PAL

Pedagogic Algorithmic Language

A Reference Manual and A Primer

Arthur Evans, Jr.

Department of Electrical Engineering

Massachusetts Institute of Technology

February 1968

Introduction and Acknowledgements

PAL -- Pedagogic Algorithmic Language -- is a computer programming language that has been developed as a teaching vehicle in connection with the subject 6.231, Programming Linguistics, in the Electrical Engineering Department at the Massachusetts Institute of Technology. 6.231 is designed to be taught to second term sophomores who have already taken an introductory subject in computer programming and who intend to take further subjects in Computer Science. The present document is both a primer and also a reference manual for the PAL language.

PAL is currently implemented in CTSS, a general purpose time sharing system running on a modified IBM 7094 computer at the MIT Information Processing Service Center, and is used interactively by students. A major part of 6.231 is a set of homework exercises to be carried out on the computer.

PAL is itself written in BCPL, a general purpose programming language. Since BCPL has been designed to bootstrap itself easily onto other computers, it should be possible to implement PAL on a new computer without excessive difficulty (as these things go).

PAL is a product of the effort of many people. Intellectually, PAL is a direct descendant of ISWIM, a language developed by Peter J. Landin. (See "The next 700 programming languages", by P. J. Landin, Comm. ACM 9, 3 (March 1966), 157-164.) The initial implementation of PAL was by Landin and James H. Morris, Jr., in LISP. The PAL language described in the present document has been designed by the joint efforts of Thomas J. Barkalow, Arthur Evans, Jr., Robert M. Graham, Morris, Martin Richards, and John M. Wozencraft. The implementation is the work of Barkalow and Richards. The language BCPL is the work of Richards. The present document is by Evans, with critical assistance from the rest of the team. Barkolow wrote appendix 3.3, and appendix 2.1 is the work of Richards and Morris.

Forward to the Student

This document is your principle source of information about the PAL language. When it has been supplemented with class handouts on the mechanics of using the computer, you will have all of the information you need to write programs in PAL and carry them through execution on the computer.

A short glance at this manual will reveal that it is not yet complete, there being sections referred to in the table of contents that do not yet exist. Some of the missing sections will be passed out in class during the semester.

Undoubtedly typographic and other errors will be found in this manual. The 6.231 staff will be grateful for all such errors reported.

In your initial approach to this manual, the following may be found useful. Read first the introductory sections to the four main chapters: sections 1.0, 2.0, 3.0 and 4.0. Doing so will give you an overall introduction to what is in PAL. (Were section 0.3 written it would serve this purpose.) Next read sections 0.1 and 0.2, which explain the format of the manual and many of the conventions used. After that all there is to do is to read chapters 1 to 4 in order.

Before attempting to write a PAL program, study section 0.4 and appendix 3. You will also need some material passed out in class.

The worked examples in appendix 4 will undoubtedly be found useful. Another source of correct PAL programming available to you is the PAL library, accessible on the computer. It contains some examples of well written PAL text.

Good luck!

Table of Contents

Table of Contents

The Table of Contents was last modified on 02/04/68 at 16:22 by Evans.

(This section was last modified on 02/03/68 at 23:10 by Evans.)

This document is designed to serve two purposes in that it is to be both a primer and also a reference manual for the language PAL. There seems to be ample evidence in the computer profession that these two goals are incompatible in a single document. To the extent this is true, then, this document seems to be more successful as a reference manual than as a primer. Nonetheless, the newcomer to PAL should find it adequate.

The Plan of the Manual

Following this introduction, there are five chapters describing the PAL language and then several appendices on various topics. The first four chapters, which make up the bulk of the manual, are divided into sections each describing some aspect of PAL. Each section is subdivided into (up to) six parts, as follows:

Introduction
Formulae
Notes
Semantics
Examples
Advice

The first letter of each of these words identifies the subsection. Thus a reference to section 1.2/N refers to the Notes in part 2 of chapter 1. Chapter 5 is on miscellaneous language topics and is organized less formally.

A few more words about the six subdivisions are in order. In some sections, one or more of the subdivisions will be missing, since they would serve no useful purpose. When the sections are present, they describe a new concept as follows:

- Introduction: Informal introductory comment about the new concept is given. If the concept requires understanding of some aspect of PAL which appears later in the manual, a few words are given here about the missing ideas.

- Formulae: The syntax of those aspects of the language relevant to the new concept is given. The notation used is described in section 0.2.

- Notes: Frequently, further notes and descriptive material are necessary concerning the syntax. Sometimes this is because the syntax equations do not tell the whole story, and it is necessary to give more information.

- Semantics: In this subdivision, we give the meaning of PAL program elements the user might write using the new concept. The discussion here is informal, assuming the existence of a "friendly" reader.

- Examples: Examples are given of PAL program elements using the new concept.

- Advice: Frequently there are issues of convenience or running efficiency related to the new concept of which the user should be aware but which find no home elsewhere.

Page Numbering Conventions:

The pages of this manual are numbered to indicate their section, and are not numbered consecutively as usual. Thus the first page of the present section is numbered "0.1 - 1". The first page in the Notes part of section 3 of chapter 2 would be numbered "2.3/N - 1". Page numbers in appendices are prefixed with "Ap", so the first page of Appendix 1.1 is numbered "Ap 1.1 - 1".

Underlining and Quoting Conventions

In a manual such as this one, it is frequently necessary to refer in text to constructs of the language. To distinguish them from the English text that is the manual, such constructs will usually be surrounded by the double-quote mark " . On occasion where no ambiguity may result, such constructs will be underlined. These conventions apply only to in-line text and do not apply to text displayed on a line by itself (usually centered). For example, we might be discussing the PAL expression

```
(x + 1)*(x + 2) where x = 7
```

and refer in text to the functor "+" or to the variable <u>x</u> or to the punctuation "where".

The words "Interpret" and "Denote"

Certain words will always be used in this manual in a technical sense only.

(This section was last modified on 02/03/68 at 23:14 by Evans.)

Introduction

In specifying a computer language, there are two problems to solve: It is first necessary to specify exactly which strings of characters (out of all possible such strings) are legal sentences in the language; and it is next necessary to specify exactly what is the meaning of each such legal sentence. The first problem has to do with the syntax of the language, and the second has to do with semantics. In this section the technique used in this manual for the specification of syntax will be discussed.

The Formalism used for Syntax

The term "syntax" has to do with specifying those strings of characters that are legal instances of a particular class. It has been found that specifying syntax is done most conveniently with the use of a formalism. That is, a particular notation is used that gives an exact definition of what are the legal strings. Since the strings being defined are the strings of the language, we use the term "meta-linguistic" to refer to marks used in the definition.

Six meta-linguistic marks are used in our formalism:

< > ::= | { }

The first two of these marks are "meta-linguistic brackets", used to enclose the name of a construct. The third mark is a single entity: "colon-colon-equal". It can best be read as "is defined to be". The fourth mark is best read "or". The last two marks serve as parentheses at the meta-linguistic level. These marks are used in the formal description of the syntax, and it is assumed that identical marks are not used in PAL. (In section 1.1 we will see that special provision is needed to talk about the fact that each of these symbols other than "::=" is actually used in PAL.) The use of this notation will now be explained.

