f) \rightarrow equal(arg1(f), a)\big)$ II.15

$get = \lambda(e, x).$ II.16

$\big(init?(e) \rightarrow \big(x = \text{“succ”} \rightarrow mk\text{-}sc(), x = \text{“equal”} \rightarrow mk\text{-}eq1()\big),$ II.17

$simp?(e) \rightarrow \mathbf{if}\ x = bvar(e)\ \mathbf{then}\ bval(e)\ \mathbf{else}\ get(old(e), x),$ II.18

$rec?(e) \rightarrow \mathbf{if}\ x = dvar(letx(e))$ II.19

$\mathbf{then}\ mk\text{-}closr(dexp(letx(e)), e)\ \mathbf{else}\ get(old(e), x)\big).$ II.20

Just as with FUNVAL, we may examine the different kinds of records in ENV with regard to their role in the interpretation of the defined language. The unique record in INIT has no subfields, while the records in SIMP and REC each have one field (selected by *old*) that is another member of ENV. Thus environments in our second interpreter are linear lists (in which each element specifies the binding of a single variable), and the unique record in INIT serves as the empty list.

It is easily seen that $get(e, x)$ searches such a list to find the binding of the variable x. When *get* encounters a record in SIMP, it compares x with the *bvar* field, and if a match occurs, it returns the value stored in the *bval* field. When get encounters a record in REC, it compares x with $dvar(letx(e))$ (the declared variable of the recursive **let** expression that created the binding), and if a match occurs, it returns the value obtained by evaluating $dexp(letx(e))$ (the declaring subexpression of the same recursive **let** expression) in the environment e. The fact that e includes the very binding that is being “looked up” reflects the essential recursive characteristic that the declaring subexpression should “feel” the effect of the declaration in which it is embedded. When *get* encounters the empty list, it compares x with each of the predefined variables, and if a match is found, it returns the appropriate value.

The definition of *get* reveals the consequences of our restricting recursive **let** expressions by requiring that their declaring subexpressions should be lambda expressions. Because of this restriction, the declaring subexpressions are always evaluated by the trivial operation of forming a closure. Therefore, the function *get* always terminates, since it never calls any other recursive function, and can never call itself more times than the length of the list that it is searching. (On the other hand, if we had permitted arbitrary declaring subexpressions, line II.20 would contain $eval(dexp(letx(e)), e)$ instead of $mk\text{-}closr(dexp(letx(e)), e)$. This seemingly slight modification would convert *get* into a function that might run on forever, as for example, when looking up the variable k in an environment created by the defined-language construction **letrec** $k = k + 1$ **in** $\cdots$.)

The second interpreter is similar in style, and in many details, to McCarthy's definition of LISP [1]. The main differences arise from our insistence upon FUNARG binding, the use of recursive **let** expressions instead of label expressions, and the use of predefined variables instead of variables with flagged property lists.

7. Continuations

The transition from the meta-circular interpreter to our second interpreter has not eliminated order-of-application dependence. It can easily be seen that a change in the order of application used in the defining-language expression (in II.5)

$$apply\big(eval(opr(r), e), eval(opnd(r), e)\big)$$

will cause a similar change for all application expressions of the defined language.

To eliminate this dependence, we must first identify the circumstances under which an arbitrary program in the defining language will be affected by the order of application. The essential effect of switching from call by value to call by name is to postpone the evaluation of the operands of application expressions (and declaring subexpressions of **let** expressions), and to alter the number of times these operands are evaluated. We have already seen that in a purely applicative language, the only way in which this change can affect the meaning of a program is to avoid the evaluation of a nonterminating operand. Now suppose we define an expression to be *serious* if there is any possibility that its evaluation might not terminate. Then a sufficient condition for order-of-application independence is that a program should contain no serious operands or declaring expressions.

Next, suppose that we can divide the functions that may be applied by our program into *serious* functions, whose application may sometimes run on forever, and *trivial* functions, whose application will always terminate. (Of course, it is well-known that one cannot effectively decide whether an arbitrary function will always terminate, but one can still establish this classification in a "fail-safe" manner, i.e., classify a function as serious unless it can be shown to terminate for all arguments.) Then an expression will only be serious if its evaluation can cause the application of a serious function, and a program will be independent of order-of-application if no operand or declaring expression can cause such an application.

At first sight, this condition appears to be so restrictive that it could not be met in a nontrivial program. As can be seen with a little thought, the condition implies that whenever some function calls a serious function, the calling function must return the same result as the called function, without performing any further computation. But any function that calls a serious function must be serious itself. Thus by induction, as soon as any serious function returns a result, every function must immediately return the same result, which must therefore be the final result of the entire program.

Nevertheless, there is a method for transforming an arbitrary program into one that meets our apparently restrictive condition. The underlying idea has appeared in a variety of contexts [26, 27, 28], but its application to definitional interpreters is due to L. Morris [20] and Wadsworth. Basically, one replaces each serious function f_{old} (except the main program) by a new serious function f_{new} that accepts an additional argument c called a *continuation*. The continuation will be a function itself, and f_{new} is expected to compute the same result as f_{old}, apply the continuation to this result, and then return the result of the continuation, i.e.,

$$f_{new}(x_1, \ldots, x_n, c) = c\big(f_{old}(x_1, \ldots, x_n)\big).$$

This introduction of continuations provides an additional "degree of freedom" that can be used to meet the condition of order-of-application independence. Essentially, instead of performing further actions after a serious function has returned, one embeds the further actions in the continuation that is passed to the serious function.

To transform our second interpreter, we must first classify its functions. Since the defined language contains expressions and functions whose evaluation and application may never terminate, the defining-language functions *eval* and *apply* are serious and must be altered to accept continuations. On the other hand, since we have seen that *get* always terminates, it is trivial and will not be altered. (Note that this situation would change if the defined language permitted recursive **let** expressions with arbitrary declaring subexpressions.)

Both *eval* and *apply* produce results in the set VAL, so that the arguments of continuations will belong to this set. The result of a continuation will always be the value of the entire program being interpreted, which will also belong to the set VAL. Thus the set of continuations is:

$$\text{CONT} = \text{VAL} \rightarrow \text{VAL}.$$

(In a more complicated interpreter in which different serious functions produced different kinds of results, we would introduce different kinds of continuations.)

