PAL—a language designed for teaching programming linguistics*

by ARTHUR EVANS, JR.
Massachusetts Institute of Technology
Cambridge, Massachusetts

INTRODUCTION

This paper describes PAL—a new computer language. Given the fact that new languages seem to appear in computer literature at the rate of several per month, it seems incumbent on one who creates a new language to justify having done so. In the present case, there are two important considerations: control and specification. Let us consider each of these in turn.

By virtue of our having designed PAL, it is ours. There is no PAL Users Group or Committee of Vested Interests concerned with retaining upward compatibility with what was done last year (or last month). This doesn't mean we change the specifications of the language every few weeks (our students are, in a real sense, our Committee of Vested Interests), but it does mean we can make decisions on changes solely on technical grounds. More important, though, we can design the language to meet the criteria *we* think important. For example, the language almost demands interpretive execution. Since no one writes production programs in PAL we are able to put up with inefficiencies in the implementation that would otherwise be intolerable. Thus we have designed our own language so that we will have *control* over it.

Control alone is not adequate to justify a language design effort—there must be more, and there is. The designer of a computer language has an obligation to describe it—that is, if he expects anyone other than himself and a few of his friends to use it. We felt that a language for student use must be completely and accurately specified. This leads directly to a problem currently attracting much attention in Computer Science research: formalizing the specification of the syntax and semantics of computer languages. The *syntax* part of the problem has been under fairly good control since about 1960, when John Backus and Peter Naur created what is now referred to as Backus-Naur Form. The semantics part of the problem, however, is very far from being under control. Indeed, it is currently the subject of research by many groups. While we by no means feel that we have solved the problem in the general case, we do feel that we can present to our students an honest formalization of PAL's semantics. Of course a large part of our success in doing so is a result of our having chosen a fairly simple language to describe, and this reasoning is the second justification for our having produced a new language. An important design criterion in adding features to PAL has been that their formal specification be clean.

Before proceeding with a discussion of PAL, it seems worthwhile to discuss briefly the use for which it is intended: as a pedagogical vehicle in an undergraduate subject called "Programming Linguistics". This subject is designed primarily for sophomores who anticipate a major professional interest in computer science, and has two objectives. The first is to study linguistic constructs for the specification of algorithms, and students are expected to learn some of the interesting and important intellectual ideas which are relevant to programming languages. (Examples of such ideas are the application of a function to arguments, the "creation" of new variables and allocation of their storage, the updating of the value associated with a variable, etc.) PAL was designed to reveal clearly these various intellectual ideas, with a minimum of syntactic decoration and a maximum of semantic clarity. The second objective of the subject is that the students improve their proficiency in computer programming. PAL is an adequately clean and powerful programming language that can readily be used by the students to perform fairly complex programming exercises as homework. As a final word, it should be remarked that students taking this subject have already had exposure to computer languages, either through experience or from an introductory subject.

*This work was supported in part by Project MAC, an M.I.T. research project sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Nonr-4102(01).

The world view of computation underlying the design of PAL may be summarized as follows: We regard computation as having to do with the manipulation of abstract objects. Thus we are led to consider a *universe of discourse* made up of such abstract objects. It should be clear that the nature of this universe of discourse is invariant with respect to the representation of these objects: The properties of the integers remain invariant regardless of whether they are represented as Arabic numerals or as Roman numerals. Of course, in communicating with computers or in writing programs, we are deeply concerned with the external representation of an abstract object rather than the abstract object itself. With this fact in mind then, we suggest the following:

1. An *algorithm* is the specification of a transformation on abstract objects, expressed in terms of other transformations which are themselves sufficiently simple that they need no further specification.
2. A *programming language* is a set of conventions for communicating algorithms.
3. A *program* is an algorithm represented in a programming language.

This point of view on the nature of computation influences the design and nature of the PAL language. It surely influences the way we choose to describe PAL. For example, we distinguish carefully between *strings*, which are abstract objects, and *quotations* which are the written representation of strings.