A definition such as

```
<letter> ::= a | b | c | d | e | f | g
```

may be read as, "An element of the meta-linguistic class <letter> is defined to be either "a" or "b" or "c" or "d" or "e" or "f" or "g". Thus the definition specifies that each of the first six characters of the alphabet, lower case, is a member of the syntactic category <letter>. Having this definition, we may write

```
<string> ::= <letter> | <string> <letter>
```

This can be read, "The class <string> is defined to be either a <letter> or a <string> followed by a <letter>." Note that <string> is defined in terms of itself. We will now show that "abc" is a <string> in terms of the above definition. It is not a <letter>, so we ask if it is a <string> followed by a <letter>. Since "c" is a letter, the answer is yes providing that we can show that "ab" is a <string>. Similarly, we can deduce that "ab" is a <string> if we can show that "a" is one, since "b" is a

<letter>. But "a" is a <string> because it is a <letter>, and we conclude that "abc" is a string.

An English sentence equivalent to the above definition of the class <string> might be, "A <string> is a sequence of any length of items from the class <letter>." Although it is easy to give an English sentence equivalent to the syntax equation just given, we shall see many syntax equations that are sufficiently complicated that an equivalent English description would be awkward.

Let us continue this example further. Suppose that we define

```
<LETTER>  ::=  A | B | C | D | E | F | G
<digit>   ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

In PAL, the programmer may use as names identifiers made up out of arbitrarily long strings of <letter>s, <LETTER>s and <digit>s. (Of course, the entire alphabet is available to the PAL user instead of just the first six letters, but the present explanation is more easily given this way.) Thus we might propose defining <name> by

```
<char>  ::=  <letter> | <LETTER> | <digit>
<name>  ::=  <char> | <name> <char>
```

As it happens, however, there are two restrictions that we wish to have in PAL: First, if the name consists entirely of digits, it is interpreted as an integer and not as a programmer's name;

and second, a name two or more characters long consisting entirely of lower case letters is interpreted as a "system word" built into PAL with a predefined meaning, and is not available to the programmer for his use as an identifier. We will now provide, as our example of the syntax notation to be used, a complete description of a class <variable> which does not include integers or system words.

Before doing so, we will expand the notation. The syntax equation given above for <name> has a form that appears quite often in this sort of work. It defines one item as a string of arbitrary length of instances of some other item. Because this is common, it is expedient to introduce a notation. We write

$$\langle\text{name}\rangle \; ::= \; \langle\text{char}\rangle_1^\infty$$

to indicate that a <name> consists of one to arbitrarily many instances of <char>. Note that the mark "∞" as used here means "arbitrarily many", or "as many as desired", neither of which is quite the same as "infinity".

Actually, we do not need the definition of <char>, since we can write

$$\langle\text{name}\rangle \; ::= \; \{ \; \langle\text{letter}\rangle \mid \langle\text{LETTER}\rangle \mid \langle\text{digit}\rangle \; \}_1^\infty$$

The intent of this notation is that it define the same class of strings as that defined by the previous notation. Possible instances of the class name include the following:

a A 4 aAf3g 1ab 9o00 a1b2c3d4

With this notation, the classes <integer> and <system word> mentioned earlier can be defined by the equations

<integer> ::= <digit>$_1^\infty$

<system word> ::= <letter>$_2^\infty$

Here a system word consists of two or more <letter>s, according to the definition. Similarly, a class consisting of two or more <name>s separated by plus signs could be defined by

<summation> ::= <name> { + <name> }$_1^\infty$

In a language like Fortran where symbols are no more than six characters long and must start with a letter, we might have

<symbol> ::= <LETTER> { <LETTER> | <digit> }$_0^5$

(Remember that only upper case letters are available in Fortran.)

We now provide the definition of <variable> promised earlier. Assuming that the classes <digit>, <letter> and <LETTER> are already defined, we have

<symbol head> ::= <digit>$_1^\infty$ { <letter> | <LETTER> }
| <letter>$_1^\infty$ { <LETTER> | <digit> }
| <LETTER>

<variable> ::= <letter>
| <symbol head> { <letter> | <LETTER> | <digit> }$_0^\infty$

The reader should satisfy himself that <variable> as defined here has all the properties asked for.

Syntactic Ambiguity

It is quite easy to write syntax with the property that certain strings are defined by the syntax in more than one way. In such a case, we say that the syntax is ambiguous. Consider

```
<sheep noise>  ::=  baa | <sheep noise> <sheep noise>
```

This definition is ambiguous, since the phrase

```
baa baa baa
```

can be a <sheep noise> in more than one way. The difficulty is that we do not know whether the baa in the middle should be associated with the first or third baa to make a <sheep noise>, and there is nothing in the definition to tell us which. Alternatively, we could have

```
<sheep noise 1>  ::=  baa | <sheep noise 1> baa

<sheep noise 2>  ::=  baa | baa <sheep noise 2>
```

It should be clear that each of these two definitions is unambiguous, the first associating baas to the left and the second associating them to the right. Further, each of the three definitions defines the same class of strings.

In a language such as PAL, it is usually possible to write syntax definitions unambiguously. The definition of <variable>

above was somewhat complex because of the desire to make it unambiguous. (The reader should satisfy himself that it is, indeed, unambiguous.) However, we shall frequently choose to give ambiguuous syntax, mainly because removing the ambiguity from the formulae seems to introduce more problems than it solves. (Of course, we must remove the ambiguity somewhere if we are no have a reasonable definition of a language. Usually the Notes section will address this issue.) Consider the case of arithmetic expressions on integers. We might have

```
<operator>  ::=  + | - | * | /

<expression>  ::=  <integer> | ( <expression> )
                 | <expression> <operator> <expression>
```

(Here <integer> is as above.) This syntax is ambiguous, since phrases like

2 + 3 * 4

can be interpreted in more than one way. It should be clear that this problem is serious, since one interpretation gives <u>14</u> as an answer and the other gives <u>20</u>.

We first show that it is possible, in definitions such as this, to produce an unambiguous syntax. Using <integer> as above, we define