The overall form of our transformed interpreter will be:

$$
\begin{array}{ll}
\mathit{interpret} = \lambda r.\ \mathit{eval}(r, \mathit{mk\text{-}init}(), \lambda a.\ a) & \text{II.1}' \\
\mathit{eval} = \lambda(r, e, c).\ \cdots & \text{II.2}' \\
\mathit{apply} = \lambda(f, a, c).\ \cdots & \text{II.10}' \\
\mathit{get} = \text{same as in Interpreter II.} & \text{II.16–20}
\end{array}
$$

Note that the "main level" call of *eval* by *interpret* provides an identity function as the initial continuation.

We must now alter each branch of *eval* and *apply* to apply the continuation c to the former results of these functions. In lines II.3, 4, 6, 13, 14, and 15, the branches evaluate expressions which are not serious, and which are therefore permissible operands. Thus in these cases, we may simply apply the continuation c to each expression:

$$
\begin{array}{ll}
\mathit{eval} = \lambda(r, e, c). & \text{II.2}' \\
\quad \big(\mathit{const?}(r) \rightarrow c(\mathit{evcon}(r)), & \text{II.3}' \\
\quad \mathit{var?}(r) \rightarrow c(\mathit{get}(e, r)), & \text{II.4}' \\
\qquad \vdots & \\
\quad \mathit{lambda?}(r) \rightarrow c(\mathit{mk\text{-}closr}(r, e)),\ \ldots\big) & \text{II.6}' \\
\mathit{apply} = \lambda(f, a, c).\ \big(\ldots\ , & \text{II.10}' \\
\quad \mathit{sc?}(f) \rightarrow c(\mathit{succ}(a)), & \text{II.13}' \\
\quad \mathit{eq1?}(f) \rightarrow c(\mathit{mk\text{-}eq2}(a)), & \text{II.14}' \\
\quad \mathit{eq2?}(f) \rightarrow c(\mathit{equal}(\mathit{arg1}(f), a))\big). & \text{II.15}'
\end{array}
$$

In lines II.9 and II.12, the branches evaluate expressions that are serious themselves but contain no serious operands. By themselves, these expressions are permissible, but they must not be used as operands in applications of the continuation. The solution is straightforward; instead of applying the continuation c to the result of *eval*, we pass c as an argument to *eval*, i.e., we "instruct" *eval* to apply c before returning its result:

$$\mathit{letrec?}(r) \rightarrow \mathit{eval}(\mathit{body}(r), \mathit{mk\text{-}rec}(r, e), c)) \qquad \text{II.9}'$$

$$\vdots$$

$$\big(\mathit{closr?}(f) \rightarrow \qquad \text{II.11}'$$

$$\mathit{eval}\big(\mathit{body}(\mathit{lam}(f)), \mathit{mk\text{-}simp}(\mathit{fp}(\mathit{lam}(f)), a, \mathit{en}(f)), c\big). \qquad \text{II.12}'$$

The most complex part of our transformation occurs in the branch of eval that evaluates application expressions in line II.5. Here we must perform four serious operations:

1. Evaluate the operator.

2. Evaluate the operand.

3. Apply the value of the operator to the value of the operand.

4. Apply the continuation c to the result of (3).

Moreover, we must specify explicitly that these operations are to be done in the above order. This will insure that the defined language uses call by value, and also that the subexpressions of an application expression are evaluated from left to right (operator before operand).

The solution is to call *eval* to perform operation (1), to give this call of *eval* a continuation that will call *eval* to perform operation (2), to give the second call of *eval* a continuation that will call *apply* to perform (3), and to give *apply* a continuation (the original continuation c) that will perform (4). Thus we have:

$$\mathit{appl?}(r) \rightarrow \mathit{eval}\big(\mathit{opr}(r), e, \lambda f.\ \mathit{eval}\big(\mathit{opnd}(r), e, \lambda a.\ \mathit{apply}(f, a, c)\big)\big). \qquad \text{II.5}'$$

A similar approach handles the branch that evaluates conditional expressions in lines II.7 and 8. Here there are three serious operations to be performed successively:

1. Evaluate the premiss.

2. Evaluate the conclusion or the alternative, depending on the result of (1).

3. Apply the continuation c to the result of (2).

The transformed branch is:

$$\mathit{cond?}(r) \rightarrow \mathit{eval}\big(\mathit{prem}(r), e, \qquad \text{II.7}'$$

$$\lambda b.\ \textbf{if}\ b\ \textbf{then}\ \mathit{eval}(\mathit{conc}(r), e, c)\ \textbf{else}\ \mathit{eval}(\mathit{altr}(r), e, c)\big). \qquad \text{II.8}'$$

Combining the scattered pieces of our transformed interpreter, we have:

$$
\begin{array}{lr}
\mathit{interpret} = \lambda r.\ \mathit{eval}(r, \mathit{mk\text{-}init}(), \underline{\lambda a.\ a}) & \text{II.1}' \\
\mathit{eval} = \lambda(r, e, c). & \text{II.2}' \\
\quad \big(\mathit{const?}(r) \rightarrow c(\mathit{evcon}(r)), & \text{II.3}' \\
\quad \mathit{var?}(r) \rightarrow c(\mathit{get}(e, r)), & \text{II.4}' \\
\quad \mathit{appl?}(r) \rightarrow \mathit{eval}\big(\mathit{opr}(r), e, \underline{\lambda f.\ \mathit{eval}\big(\mathit{opnd}(r), e, \underline{\lambda a.\ \mathit{apply}(f, a, c)}\big)}\big), & \text{II.5}' \\
\quad \mathit{lambda?}(r) \rightarrow c(\mathit{mk\text{-}closr}(r, e)), & \text{II.6}' \\
\quad \mathit{cond?}(r) \rightarrow \mathit{eval}\big(\mathit{prem}(r), e, & \text{II.7}' \\
\qquad \underline{\lambda b.\ \mathbf{if}\ b\ \mathbf{then}\ \mathit{eval}(\mathit{conc}(r), e, c)\ \mathbf{else}\ \mathit{eval}(\mathit{altr}(r), e, c)}\big), & \text{II.8}' \\
\quad \mathit{letrec?}(r) \rightarrow \mathit{eval}(\mathit{body}(r), \mathit{mk\text{-}rec}(r, e), c)\big) & \text{II.9}' \\
\mathit{apply} = \lambda(f, a, c). & \text{II.10}' \\
\quad \big(\mathit{closr?}(f) \rightarrow & \text{II.11}' \\
\qquad \mathit{eval}\big(\mathit{body}(\mathit{lam}(f)), \mathit{mk\text{-}simp}(\mathit{fp}(\mathit{lam}(f)), a, \mathit{en}(f)), c\big), & \text{II.12}' \\
\quad \mathit{sc?}(f) \rightarrow c(\mathit{succ}(a)), & \text{II.13}' \\
\quad \mathit{eq1?}(f) \rightarrow c(\mathit{mk\text{-}eq2}(a)), & \text{II.14}' \\
\quad \mathit{eq2?}(f) \rightarrow c(\mathit{equal}(\mathit{arg1}(f), a))\big) & \text{II.15}' \\
\mathit{get} = \text{same as in Interpreter II.} & \text{II.16–20}
\end{array}
$$