One of the basic intellectual ideas illustrated in PAL is so important that we refer to it by name: the Principle of Referential Transparency. This means that any subexpression may be replaced by any other expression, providing they both denote the same value. (We choose to ignore for the moment the problem of side effects. Clearly, the principle does *not* hold if the evaluation of one of the expressions produces side effects different from those produced by the evaluation of the other.) For example, suppose that f is a function that requires three arguments. Then we could write

$$f(Arg1, Arg2, Arg3)$$

(where the *Argi* are any suitable expressions) to denote the value of applying f to the arguments. On the other hand, we could equivalently have written

$$T := Arg1, Arg2, Arg3;$$
$$f(T)$$

Here T is some variable, and ":=" indicates assignment, as in Algol. Since f is to be applied to a 3-tuple, its argument may be any expression whose value, at the time of execution, is a 3-tuple. (The principle of referential transparency is perhaps more familiar to those interested in natural languages than to those interested in computer languages. See, for example, Strachey.[1]

A very important aspect of PAL is that its universe of discourse includes functions. This means that a function may be stored as the value of a variable. Further, it means that a function application may have as value still another function. This particular feature influences strongly many things that go on in PAL, both linguistic features of the language itself and features of the implementation. We discuss first some linguistic aspects of this. Consider a function which requires four arguments, and in which in a particular computation three of them will always be the same. What we might like to do is to supply three of the arguments at once, and supply the fourth argument as needed. Another way to state this requirement is that we want a function of one variable which is the fourth argument. Still another way to state the requirement is that we wish to write a function of three arguments whose value will be a function requiring a single argument. This is easy and natural to do in PAL. For example, suppose that *Funct* is our function of four arguments and we want *Functx* to have been applied to *Arg1*, *Arg2* and *Arg3*. We might write

$$\text{let } Functx\ (x) = Funct\ (Arg3, Arg2, Arg1, x)$$

This will work even if the *Argi* are complex expressions, although in such a case it suffers the (perhaps relevant) disadvantage that the *Argi* will be evaluated each time *Functx* is applied. If this is an important consideration, we might instead write

```
let Functx =
    let F1  = Arg1
    and F2  = Arg2
    and F3  = Arg3
    in
    g
    where g (x) = Funct (F1, F2, F3, x)
```

We will explain this notation further later. Hopefully, the reader will find its meaning "fairly obvious".

It is important to consider also the effects on the implementation of the availability of function-producing functions. An important design criterion in PAL was that the user never has to worry about the management of storage: the PAL system does it all for him. Consider now the following program:

```
let a = 20
in
let f (x) = g where g (y) = x + y
in
let h = f (a)
in
Print (h (6));
a := 30;
Print (h (6))
```

Here we define a variable a and then a function f. This latter is a function-producing function. When it is applied to an integer n, it produces a function which adds n to its argument. Thus, for example, $f(1)$ is the successor function and $f(-1)$ is the predecessor function. In the present example, h is defined to be a function that adds a to its argument. Thus the first *Print* statement causes *26* to be printed, since a is *20*. However, updating a has the effect of changing h, and the second *Print* statement produces *36*. The situation is actually even more complex. Suppose that t is a variable defined in a block containing the text shown. Then executing

$$t := h$$

permits references to a after the block in which a is declared has been exitted from. This sort of construction works correctly in PAL. Permitting it clearly has implications on storage management at runtime: An item remains in existence as long as there is a reference to it, as in LISP. A LISP-type garbage collection scheme is used. (This issue is discussed at length by Weizenbaum, who calls it the FUNARG problem.[2] It is well known to LISP aficionados.)

As a final preliminary to discussing the details of PAL, it is worth mentioning a few of the more important points in which it differs from other languages. The PAL compiler knows nothing whatsoever about types: the type of a variable is known only at run-time. Stated differently, the type is associated with the value and not with the variable. Since there is complete type checking, it therefore follows that the running system must be interpretive. For example, consider the application of the value of a variable to some argument. The variable must denote a value which is something which can properly be applied. It may be a basic function built into PAL, a function defined by the programmer, or (as we will see later) a tuple. In any case, though, it is necessary that the argument be checked to insure that it is an appropriate one. For example, a function defined by the programmer to have three arguments cannot be applied other than to a three-tuple.

