The Next 700 Programming Languages

P. J. Landin

Univac Division of Sperry Rand Corp., New York, New York

"... today ... 1,700 special programming languages used to 'communicate' in over 700 application areas."—*Computer Software* Issues, an American Mathematical Association Prospectus, July 1965.

A family of unimplemented computing languages is described that is intended to span differences of application area by a unified framework. This framework dictates the rules about the uses of user-coined names, and the conventions about characterizing functional relationships. Within this framework the design of a specific language splits into two independent parts. One is the choice of written appearances of programs (or more generally, their physical representation). The other is the choice of the abstract entities (such as numbers, character-strings, lists of them, functional relations among them) that can be referred to in the language.

The system is biased towards "expressions" rather than "statements." It includes a nonprocedural (purely functional) subsystem that aims to expand the class of users' needs that can be met by a single print-instruction, without sacrificing the important properties that make conventional right-hand-side expressions easy to construct and understand.

1. Introduction

Most programming languages are partly a way of expressing things in terms of other things and partly a basic set of given things. The ISWIM (If you See What I Mean) system is a byproduct of an attempt to disentangle these two aspects in some current languages.

This attempt has led the author to think that many linguistic idiosyncracies are concerned with the former rather than the latter, whereas aptitude for a particular class of tasks is essentially determined by the latter rather than the former. The conclusion follows that many language characteristics are irrelevant to the alleged problem orientation.

ISWIM is an attempt at a general purpose system for describing things in terms of other things, that can be problem-oriented by appropriate choice of "primitives." So it is not a language so much as a family of languages, of which each member is the result of choosing a set of primitives. The possibilities concerning this set and what is needed to specify such a set are discussed below.

ISWIM is not alone in being a family, even after mere syntactic variations have been discounted (see Section 4). In practice, this is true of most languages that achieve more than one implementation, and if the dialects are well disciplined, they might with luck be characterized as differences in the set of things provided by the library or operating system. Perhaps had ALGOL 60 been launched as a family instead of proclaimed as a language, it would have fielded some of the less relevant criticisms of its deficiencies.

At first sight the facilities provided in ISWIM will appear comparatively meager. This appearance will be especially misleading to someone who has not appreciated how much of current manuals are devoted to the explanation of common (i.e., problem-orientation independent) logical structure rather than problem-oriented specialties. For example, in almost every language a user can coin names, obeying certain rules about the contexts in which the name is used and their relation to the textual segments that introduce, define, declare, or otherwise constrain its use. These rules vary considerably from one language to another, and frequently even within a single language there may be different conventions for different classes of names, with near-analogies that come irritatingly close to being exact. (Note that restrictions on what names can be coined also vary, but these are trivial differences. When they have any logical significance it is likely to be pernicious, by leading to puns such as ALGOL's integer labels.)

So rules about user-coined names is an area in which we might expect to see the history of computer applications give ground to their logic. Another such area is in specifying functional relations. In fact these two areas are closely related since any use of a user-coined name implicitly involves a functional relation; e.g., compare

$x(x+a)$	$f(b+2c)$
where $x = b + 2c$	**where** $f(x) = x(x+a)$

ISWIM is thus part programming language and part program for research. A possible first step in the research program is 1700 doctoral theses called "A Correspondence between x and Church's λ-notation."[1]

2. The where-Notation

In ordinary mathematical communication, these uses of '**where**' require no explanation. Nor do the following:

$f(b+2c) + f(2b-c)$
where $f(x) = x(x+a)$

$f(b+2c) + f(2b-c)$
where $f(x) = x(x+a)$
and $b = u/(u+1)$
and $c = v/(v+1)$

$g(f$ **where** $f(x) = ax^2 + bx + c,$
$u/(u+1),$
$v/(v+1))$
where $g(f, p, q) = f(p+2q, 2p-q)$

Presented at an ACM Programming Languages and Pragmatics Conference, San Dimas, California, August 1965.

[1] There is no more use or mention of λ in this paper—cognoscenti will nevertheless sense an undercurrent. A not inappropriate title would have been "Church without lambda."

A phrase of the form '**where** definition' will be called a "**where**-clause." An expression of the form 'expression **where**-clause' is a "**where**-expression." Its two principal components are called, respectively, its "main clause" and its "supporting definition." To put '**where**' into a programming language the following questions need answers.

Linguistic Structure. What structures of expression can appropriately be qualified by a **where**-clause, e.g., conditional expressions, operand-listings, statements, declarations, **where**-expressions?

Likewise, what structures of expression can appropriately be written as the right-hand side (*rhs*) of a supporting definition? What *contexts* are appropriate for a **where**-expression, e.g., as an arm of a conditional expression, an operator, the main-clause of a **where**-expression, the left-hand side (*lhs*) of a supporting definition, the rhs of a supporting definition?

Syntax. Having answered the above questions, what are the rules for writing the acceptable configurations unambiguously? E.g., where are brackets optional or obligatory? or other punctuation? or line breaks? or indentation? Note the separation of decisions about structure from decisions about syntax. (This is not a denial that language designers might iterate, like hardware designers who distinguish levels of hardware design.)

Semantic Constraints on Linguistic Structure. In the above examples each main clause was a *numerical* expression; i.e., given appropriate meanings for the various identifiers in it, it denoted a number. What other kinds of meaning are appropriate for a mainclause, e.g., arrays, functions, structure descriptions, print-formats?

Likewise what kinds of meaning are appropriate for *rhs*'s of supporting definitions? Notice there is not a third question analogous to the third question above under *linguistic structure*. This is because a **where**-expression must mean the same kind of thing as its main clause and hence raises no new question concerning what contexts are meaningful. Notice also that the questions about meaning are almost entirely independent of those about structure. They depend on classifying expressions in two ways that run across each other.

Outcome. What is the outcome of the more recondite structural configurations among those deemed admissible, e.g. mixed nests of **where**-expressions, function definitions, conditional expressions, etc.?

Experimental programming has led the author to think that there is no configuration, however unpromising it might seem when judged cold, that will not turn up quite naturally. Furthermore, some configurations of '**where**' that might first appear to reflect somewhat pedantic distinctions, in fact provide close matches for current language features such as **name/value** and **own** (see [2, 3]).