At this stage, since continuations are functional arguments, we have achieved order-of-application independence at the price of re-introducing higher-order functions. Fortunately, we can now "defunctionalize" the set CONT in the same way as FUNVAL and ENV. To interpret the new members of CONT we introduce a function *cont* such that if c_{new} represents the continuation c_{old} and a is a member of VAL then $\mathit{cont}(c_{new}, a) = c_{old}(a)$. The application of *cont* must be introduced at each point in *eval* and *apply* where a continuation is applied to a value, i.e., in lines II.3′, 4′, 6′, 13′, 14′, and 15′.

There are four lambda expressions, indicated by solid underlining, that create continuations. The following table gives their locations and global variables, and the equations defining the new sets of records that will represent their values:

Location	Global Variables	New Record Equation
II.1′	none	FIN = []
II.5′ (outer)	$r\ e\ c$	EVOPN = [*ap*: APPL, *en*: ENV, *next*: CONT]
II.5′ (inner)	$f\ c$	APFUN = [*fun*: VAL, *next*: CONT]
II.8′	$r\ e\ c$	BRANCH = [*cn*: COND, *en*: ENV, *next*: CONT]

By replacing these lambda expressions by record-creation operations and moving their bodies into the new function *cont* (within **let** expressions that rebind their formal parameters and global variables appropriately), we obtain a third interpreter, which is independent of order-of-application and does not use higher-order functions:

$$
\begin{array}{ll}
\text{CONT} = \text{FIN} \cup \text{EVOPN} \cup \text{APFUN} \cup \text{BRANCH} & \\
\text{FIN} = [\,] & \\
\text{EVOPN} = [\mathit{ap}\colon \text{APPL},\ \mathit{en}\colon \text{ENV},\ \mathit{next}\colon \text{CONT}] & \\
\text{APFUN} = [\mathit{fun}\colon \text{VAL},\ \mathit{next}\colon \text{CONT}] & \\
\text{BRANCH} = [\mathit{cn}\colon \text{COND},\ \mathit{en}\colon \text{ENV},\ \mathit{next}\colon \text{CONT}] & \\
\text{FUNVAL, ENV, etc.} = \text{same as in Interpreter II.} & \\
\mathit{interpret} = \lambda r.\ \mathit{eval}(r, \mathit{mk\text{-}init}(), \mathit{mk\text{-}fin}()) & \\
\mathit{eval} = \lambda(r, e, c). & \\
\quad \big(\mathit{const?}(r) \to \mathit{cont}(c, \mathit{evcon}(r)), & \\
\quad \mathit{var?}(r) \to \mathit{cont}(c, \mathit{get}(e, r)), & \\
\quad \mathit{appl?}(r) \to \mathit{eval}(\mathit{opr}(r), e, \mathit{mk\text{-}evopn}(r, e, c)), & \\
\quad \mathit{lambda?}(r) \to \mathit{cont}(c, \mathit{mk\text{-}closr}(r, e)), & \\
\quad \mathit{cond?}(r) \to \mathit{eval}(\mathit{prem}(r), e, \mathit{mk\text{-}branch}(r, e, c)), & \\
\quad \mathit{letrec?}(r) \to \mathit{eval}(\mathit{body}(r), \mathit{mk\text{-}rec}(r, e), c)\big) & \text{III} \\
\mathit{apply} = \lambda(f, a, c). & \\
\quad \big(\mathit{closr?}(f) \to & \\
\qquad \mathit{eval}\big(\mathit{body}(\mathit{lam}(f)), \mathit{mk\text{-}simp}(\mathit{fp}(\mathit{lam}(f)), a, \mathit{en}(f)), c\big), & \\
\quad \mathit{sc?}(f) \to \mathit{cont}(c, \mathit{succ}(a)), & \\
\quad \mathit{eq1?}(f) \to \mathit{cont}(c, \mathit{mk\text{-}eq2}(a)), & \\
\quad \mathit{eq2?}(f) \to \mathit{cont}\big(c, \mathit{equal}(\mathit{arg1}(f), a)\big)\big) & \\
\mathit{cont} = \lambda(c, a). & \\
\quad \big(\mathit{fin?}(c) \to a, & \\
\quad \mathit{evopn?}(c) \to \textbf{let } f = a \textbf{ and } r = \mathit{ap}(c) \textbf{ and } e = \mathit{en}(c) \textbf{ and } c = \mathit{next}(c) & \\
\qquad \textbf{in } \mathit{eval}(\mathit{opnd}(r), e, \mathit{mk\text{-}apfun}(f, c)), & \\
\quad \mathit{apfun?}(c) \to \textbf{let } f = \mathit{fun}(c) \textbf{ and } c = \mathit{next}(c) \textbf{ in } \mathit{apply}(f, a, c), & \\
\quad \mathit{branch?}(c) \to \textbf{let } b = a \textbf{ and } r = \mathit{cn}(c) \textbf{ and } e = \mathit{en}(c) \textbf{ and } c = \mathit{next}(c) & \\
\qquad \textbf{in if } b \textbf{ then } \mathit{eval}(\mathit{conc}(r), e, c) \textbf{ else } \mathit{eval}(\mathit{altr}(r), e, c)\big) & \\
\mathit{get} = \text{same as in Interpreter II.} &
\end{array}
$$

From their abstract syntax, it is evident that continuations in our third interpreter are linear lists, with the unique record in FIN acting as the empty list, and the *next* fields in the other records acting as link fields. In effect, a continuation is a list of instructions to be interpreted by the function *cont*. Each instruction accepts a "current value" (the second argument of *cont*) and produces a new value that will be given to the next instruction. The following list gives approximate meanings for each type of instruction:

FIN: The current value is the final value of the program. Halt.

EVOPN: The current value is the value of an operator. Evaluate the operand of the application expression in the *ap* field, using the environment in the *en* field. Then obtain a new value by applying the current value to the value of the operand.

APFUN: The current value is the value of an operand. Obtain a new value by applying the function stored in the *fun* field to the current value.