As suggested by the last sentence of the previous paragraph, it is the case that all functions in PAL are monadic. This means that each function is applied to only a single argument. Since it is obviously necessary to have functions of "several arguments", the convention is that such a function is to be applied to a tuple. An n-tuple is a constructed object with n components. Transformationally, an n-tuple behaves as function on integers, and may be applied to an integer between one and n to select the relevant component. Thus, a "function of three arguments" is one which is properly applied to a three-tuple. For convenience of discourse, though, we will frequently refer to "functions of n arguments" in what follows, tolerating the abuse of language.

A final important difference between PAL and most other computer languages has to do with *sharing*. Two variables are said to *share* if updating one of them (as, with an assignment statement) causes the value of the other to be changed also. (Equivalently, the two variables occupy the same memory location.) For example, suppose that P denotes some 5-tuple and that i denotes a positive integer less than six. Then the declaration

$$\text{let } w = P(i)$$

causes w to share with the i^{th} component of P. Thus the effect achieved in Fortran at compile time with EQUIVALENCE statements is available dynamically in PAL.

The PAL Language

There are two important subsets of PAL: the applicative subset and the imperative subset. The applicative subset has to do with the application of functions to arguments. Using this part of PAL only, it is possible to write arbitrarily complicated expressions whose evaluation gives a result in terms of an initial set of values. Since this part of PAL includes the ability to state definitions and the ability to write conditionals, and since the ability to write definitions includes the ability to define recursive functions, it should be clear that this is all that is needed. (What we have just stated is that Church's lambda-calculus is equivalent in computational power to any of the various other schemes, such as Turing machines.) Indeed, this part of PAL matches very closely the computational power of pure LISP, or LISP-1. Although it would in principle be possible to have nothing else in PAL, it turns out that programming convenience requires imperative abilities. Equivalently, the users of LISP-1 found the need for the features which show up in LISP-1.5.

The imperative features of PAL include such statements as assignment statements and go-to statements. Thus the evaluation of a program is more than just the value of a complicated expression: it is reflected also in the contents of a memory unit. The availability of assignment statements implies the existence of a memory.

We will now summarize briefly PAL's syntax and semantics. We give first PAL's *alphabet* and the set of *words* which make up PAL's vocabulary. We then discuss PAL's universe of discourse (the set of objects with which the user of PAL may manipulate), and then show how these objects are represented in PAL. Following a discussion of the basic operators provided, we discuss how expressions may be constructed. Finally, we look at the imperative part of PAL.

PAL's Alphabet

PAL uses essentially the ASCII alphabet. Actually, the current implementation requires only those characters

available on an IBM 2741 with printing element ("golf ball") 963, although the entire alphabet may be used in data. Probably the most notable aspect of ASCII—particularly for one not used to it—is the availability of both upper and lower case letters. In addition to the 52 letters and 10 digits, the following characters are available:

! " # $ % & ' ( ) * + , - . / : ;
< = > ? @ [\ ] ^ – ' { | } ~

All of these except

" # ? @ \ ^ / ~

are actually used in PAL programming, and these latter are available in data.

PAL's Universe of Discourse

An important part of the specification of a programming language is the specification of the objects which the programmer may manipulate. In PAL these objects include *numbers*, *character strings*, *truth values*, *tuples*, *labels* and *functions*. A *number* in PAL is either an integer or a "real". As suggested earlier, a character string is any collection of characters from PAL's alphabet. (Special provisions are made for quoting quote-marks and format effectors such as new-line or tab.) The truth values are available with names *true* and *false*.

A *tuple* is an ordered collection of objects, the objects being any of the permissable data objects in PAL. Thus, for example, an element of a tuple may itself be a tuple. There is no requirement that successive elements of a tuple be of the same type: for example, the first may be an integer, the second element another tuple and the third element a label. A tuple transforms as a function on integers: The application of a tuple as an operator to an integer as an operand results in the selection of the relevant element of the tuple. Thus if *A* is a tuple, then the result of applying *A* to *2* is the second element of *A*. Such a construction may appear on the left side of an assignment statement. Updating *A(2)* causes any variable that shares with it to be updated also.