All these questions are not answered in this paper. The techniques for answering them are outlined in Section 4.

One other issue arises when '**where**' is added to a programming language—types and specifications. A method of expressing these functionally is explained in [2]. It amounts to using named transfer-functions instead of class names like **integer**, i.e., writing

$$\textbf{where } n = round(n)$$

instead of the specification

$$\textbf{integer } n$$

Thus the use of functional notation does not jeopardize the determination of type from textual evidence.

3. Physical ISWIM and Logical ISWIM

Like ALGOL 60, ISWIM has no prescribed physical appearance. ALGOL 60's designers sought to avoid commitment to any particular sets of characters or type faces. Accordingly they distinguish between "publication language," "reference language" and "hardware languages." Of these the reference language was the standard and was used in the report itself whenever pieces of ALGOL 60 occurred. Publication and hardware languages are transliterations of the reference language, varying according to the individual taste, needs and physical constraints on available type faces and characters.

Such variations are different physical representations of a single abstraction, whose most faithful physical representation is the reference language. In describing ISWIM we distinguish an abstract language called "logical ISWIM," whose texts are made up of "textual elements," characterized without commitment to a particular physical representation. There is a physical representation suitable for the medium of this report, and used for presenting each piece of ISWIM that occurs in this report. So this physical representation corresponds to "reference ALGOL 60," and is called "reference ISWIM," or the "ISWIM reference representation," or the "ISWIM reference language."

To avoid imprecision one should never speak just of "ISWIM," but always of "logical ISWIM" or of "such-and-such physical ISWIM." However, in loose speech, where the precise intention is clear or unimportant, we refer to "ISWIM" without qualification. We aim at a more formal relation between physical and logical languages than was the case in the ALGOL 60. This is necessary since we wish to systematize and mechanize the use of different physical representations.

4. Four Levels of Abstraction

The "physical/logical" terminology is often used to distinguish features that are a fortuitous consequence of physical conditions from features that are in some sense more essential. This idea is carried further by making a similar distinction among the "more essential" features. In fact ISWIM is presented here as a four-level concept comprising the following:

(1) physical ISWIM's, of which one is the reference language and others are various publication and hardware languages (not described here).

(2) logical ISWIM, which is uncommitted as to character sets and type faces, but committed as to the sequence of textual elements, and the grammatical rules for grouping them, e.g., by parentheses, indentation and precedence relations.

(3) abstract ISWIM, which is uncommitted as to the grammatical rules of sequence and grouping, but committed as to the grammatical categories and their nesting structure. Thus abstract ISWIM is a "tree language" of which logical ISWIM is one linearization.

(4) applicative expressions (AEs), which constitute another tree language, structurally more austere than abstract ISWIM, and providing certain basic grammatical categories in terms of which all of ISWIM's more numerous categories can be expressed.

The set of acceptable texts of a physical ISWIM is specified by the relations between 1 and 2, and between 2 and 3. The outcome of each text is specified by these relations, together with a "frame of reference," i.e., a rule that associates a meaning with each of a chosen set of identifiers.

These are the things that vary from one member of our language family to the next. The specification of the family is completed by the relation between abstract ISWIM and AEs, together with an abstract machine that interpret AEs. These elements are the same for all members of the family and are not discussed in this paper (see [1, 2, 4]).

The relationship between physical ISWIM and logical ISWIM is fixed by saying what physical texts represent each logical element, and also what layout is permitted in stringing them together. The relationship between logical ISWIM and abstract ISWIM is fixed by a formal grammar not unlike the one in the ALGOL 60 report, together with a statement connecting the phrase categories with the abstract grammatical categories.

These two relations cover what is usually called the "syntax" or "grammar" of a language. In this paper syntax is not discussed beyond a few general remarks and a few examples whose meaning should be obvious.

The relationship between abstract ISWIM and AEs is fixed by giving the form of AE equivalent to each abstract ISWIM grammatical category. It happens that these latter include a subset that exactly matches AEs. Hence this link in our chain of relations is roughly a mapping of ISWIM into an essential "kernel" of ISWIM, of which all the rest is mere decoration.

5. Abstract ISWIM

The texts of abstract ISWIM are composite information structures called *amessage's*. The following structure definition defines[2] the class *amessage* in terms of a class called *identifier*. It also defines several functions for manipulating *amessage's*. These comprise the predicates *demand*, *simple*, *infixed*, etc; also the selectors *body*, *rator*, *leftarm*, *nee*, etc; also (taking for granted certain unformalized conventions concerning structure definitions) the constructors, *consdemand*, *conscombination* (elsewhere abbreviated to *combine*), *consstandardadef*, etc. Examples of reference ISWIM are given alongside, against the right margin.

An amessage is
  either a *demand*, and has [**Print** $a+2b$
    a *body* which is an aexpression,
  or **else** a definition, [**Def** $x=a+2b$
where rec
an aexpression (aexp) is
  either *simple*, and has [*CAth*231"
    a *body* which is an identifier
  or a *combination*, in which case it has [$sin(a+2b)$
    a *rator*, which is an aexp, or
    and a *rand*, which is an aexp, $a + 2b$
  or *conditional*, in which case it is
    either *two-armed*, and has [$p\rightarrow a+2b;\ 2a-b$
      a *condition*, which is an aexp,
      and a *leftarm*, which is an aexp,
      and a *rightarm*, which is an aexp,
    or *one-armed*, and has [$q\rightarrow 2a-b$
      a *condition*, which is an aexp,
      and an *arm*, which is an aexp,
  or a *listing*, and has [$a+b,\ c+d,\ e+f$
    a *body* which is an aexp-list,
  or *beet*, and has [$x(x+1)$ **where** x = a + $2b$
    a *mainclause*, which is an aexp, or
    and a *support* **let** $x = a + 2b;\ x(x+1)$
      which is an adef,
and
an adefinition (adef) is
  either *standard*, and has [$x=a+2b$
    a *definee* (*nee*), which is an abv,
    and a *definiens* (*niens*), which is an aexp,
  or *functionform*, and has [$f(x)=x(x+1)$
    a *lefthandside* (*lhs*),
      which is an abv-list of length ≥ 2,
    and a *righthandside* (*rhs*), which is an aexp
  or *programpoint*, and has [**pp** $f(x)=x(x+1)$
    a *body* which is an adef,
  or *circular*, and has [**rec** $f(n)=(n=0)\rightarrow 1;\ nf(n-1)$
    a *body* which is an adef,
  or *simultaneous*, and has [$x=a+2b$ **and** $y=2a-b$
    a *body*, which is an adef-list,
  or *beet*, and has [$f(y)=x(x+y)$
    a *mainclause*, **where** $x=a+2b$
    which is an adef,
    and a *support*, which is an adef.
where an abv is
  either *simple*, and has
    a *body*, which is an identifier,
  or **else**, is an abv-list. [$x,\ (y,\ z),\ w$