BRANCH: The current value is the value of a premiss. If it is **true** (**false**) obtain a new value by evaluating the conclusion (alternative) of the conditional expression stored in the *cn* field, using the environment in the *en* field.

Each of the three serious functions, *eval*, *apply*, and *cont*, does a branch on the form of its first argument, performs trivial operations such as field selection, record creation, and environment lookup, and then calls another serious function. Thus our third interpreter is actually a state-transition machine, whose states each consist of the name of a serious function plus a list of its arguments.

This interpreter is similar in style to Landin's SECD machine [7], though there is considerable difference in detailed mechanisms. (Very roughly, one can construct the continuation by merging Landin's stack and control and concatenating this merged stack with the dump.)

8. Continuations with Higher-Order Functions

In transforming Interpreter I into Interpreter III, we have moved from a concise, abstract definition to a more complex machine-like one. If clarity consists of the avoidance of subtle characteristics of the defining language, then Interpreter III is certainly clearer than Interpreter I. But if clarity consists of conciseness and the absence of unnecessary complexity, then the reverse is true. The machine-like character of Interpreter III includes a variety of "cogs and wheels" that are quite arbitrary, i.e., one can easily construct equivalent interpreters (such as the SECD machine) with different cogs and wheels.

In fact, these "cogs and wheels" were introduced when we defunctionalized the sets FUNVAL, ENV, and CONT, since we replaced the functions in these sets by representations that were correct, but not unique. Had we chosen different representations, we would have obtained an equivalent but quite different interpreter.

This suggests the desirability of retaining the use of higher-order functions, *providing* these entities can be given a mathematically rigorous definition that is independent of any specific representation. Fortunately, such a definition has recently been provided by D. Scott's new theory of computation [12, 13, 14, 15], which is based on concepts of lattice theory and topology. (The central technical problem that Scott has solved is to define functions that are not only higher-order, but also *typeless*, so that any function may be applied to any other function, including itself.) Although a description of this work would be beyond the scope of this paper, we may summarize its main implication for definitional interpreters: Scott has developed a mathematical model of the lambda calculus, which is thereby a model for a purely applicative higher-order defining language. But the defining language modelled by Scott uses call by name rather than call by value. (In terms of the lambda calculus, it uses normal order of evaluation.) Thus to apply Scott's work to a defined language that uses call by value, we need a definitional interpreter that retains higher-order functions but is order-of-application independent.

An obvious approach to this goal is to introduce continuations directly into the meta-circular interpreter. At first sight, this appears to be straightforward. Referring back to Interpreter I, we see that the function *eval* is obviously serious, while *evlambda*, *ext* and *initenv* are trivial. (*evlambda* is trivial since the evaluation of lambda expressions always terminates.) Apparently *eval* is the only function that must accept continuations.

But when we transform the branch of *eval* that evaluates application expressions, the construction described in the previous section seems to give:

$$appl?(r) \rightarrow eval\big(opr(r), e, \lambda f.\ eval\big(opnd(r), e, \lambda a.\ c(f(a))\big)\big).$$

Unfortunately, the subexpression $c(f(a))$ is not independent of the order-of-application, since the evaluation of the operand $f(a)$ may never terminate, while the function c may be independent of its argument.

The difficulty is that the class of serious functions must include every potentially nonterminating function that may be applied during the execution of the interpreter; in addition to *eval*, this class contains the members of the set FUNVAL of defined-language functional values. Thus we must modify the functions in FUNVAL to accept continuations:

$$\text{FUNVAL} = \text{VAL, CONT} \rightarrow \text{VAL},$$

replacing each function f_{old} by an f_{new} such that $f_{new}(a, c) = c(f_{old}(a))$. This allows us to replace the order-dependent expression $c(f(a))$ by the order-independent expression $f(a, c)$. Of course, we must add continuations as an extra formal parameter to each lambda expression that creates a member of FUNVAL.

(A similar modification of the functions in ENV is unnecessary, since it can be shown that the functions in this set always terminate. Just as with *get*, this depends on the exclusion of recursive **let** expressions with arbitrary declaring subexpressions.)

Once the necessity of altering FUNVAL has been realized, the transformation of Interpreter I follows the basic lines described in the previous section. We omit the details and state the final result:

$$\text{VAL} = \text{INTEGER} \cup \text{BOOLEAN} \cup \text{FUNVAL}$$
$$\text{FUNVAL} = \text{VAL, CONT} \rightarrow \text{VAL}$$
$$\text{ENV} = \text{VAR} \rightarrow \text{VAL}$$
$$\text{CONT} = \text{VAL} \rightarrow \text{VAL}$$
$$interpret = \lambda r.\ eval(r, initenv, \lambda a.\ a)$$

$$
\begin{aligned}
&eval = \lambda(r, e, c).\\
&\qquad \big(const?(r) \rightarrow c(evcon(r)),\\
&\qquad var?(r) \rightarrow c(e(r)),\\
&\qquad appl?(r) \rightarrow eval\big(opr(r), e, \lambda f.\ eval(opnd(r), e, \lambda a.\ f(a, c))\big),\\
&\qquad lambda?(r) \rightarrow c(evlambda(r, e)),\\
&\qquad cond?(r) \rightarrow eval\big(prem(r), e,\\
&\qquad\qquad \lambda b.\ \textbf{if}\ b\ \textbf{then}\ eval(conc(r), e, c)\ \textbf{else}\ eval(altr(r), e, c)\big),\\
&\qquad letrec?(r) \rightarrow \textbf{letrec}\ e' =\\
&\qquad\qquad\qquad \lambda x.\ \textbf{if}\ x = dvar(r)\ \textbf{then}\ evlambda(dexp(r), e')\ \textbf{else}\ e(x)\\
&\qquad\qquad \textbf{in}\ eval(body(r), e', c)\big)\\
&evlambda = \lambda(\ell, e).\ \lambda(a, c).\ eval\big(body(\ell), ext(fp(\ell), a, e), c\big)\\
&ext = \lambda(z, a, e).\ \lambda x.\ \textbf{if}\ x = z\ \textbf{then}\ a\ \textbf{else}\ e(x)\\
&initenv = \lambda x.\ \big(x = \text{“succ”} \rightarrow \lambda(a, c).\ c(succ(a)),\\
&\qquad x = \text{“equal”} \rightarrow \lambda(a, c).\ c\big(\lambda(b, c').\ c'(equal(a, b))\big)\big).
\end{aligned}
\qquad \text{IV}
$$

This is basically the form of interpreter devised by L. Morris [20] and Wadsworth. It is almost as concise as the meta-circular interpreter, yet it offers the advantages of order-of-application independence and, as we will see in the next section, extensibility to accommodate imperative control features.