```
<primary>  ::=  <integer> | ( <expression 1> )
```

```
        <term>  ::=  <primary> | <term> { * | / } <primary>

        <expression 1>  ::=  <term>
            | <expression 1> { + | - } <term>
```

The reader should satisfy himself that the same strings that are <expression>s are also <expression 1>s, but that <expression 1>s are unambiguous.

Now that we have shown that it is possible (at least in this case) to write unambiguous syntax, we explain why we frequently choose not to. The definition of <expression> is simpler and more straightforward than that of <expression 1>. It is clear at a glance what strings are <expression>s, but deducing what strings are <expression 1>s takes a bit more study. Thus from the point of view of the learner, it seems preferable in such cases as this one to choose the ambiguous syntax for the formulae.

Of course, there is still a problem to solve: The purpose of a manual such as this one is to convey, accurately and unambiguously, just what is the meaning of each sentence in the language. This is done in the present manual by appending to the formal syntax in the Formulae section such English description as seems appropriate, and it is this latter that goes into the Notes section. In a definition such as the above of <expression>, the following discussion would be provided:

> The definition of <expression> is ambiguous. To deduce the meaning of ambiguous phrases, note the precedence table

given below.

Operator	Precedence
+	1
-	1
*	2
/	2

Here each operator is assigned a precedence. In cases where a particular subexpression may be associated with the operator on either its left or its right, it will be associated with whichever operator has higher precedence. In cases of equal precedence, the operator on the left will be used.

The reader should satisfy himself that the informal discussion just given will have the same effect as the formal definition of <expression 1>.

(This section was last modified on 02/04/68 at 16:37 by Evans.)

The term "the PAL language" is actually somewhat ambiguous, since there are several different possibilities as to what it means. The problem has to do with the available character set. Ideally, we could write programs in _canonic PAL_, a version of PAL which assumes the existence of an arbitratily large character set. In practice, we are concerned in this manual with the implementation of PAL as it exists (in the spring of 1968) on CTSS on the IBM 7094 at MIT. In addition, unfortunately, we are also concerned with problems arising from the fact that some console devices attached to CTSS differ from others, and different characters are available on different consoles.

The present section of the manual concerns itself with the distinction between canonic PAL and the PAL implemented on CTSS. The problems of differing console devices are discussed in Appendix 3.2.

It is convenient for PAL's designers to assume the existence of a very large character set. For example, it would be pleasant to use the marks "∧" and "∨" for the Boolean connectives "and" and "or", respectively, since these are the marks usually used in logic. Unfortunately these marks are not available on typewriter devices attached to the computer to which we have access. Thus the writer of a manual such as this one faces a problem: Should he use "∧", or should he replace that mark by something that can be input to the computer?

In the present manual, we opt for the latter choice. The purpose of this manual is to assist a new user of PAL to understand PAL. Since the reader's objective is to be able to use PAL on a computer, it seems most useful to describe PAL in a manner as close as is convenient to the notation actually used on the computer. Thus this manual describes an *implementation* of PAL -- indeed, a particular implementation on a particular computer at a particular time.

On the other hand, the main part of this manual *ignores* the problem (mentioned above) of differing consoles. For example, the relation of *a* being greater than *b* is expressed in this manual as

$$a > b$$

even though there are some consoles not equipped to type the "greater than" mark. From such consoles one might type

a gr b

(as explained in detail in Appendix 3.2).

Chapter 1:

The Basic Elements of PAL

(This section was last modified on 02/02/68 at 11:55 by Evans.)

In defining a conventional language, we start with an *alphabet*. Using this we make up first *words* and then larger constructs such as *sentences* and *paragraphs*. A similar technique is used in defining the PAL language. PAL's alphabet is (roughly) the set of characters that can be typed on the devices available. (A complete description of PAL's alphabet is found in section 1.1.)

PAL's alphabet is used to make up *words*. There are several ways to organize PAL's words into hierarchical classes, one of which is the following:

- identifier
  - variable
  - constant
    - quotation
    - numeric
    - literal
- functor
- punctuation

The classes just shown are based on the purpose to which their members are put in PAL. Another way to categorize PAL's words is snyntactically, but that is less useful for the present purposes. We consider first the meaning of each of the above classes, and defer till later in this section the syntactic categorization.

Identifiers are words available to the programmer to denote the values which he wishes to manipulate in the course of a computation. Certain identifiers, such as "2", always denote the same value -- a value which can be deduced from the form of the identifier. Such identifiers are called constants, and come in three forms: quotations, which denote strings; numerics, which denote numeric values; and literals, which denote certain "built-in" values. (We see later that there is a further breakdown of numerics into those that denote integer values and those that denote real values.)

Identifiers other than constants are variables. These denote values of the programmer's choosing -- frequently denoting different values during the course of a computation.

Identifiers, both variable and constant, are discussed in section 1.4.

A functor is a word whose effect, when the program is evaluated, is the execution of an operation. For example, the functor "+" indicates addition, and the functor "aug" indicates the operation of augmenting a tuple. Functors are discussed in section 1.2.

The remaining words in PAL's vocabulary are called punctuations. These include "let", "and", "||" and others. Punctuations are discussed in section 1.3.

As suggested above, an alternate way to categorize PAL's words is syntactically, or by the form rather than by the meaning of the word. We might then have the following classes:

variable

quotation

numeric

reserved word

special

Here the first three classes are as above. A reserved word is one made up of two or more lower case letters. This class includes "true" (a literal in the first categorization), "aug" (a functor), and "let" (a punctuation). The class special then includes "+" (a functor) and "." (a punctuation). It is always possible to tell by looking at a word which syntactic class it is a member of, but the first categorization does not have that property. (The only way to tell that "let" serves as a punctuation is to look at a list. On the other hand, it is clearly a reserved word, by its form.)

All of this discussion is best summarized by the following table. Note that each of PAL's words (other than variables, quotations and numerics) is shown exactly once in this table, with the exception of "=" which is shown twice. (This symbol is used as both a functor and also as a punctuation.)

	reserved word	special
literal	true false nil dummy	
functor	not aug val res jj	+ - * / ** & |
		< = > % $
punctuation	let in where within	( ) [ ] { { = .
	rec and ll goto	-> ! ; := : ,

Introduction: (This section was last modified on 02/02/68 at 12:28 by Evans.)

As mentioned in section 1.0, text in PAL language is written using a particular alphabet which is made up of characters. This section contains a listing of PAL's character alphabet, along with certain other information relevant to creating PAL text.

The alphabet chosen for PAL is the so-called ASCII alphabet. ASCII -- the word is acronymic for American Standard Code for Information Interchange -- represents a standard character set which (hopefully) will some day be available on all computers and all console devices. Because of ASCII's growing acceptance in the computer world in general and at MIT in particular, PAL's designers have chosen it for the implementation language of PAL. The full ASCII character set, as defined by USASI (the United States of America Standards Institute), consists of 128 characters. Of these, 94 are printable graphics; some others are format effectors such as space, backspace, tabulate, new-line, etc.; and the rest are control characters of no interest to the present discussion. PAL's alphabet includes the graphics and a selection from the format effectors. (The interested reader may learn more about ASCII by consulting Communications of the ACM, Volume 8, Number 4 (April 1965), pages 207-214.)

In this less than perfect world, many console devices are deficient in that they do not provide for direct input of all of the character set. Appendix 3.2 contains the information needed to use PAL from the various devices which are actually available. In this primer, the entire ASCII set is assumed to be available, since it is possible (albeit somewhat awkward) to input or print any character with any device.

A PAL program should be regarded as one continuous character stream rather than as a sequence of lines or card images, in that the transition from one line to the next has no significance in the language. (There are two exceptions to this rule: The newline character is treated as a space, so a construct such as an identifier which may not have embedded spaces may not be continued from one line to the next. Also, newline terminates the comment convention.) In the program text, the actual transition from one line to the next is indicated by the character "newline", the character produced when the "carriage return" key is typed.

Formulae:

<upper case letter> ::=

A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

<lower case letter> ::=

a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

<letter> ::= <upper case letter> | <lower case letter>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<alphanumeric> ::= <letter> | <digit>

<special character> ::= ! | " | # | $ | % | & | ' | (|) | * |
+ | , | - | . | / | : | ; | <less than> | = | <greater than>
| ? | @ | [| \ |] | ^ | _ | ` | <left brace> | <bar> |
<right brace> | ~

<format effector> ::= <space> | <tab> | <newline> | <backspace>
| <black ribbon shift> | <red ribbon shift>

<character> ::= <alphanumeric> | <special character>

<composite token> ::= // | := | -> | **

<quotation> ::= ' <any character other than '>$_0^\infty$ '

<reserved word> ::= <lower case letter>$_2^\infty$

Special Definitions:

<less than> ::=

<greater than> ::=

<bar> ::=

<left brace> ::=

<right brace> ::=

<space> ::=

<tab> ::=

<newline> ::=

<backspace> ::=

<black ribbon shift> ::=

<red ribbon shift> ::=

Notes:

1. Omission of Spaces. In PAL as in English, the presence or absence of spaces can affect the meaning of what is written: "DIGIT" and "DIG IT" have different meanings. On the other hand, there are many places in PAL where the user may insert spaces as he chooses to improve the readability of the text. (See appendix 4 for some very readable examples of PAL.) The following rules specify where spaces are optional in PAL. In these rules (and only in these rules), the word "namer" is used to stand for a variable, a quotation, a numeric or a reserved word. Further, the word "space" stands for any arbitrary concatenation (one or more characters long) of the characters space, tab and newline. In preparing PAL text, proceed as follows:

1. Provide space on each side of each word. (Here word is used in the sense of section 1.0.)

2. Place arbitrary space between any two characters, providing they are not both part of the same namer and that they do not make up a composite token.

3. Eliminate space at will, providing only that the effect is not to cause two alphanumerics that were previously separated by space to become adjacent.

The above three rules are to be obeyed in order. Note that following rule 3 can result in the elimination of space required by rule 1. Since alphanumerics are used in PAL only in namers, the effect of these three rules may be seen to be the following:

Space is required between namers that would otherwise be adjacent, and space is optional at all places except in the middle of a namer or a composite token.

2. Special definitions. The 94 ASCII printable graphics are included in the class <character>. In addition, the following format effector classes will be used whenever they are needed, although typographic problems make it awkward to define them in the usual way.

<space>
<tab>
<newline>
<backspace>
<black ribbon shift>
<red ribbon shift>

In addition, the following five classes represent characters that are used in the syntax definitions of this manual. For this reason, these characters cannot appear in syntactic equations.

<less than>
<greater than>
<bar>
<left brace>
<right brace>

Each above class consists of the single character represented by its name.

One other sort of syntactic class is left undefined. For example, the class

<any string not including "newline">

is used in the next paragraph but never defined, since the membership of the class is obvious from the name of the class.

3. Comment conventions. The user may insert certain arbitrary sequences of characters into his programs to improve the self-documentation of what he writes. The presence of the sequence

// <any string not including "newline"> <newline>

will not affect a program. That is, a double-slash and all characters to its right will be ignored on any line.

4. Quotations. As will be seen in more detail in section 1.4, one of the basic data types the programmer may manipulate is strings of characters. A string is an ordered sequence of characters from PAL's alphabet. A quotation is (roughly) any sequence of characters not including the single quote mark ' , surrounded by quotes. Inside of a quotation, and only in a quotation, the characters space, tab and newline have meaning. In other words, the newline character may be quoted. However, because doing so is awkward, PAL recognizes certain special conventions inside of quotations, as described in detail in section 1.4. Because of the existence of these conventions, the syntax shown is not completely correct. (Correct syntax appears

in section 1.4.)

5. Interaction between comment and quotation conventions. Two rules have been given for special processing of text by PAL: between the composite token "//" and the next succeeding newline, and between successive instances of the single quote. These rules interact in the following way: a "character reading" program that is part of the PAL translator examines text in the same order that it is written, from left to right on each line and from top to bottom of the page. Whenever processing of either of these rules begins, the other rule is ignored until the end of the effect of the first rule. Thus // in a quotation does not signify a comment, and single quotes need not be paired in comments.

6. Composite Tokens. A composite token is made up of two special characters from the basic alphabet which have a special meaning when used together. They must be typed in adjacent positions on the paper, with no intervening space or newline. The composite tokens are now listed.

// comment convention. See Note 3 above.

:= the assignment operator. See section 3.2.

-> the conditional. See sections 2.5 and 3.4.

** the infixed exponentiation operator. See section 1.2/S.

7. Unused characters. The observant reader will have noticed that the class <special character> includes the following

characters that are not (apparently) used in PAL:

" # ? @ \ ^ ` ~

These characters are available for programmer use in quotations.

Introduction: (This section was last modified on 02/04/68 at 16:47 by Evans.)

Functors are those words in PAL's vocabulary whose effect, when the program of which they are a part is evaluated, is the execution of an operation. For example, the "+" in the expression