A program-point definition introduces a deviant kind of function. Applying such a function precipitates premature termination of the **where**-expression containing it, and causes its result to be delivered as the value of the entire **where**-expression.

Program-points are ISWIM's, nearest thing to jumping. Assignment is covered as a particular case of an operator. For both of these the precise specification is in terms of the underlying abstract machine (see [2]).

[2] Writing a structure definition involves coining names for the various alternative formats of *amessage*'s and their components. The only obscure coinage is "beet," which abbreviates "beta-redex," i.e., "an expression amenable to rule (β)"; see Section 7.

6. Relationship to LISP

ISWIM can be looked on as an attempt to deliver LISP from its eponymous commitment to lists, its reputation for hand-to-mouth storage allocation, the hardware dependent flavor of its pedagogy, its heavy bracketing, and its compromises with tradition. These five points are now dealt with in turn:

(1) ISWIM has no particular problem orientation. Experiments so far have been mainly in numerical work and language processing with brief excursions into "commercial" programming and elsewhere. No bias towards or away from a particular field of application has emerged.

(2) Outside a certain subset (corresponding closely to ALGOL 60 without dynamic own arrays), ISWIM needs garbage collection. An experimental prototype implementation followed common ALGOL 60 practice. It used dynamic storage allocation with two sources, one LIFO and the other garbage collected, with the intention that the LIFO source should take as big a share as possible.

However, as with ALGOL 60, there is a latent potential for preallocating storage in certain favorable and commonly useful situations. Compared with LISP the chief amelioration of storage allocation comes out of a mere syntactic difference, namely, the use of **where** (or **'let'** which is exactly equal in power and program structure). This provides a block-structure not dissimilar in textual appearance from ALGOL 60's, though it slips off the pen more easily, and is in many respects more general.

(3) LISP has some dark corners, especially outside "pure LISP," in which both teachers and programmers resort to talking about addresses and to drawing storage diagrams. The abstract machine underlying ISWIM is aimed at illuminating these corners with a minimum of hardware dependence.

(4) The textual appearance of ISWIM is not like LISP's *S*-expressions. It is nearer to LISP's *M*-expressions (which constitute an informal language used as an intermediate result in hand-preparing LISP programs). ISWIM has the following additional features:

(a) "Auxiliary" definitions, indicated by **'let'** or **'where'**, with two decorations: **'and'** for simultaneous definitions, and **'rec'** for self-referential definitions (not to be mistaken for a warning about recursive *activation*, which can of course also arise from self-application, and without self-reference).

(b) Infixed operators, incorporated systematically. A logical ISWIM can be defined in terms of four unspecified parameters: three subsets of the class of *identifiers*, for use as prefixed, infixed and postfixed operators; and a precedence relation defined over the union of these subsets.

(c) Indentation, used to indicate program structure. A physical ISWIM can be defined in terms of an unspecified parameter: a subset of phrase categories, instances of which are restricted in layout by the following rule called "the offside rule." The southeast quadrant that just contains the phrase's first symbol must contain the entire phrase, except possibly for bracketed subsegments. This rule has three important features. It is based on vertical alignment, not character width, and hence is equally appropriate in handwritten, typeset or typed texts. Its use is not obligatory, and use of it can be mixed freely with more conventional alternatives like punctuation. Also, it is incorporated in ISWIM in a systematic way that admits of alternatives without changing other features of ISWIM and that can be applied to other languages.

(5) The most important contribution of LISP was not in listprocessing or storage allocation or in notation, but in the logical properties lying behind the notation. Here ISWIM makes little improvement because, except for a few minor details, LISP left none to make. There are two equivalent ways of stating these properties.

(a) LISP simplified the equivalence relations that determine the extent to which pieces of program can be interchanged without affecting the outcome.

(b) LISP brought the class of entities that are denoted by expressions a programmer can write nearer to those that arise in models of physical systems and in mathematical and logical systems.

These remarks are expanded in Sections 7 and 8.

7. The Characteristic Equivalences of ISWIM

For most programming languages there are certain statements of the kind, "There is a systematic equivalence between pieces of program like this, and pieces like that," that *nearly* hold but not quite. For instance in ALGOL 60 there is a *nearly* true such statement concerning procedure calls and blocks.

At first sight it might appear pedantic to quibble about such untidiness—"What's the point of having two different ways of doing the same thing anyway? ISN't it better to have two facilities than just one?" The author believes that expressive power should be by design rather than accident, and that there is great point in equivalences that hold without exception. It is a platitude that any given outcome can be achieved by a wide variety of programs. The practicability of all kinds of program-processing (optimizing, checking satisfaction of given conditions, constructing a program satisfying given conditions) depends on there being elegant equivalence rules. For ISWIM there are four groups[3], concerning:

(1) the extent to which a subexpression can be replaced by an equivalent subexpression without disturbing the equivalence class of the whole expression. Without this group the other rules would be applicable only to complete expressions, not to subexpressions.

(2) user-coined names, i.e., in definitions and, in particular, function definitions.

(3) built-in entities implicit in special forms of expression. The only instances of this in ISWIM are conditional expressions, listings and self-referential definitions.

(4) named entities added in any specific problem-orientation of ISWIM.

[3] To facilitate subsequent discussion each rule is preceded by a name, e.g., "(μ)", "(ν)", etc. These are chosen to conform with precedents in Curry's *Combinatory Logic*.