A *label* in PAL is very much like a label in most programming languages, although there are differences we will see later. Syntactically, a label is indicated by following it with a colon. A label is actually a variable and has the same syntax as a variable.

A *function* is just that. Functions are defined in PAL with a syntax explained later.

Consider now what are called *functors*. The usual arithmetic plus, minus, times and divide are provided, as is an exponentiation operator. The first four of these operators will operate on two integers or on two reals, but will not do mixed-mode arithmetic. Operating on truth values are the logical operators *and*, *or* and *not*. Operating on numbers and producing truth values are the relational operators less than and greater than. Equality will operate on any two objects from the classes number, string and truth value. If both objects have the same type and have the same value, the equality relation between them is true, while it is false otherwise. The infixed operator *aug* takes an *n*-tuple as its left operand and any object as its right operand and produces as result an *n+1*-tuple. Other functors will be mentioned later in this discussion.

The Applicative Subset of PAL

The applicative subjset of PAL consists of those aspects of the language which permit new values to be expressed in terms of (perhaps complex) operations on existing ones. Starting with the set of constants provided by the designers of PAL and with a built in set of operations (such as "+", "–", "*", etc.) for expressing new values in terms of existing ones, the programmer can write expressions such as

$$2 + 3$$

or

$$(7 * (8 + 9))/5$$

We are concerned for the moment with such expressions. To restate our concern in terms of languages such as Fortran, we are concerned with the sort of construction that can appear on the right side of an assignment statement. We defer for now a discussion of assignment statements themselves.

There are several ways available to the programmer to indicate the application of a function to arguments. The most obvious is the case where the application is explicit. That is, the programmer writes both the function and its argument(s) in the usual form. For example, writing

$$Add\ (2, 3)$$

means the application of the function *Add* to the arguments *2* and *3*. Parentheses are *not* needed to indicate functional application—only to indicate grouping. (They are needed above for that purpose.) Thus PAL permits

$$f\,x$$

or

$$(f)\ \ (x)$$

or even

$$(f)\ x$$

where other languages might require

$$f\ \ (x)$$

In addition to *prefix* forms such as the above, *infix* forms are also permitted. The programmer may write such expressions as

$$2*x*x + 3*y - z/7$$

just as in most high level computer languages.

The user of PAL is provided with a set of built in functions to do certain things which he could not conveniently (or sometimes at all) program himself. Included are type-checking predicates to determine whether or not a particular object is of a given type; string manipulating functions to take the first character of a string or the remainder of a string or to concatenate two strings; type conversion functions for conversion between real and integer and vice versa; and a set of miscellaneous functions. This latter set includes *Order*, which gives the "length" of a tuple; functions for reading data and printing data; and a few other useful predicates.

A basic form of expression in PAL is the *conditional expression*. The syntax of these is very much akin to that of LISP. For example, the expression that in Algol could be written as

$$\text{if } x < 0 \text{ } \textit{then } (-x) \textit{ else } + x$$

would be written in PAL as

$$x < 0 \rightarrow -x \mid + x$$

The Algol *if* is not used (it is not needed), *then* is replaced by the right arrow as in LISP, and *else* is replaced by the exclamation point.(It's in ASCII, and it's not used for anything else.) In PAL, either arm of the conditional may itself be a conditional.

Lambda Notation

Still another kind of expression available to the PAL user is the lambda expression. This is a direct implementation of Church's lambda calculus[3] with free variables. Unfortunately, there seems to be no good elementary reference available on the subject, Church's original paper defining a notation which differs from ours. Other possible references are [4] and [5]. Thus we now provide a very brief introduction which, hopefully, will be adequate for reading the rest of this paper.

In conventional mathematics, writing

$$f(x) = x + 1$$

indicates the definition of a function f of one argument. It seems clear that writing

$$f(y) = y + 1$$

defines the same function. What we are concerned with is the essential nature of the value of f. That is, f seems to be

that function of x that "$x+1$" is

and we require a notation for that idea. The lambda notation provides an answer to this need. The object which can be written as

$$\lambda x.x + 1$$

is precisely the desired object, and the definition

$$f = \lambda x.x + 1$$

has precisely the same meaning as the two previous definitions. Similarly, the lambda expression

$$\lambda(x, y) \,.\, x + y$$

is

that function of x and y that "x+y" is

so that

$$\text{Add} = \lambda(x, y) \,.\, x + y$$

and

$$\text{Add } (x, y) = x + y$$

have precisely the same meaning.