(The zealous reader may wish to verify that defunctionalization and the introduction of continuations are commutative, i.e., by replacing FUNVAL, ENV, and CONT by appropriate nonfunctional representations, one can transform Interpreter IV into Interpreter III.)

9. Escape Expressions

We now turn to the problem of adding imperative features to the defined language (while keeping the defining language purely applicative). These features may be divided into two classes:

1. Imperative control mechanisms, e.g., statement sequencing, labels and jumps.

2. Assignment.

We will first introduce control mechanisms and then consider assignment.

At first sight, this order of presentation seems facetious. In a language without assignment, it seems pointless to jump to a label, since there is no significant way for the part of the computation before the jump to influence the part afterwards. However, in Reference [29], Landin introduced an imperative control mechanism that is more general than labels and jumps, and that significantly enhances the power of a language without assignment. The specific mechanism that he introduced was called a J-operator, but in this paper we will develop a slightly simpler mechanism called an escape expression.

If (in the defined language) x is a variable and r is an expression, then

escape x **in** r

is an *escape expression*, whose *escape variable* is x and whose *body* is r. The evaluation of an escape expression in an environment e proceeds as follows:

1. The body r is evaluated in the environment that is the extension of e that binds x to a function called the *escape function.*
2. If the escape function is never applied during the evaluation of r, then the value of r becomes the value of the escape expression.
3. If the escape function is applied to an argument a, then the evaluation of the body r is aborted, and a immediately becomes the value of the escape expression.

Essentially, an escape function is a kind of label, and its application is a kind of jump. The greater generality lies in the ability to pass arguments while jumping.

(Landin's J-operator can be defined in terms of the escape expression by regarding **let** $g = J\ \lambda x.\ r_1$ **in** r_0 as an abbreviation for **escape** h **in let** $g = \lambda x.\ h(r_1)$ **in** r_0, where h is a new variable not occurring in r_0 or r_1. Conversely, one can regard **escape** g **in** r as an abbreviation for **let** $g = J\ \lambda x.\ x$ **in** r.)

In order to extend our interpreters to handle escape expressions, we begin by extending the abstract syntax of expressions appropriately:

$$\begin{aligned}&\text{EXP} = \ldots \cup \text{ESCP}\\&\text{ESCP} = [\mathit{escv}\text{: VAR, } \mathit{body}\text{: EXP}].\end{aligned}$$

It is evident that in each interpreter we must add a branch to *eval* that evaluates the new kind of expression.

First consider Interpreter IV. Since an escape expression is evaluated by evaluating its body in an extended environment that binds the escape variable to the escape function, and since the escape function must be represented by a member of the set FUNVAL = VAL, CONT $\rightarrow$ VAL, we have

$$\begin{aligned}&\mathit{eval} = \lambda(r, e, c).\ \big(\ \ldots\ ,\\&\qquad \mathit{escp?}(r) \rightarrow \mathit{eval}\big(\mathit{body}(r), \mathit{ext}(\mathit{escv}(r), \lambda(a, c').\ \ldots, e), c\big)\big),\end{aligned}$$

where the value of $\lambda(a, c').\ \ldots$ must be the member of FUNVAL representing the escape function.

Since *eval* is a serious function, its result, which is obtained by applying the continuation c to the value of the escape expression, must be the final result of the entire program being interpreted. This means that c itself must be a function that will accept the value of the escape expression and carry out the interpretation of the remainder of the program. But the member of FUNVAL representing the escape function is also serious, and must therefore also produce the final result of the entire program. Thus to abort the evaluation of the body and treat the argument a as the value of the escape expression, it is only necessary for the escape function ignore its own continuation c', and to apply the higher-level continuation c to a. Thus we have:

$$\begin{aligned}&\mathit{eval} = \lambda(r, e, c).\ \bigl(\ \ldots\ ,\\ &\qquad \mathit{escp?}(r) \to \mathit{eval}\bigl(\mathit{body}(r), \mathit{ext}(\mathit{escv}(r), \lambda(a, c').\ c(a), e), c)\bigr).\end{aligned}$$

The extension of Interpreter III is essentially similar. In this case, we must add to the set FUNVAL a new kind of record that represents escape functions:

$$\begin{aligned}&\mathrm{FUNVAL} = \ldots \cup \mathrm{ESCF}\\ &\mathrm{ESCF} = [\mathit{cn}\colon \mathrm{CONT}].\end{aligned}$$

These records are created in the new branch of *eval*:

$$\begin{aligned}&\mathit{eval} = \lambda(r, e, c).\ \bigl(\ \ldots\ ,\\ &\qquad \mathit{escp?}(r) \to \mathit{eval}\bigl(\mathit{body}(r), \mathit{mk\text{-}simp}(\mathit{escv}(r), \mathit{mk\text{-}escf}(c), e), c)\bigr),\end{aligned}$$

and are interpreted by a new branch of *apply*:

$$\begin{aligned}&\mathit{apply} = \lambda(f, a, c).\ \bigl(\ \ldots\ ,\\ &\qquad \mathit{escf?}(f) \to \mathit{cont}(\mathit{cn}(f), a)\bigr).\end{aligned}$$

From the viewpoint of this interpreter, it is clear that the escape expression is a significant extension of the defined language, since it introduces the possibility of embedding continuations in values.

(The reader should be warned that either of the above interpreters is a more precise definition of the escape expression than the informal English description given beforehand. For example, it is possible that the evaluation of the body of an escape expression may not cause the application of the escape function, but may produce the escape function (or some function that can call the escape function) as its value. It is difficult to infer the consequences of such a situation from our informal description, but it is precisely defined by either of the interpreters. In fact, the possibility that an escape function may propagate outside of the expression that created it is a powerful facility that can be used to construct control-flow mechanisms such as coroutines and nondeterministic algorithms.)

When we consider Interpreters I and II, we find an entirely different situation. The ability to "jump" by switching continuations is no longer possible. An escape function must still be represented by a member of FUNVAL, but now this implies that, if the function terminates without an error stop, then its result must become the value of the application expression that applied the function. As far as is known, there is no way to define the escape expression by adding branches to Interpreter I or II (except by the "cheat" of adding imperative control mechanisms to the defining language, as in Reference [19]). The essential problem is that the information that was explicitly available in the continuations of Interpreters III and IV is implicit in the recursive structure of Interpreters I and II, and in this form it cannot be manipulated with sufficient flexibility.