GROUP 1

(μ)	If	$L \equiv L'$	then	$L\ (M) \equiv L'\ (M)$
(ν)	If	$M \equiv M'$	then	$L\ (M) \equiv L\ (M')$
(ν')	If	$M \equiv M'$	then	$(L, \cdots, M, \cdots, N) \equiv (L, \cdots, M', \cdots N)$
(ν'')	If	$L \equiv L'$	then	$(L \rightarrow M;\ N) \equiv (L' \rightarrow M, N)$
(ν''')	If	$M \equiv M'$	then	$(L \rightarrow M;\ N) \equiv (L \rightarrow M';\ N)$
(ν^{iv})	If	$N \equiv N'$	then	$(L \rightarrow M;\ N) \equiv (L \rightarrow M;\ N')$
(ν^{v})	If	$M \equiv M'$	then	$(L$ **where** $x = M) \equiv (L$ **where** $x = M')$

The significant omissions here are the main-clause in the last case above, the *rhs* of a function definition "$f(x) = M$" and of a circular definition "**rec** $x = M$".

GROUP 2

(**let**)	**let** $x = M;\ L$	$\equiv\ L$ **where** $x = M$
(I')	$f(x) = L$	$\equiv f = (g$ **where** $g(x) = L)$
	$f(a, b, c)(x, y) = L$	$\equiv f(a, b, c) = (g$ **where** $g(x, y) = L)$
	and so on for each shape of left-hand side	
(I)	$(f$ **where** $f(x) = L)\ M$	$\equiv\ L$ **where** $x = M$
(β')	$(x = L)$ **where** $y = M$	$\equiv x = (L$ **where** $y = M)$
(D')	$x = L$ **and** $y = M$ **and**	$\equiv \cdots$ **and** $z = N$
		$\equiv (x, y, \cdots, z) = (L, M, \cdots, N)$

Rules (I'), (β'), (D'), together with (Y) below, enable any definition to be "standardized," i.e., expressed in a *lhs/rhs* form, in which the *lhs* is precisely the definee. Thus a nonstandard definition can be transformed so as to be amenable to rules (ν^v) and (β) (see Group 2′).

GROUP 2′

(β) L **where** $x = M$ $\equiv$ Subst $\begin{matrix} M \\ x \end{matrix}$ L

where "Subst $\begin{matrix} A \\ B \end{matrix}$ C" means roughly the expression resulting from substituting A for B throughout C. Here 'x' may be any list-structure of *distinct* identifiers, provided that 'M' has structure that fits it.

This rule is the most important, but it has only limited validity, namely, within the "purely functional" subset of ISWIM that results from not using the program-point feature or assignment.

Its importance lies in a variant of a famous theorem of mathematical logic, the Church-Rosser theorem. This concerns the possibility of eliminating '**where**' from an expression by repeatedly applying the rules stated above, including crucially (β). The theorem ensures that if there are several ways of doing this they all reach the same result.

The author thinks that the fruitful development to encompass all ISWIM will depend on establishing "safe" areas of an ISWIM expression, in which imperative features can be disregarded. The usefulness of this development will depend on how successfully ISWIM's nonimperative features supersede conventional programming.

GROUP 3

($\rightarrow$)	**true** $\rightarrow M;\ N$	$\equiv M$
($\rightarrow'$)	**false** $\rightarrow M;\ N$	$\equiv N$
($\rightarrow''$)	$P \rightarrow M$	$\equiv P \rightarrow M;$ *undefined*
(*undefined*)	*undefined*	$\equiv$ *selfapply* (*selfapply*) **where** *selfappiy* $(f) = f(f)$
(Y)	**rec** $x = L$	$\equiv x = (L$ **where rec** $x = L)$
(D'')	$(x, \cdots, z) = M$	$\equiv (x, \cdots, z) =$ *null* $(t^k w) \rightarrow$ $hw, \cdots, h(t^{k-1}w)$ **where** $w = M$ (for $k \geq 2$)
	$(x, (u, v), z) = M$	$\equiv (x, (u, v), z) =$ *null* $(t^3 w) \rightarrow$ $h(w)$, (*null* $(t^2 w') \rightarrow$ $h(w'), h(t(w'))$ **where** $w' = h(t(w))$) $h(t^2 w)$ **where** $w = M$
	and so on for each shape of definee	
(*null*)	*null* (*nullist*)	$\equiv$ **true**
(*null'*)	*null* $(L_1, \cdots, L_k)$	$\equiv$ **false** **where** $(x, \cdots, z) = L_1, \cdots, L_k$ $(k \geq 2)$
(h)	$h(L_1, \cdots, L_k)$	$\equiv x$ **where** $(x, \cdots, z) = L_1, \cdots, L_k$ $(k \geq 2)$
(t)	$t(L_1, \cdots, L_k)$	$\equiv y, \cdots, z$ **where** $(x, y, \cdots, z) = L_1, \cdots, L_k$ $(k \geq 3)$
(t')	$t(t(L_1, L_2))$	$\equiv$ *nullist* **where** $(x, y) = L_1, L_2$

The rules about listings may appear willfully indirect. The more natural transformations are those effected by applying, for example, (D'') then (β). But these would have suffered the same limited validity as (β). In their above slightly cautious formulation the validity of (D''), etc. is unrestricted, and the more powerful equivalences that hold for nonimperative expressions arise entirely from (β).

GROUP 4

A problem-orientation of ISWIM can be characterized by additional axioms. In the simplest case such an axiom is an ISWIM definition. The resulting modification is called a "definitional extension" of the original system.

In more elaborate cases axioms may mutually constrain a group of identifiers; e.g. the following rule for equality among integers:

(=) Suppose L and M are ISWIM written integers (i.e., strings of digits); then either one or the other of the following holds:

$L = M \quad \equiv$ **true**
$L = M \quad \equiv$ **false**

according as L and M differ at most in lefthand zeros, or not.

Another example, presented even less formally, is the structure definition for abstract ISWIM.

Group 1 above makes no provision for substitutions within expressions that are qualified by a supporting definition or are used to define a function. However, such a substitution is legitimized as long as it does not involve the definees or variables, by encasing it within applications of rule (β) and its inverse (with any other rules that might

be needed to produce something that is amenable to (β), i.e., a beet with a *standard* definition).