```
3 + 4
```

is a functor indicating addition. Even assuming that Add denotes a function of two arguments that returns their sum, there is an important difference between the previous expression and

```
Add(3, 4)
```

Functors may be infix operators; i.e., they may appear between their operands. Thus there is a syntactic issue in deducing that the left operand of "+" in

```
3 * 4 + 5
```

is the product of 3 and 4 and is not just 4. There is no such problem in an expression such as

```
Add(Mult(3, 4), 5)
```

Formulae:

```
<functor> ::=
        <infix arithmetic functor>
      | <prefix arithmetic functor>
      | <infix logical functor>
      | <prefix logical functor>
      | <arithmetic relational functor>
      | <equality relational functor>
      | <tuple-making functor>
      | <jumping functor>
      | <operator defining functor>
      | <unsharing functor>

<infix arithmetic functor>  ::=  + | - | * | / | **

<prefix arithmetic functor>  ::=  + | -

<infix logical functor>  ::=  & | <vertical bar>

<prefix logical functor>  ::=  not

<arithmetic relational functor>  ::=
        <less than> | <greater than>

<equality relational functor>  ::=  =

<tuple-making functor>  ::=  aug

<jumping functor>  ::=  jj | val | res
```

```
<operator defining functor>  ::=  %

<unsharing functor>  ::=  $


<infix functor>  ::=
       <infix arithmetic functor>
     | <infix logical functor>
     | <arithmetic relational functor>
     | <equality relational functor>
     | <tuple-making functor>