We have asserted that the escape mechanism encompasses less general control mechanisms such as labels and jumps. The following description outlines the way in which these more specialized operations can be expressed in terms of the escape expression. (A more detailed exposition is given in Reference [29].)

1. In the next section we will introduce assignment in such a way that assignments can be executed during the evaluation of expressions. In this situation it is unnecessary to make a semantic distinction between expressions and statements; any statement can be regarded as an expression whose evaluation produces a dummy value.

2. A label-free sequence of statements $s_1; \cdots ; s_n$ can be regarded as an abbreviation for the expression

   $$(\cdots((\lambda x_1. \ \ldots \ \lambda x_n. \ x_n)(s_1)) \ \cdots \ (s_n)).$$

   The effect is to evaluate the statements sequentially from left to right, ignoring the value of all but the last.

3. If $s_0, \ldots, s_n$ are label-free statement sequences, and $\ell_1, \ldots, \ell_n$ are labels, then a block of the form

   $$\mathbf{begin}\ s_0, \ell_1 \colon s_1; \cdots ; \ell_n \colon s_n\ \mathbf{end}$$

   can be regarded as an abbreviation for

   $$\begin{aligned}&\mathbf{escape}\ g\ \mathbf{in\ letrec}\ \ell_1 = \lambda x.\ g(s_1; \cdots ; s_n)\ \mathbf{and}\ \ell_2 = \lambda x.\ g(s_2; \cdots ; s_n)\\ &\qquad \mathbf{and}\ \cdots\ \mathbf{and}\ \ell_n = \lambda x.\ g(s_n)\ \mathbf{in}\ (s_0; \cdots ; s_n)\end{aligned}$$

   (where g and x are new variables not occurring in the original block). The effect is that each label denotes a function that ignores its argument, evaluates the appropriate sequence of statements, and then escapes out of the enclosing block.

4. An expression of the form **goto** r can be regarded as an abbreviation for $r(0)$, i.e., a jump to a label becomes an application of the function denoted by the label to a dummy argument.

10. Assignment

Although the basic concept of assignment is well understood by any competent programmer, a surprising degree of care is needed to combine this concept with the language features we have discussed previously. Intuitively, the notion of assignment presupposes that the operations that are performed during the evaluation of a program will occur in a definite temporal order. Some of these operations will *assign* values to "variables". Other operations may be affected by these assignments; specifically, an operation may depend upon the value most recently assigned to each "variable", which we will call the value currently *possessed* by the "variable".

This suggests that for each instant during program execution, there should be an entity which specifies the set of "variables" that are present and the values that they currently possess. We will call such an entity a *memory*, and denote the set of possible memories by MEM.

The main subtlety is to realize that the "variables" discussed here are distinct from the *variables* used in previous sections. This is necessitated by the fact that most programming languages permit situations (such as might arise from the use of "call by address") in which several *variables* denote the same "variable", in the sense that assignment to one of them will change the value possessed by all. This suggests that a "variable" is actually a new kind of object to which a *variable* can be bound. Henceforth, we will call these new objects *references* rather than "variables". (Other terms used commonly in the literature are *L-value* and *name*.) We will denote the set of references by REF.

Abstractly, the nature of references and memories can be characterized by specifying an initial memory and four functions:

initmem: Contains no references.

$nextref(m)$: Produces a reference not contained in the memory m.

$augment(m, a)$: Produces a memory containing the new reference $nextref(m)$ plus the references already in m. The new reference possesses the value a, while the remaining references possess the same values as in m.

$update(m, rf, a)$: Produces a memory containing the same references as m. The reference rf (assuming it is present) possesses the value a, while the remaining references possess the same value as in m.

$lookup(m, rf)$: Produces the value possessed by the reference rf in memory m.

A simple "implementation" can be obtained by numbering references in the order of their creation [25]:

$$
\begin{aligned}
&\mathrm{REF} = [\mathit{number}\colon \mathrm{INTEGER}]\\
&\mathrm{MEM} = [\mathit{count}\colon \mathrm{INTEGER}, \mathit{possess}\colon \mathrm{INTEGER} \to \mathrm{VAL}]\\
&\mathit{initmem} = \mathit{mk\text{-}mem}(0, \lambda n.\ 0)\\
&\mathit{nextref} = \lambda m.\ \mathit{mk\text{-}ref}(\mathit{count}(m) + 1)\\
&\mathit{augment} = \lambda(m, a).\ \mathit{mk\text{-}mem}\big(\mathit{count}(m) + 1,\\
&\qquad \lambda n.\ \mathbf{if}\ n = \mathit{count}(m) + 1\ \mathbf{then}\ a\ \mathbf{else}\ (\mathit{possess}(m))(n)\big)\\
&\mathit{update} = \lambda(m, \mathit{rf}, a).\ \mathit{mk\text{-}mem}\big(\mathit{count}(m),\\
&\qquad \lambda n.\ \mathbf{if}\ n = \mathit{number}(\mathit{rf})\ \mathbf{then}\ a\ \mathbf{else}\ (\mathit{possess}(m))(n)\big)\\
&\mathit{lookup} = \lambda(m, \mathit{rf}).\ (\mathit{possess}(m))(\mathit{number}(\mathit{rf})).
\end{aligned}
$$

Our next task is to introduce memories into our interpreters. Although any of our interpreters could be so extended, we will only consider Interpreter IV.

It is evident that the operation of evaluating a defined-language expression will now depend upon a memory m and will produce a (possibly) altered memory m'. Thus the function *eval* will accept m as an additional argument. However, because of the use of continuations, m' will not be part of the result of *eval*. Instead, m' will be passed on as an additional argument to the continuation that is applied by *eval* to perform the remainder of program execution.

In a similar manner, the application of a defined-language function will depend upon and produce memories. Thus each function in the set FUNVAL will accept a memory as an additional argument, and will also pass on a memory to its continuation.

On the other hand, there are particular kinds of expressions, specifically constants, variables, and lambda expressions, whose evaluation cannot cause assignments. For this reason, the functions *evcon* and *evlambda*, and the functions in the set ENV, will not accept or produce memories.