Equivalence rules can be used to prove things about the system. For example, the reader will readily verify that the equivalence of

f (6) **where rec** $f(n) = (n=0) \rightarrow 1;\ nf\ (n-1)$

and

6 $(f(5)$ **where rec** $f\ (n) = (n=0) \rightarrow 1;\ nf\ (n-1))$

can be established with the following steps:

(I'), (Y), (β), (Y), (β), (I), ($=$), (β) backwards, (Y) backwards, (I') backwards.

in this sequence we omit the auxiliary applications of (β), etc. that are needed at almost every step to legitimize the substitution.

8. Application and Denotation

The commonplace expressions of arithmetic and algebra have a certain simplicity that most communications to computers lack. In particular, (a) each expression has a nesting subexpression structure, (b) each subexpression denotes something (usually a number, truth value or numerical function), (c) the thing an expression denotes, i.e., its "value", depends only on the *values* of its subexpressions, not on other properties of them.

It is these properties, and crucially (c), that explains why such expressions are easier to construct and understand. Thus it is (c) that lies behind the evolutionary trend towards "bigger righthand sides" in place of strings of small, explicitly sequenced assignments and jumps. When faced with a new notation that borrows the functional appearance of everyday algebra, it is (c) that gives us a test for whether the notation is genuinely functional or merely masquerading.

The important feature of ISWIM's equivalence rules is that they guarantee the same desirable properties to ISWIM's nonimperative subset. We may equate "abstract object" with "equivalence class," and equate "denotes" with "is a member of." Then the properties (μ) and (ν) ensures anologies of (c) above. They state that the value of an operator/operand combination depends only on the *values* of its component subexpressions, not on any other aspects of them.

Thus conditions (μ) and (ν) are equivalent to the existence of a dyadic operation among the abstract objects; we call this operation "application."

The terminology of "abstract objects," "denoting" and "application" is frequently more convenient than that of equivalence relations. For example, it suggests another way of characterizing each problem-orientation of ISWIM. We can think of a set of abstract objects with a partially defined dyadic "application" operation and a monadic "designation" operation that associates a "primitive" abstract object with each of some chosen set of names, called the "constants" of the special system.

Consider for example a programming language that contains expressions such as

'wine'

Anyone with a generous ontology will admit that this 6-character expression denotes the 4-character-string

wine

For such a person its use in the language is characterized by

- the objects that it is applicable to, and the object it produces in each case (e.g., strings might be used like vectors, whose application to an integer produces an item of the string).
- The objects that it is amenable to, and the object it yields in each case (e.g., prefixing, appending, selection, etc.).

The sceptic need not feel left out. He just has to talk, a bit more clumsily, about

'wine'

being in the equivalence class that also contains

concatenate (*'wi'*, *'ne'*)

and

append (*fifthletterof* (*romanalphabet*), (*threeletterstemof* (*'winter'*))

Then he goes on to speak of the equivalence class of expressions that can serve as operand or operator to any of the above, and the equivalence class of the resulting operator/operand combination.

9. Note on Terminology

ISWIM brings into sharp relief some of the distinctions that the author thinks are intended by such adjectives as procedural, nonprocedural, algorithmic, heuristic, imperative, declarative, functional, descriptive. Here is a suggested classification, and one new word.

First, none of these distinctions are concerned with the use of pidgin English rather than pidgin algebra. Any pidgin algebra can be dressed up as pidgin English to please the generals. Conversely, it is a special case of the thesis underlying ISWIM that any pidgin English that has so far been implemented can be stripped to pidgin algebra. There is nevertheless an important possibility of having languages that are heuristic on account of their "applicative structure" being heuristic.

An important distinction is the one between indicating what behavior, step-by-step, you want the machine to perform, and merely indicating what outcome you want. Put that way, the distinction will not stand up to close investigation. I suggest that the conditions (a-c) in Section 8 are a necessary part of "merely indicating what outcome you want." The word "denotative" seems more appropriate than nonprocedural, declarative or functional. The antithesis of denotative is "imperative." Effectively "denotative" means "can be mapped into ISWIM without using jumping or assignment," given appropriate primitives.

It follows that functional programming has little to do with functional notation. It is a trivial and pointless task to rearrange some piece of symbolism into prefixed operators and heavy bracketing. It is an intellectually demanding activity to characterize some physical or logical system as a set of entities and functional relations among them. However, it may be less demanding and more revealing than characterizing the system by a conventional program, and it may serve the same purpose. Having formulated the model, a specific desired feature of the system can be systematically expressed in functional notation. But other notations may be better human engineering. So the role of functional notation is a standard by which to describe others, and a standby when they fail.

The phrase "describe in terms of" has been used above with reference to algorithmic modes of expression, i.e., interchangeably with "express in terms of." In this sense "3 + 4" is a description of the number 7 in terms of the numbers 3 and 4. This conflicts with current use of the phrase "descriptive languages," which appears to follow the logicians. For example, a language is descriptive in which the machine is told

Print the x such that $x^2 - x - 6 = 0 \wedge x \geq 0$

Such a classification of languages (as opposed to merely expressions within languages) is useless, and even harmful by encouraging stupidly restrictive language design, if it excludes the following:

Print *square* (the x such that $x^2 - x - 6 = 0 \wedge x \geq 0$)
Print $u(u+1)$
  where u = the x such that $x^2 - x - 6 = 0 \wedge x \geq 0$.
Print $f(1, -1, 6)$
  where $f(a, b, c)$ = the x such that $ax^2 + bx + c = 0 \wedge x \geq 0$

On the other hand it might reasonably exclude

Print *solepositivezeroof* $(1, -1, -6)$

where *solepositivezeroof* happens to be a library function.

The author therefore suggests that there is a useful distinction that can be made here concerning *languages*. Consider the function i, which operates on a class (or property) having a sole member (or instance), and transforms it into its sole member. We are interested in whether or not a language permits reference to i, with more or less restricted domain.