Now consider the definition

$$f = \lambda x.\ (\lambda y.\ x + y)$$

Although it is somewhat hard to express in English, the careful reader should be able to see that "f is that function of x that 'that function of y that $x+y$ is' is".

Since the character "λ" is not part of the PAL alphabet, the punctuation "11" is used instead. However, we will continue to use λ in this paper, to improve readability.

Definitions

In what has gone so far, we have assumed that variables existed and have felt free to use as examples such expressions as

$$x + 1$$

without explaining how the variable x came to denote an integer. PAL provides two syntactic forms to permit the user to "create" new variables: the *let* expression and the *where* expression. Each of these uses the idea of a *definition*.

Consider the PAL expression

```
let x = 1 and y = 2
in
let z = 3*x + 4*y
in
x * y * z
```

In evaluating this expression, x and y are first "created" as new variables, with values *1* and *2* respectively. Next z is created with value *11*. The value of the entire expression is then determined to be *22*. An equivalent expression is

```
(x * y * z
where z = 3*x + 4*y)
    where x = 1 and y = 2
```

(The parentheses are required to give the correct parse.) Still another equivalent expression is

$$[\ \lambda\ (x,y).\ (\lambda z.\ x*y*z)\ (3*x\ +\ 4*y)\ ]\ (1,\ 2)$$

The Imperative Subset of PAL

In PAL as in many languages, the programmer may write *sequences* of commands, separated by semi-colons. An element of a sequence may be a *goto statement*, an *assignment statement* or a function call. In the latter case, the function is (presumably) one whose useful effect is its side effect rather than the value it delivers.

An assignment statement has the form

$$V := E$$

or, alternatively,

$$V1,\ V2,\ \ldots,\ Vn := E$$

The latter is a *simultaneous assignment* and is defined only if E denotes an n-tuple. It is quite important that the assignment be "simultaneous", since a statement such as

$$x,\ y := y,\ x$$

may be used to interchange the values of x and y. The idea is that the values on the right and the locations on the left are all evaluated before any assignment is made.

Statements such as

$$x > y \rightarrow x\ !\ y := 0$$

are permitted, the effect here being that the larger of x and y is to be set to zero.

A *goto* statement in PAL is much like one in any other language. Of course, the "operand" of the *goto* may be any expression which denotes a label. Since labels may be assigned (with assignment statements), it is quite possilbe by assigning a label to a global quantity to go to it from a point which, in most languages, would be outside of the scope of the label. This is permitted in PAL, and it works. Of course, this also has effects on run-time storage management.

Formalization of PAL

We now touch briefly on the technique used to formalize PAL. Of necessity, the discussion is somewhat superficial, since the details are quite complex. Our objective to formalize completely PAL's semantics requires that we supply a set of rules which, given any PAL program, will enable one to tell "what it does", or "what it computes". The rules provide a four step evaluation process: parsing, unsugaring, translation, and evaluation. We consider each of these in turn.

Parsing

To understand a program one must be able to parse it, and parsing is the first step. This is precisely the syntactic analysis phase that is the front end of any compiler. The result of the parse is an abstract syntax tree—a tree form representation of the source code.

Sugared Syntax and Unsugaring

Although PAL is not particularly rich in syntactic decoration, it does in many cases provide several alternate ways to express a given idea. Usually, one alternate will be more convenient for the human user than the other. For example, the function form definition

$$\text{let}\ f(x) = E$$

(where E is some expression) could equally well have been written as

$$\text{let}\ f = \lambda x.\ E$$

The first form is surely more convenient for people, although the second form seems conceptually simpler. We regard the first form as a *syntactic sugaring* of the second form. The second step in the formal evaluation of a program is unsugaring it, replacing it by a more austere form. (Landin[6] exhibits Algol as a sugaring of simpler forms.)