These considerations lead to the following interpreter, in which memories propagate through the various operations in a manner that correctly reflects the temporal order of execution:

$$
\begin{array}{l}
\mathrm{VAL} = \mathrm{INTEGER} \cup \mathrm{BOOLEAN} \cup \mathrm{FUNVAL} \\
\mathrm{FUNVAL} = \mathrm{VAL},\, \mathrm{MEM},\, \mathrm{CONT} \to \mathrm{VAL} \\
\mathrm{ENV} = \mathrm{VAR} \to \mathrm{VAL} \\
\mathrm{CONT} = \mathrm{MEM},\, \mathrm{VAL} \to \mathrm{VAL} \\
interpret = \lambda r.\ eval(r, initenv, initmem, \lambda(m,a).\ a) \\
eval = \lambda(r,e,m,c). \\
\quad \big(const?(r) \to c(m, evcon(r)), \\
\quad var?(r) \to c(m, e(r)), \\
\quad appl?(r) \to eval\big(opr(r), e, m, \\
\qquad \lambda(m', f).\ eval(opnd(r), e, m', \\
\qquad\quad \lambda(m'', a).\ f(a, m'', c))\big), \\
\quad lambda?(r) \to c(m, evlambda(r,e)), \\
\quad cond?(r) \to eval\big(prem(r), e, m, \\
\qquad \lambda(m', b).\ \textbf{if}\ b\ \textbf{then}\ eval(conc(r), e, m', c)\ \textbf{else}\ eval(altr(r), e, m', c)\big), \\
\quad letrec?(r) \to \textbf{letrec}\ e' = \\
\qquad\quad \lambda x.\ \textbf{if}\ x = dvar(r)\ \textbf{then}\ evlambda(dexp(r), e')\ \textbf{else}\ e(x) \\
\qquad \textbf{in}\ eval(body(r), e', m, c), \\
\quad escp?(r) \to eval\big(body(r), ext(escv(r), \lambda(a, m', c').\ c(m', a), e), m, c\big)\big) \\
evlambda = \lambda(\ell, e).\ \lambda(a, m, c).\ eval\big(body(\ell), ext(fp(\ell), a, e), m, c\big) \\
ext = \lambda(z, a, e).\ \lambda x.\ \textbf{if}\ x = z\ \textbf{then}\ a\ \textbf{else}\ e(x) \\
initenv = \lambda x.\ \big(x = \text{“succ”} \to \lambda(a, m, c).\ c(m, succ(a)), \\
\quad x = \text{“equal”} \to \lambda(a, m, c).\ c\big(m, \lambda(b, m', c').\ c'(m', equal(a,b))\big)\big).
\end{array}
$$

At this stage, although we have "threaded" memories through the operations of our interpreter, we have not yet introduced references, nor any operations that alter or depend upon memories. To proceed further, however, we must distinguish between two approaches to assignment, each of which characterizes certain programming languages.

In the "L-value" approach, in each context of the evaluation process where a value would occur, a reference (i.e., L-value) possessing that value occurs instead. Thus, for example, expressions evaluate to references, functional arguments and results are references, and environments bind variables to references. (In richer languages, references would occur instead of values in still other contexts, such as array elements.) This approach is used in the languages PAL [3] and ISWIM [2], and in somewhat modified form (i.e., references always occur in certain kinds of contexts, while values always occur in others) in such languages as FORTRAN, ALGOL 60, and PL/I. Its formalization is due to Strachey [30], and is used extensively in the Vienna definition of PL/I [18].

In the "reference" approach, references are introduced as a new kind of value, so that either references or "normal" values can occur in any meaningful context. This approach is used in ALGOL 68 [31], BASIL [32], and GEDANKEN [4].

The relative merits of these approaches are discussed briefly in Reference [4]. Although either approach can be accommodated by the various styles of interpreter discussed in this paper, we will limit ourselves to incorporating the reference approach into the above extension of Interpreter IV. We first augment the set of values appropriately:

$$\text{VAL} = \text{INTEGER} \cup \text{BOOLEAN} \cup \text{FUNVAL} \cup \text{REF}.$$

Next we introduce basic operations for creating, assigning, and evaluating references. For simplicity, we will make these operations basic functions, denoted by the predefined variables *ref*, *set*, and *val*. The following is an informal description:

$ref(a)$: Accepts a value a and returns a new reference initialized to possess a.

$(set(rf))(a)$: Accepts a reference rf and a value a. The value a is assigned to rf and also returned as the result. (Because of our restriction to functions of a single argument, this function is Curried, i.e., *set* accepts rf and returns a function that accepts a.)

$val(rf)$: Accepts a reference rf and returns its currently possessed value.

To introduce these new functions into our interpreter, we extend the initial environment as follows:

$$\begin{aligned}
&initenv = \lambda x.\ \big(\cdots\\
&\quad x = \text{"ref"} \rightarrow \lambda(a, m, c).\ c(augment(m, a), nextref(m)),\\
&\quad x = \text{"set"} \rightarrow \lambda(rf, m, c).\ c\big(m, \lambda(a, m', c').\ c'(update(m', rf, a), a)\big),\\
&\quad x = \text{"val"} \rightarrow \lambda(rf, m, c).\ c(m, lookup(m, rf))\big).
\end{aligned}$$

The main shortcoming of the reference approach is the incessant necessity of using the function *val*. This problem can be alleviated by introducing *coercion* conventions, as discussed in Reference [4], that cause references to be replaced by their possessed values in appropriate contexts. However, since these conventions can be treated as abbreviations, they do not affect the basic structure of the definitional interpreters.

11. Directions Of Future Research

Within this paper we have tried to present a systematic, self-contained, and reasonably complete description of the current state of the art of definitional interpreters. We conclude with a brief (and hopeful) list of possible future developments:

1. It would still be very desirable to be able to define higher-order languages logically rather than interpretively, particularly if such an approach can lead to practical correctness proofs for programs. A major step in this direction, based on the work of Scott [12, 13, 14, 15], has been taken by R. Milner [16]. However, Milner's work essentially treats a language using call by name rather than call by value.

2. It should be possible to treat languages with multiprocessing features, or other features that involve "controlled ambiguity". An initial step is the work of the IBM Vienna Laboratory [18], using a nondeterministic state-transition machine.

3. It should also be possible to define languages, such as ALGOL 68 [31], with a highly refined syntactic type structure. Ideally, such a treatment should be meta-circular, in the sense that the type structure used in the defined language should be adequate for the defining language.

4. The conciseness of definitional interpreters makes them powerful tools for language design, particularly when one wishes to add new capabilities to a language with a minimum of increased complexity. Of particular interest (at least to the author) are the problems of devising better type systems and of generalizing assignment (for example, by permitting memories to be embedded in values.)