For example the above programs become:

Print i(*p* **where** $p(x) = x^2 - x - 6 \wedge x \geq 0$)
Print *square* (i(*p* **where** $p(x) = x^2 - x - 6 \wedge x \geq 0$))
Print $u(u+1)$
  where $u = i$ (*p* **where** $p(x) = x^2 - x - 6 \wedge x \geq 0$)
Print $f(1, -1, -6)$
  where $f\ (a, b, c) = i$(*p* **where** $p(x) = ax^2 + bx + c \wedge x \geq 0$)

More precisely, the distinction hinges on whether, when "applicative structure" is imputed to the language, it can be done without resorting to i, or to primitives in terms of which i can be defined.

This discussion of i reveals the possibility that primitives might be sensationally nonalgorithmic. So the algorithmic/heuristic distinction cuts across the denotative/imperative (i.e., nonprocedural/procedural) distinction. On the other hand if limited forms of i can be algorithmized, they still deserve the term "descriptive." So this factor is also independent.

10. Eliminating Explicit Sequencing

There is a game sometimes played with ALGOL 60 programs—rewriting them so as to avoid using labels and **go to** statements. It is part of a more embracing game—reducing the extent to which the program conveys its information by explicit sequencing. Roughly speaking this amounts to using fewer and larger statements. The game's significance lies in that it frequently produces a more "transparent" program—easier to understand, debug, modify and incorporate into a larger program.

The author does not argue the case against explicit sequencing here. Instead he takes as point of departure the observation that the user of any programming language is frequently presented with a choice between using explicit sequencing or some alternative feature of the language. Furthermore languages vary greatly in the alternatives they offer. For example, our game is greatly facilitated by ALGOL 60's conditional statements and conditional expressions. So the question considered here is: What other such features are there? This question is considered because, not surprisingly, it turns out that an emphasis on describing things in terms of other things leads to the same kind of requirements as an emphasis against explicit sequencing.

Though ALGOL 60 is comparatively favorable to this activity, it shares with most other current languages certain deficiencies that severely limit how far the game can go. The author's experiments suggest that two of the most needed features are:

- Treat a listing of expressions as a special case of the class of expressions, especially in the arms of a conditional expression, and in defining a function.
- Treat argument lists as a special case of lists. So a triadic function can have its arguments supplied by a conditional whose arms are 3-listings, or by application of a function that produces a 3-list. A similar situation arises when a 3-listing occurs as a definee. (Even LISP trips up here, over lists of length one.)

To clarify their practical use, here are some of the steps by which many a conventional ALGOL 60 or PL/1 program can be transformed into an ISWIM program that exploits ISWIM's nonimperative features.

(1) Rewrite the program so as to use two-dimensional layout and arrows to illuminate the explicit sequencing, i.e., as a flowchart with algebraic steps. Rearrange this to achieve the least confusing network of arrows.

(2) Apply the following changes repeatedly wherever they are applicable:

(a) Replace a string of independent assignments by one multiple assignment.

(b) Replace an assignment having purely local significance by a **where**-clause.

(c) Replace procedures by type-procedures (possibly

with multiple type), and procedure statements by assignment statements.

(d) Replace conditional jumps by conditional statements having bigger arms.

(e) Replace a branch whose arms have assignees in common by an assignment with conditional right-hand side.

(f) Replace a join by two calls for a procedure.

It should be observed that translating into ISWIM does not *force* such rearrangements; it merely facilitates them. One interesting observation is that the most recalcitrant uses of explicit sequencing appear to be associated with success/failure situations and the action needed on failure.

Section 2 discussed adding '**where**' to a conventional programming language. Theory and experiment both support the opposite approach, that taken in LISP, of adding imperative features to a basically nonimperative language. One big advantage is that the resulting language will have a nonimperative subset.

The special claim of ISWIM is that it grafts procedural notions onto a purely functional base without disturbing many of the desirable properties. The underlying ideas have been presented in [2]. This paper can do no more than begin the task of explaining their practical significance.

11. Conclusion

The languages people use to communicate with computers differ in their intended aptitudes, towards either a particular application area, or a particular phase of computer use (high level programming, program assembly, job scheduling, etc). They also differ in physical appearance, and more important, in logical structure. The question arises, do the idiosyncracies reflect basic logical properties of the situations that are being catered for? Or are they accidents of history and personal background that may be obscuring fruitful developments? This question is clearly important if we are trying to predict or influence language evolution.

To answer it we must think in terms, not of languages, but of families of languages. That is to say we must systematize their design so that a new language is a point chosen from a well-mapped space, rather than a laboriously devised construction.

To this end the above paper has marshalled three techniques of language design: abstract syntax, axiomatization, and an underlying abstract machine.

It is assumed that future calls on language development cannot be forstalled without generalizing the alternatives to explicit sequencing. The innovations of "program-points" and the "off-side rule" are directed at two of the problems (respectively in the areas of semantics and syntax) that must consequently be faced.

Acknowledgments. The author is grateful for helpful discussions with W. H. Burge. Wider influences on the investigation of which this paper is one outcome are mentioned in [1]. Of these the main ones are the publications of Curry and of McCarthy.

REFERENCES

1. LANDIN, P. J. The mechanical evaluation of expressions. *Comput. J. 6*, 4 (Jan. 1964), 308–320.
2. ——. A correspondence between ALGOL 60 and Church's Lambda-notation. *Comm. ACM 8* (1965), 89–101; 158–165.
3. ——. A formal description of ALGOL 60. In *Formal Language Description Languages for Computer Programming*, T. B. Steel, Jr. (Ed.), North Holland, Amsterdam, 1965.
4. ——. An abstract machine for designers of computing languages. (Summary). IFIP65 Proc., Part II.

DISCUSSION

Naur: Regarding indentation, in many ways I am in sympathy with this, but I believe that if it came about that this notation were used for very wide communication and also publication, you would regret it because of the kind of rearrangement of manuscripts done in printing, for example. You very frequently run into the problem that you have a wide written line and then suddenly you go to the *Communications of the ACM* and radically, perhaps, you have to compress it. The printer will do this in any way he likes; he is used to having great freedom here and he will foul up your notation.