Translation

The third step is the translation of the abstract syntax tree to an *evaluation tree*. (In compiler terminology, this is the part of the compiler that generates object code.) The evaluation tree is interpreted by the evaluator. It consists of a sequence of control items, each of which corresponds (roughly) to a single instruction of a rather complex computer.

These three parts of PAL's formalization are distinctly less well formalized than the fourth part, since the rules governing these phases are expressed in English or in flow chart form. To complete this part of the formalization would require a notation to formalize the algorithms. Although this is probably not an exceptionally difficult problem, we have not chosen to attack it. We have felt that other aspects of the problem—particularly the evaluator—are of greater intellectual interest. Nonetheless, the present problem must be solved eventually to complete the task. The techniques developed by IBM in connection with the formal definition of PL/I[7, 8] would undoubtedly be more than adequate for the task, although for our purposes they may well suffer from being *too* much more than adequate.

The Evaluator

The fourth part of the formalization—the part that does the evaluation of the evaluation tree, is the part we feel to be of the greatest intellectual interest. Our evaluator is a direct descendent of that of Landin,[5] although

there are important differences. Following Landin, our evaluator is an abstract machine with four components: a *c*ontrol, a *s*tack, an *e*nvironment and a *d*ump, and is called the CSED machine. We will discuss first each of the four components and explain briefly how the machine operates.

The *control* contains a set of control items to be executed. In the sense that the CSED machine resembles a computer, the control may be regarded as the program of the machine. The machine is started by placing into the control the evaluation structure produced by the translator. The machine operates by considering at each step the "next" item in the control, and acting on it.

The *stack* is a stack of partial results, quite conventional in operation. The *environment* contains name-value pairs by which variables are associated with values. The process of evaluating a variable consists of looking it up in the environment. If the variable appears more than once in the environment, only the "latest" instance of it will be accessed. As suggested below, the environment is actually a tree rather than a list. Finally, the *dump* is used to store the entire state of the machine when a subproblem is to be evaluated. A subproblem evaluation occurs in connection with the evaluation of the body of a function, after its formal parameters have been properly associated with the arguments.

The evaluation of a PAL program starts by placing into the control the entire control structure produced by the translator. Items from the control are processed one at a time, roughly in the order in which they occur in the text of the original program. When a variable is encountered in the control, it is looked up in the environment, and the associated value is placed on the stack. From time to time, the control item *gamma* becomes the current control item. This item signifies functional application, and the element at the top of the stack (the rator) is applied to the second element of the stack (the rand), the result replacing them both on the stack. Consider the case where the rator is a function defined by the programmer. It is the nature of a programmer-defined function that we are to evaluate the body of the function in an environment which has been modified by associating the actual parameter with the formal parameters. Without going into detail, this is precisely what happens. The actual parameters are evaluated in the environment in which the function call appears. At the time of the function call, the formal parameters of the function are associated in a new "environment layer" with the actual parameters supplied. It is at this time that the tree-like nature of the environment becomes important, since it is necessary that free variables appearing in functions be evaluated properly, depending on the textual position of the function rather than upon its place of invocation. As the earlier discussion of the FUNARG problem indicates, this is quite important.

It is not possible in this paper to describe further PAL's evaluator. Weizenbaum[2] describes a similar scheme, as does Landin[5].

A Sample PAL Program

Shown is a listing and a run of a sample PAL program called SQRTY. All it does is to print the square roots of integers from zero to ten. The following comments may assist the reader. The characters "//" introduce a comment, which extends to the end of the line. The program first defines the functions *Sqrt* and *Abs*. *Sqrt* uses an internal function *f* to compute the square root recursively by Newton's method. *Sqrt* and *f* use only the applicative part of PAL. The second part of the program is imperative, using assignment statements and *gotos*. The function *Write* prints output. **t* in a quotation stands for tabulate (as on a typewriter), and **n* stands for newline. *ItoR* converts from integer to real.

In the run shown, the person typed two lines. The first, *r printa sqrty pal*, caused the file SQRTY PAL to be printed. Next, the line *r pal sqrty* invoked the PAL system to compile and run SQRTY. The lines through *Execution* were printed by the system, and the next 13 lines are output from SQRTY. The last two lines are PAL system output.