<prefix functor>  ::=
       <prefix arithmetic functor>
     | <prefix logical functor>
     | <unsharing functor>
```

Notes:

Note that the infix logical functors are "&" (read "and") and "|" (read "or"). The reserved word and is also used in PAL, but for a completely different purpose. (It is used as a conjunction between the components of a simultaneous definition. See section 4.3.)

The mark "=" is used both as a functor and as a punctuation. PAL's syntax is such that its purpose can never be confused.

The classes <infix functor> and <prefix functor> are not used elsewhere in this section, but they are used in other parts of the manual. Their definitions are presented here for later convenience.

Semantics:

A short description is now given of each of the functors. The functors are described in this section in the same order that they appear in 1.2/F.

Infix Arithmetic Functors

+ indicates addition, either between two integers or between two reals.

- indicates subtraction, either between two integers or between two reals.

* indicates multiplication, either between two integers or between two reals.

/ indicates division, either between two integers or between two reals.

** indicates exponentiation, in that "a**b" in PAL is the same as "a^b" in conventional mathematical notation. The left operand may be integer or real, and the right operand must be integer. The value is the same type as the left operand.

Prefix Arithmetic Functors

+ leaves unchanged any numeric operand, and is undefined for non-numeric operands.

- changes the sign of any numeric operand.

Infix Logical Functors

& The expression "p & q" is defined if both *p* and *q* denote logical values. It denotes the value *true* if both *p* and *q* do, and denotes *false* otherwise.

| The expression "p | q" is defined if both *p* and *q* denote logical values. It denotes *true* if either *p* or *q* do, and denotes *false* otherwise.

Prefix Logical Functor

not Changes the sense of a logical operand, in that "not true" is "false" and "not false" is "true".

Arithmetic Relational Functor

< less than

> greater than

Each of the relational functors is defined if its two operands are either both integer or both real. The expression "a relat b" (where *relat* is one of the two relational functors just listed) will be denote *true* if the value denoted by *a* stands in relation *relat* to that denoted by *b*.

Equality Relational Functor

= This infix binary functor is defined if its operands are integers, reals, truth values or strings. If both operands are of the same type and have the same value, the relation is *true*, while otherwise it is *false*. Note that the result is defined (and *false*) if the two operands are of different type.

Tuple-Making Functors

aug If *T* denotes an *n*-tuple and *E* denotes any value, the expression "T aug E" denotes an *n+1*-tuple whose first *n* elements have the same values as (and share with) the corresponding elements of *T* and whose *n+1*-st element shares with *E*. For further details on tuples, see section 2.2. For more on sharing, see Appendix 1.5.

Jumping Functors

val see section 5.5

res see section 5.5

jj see section 5.4

Operator Defining Functor

% Suppose that *E* and *F* are any expressions and *N* is a name. Then the expression "E %N F" denotes the same value as does the expression "N(E, F)". In other words, % permits writing a prefix binary function in infix form.

Unsharing Functor

$ This prefix unary functor may be applied to any object whatsoever. The result of the application denotes the same value as does the argument, but the result does not share with any other object. (Of course, components of the result may share. See appendix 1.5 on sharing.)

Comment

Throughout the previous discussion, the issue of identifying the left and right operand of the various infix functors has been ignored. That is, this section does not specify what interpretation is to be placed on an expression such as

p & q | r

Is the left operand of "|" to be q or is it to be "p & q"? Similarly, we do not know the interpretation of

a + b %f c

These issues are discussed in section 2.3 on infix operators and in section 5.3 on %.

Introduction: (This section was last modified on 02/04/68 at 16:49 by Evans.)

Those words of PAL's vocabulary that are neither identifiers nor functors are called punctuations. This last is a sort of catch all set that includes all of those constructs that seem to find no home elsewhere.

Formulae:

```
<punctuation> ::=

       <bracket>

     | <definitional punctuation>

     | <conditional punctuation>

     | <lambda-expression punctuation>

     | <imperative punctuation>

     | <label defining punctuation>

     | <tuple punctuation>

<bracket> ::= ( | ) | [ | ] | { | }

<definitional punctuation> ::= let | in | where | within | rec

     | and | =

<conditional punctuation> ::= -> | !

<lambda-expression punctuation> ::= ll | .

<imperative punctuation> ::= ; | goto | :=

<label defining punctuation> ::= :

<tuple punctuation> ::= ,
```

Notes:

It should be noted that "=" is both a punctuation and a functor. PAL's syntax is such that it is always possible to tell at any occurrence which class it belongs to.

All three types of brackets may be used, and they are equivalent in effect. The only proviso is that they must be matched. (For example, the expression

(x+1]/[x-1)

is not correct.) In the syntax formulae used hereafrer only round brackets will be used, but the reader should understand any pair may be used.

Introduction: (This section was last modified on 02/03/68 at 24:10 by Evans.)

Identifiers permit the programmer to identify and to reference the various values which enter into a computation. Identifiers are either variables or constants. A variable is an identifier selected by the programmer, which he may use to denote any value produced during the course of a computation. Thus variables are of the programmer's choosing and will, in general, denote different values during the course of a computation. A constant identifier, on the other hand, has the property that the value it denotes is "obvious" from the form of the identifier, and further that this value cannot change during the course of a computation. For example, the constant identifier 2 always denotes the second positive integer.

Certain variable identifiers are predefined when the user begins to use PAL, in that they denote values provided by the PAL system designers. For the most part, these values are functions that the user could not write himself. They are listed in section 2.4.

Formulae:

<identifier> ::= <variable> | <constant>

<variable> ::= <lower case letter>
    | <variable head> <alphanumeric>$_0^\infty$

<variable head> ::=
    <digit>$_1^\infty$ <letter>
    | <lower case letter>$_1^\infty$ { <upper case letter> | <digit> }
    | <upper case letter>

<constant> ::= <quotation> | <numeric> | <literal>

<quotation element> ::=
    <any character other than * or ' >
    | *n | *t | *b | *s | ** | *' | *k | *r

<quotation> ::= ' <quotation element>$_0^\infty$ '

<numeric> ::= <integer numeric> | <real numeric>

<integer numeric> ::= <digit>$_1^\infty$

<real numeric> ::= <digit>$_1^\infty$. <digit>$_1^\infty$

<literal> ::= true | false | nil | dummy

Notes:

In general, the intent is that any sequence of one or more alphanumerics may be used as an identifier. However, some such sequences have special meanings. An identifier consisting entirely of digits is a constant denoting an integer. Identifiers two or more characters long consisting entirely of lower case letters are not available for arbitrary use by the user but are instead specified by the designer of the PAL system. Some such identifiers are literals, denoting built-in values. Others are functors (see section 1.2) or punctuations (see section 1.3). The rather complex syntax for variable excludes both classes. It also provides an unambiguous definition, as explained in section 0.2.

Note that both "1." and ".1" are not members of the class <real numeric> according to the syntax, since at least one digit is rwquired on each side of the decimal point. They would have to be written as "1.0" and "0.1", respectively.

Semantics:

The value denoted by a variable depends on the evaluation of the program of which it is a part.

A constant denotes a value that can be deduced from the form of the constant. Constants in PAL have the property that they never share with anything. (Thus, for example, a constant cannot be updated. See 3.2/N for further discussion of this point.) We now discuss how the value denoted by a constant is determined.

A quotation denotes a string whose characters are the characters appearing between the quote marks. In a quotation (and nowhere else) the characters space, tab and newline have meaning. In addition, the character "*" has special meaning in a quotation. The "*" and the character immediately following the "*" are replaced by a single character, according to the following table:

*n	newline
*t	tab
*b	backspace
*s	space
*'	'
*k	black ribbon shift
*r	red ribbon shift

If "*" is followed by any character other than one listed in this table, the effect is undefined. In any particular implementation of PAL, there will be some upper limit on the maximum length string permitted. See appendix 3.1 for current details.

An integer numeric denotes that integer represented by the digits which make up the identifier. In any particular implementation, there will be limits on the range of possible values that can be denoted. For example, there will be an upper limit on the magnitude of the integers that can be represented. The effect of using an integer numeric that denotes a value too large for the implementation is not defined. (See appendix 3.1 for details.)

A real numeric denotes that rational number represented by the digits that make up the identifier. In any particular implementation, the largest possible value, the smallest positive value and the precision will be limited. The effect of exceeding these limits is not defined. (See appendix 3.1 for details.)

The literal true denotes the abstract object truth, and false denotes falsehood. (Each occurrence of true could be replaced by the expression "(0=0)" and each occurrence of false by "(0=1)".) The literal nil denotes the zero-tuple. The literal dummy denotes the same value denoted by an assignment statement. See section 3.1 for further details.

Chapter 2:

The Simple Applicative Subset of PAL

(This section was last modified on 02/04/68 at 03:20 by Evans.)

The simple applicative subset of PAL consists of those aspects of the language which permit new values to be expressed in terms of (perhaps complex) operations on existing values. 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 in this chapter 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 till the next chapter a discussion of assignment statements themselves.

In the first three sections of this chapter, we will discuss the application of a function to arguments. First we will consider 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*. (Assume for the moment that *Add* denotes a function that does addition.) Next we will discuss infix operations that the programmer may write. (An *infix operator* is one that appears between its arguments, rather than before them as did *Add* in the previous example. The latter case is sometimes referred to as *prefix*.) The programmer may write

2 + 3

instead of the previous expression. Clearly, writing

2 + 3 * 4

is much more convenient than writing

Add(2, Mult(3, 4))

but there are problems with the simpler notation. It is only by convention that the example is interpreted as shown and not as

Mult(Add(2,3), 4)

In other words, the interpretation of the prefix form is unambiguous, but the infix form permits ambiguity unless suitable rules are supplied. These rules are given in section 2.3.

The examples given above use integers as the arguments to the functions. In many of the later examples given in this chapter, expressions such as

$$x + y$$

will be used. Clearly, such expressions are meaningful only if (a) _x_ and _y_ have values, and (b) the values are such that "+" can be applied to them (i.e., the values are numbers). We will assume in the examples given in this chapter that similar such conditions hold. In sections 2.6 and 2.7 we will see how variables "are created", with the final details being supplied in chapter 4. In section 3.2 we see how a variable, once created, may have its value changed.

<u>Introduction</u>: (This section was last modified on 02/04/68 at 03:30 by Evans.)

In this section, we are concerned with the application of a function to arguments. In the simplest case, both the function and its argument are given. For example, the expression