Landin: I have great experience with this. (Laughter) I think I am probably the only person who has run through three versions of the galley proofs for the *Communications of the ACM*. However, I think that next time I could do better, and I think it is worth looking into. At any rate, the principle that I have described here is a good deal better than some that one might think of; for example it does not depend on details of character width, character by character—it is just as good handwritten as it is printed. Secondly, limiting the breadth of the page, I agree with you, needs more consideration. By the time I got through with the particular example I am talking about, by getting it printed, I had devised what I thought was a fairly reasonable method of communicating the principles that have been used in indentation.

Floyd: Another objection that I think is quite serious to indentation is that while it works on the micro-scale—that is, one page is all right—when dealing with an extensive program, turning from one page to the next there is no obvious way of indicating how far indentation stretches because there is no printing at all to indicate how far you have indented. I would like you to keep that in mind.

Landin: Yes, I agree. In practice I deal with this by first making the page breaks in sensible places.

Floyd: That's all right as long as you don't have an indented region which is simply several pages long.

Landin: Well in that case the way I did it was to cut down the number of carryover levels to about four or five from one page to another. You can at least make it simpler when you are handwriting by putting some kind of symbols at the bottom of the page and top of the continuation.

Floyd: Even if you regard your indentation spaces as characters there still doesn't seem to be any way—in fact, I am fairly sure there is no way—of representing the indentation conventions within a phrase-structure grammar.

Landin: Yes, but some indentation conventions can be kept within phrase structure grammars by introducing two terminal symbols that are grammatically like parentheses, but are textually like typewriter keys for setting and clearing tabulation positions. More precisely, the textual representation of the second of these symbols can be explained as the following sequence of typewriter actions: 1) line-feed; 2) back-space as far as the right-most tab position that is still currently active; 3) clear tab position; and 4) do step 2 again.

While this fits some indentation conventions, the one I propose is too permissive to be included. For my language I have written a formal grammar that is not phrase structure and includes one departure that meets this problem.

Leavenworth: I should like to raise the question of eliminating explicit jumps, I mean of using recursion as against interation.

Landin: It seems to me that there are rather a small number of functions which you could use if you were writing a LISP program in the places where ordinary programs would use iterations, and that if you were to use these the processor might do as well as if you had written a loop. For example, **iterate** (m, f, x) might apply f, m times to x with the result $f^m(x)$. This is the simplest kind of loop I know and the function **iterate** provides a purely functional notation for this rather simple kind of loop. If a lot of familiar types of loop can be represented by a few functions which could be defined recursively, I think it is sensible to take these as primitive. Another such function is **while** (p, f, x) which goes on applying f to x until the predicate p becomes false.

Strachey: I must just interpolate here something which is a bit of advertising I suppose. Nearly all the linguistic features, such as **where** and **while** and **and** and **recursive,** that Peter Landin has been talking about are incorporated as an integral part of a programming language being developed at Cambridge and London called CPL. In fact the **where** clauses are a very important feature of this language.

Irons: I have put together a program which uses some of these features and which has a standard output which prints the program in an indented manner. If it runs off the right end of the page, it produces another page to go on the right, and so forth. While certainly there are some situations that occur when it would be a bit awkward to make the paper go around the room, I have found that in practice, by and large it is true that this is a very profitable way of operating.

Strachey: I should like to intervene now and try to initiate a slightly more general discussion on declarative or descriptive languages and to try to clear up some points about which there is considerable confusion. I have called the objects I am trying to discuss DLs because I don't quite know what they are. Here are some questions concerning DLs: (1) What are DLs? (2) What is their relationship to imperative languages? (3) Why do we need DLs? (4) How can we use them to program? (5) How can we implement them? (6) How can we do this efficiently? (7) Should we mix DLs with imperative languages?

It seems to me that what I mean by DLs is not exactly what other people mean. I mean, roughly, languages which do not contain assignment statements or jumps. This is, as a matter of fact, not a very clear distinction because you can always disguise the assignments and the jumps, for that matter, inside other statement forms which make them look different. The important characteristic of DLs is that it is possible to produce equivalence relations, particularly the rule for substitution which Peter Landin describes as (β) in his paper. That equivalance relation, which appears to be essential in almost every proof, does not hold if you allow assignment statements. The great advantage then of DLs is that they give you some hope of proving the equivalence of program transformations and to begin to have a calculus for combining and manipulating them, which at the moment we haven't got.

I suggest that an answer to the second question is that DLs form a subset of all languages. They are an interesting subset, but one which is inconvenient to use unless you are used to it. We need them because at the moment we don't know how to construct proofs with languages which include imperatives and jumps.

How should we use them to program? I think this is a matter of learning a new programming technique. I am not convinced that all problems are amenable to programming in DLs but I am not convinced that there are any which are not either; I preserve an open mind on this point. It is perfectly true that in the process of rewriting programs to avoid labels and jumps, you've gone half the way towards going into DLs. When you have also avoided assignment statements, you've gone the rest of the way. With many problems you can, in fact, go the whole way. LISP has no assignment statements and it is remarkable what you can do with pure LISP if you try. If you think of it in terms of the implementations that we know about, the result is generally intolerably inefficient—but then that is where we come to the later questions.

How do we implement them? There have not been many attempts to implement DLs efficiently, I think. Obviously, it can be done fairly straightforwardly by an interpretive method, but this is very slow. Methods which compile a runable program run into a lot of very interesting problems. It can be done, because DLs are a subset of ordinary programming languages; any programming language which has sufficient capabilities can cope with them. There are problems, however: we need entities whose value is a function—not the application of a function but a function—and these involve some problems.

How to implement efficiently is another very interesting and difficult problem. It means, I think, recognizing certain subsets and transforming them from, say, recursions into loops. This can certainly be done even if they have been written in terms of recursions and not, as Peter Landin suggested, in terms of already transformed functions like **iterate** or **while.**

I think the last question, "Should DLs be mixed with imperative languages?", clearly has the answer that they should, because at the moment we don't know how to do everything in pure DLs. If you mix declarative and imperative features like this, you may get an apparently large programming language, but the important thing is that it should be simple and easy to define a function. Any language which by mere chance of the way it is written makes it extremely difficult to write compositions of functions and very easy to write sequences of commands will, of course, in an obvious psychological way, hinder people from using descriptive rather than imperative features. In the long run, I think the effect will delay our understanding of basic similarities, which underlie different sorts of programs and different ways of solving problems.