References

1. John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4):184–195, April 1960.
2. Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March 1966.
3. Arthur Evans, Jr. PAL – A language designed for teaching programming linguistics. In *Proceedings of 23rd National ACM Conference*, pages 395–403. Brandin/Systems Press, Princeton, New Jersey, 1968.
4. John C. Reynolds. GEDANKEN – A simple typeless language based on the principle of completeness and the reference concept. *Communications of the ACM*, 13(5):308–319, May 1970.
5. Alonzo Church. *The Calculi of Lambda-Conversion*, volume 6 of *Annals of Mathematics Studies*. Princeton University Press, Princeton, New Jersey, 1941.

6. Haskell Brookes Curry and Robert Feys. *Combinatory Logic, Volume 1*. Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1958. Second printing 1968.
7. Peter J. Landin. A λ-calculus approach. In Leslie Fox, editor, *Advances in Programming and Non-Numerical Computation: Proceedings of A Summer School*, pages 97–141. Oxford University Computing Laboratory and Delegacy for Extra-Mural Studies, Pergamon Press, Oxford, England, 1966.
8. Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, New York City, April 5–7, 1966. American Mathematical Society, Providence, Rhode Island, 1967.
9. Zohar Manna. The correctness of programs. *Journal of Computer and System Sciences*, 3(2):119–127, May 1969.
10. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580 and 583, October 1969. Reprinted in [11].
11. David Gries, editor. *Programming Methodology*. Springer-Verlag, New York, 1978.
12. Dana S. Scott. Outline of a mathematical theory of computation. Technical Monograph PRG–2, Programming Research Group, Oxford University Computing Laboratory, Oxford, England, November 1970. A preliminary version appeared in Proceedings of the Fourth Annual Princeton Conference on Information Sciences and Systems (1970), 169–176.
13. Dana S. Scott. Lattice theory, data types and semantics. In Randell Rustin, editor, *Formal Semantics of Programming Languages: Courant Computer Science Symposium 2*, pages 65–106, New York University, New York, September 14–16, 1970. Prentice-Hall, Englewood Cliffs, New Jersey, 1972.
14. Dana S. Scott. Models for various type-free calculi. In Patrick Suppes, Leon Henkin, Athanase Joja, and Gr. C. Moisil, editors, *Logic, Methodology and Philosophy of Science IV: Proceedings of the Fourth International Congress*, volume 74 of *Studies in Logic and the Foundations of Mathematics*, pages 157–187, Bucharest, Romania, August 29–September 4, 1971. North-Holland, Amsterdam, 1973.
15. Dana S. Scott. Continuous lattices. In F. William Lawvere, editor, *Toposes, Algebraic Geometry and Logic*, volume 274 of *Lecture Notes in Mathematics*, Dalhousie University, Halifax, Nova Scotia, January 16–19, 1971. Springer-Verlag, Berlin, 1972.
16. Robin Milner. Implementation and applications of Scott's logic for computable functions. In *Proceedings of an ACM Conference on Proving Assertions about Programs*, pages 1–6, Las Cruces, New Mexico, January 6–7, 1972. ACM, New York. SIGPLAN Notices Volume 7, Number 1 and SIGACT News, Number 14.
17. Rodney M. Burstall. Formal description of program structure and semantics in first order logic. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence 5*, pages 79–98. Edinburgh University Press, Edinburgh, 1969.
18. Peter Lucas, Peter E. Lauer, and H. Stigleitner. Method and notation for the formal definition of programming languages. Technical Report TR 25.087, IBM Laboratory Vienna, June 28, 1968. Revised July 1, 1970.
19. John C. Reynolds. GEDANKEN – a simple typeless language which permits functional data structures and coroutines. Report ANL–7621, Applied Mathematics Division, Argonne National Laboratory, Argonne, Illinois, September 1969.
20. F. Lockwood Morris. The next 700 formal language descriptions. *Lisp and Symbolic Computation*, 6(3–4):249–257, November 1993. Original manuscript dated November 1970.
21. Jaco W. de Bakker. Semantics of programming languages. In Julius T. Tou, editor, *Advances in Information Systems Science*, volume 2, chapter 3, pages 173–227. Plenum Press, New York, 1969.
22. David M. R. Park. Fixpoint induction and proofs of program properties. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence 5*, pages 59–78. Edinburgh University Press, Edinburgh, 1969.
23. Jerome Feldman and David Gries. Translator writing systems. *Communications of the ACM*, 11(2):77–113, February 1968.
24. John McCarthy. Towards a mathematical science of computation. In Cicely M. Popplewell, editor, *Information Processing 62: Proceedings of IFIP Congress 1962*, pages 21–28, Munich, August 27–September 1, 1962. North-Holland, Amsterdam, 1963.

25. John M. Wozencraft and Arthur Evans, Jr. Notes on programming linguistics. Technical report, Department of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, Massachusetts, February 1971.
26. Adriaan van Wijngaarden. Recursive definition of syntax and semantics. In T. B. Steel, Jr., editor, *Formal Language Description Languages for Computer Programming: Proceedings of the IFIP Working Conference on Formal Language Description Languages*, pages 13–24, Baden bei Wien, Austria, September 15–18, 1964. North-Holland, Amsterdam, 1966.
27. James H. Morris, Jr. A bonus from van Wijngaarden's device. *Communications of the ACM*, 15(8):773, August 1972.
28. Michael J. Fischer. Lambda calculus schemata. In *Proceedings of an ACM Conference on Proving Assertions about Programs*, pages 104–109, Las Cruces, New Mexico, January 6–7, 1972. ACM, New York.
29. Peter J. Landin. A correspondence between ALGOL 60 and Church's lambda-notation. *Communications of the ACM*, 8(2–3):89–101, 158–165, February–March 1965.
30. D. W. Barron, John N. Buxton, D. F. Hartley, E. Nixon, and Christopher Strachey. The main features of CPL. *The Computer Journal*, 6:134–143, July 1963.
31. Adriaan van Wijngaarden, B. J. Mailloux, J. E. L. Peck, and C. H. A. Koster. Report on the algorithmic language ALGOL 68. *Numerische Mathematik*, 14(2):79–218, 1969.
32. T. E. Cheatham, Jr., Alice Fischer, and P. Jorrand. On the basis for ELF – an extensible language facility. In *1968 Fall Joint Computer Conference*, volume 33, Part Two of *AFIPS Conference Proceedings*, pages 937–948, San Francisco, December 9–11, 1968. Thompson Book Company, Washington, D.C.