The Current PAL Implementation

PAL is currently implemented on an IBM 7094 under CTSS at MIT Project MAC and at the MIT Information Processing Services Center (Computation Center). With the exception of a small package of operating system interface routines, the entire system is written in BCPL—a general purpose programming language designed with compiler writing in mind. The actual processing that goes on in the compiler and in the run-time system parallels quite closely the earlier discussion of PAL's formalization. Indeed, the run-time system is very similar to the CSED evaluator.

BCPL runs on several machines and is fully documented.[9] The PAL language is fairly well documented, a reference manual for student use having been prepared.[10]

BCPL has been designed deliberately in such a way as to facilitate its bootstrapping onto new machines, and has already been transferred successfully several times. PAL could be implemented on a new computer by first implementing BCPL. As these things go, this seems fairly straightforward.

ACKNOWLEDGEMENTS

The PAL language is a direct descendent of Peter Landin's ISWIM,[11] although there are important differences, particularly in the imperatives. The first implementation of PAL was by Landin and James H. Morris,

```
r printa sqrty pal


 SQRTY    PAL     06/14/68  0040.6

// Sample PAL program.

let Abs x =  // compute absolute value of argument
          x < 0.0  ->  -x  !  x
in

let Sqrt x =  // compute square root of x
     f(x/2.0)  // initial approximation is x/2
     where rec f t =  // a recursive function for Newton's method
          Abs( t*t - x ) < 0.005  // is t close enough?
          ->   t    // Yes, so return t as result.
          !    f ( 0.5*(t + x/t) )  // No, so keep trying...
in

let i = 0  // a counter, to have its square root taken.
in

L:   // the main loop of this rather simple little program..
     Write (i, '*t', Sqrt (ItoR i), '*n');
     i := i + 1;
     i < 11  // did we just do 10?
     ->      // not yet, so keep going
          goto L
     !       // All done, so say so and go home.
     Write '*nAll done.*n'



r pal sqrty
W 042.0
Pal compiler entered
Pal loader entered
Execution
0           0.00000E+00
1           1.00030E+00
2           1.41422E+00
3           1.73214E+00
4           2.00000E+00
5           2.23611E+00
6           2.45000E+00
7           2.64575E+00
8           2.82843E+00
9           3.00002E+00
10          3.16232E+00

All done.

Execution finished
Number of cycles: 977
```

Jr., in LISP. The language they implemented was much closer to ISWIM than to PAL as it now exists.

The present version of PAL was designed by Martin Richards along with Thomas J. Barkalow, Evans, Robert M. Graham, Morris and John M. Wozencraft. The implementation is the work of Richards and Barkalow.

The intellectual effort of which PAL is one outgrowth owes much to Christopher Strachey.

REFERENCES

1. Strachey C
   unpublished manuscript
2. Weizenbaum J
   The FUNARG problem explained
   unpublished manuscript
3. Church A
   The calculi of lambda conversion
   Princeton University Press Princeton 1941
4. Curry H B and Feys R
   Combinatory logic vol 1
   North Holland Publishing Co Amsterdam 1958
5. Landin P J
   The mechanical evaluation of expressions
   The Computer Journal 6 4 Jan 1964 pp 308-320
6. Landin P J
   A correspondence between Algol 60 and Church's lambda-notation
   Comm ACM 8 Number 2 Feb 1965 pp 89-101 and Number 3 Mar 1965 pp 158-165
7. Alber K Oliva P and Urschler G
   Concrete Syntax of PL/I
   IBM Laboratory Vienna Technical Report TR25.084 Mar 15 1968
8. Alber K and Oliva P
   Translation of PL/I into Abstract Syntax
   IBM Laboratory Vienna Technical Report TR25.086 Mar 15 1968
9. Richards M
   BCPL reference manual
   Project MAC Memorandum M-352-1 February 16 1968
10. Evans A
    PAL—A reference manual and a primer
    MIT Department of Electrical Engineering Feb 1968
11. Landin P J
    The next 700 programming languages
    Comm ACM 9 3 March 1966 pp 157-164