Smith: As I understand the declarative languages, there has to be a mixture of imperative and descriptive statements or no computation will take place. If I give you a set of simultaneous equations, you may say "yes?", meaning well, what am I supposed to do about it, or you may say "yes", meaning yes I understand, but you don't do anything until I say "now find the values of the variables." In fact, in a well-developed language there is not just one question that I can ask but a number of questions. So, in effect, the declarative statements are like data which you set aside to be used later after I give you the imperatives, of which I had a choice, which get the action.

Strachey: This is a major point of confusion. There are two ideas here and I think we should try to sort them out. If you give a quadratic equation to a machine and then say "print the value of x", this is not the sort of language that I call a *DL.* I regard it as an implicit language—that is, one where you give the machine the data and then hope that it will be smart enough to solve the problem for you. It is very different from a language such as LISP, where you define a function explicitly and have only one imperative, which says "evaluate this expression and print the result."

Abrahams: I've done a fair amount of programming in LISP, and there is one situation which I feel is symptomatic of the times when you really do want an imperative language. It is a situation that arises if you are planning to do programming in pure LISP and you find that your functions accumulate a large number of arguments. This often happens when you have a number of variables and you are actually going through a process and at each stage of the process you want to change the state of the world a little bit—say, to change one of these variables. So you have the choice of either trying to communicate them all, or trying to do some sort of essentially imperative action that changes one of them. If you try to list all of the transitions from state to state and incorporate them into one function, you'll find that this is not really a very natural kind of function because the natures of the transitions are too different.

Landin: I said in my talk that LISP had not gone quite all the way and I think that this difficulty is connected with going all the way. If we write a function definition where the right-hand side is a listing of expressions such as

$$F(x) = E_1 , E_2 , E_3$$

then we can say that this function will produce a three-list as its result. If now we have another function $G(x, y, z) = E$, on some occasion we might have an expression such as $G(a^2, b^2, c^2)$ and we often feel that we should be able to write $G(F(t))$, and another example which should be allowed is

$$G(a > b \rightarrow E_1 , E_2 , E_3 \textbf{ else } E_4 , E_5 , E_6).$$

I am not quite sure but I think you can get around your problem by treating every function as if it were in fact monadic and had a single argument which was the list structure you are trying to process.

Abrahams: This is a difficulty in other programming languages too; you cannot define a function of an indefinite number of arguments.

Naur: I still don't understand this distinction about an implicit language. Does it mean that whenever you have such a language there is a built-in feature for solving equations?

Abrahams: I think the point is whether you are concerned with the problem or are concerned with the method of solution of the problem.

Ingerman: I suggest that in the situation where you have specified everything that you want to know, though the exact sequence in which you evoke the various operations to cause the solution is left unspecified, then you have something which is effectively a descriptive language; if you have exactly the same pieces of information, surrounded with promises that you will do this and then this, then you have an imperative language. The significant point is that it is not all or nothing but there is a scale and while it is probably pretty simple to go all the way with imperatives, the chances of being able to get all the way descriptive is about zero, but there is a scale and we should recognize this scale.

Smith: I think that there is a confusion between implicit or explicit on the one hand and imperative or declarative on the other. These are two separate distinctions and can occur in all combinations. For instance, an analog computer handles implicit declaratives.

Young: I think it is fairly obvious that you've got to have the ability for sequencing imperatives in any sort of practical language. There are many, many cases in which only a certain sequence of operations will produce the logically correct results. So that we cannot have a purely declarative language, we must have a general purpose one. A possible definition of a declarative language is one in which I can make the statements (a), (b), (c) and (d) and indicate whether I mean these to be taken as a sequence or as a set; that is, must they be performed in a particular order or do I merely mean that so long as they are all performed, they may be performed in any sequence at any time and whenever convenient for efficiency.

Strachey: You can, in fact, impose an ordering on a language which doesn't have the sequencing of commands by nesting the functional applications.

Landin: The point is that when you compound functional expressions you are imposing a partial ordering, and when you decompose this into commands you are unnecessarily giving a lot of information about sequencing.

Strachey: One inconvenient thing about a purely imperative language is that you have to specify far too much sequencing. For example, if you wish to do a matrix multiplication, you have to do n^3 multiplications. If you write an ordinary program to do this, you have to specify the exact sequence which they are all to be done. Actually, it doesn't matter in what order you do the multiplications so long as you add them together in the right groups. Thus the ordinary sort of imperative language imposes much too much sequencing, which makes it very difficult to rearrange if you want to make things more efficient.

Syntax-Directed Interpretation of Classes of Pictures

R. Narasimhan

Tata Institute of Fundamental Research, Bombay, India

A descriptive scheme for classes of pictures based on labeling techniques using parallel processing algorithms was proposed by the author some years ago. Since then much work has been done in applying this to bubble chamber pictures. The parallel processing simulator, originally written for an IBM 7094 system, has now been rewritten for a CDC 3600 system. This paper describes briefly the structure of syntactic descriptive models by considering their specific application to bubble chamber pictures. How the description generated in this phase can be embedded in a larger "conversation" program is explained by means of a certain specific example that has been worked out. A partial generative grammar for "handwritten" English letters is given, as are also a few computer-generated outputs using this grammar and the parallel processing simulator mentioned earlier.

Presented at an ACM Programming Languages and Pragmatics Conference, San Dimas, California, August, 1965.

1. Introduction

Recent active interest in the area of graphic data-based "conversation programs"[1] has pointed up the urgent need for sophisticated picture processing models in a convincing manner. Kirsch [2] has very ably argued that "from the point of view of computer information processing, the important fact about natural language text and pictures is that both have a syntactic structure which is capable of being described to a machine and of being used for purposes of interpreting the information within a data processing system." "The problem of how to describe the syntactic structure of text and pictures and how to use the syntactic description in interpreting the text and pictures" has been tackled in a certain specific way by Kirsch and his coworkers. (For other references, see [9].)

[1] See [9] for a good survey of work accomplished and in progress in this area, as well as in the general field of "English question-answer" programs.