```
Sqrt 8
```

denotes the result of applying <u>Sqrt</u> to <u>8</u>. Either the function or the argument may be the result of arbitrarily complex calculations. Thus the expression

```
(x y) (z w)
```

denotes the result of applying (the result of applying <u>x</u> to <u>y</u>) to (the result of applying <u>z</u> to <u>w</u>).

Note that PAL differs from conventional notation in not requiring parentheses to surround arguments: Juxtaposition is adequate to indicate application. Thus we permit either

```
x y
```

or

```
x(y)
```

rather than require the latter.

Formulae:

```
<combination>  ::=  <expression> <expression>
      | <combination> <expression>
```

Notes:

The intent of the syntax is to show that functional application associates to the left. That is, the expression

x y z

is to be interpreted as

(x y) z

rather than as

x (y z)

As usual, parentheses may be used to indicate any grouping that the programmer wishes.

Semantics:

We will use the terms rator and rand to stand for the function and its argument, respectively. (The terms are short for "operator" and "operand".) The meaning of a function evaluation is to perform the following:

- Evaluate both the rand and the rator.

- Apply the (value of) the rator to the (value of) the rand.

The value of the functional application is the value thus produced. The order of evaluation of the rand and the rator is not specified in this manual: The user should not make any assumptions based on the order of evaluation of the rand and the rator.

In many cases, a function needs more than one argument. For example, Add needs two numbers. By convention in PAL, all functions have exactly one argument. Thus in applications such as

Add(2, 3)

it is to be understood that Add is applied to the 2-tuple denoted by 2,3. It should not be inferred that the rand of Add must be an explicit 2-tuple. All that is necessary is that the rand be any expression which denotes a 2-tuple of integers.

Part of the definition of a function is the specification of the number of components of its argument. When the function is applied, the argument must have this same number of components or the effect of the application is undefined. This condition is called the principle of conformality, and it is discussed in further detail in appendix 1.2.

If the function being applied is one defined by the programmer, the semantics of the application are found in section 2.6.

Introduction: (This section was last modified on 02/04/68 at 10:17 by Evans.)

Just as a pair is an ordered collection of two objects and a triple is an ordered collection of three objects, so we use the term *n-tuple* for an ordered collection of *n* objects. Thus 2-tuple is another word for pair. We will frequently use the word *tuple* to stand for an n-tuple of unspecified size.

The programmer's usual way to write a tuple is with commas, so that

11, 12, 13, 14

denotes that 4-tuple whose elements are as shown. The elements of a tuple may be accessed by applying the tuple to an integer. Thus if *S* denotes the 4-tuple just given, then

S 2

denotes *12*. In other words, a tuple acts as a function over positive integers.

The tuple is the only construct available to the PAL programmer to specify objects with structure. Since an element of a tuple may itself be a tuple, it should be clear that arbitrarily complex objects can be specified.

We will use phrases such as "a tuple of numbers" or "a number tuple" to refer to a tuple each of whose elements is a number. Clearly, a number tuple bears strong resemblance to a one-dimensional array in languages such as Fortran. Similarly, an m-tuple of n-tuples of numbers is similar to an _m_ by _n_ array.

Formulae:

<0-tuple> ::= nil

<big tuple> ::= <expression> { , <expression> }$_1^\infty$

<tuple> ::= <0-tuple> | <big tuple>

Notes:

A tuple is an ordered collection of zero or more items, each of which can be any object at all. The 0-tuple is denoted by the word nil. The comma may well be thought of as an infixed, non-associative tuple-maker. There is no simple representation for a 1-tuple.

It should be clear to the reader that it is the comma rather than the parentheses that indicates a tuple. The parentheses are needed only if PAL's grammatical rules would otherwise indicate an alternate grouping. The fact that comma is non-associative (as suggested by the last sentence of the preceeding paragraph) is quite important. Note that each of

1, 2, 3

and

(1, 2), 3

and

1, (2, 3)

each denotes a different object. The first denotes a 3-tuple, and the next two each denote a (different) 2-tuple. If comma were associative, one or the other of the last two would denote the same object as the first.

The tuple-making functor aug, mentioned in section 1.2, is an infixed operator whose left operand may be a tuple and whose right operand may be any expression. Let *T* be any n-tuple and *E* be any expression. Then

T aug E

denotes that *n+1*-tuple whose first *n* elements are the same as the elements of *T* (although they do not share) and whose *n+1*-st element is the value denoted by *E*. (See appendix 1.2 for further discussion of sharing.)

There is an important distinction between a 1-tuple and the object that is the single element of the 1-tuple: The former can be applied to *1* to get the latter. Probably the most convenient way to indicate a 1-tuple whose value is that denoted by *E* is by the expression

nil aug E

Note that

nil , E

would not do, since this latter denotes a 2-tuple whose first element is *nil* and whose second element is *E*.

Semantics:

A tuple denotes a bundle of information. Here the "bundle" consists of the _n_ elements of the tuple, in order. There are two essentially different ways such a bundle may be used: As an object manipulatable by the programmer, it may be assigned to a variable or used anywhere in an expression where a tuple is permitted. In addition, there is another way to use tuples. Clearly, it must be possible to extract the elements of a tuple from the tuple. This is done by applying the tuple (as a function) to an argument whose value is the integer _k_, thus yielding the _k_-th element of the tuple. The application is undefined, of course, if it is not the case that

$$1 \leq k \leq n$$

where _n_ is the order (i.e., the number of elements) of the tuple.

In section 5.1 a notation is given applying a path to a tuple. Here it is assumed that the tuple's elements are themselves tuples, etc., and the path is a selector that picks its way through the structure. In section 5.2 the idea of lists is introduced. A list is either _nil_ or a 2-tuple whose second element is a list.

Note carefully the effect of sharing. If, for example, _T_ denotes the tuple denoted by

a, b, c

then *a* shares with *T 1*, *b* with *T 2* and *c* with *T 3*.

Examples:

Suppose that

r1 denotes nil aug 1

r2 denotes 1, 2

r3 denotes 1, 2, 3

r4 denotes 1, 2, 3, 4

Thus we have four tuples, each of whose elements is an integer. Then the expression

r4 3

denotes the value 3. Here the tuple r4 is being applied to the integer 3 to produce the third element of the tuple. Similarly, then, the expression

r3 (r4 2)

denotes the value 2. (r4 is applied to 2, yielding 2, and r3 is applied to that.) Now suppose that s denotes the object denoted by

r1, r2, r3, r4

The 4-tuple denoted by s is something like a lcwer-trianguular array. Ignoring sharing, the value denoted by s could be written as

(nil aug 1), (1, 2), (1, 2, 3), (1, 2, 3, 4)

The result of applying s to an integer will be a tuple, so that the expression

s 3 2

denotes the second element of the third element of s. That is, it will be the second element of r3, or 2. (Remember that the rules of PAL require that the preceeding expression be interpreted as

(s 3) 2

rather than as

s (3 2)

which would have no meaning, since an integer cannot be applied.)

For the final example, suppose that Add, Sub, Mult and Div denote functions that do addition, subtraction, multiplication and division, respectively. Suppose further that x denotes the value denoted by

Add, Sub, Mult, Div

Thus x denotes a 4-tuple each of whose elements is a function. Then the expression

x 2 (2, 3)

denotes the value -1. (x is applied to 2, yielding Sub, and 2 minus 3 is -1.) Now suppose that y denotes that value denoted by the expression

(1, 2), (3, 4), (5, 6)

That is, y denotes a 3-tuple each of whose elements is a 2-tuple. Then the expression

x (y 2 1) (y 3)

denotes the value 30. (Evaluating y 2 1 involves applying y to 2 and the result to 1. Application of y to 2 yields the 2-tuple 3,4, and applying that to 1 yields 3. x then is applied to 3, yielding Mult. The rand of Mult is y 3, or 5,6, and 5 times 6 is 30.)

Introduction: (This section was last modified on 02/04/68 at 17:18 by Evans.)

In section 2.1, we saw that we can indicate the application of a function to arguments in the usual mathematical way: f(x) denotes the application of f to x. Here the rator (function) is written before the rand (argument), so this form is called prefix form. Conventional mathematics also permits infix operators, such as "+" or "/", which are written between their operands; and so PAL permits infix forms also. In this section we discuss such infix operators. Also discussed are certain prefix operators.

PAL uses the term functor to refer to the infix and prefix operators that are part of the language.

Formulae:

```
<expression>  ::=
       <identifier>
     | <expression> <infix functor> <expression>
     | <prefix functor> <expression>
     | <expression> % <variable> <expression>
     | ( <expression> )
```

Notes:

1. It is not convenient to extract from the complete syntax of PAL exactly that part which pertains to infix functors, so the syntax given on the preceeding page is only an approximation. Functional application (see section 2.1) and conditional expressions (see section 2.5) interact strongly, and in a real sense everything else in PAL interacts weakly. Nonetheless, the present section attempts to explain the use of infix functors. To the extent that they are used in constructions not involving other syntactic forms, the present discussion is correct. (The complete syntax of PAL is given in appendix 2.)

2. The classes <prefix functor> and <infix functor> used in the syntax are defined in section 1.2.

3. Since the syntax given above for <expression>s is ambiguous, we must provide additional information to specify whether the construction

```
x + y * z
```

is to be treated as

```
x + (y * z)
```

or as

```
(x + y) * z
```

To deduce the meaning of such ambiguous phrases, note the following precedence table:

Functor	Precedence
%	5
aug	7
|	10
&	15
not	17
=	20
>	20
<	20
+	25
-	25
*	30
/	30
**	35
$	40

Here each infix functor is associated with a numerical value, called its precedence. In cases where a particular subexpression may be associated with the functor on either its left or its right, it will be associated with that one which has higher precedence. In cases of equal precedence, the functor on the left will be used. "%" is special in that it is ternary functor (i.e., it has three operands). The above rule may be followed if the "%" and the variable immediately to its right are taken as a single functor, with precedence 5.

In the ambiguous example shown above, the y can go with either the "+" on its left or the "*" on its right. Since the precedence of times (30) is higher than that of plus (25), y is seen to be an operand of times, and the first interpretation shown above is used. (This is fortunate, since that interpretation is the "usual" one. Of course, it is not just fortuitous.)

Careful attention to the rule given for the interpretation of equal precedence is particularly important in situations involving multiplication and division. In choosing an interpretation of

x / y / z

we are concerned with y. Since the functor on its left has the same precedence as that on its right, our rule dictates associating y with the functor on its left. Thus the preceeding expression is interpreted as if it were written

(x / y) / z

which is mathematically equivalent to

x / (y * z)

Of course, the programmer may indicate any grouping he likes by judicious placement of parentheses.

4. Expressions such as

```
f x + y
```

are also ambiguous but are not covered by the previous rule. A convenient way to think about the interaction of functional application and infix functors is to assume that there is an (invisible) functor between the rator and the rand which indicates functional application. Then let the invisible functor have precedence 45, and the previous rule holds. In the present example, the functor "functional application" to the left of _x_ has precedence 45, higher than the precedence of "+", so that _x_ is the operand of _f_ (i.e., the right operand of "functional application") rather than the left operand of "+". Thus the example is equivalent to

```
(f x) + y
```

rather than to

```
f (x + y)
```

This rule covers not only the interaction between functional application and infix functors, but also cases such as

```
f x y
```

As explained in section 2.1/N, functional application associates to the left, so this example is equivalent to

```
(f x) y
```

Semantics:

An expression involving infix or prefix functors denotes a value, and this section describes how that value may be deduced. Proceed as follows:

Step 1: Associate with each infix functor its left and right operands, using the rules of the previous part of this section. Similarly, associate with each prefix functor its operand, which is the expression immediately to its right.

Step 2: If the entire expression has been evaluated, then quit; otherwise, continue with either step 3 or step 4.

Step 3: Replace an identifier by the value which it denotes. Continue at step 2.

Step 4: Select an infix functor both of whose operands have been evaluated, or a prefix functor whose operand has been evaluated, and replace the functor and operand(s) by that value which is the result of applying the functor to the operand(s). Continue at step 2.

Several points should be noted.

1. No order of evaluation is to be inferred from this description, in in that step 2 may be followed by either step 3 or step 4. The programmer is cautioned not to write a program whose successful evaluation depends on a particular order of evaluation of expressions. (Of course, sequences are executed in the order written, and the programmer may be quite confidant

about certain aspects of the order of evaluation of conditionals, etc.)

2. The evaluation of step 3 is described in section 1.4/S. In general, the value denoted by a variable depends on the program of which it is a part.

3. Step 4 involves the semantics of the various functors of PAL. These are explained in section 1.2/S.

Examples:

Each example gives, on successive lines, a PAL expression involving infix functors and an equivalent PAL expression without infix functors. For the purpose of these examples, assume that the functions Add, Sub, Mult and Div correspond to the functors "+", "-", "*" and "/", respectively. (That is, "Add(x, y)" denotes the same value as does "x + y", etc.)

1. An arithmetic expression:

```
x + y * (z - w) / t

Add(x, Div(Mult(y, Sub(z, w)), t))
```

Note how parentheses are used in the first line to override the interpretation PAL would otherwise give.

2. Another one:

```
(-b + Sqrt(b*b - 4*a*c) ) / (2*a)

Div( Add(Negate b, Sqrt(Sub(Mult(b, b),
   Mult(Mult(4, a), c)))), Mult(2, a))
```

Here Negate is the function that does unary minus. This example should convince anyone not previously convinced that infix form has advantages for people.

Advice:

The alert programmer who is well versed on the precedence values of the operators can often reduce the use of parentheses in complex expressions to a minimum. While there is nothing wrong with doing so, it should be realized that the insertion of redundant parentheses in PAL expressions never affects the computation. Frequently, inserting such redendant parentheses will improve considerably the readability of the resulting text, with at least three adavantages:

- It is less likely that the programmer will make mistakes. The alert programmer referred to in the first sentence may happen to remember incorrectly the relative precedence of, say, "&" and "|".

- The programmer will find it easier later to modify what he has written, since he is more easily able to tell later what he originally intended. Since in the nature of things every program will be modified several times during its lifetime, this is an important consideration.

- Anyone else reading the program will be able to tell more easily what was intended. Since most PAL programs will be turned in as homework, making them easier to read by a grader has definite advantages.

The programmer should take these points seriously, as they represent a philosophy of programming practice that has been found, over the years, to pay off.

Introduction: (This section was last modified on 02/04/68 at 10:58 by Evans.)

The designers of PAL have provided for the user certain "built in" functions that are available in PAL with no special effort on the user's part. For the most part, these are functions that the user could not himself write in PAL. (Some of them could be so written, but only awkwardly.) These functions are listed and described in this section.

The functions that are built in to PAL are now listed. They fall into four categories: type-checking predicates, string manipulating functions, type conversion functions and miscellaneous.

Type-checking Predicates:

Each of the following functions is a predicate whose domain is all objects. Its value will be *true* if (and only if) it is applied to an object of the type that is part of its name.

```
isboolean

isstring

isfunction

isprogramclosure

islabel

istuple

isreal

isinteger
```

If *x* denotes any object whatsoever, then the expression

```
        (isboolean x) | (isstring x) | (isfunction x)
   | (isprogramclosure x) | (islabel x) | (istuple x)
                 | (isreal x) | (isinteger x)
```

is always defined and is always true. (Remember that "|" is the infix logical functor "or".) In other words, if we consider for each predicate shown the set of objects for which the predicate is *true*, then the union of these sets is all possible objects.

Further, there is no object that is in more than one of these sets.

String-manipulating Functions

The following three functions are used for manipulating strings. For further details about these functions (including the effect of their application to improper arguments), see Appendix 3.3.

Stem This function is to be applied to a string of length one or more. The value of the function is the first character of the string.

Stern This function is to be applied to a string of length one or more. The value of the function is that same string with its first character removed.

Conc This function is to be applied to two strings, of any lengths. Its value is that string obtained by concatenating the second string to the end of the first.

Type Conversion Functions

For further details about these functions, see Appendix 3.3.

ItoR This function, applied to an integer, returns a real number with the same value.

RtoI This function, applied to a real, returns the largest possible integer which is less than equal to the argument.

Stol If this function is applied to a string each of whose characters is a digit, the value of the function is that integer represented by the string.

Miscellaneous Functions

LookupinJ This function is useful for determining the value associated with a particular identifier, given the name of the identifier as a string. Its use is given in section 5.4.

Order This function, applied to a tuple, returns the number of elements of the tuple at the top level. More precisely, it returns the largest integer k such that the application of the tuple to k is defined. See appendix 3.3.

Print This function may be applied to any object whatsoever. Its effect is to print the value of the object, in a suitable form. See appendix 3.3.

Readch This function may be used to read characters from an input device. See appendix 3.3.

Atom This predicate is applicable to any object. It returns true if its argument is an integer, a real a boolean, or a string. (Note that these are precisely the same objects to which "=" may be applied.)

Null This predicate is applicable to any object. It returns *true* if the argument is the zero-tuple.

Share This function may be applied to any two objects. Its value is *true* if the two objects share.

Introduction: (This section was last modified on 02/01/68 at 21:59 by Evans.)

One of the principle sources of power in computing languages is the ability to express evaluations such that the course of the evaluation is dependent on values previously calculated. The conditional expression, whose value is one of two expressions depending on whether or not a given expression is true, is the linguistic feature available in PAL (as well as in many other programming languages) to accomplish this need. Thus the expression

```
(x > 0) -> x ! (-x)
```

denotes the absolute value of x. The first parenthesized expression denotes either true or false, depending on whether x is or is not greater than zero. If true, the value denoted by the entire expression is that denoted by x -- the expression between the "->" and the "!" -- while if false the value of the entire expression is that denoted by "(-x)" -- the expression to the right of the "!". (Neither set of parentheses is needed in this expression.)

Formulae:

```
<conditional expression>  ::=

     <expression> -> <expression> ! <expression>
```

Notes:

The mark "->" is a composite token in PAL, and must be typed with no space between the "-" and the ">". (See Note 6 in section 1.1/N.)

The conditional expression as shown provides for only a two-way branch. In the case where a many-way branch is needed, the usual technique is that the expression to the right of the "!" also be a conditional expression. An example of this is shown below.

Semantics:

Consider a conditional expression of the form

B -> E1 ! E2

The value denoted by such an expression is determined as follows:

Step 1: The value denoted by *B* is determined. If this value is other than *true* or *false*, the value of the entire conditional expression is undefined.

Step 2: If the value denoted by *B* is *true*, then the value denoted by the conditional expression is the value denoted by *E1*, while otherwise the value denoted by the conditional expression is the value denoted by *E2*.

Several points should be noted:

1. The value denoted by *B* is determined *before* either *E1* or *E2* is evaluated.

2. Only one of *E1* or *E2* is evaluated. The programmer may rest confidant that the expression "selected" by the evaluation of *B* is the only one evaluated.

Examples:

We wish to write an expression dependent on the value of *x* such that the value denoted by the expression is one when *x* is between zero and ten, inclusive, and is zero otherwise. A possible such expression is

```
(x < 0)  ->  0  !  (x > 10)  ->  0  !  1
```

(The parentheses shown are not needed.) An alternate expression with the same value is

```
x < 0  |  x > 10   ->   0   !   1
```

The first example shows a conditional expression with (in effect) a three-way branch.

The expression

```
x = 0  ->  0  !  1/x
```

illustrates the importance of evaluating only that one of the expressions that is "selected", since it would clearly be wrong to evaluate "1/x" in the case that *x* denoted zero.

<u>Introduction</u>: (This section was last modified on 02/04/68 at 11:02 by Evans.)

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

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

and

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

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". An alternate syntactic form for this definition is

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

The character "λ" is not part of the PAL alphabet (it is not listed in section 1.1) so the punctuation "ll" is used instead.

The lambda notation was first proposed by the mathematician Alonzo Church, and it is frequently referred to as Church's lambda calculus. For those so interested, the original description of lambda notation may be found in ?????????? *
Another paper possibly worth consulting is ?????????? *

Formulae:

<lambda expression> ::=

ll <bound variable part> = <expression>

<bound variable part> ::=

<bound variable element>$_{1}^{\infty}$

<bound variable element> ::= <variable> | ()

| (<variable> { , <variable> }$_{0}^{\infty}$)

Semantics:

A lambda expression denotes a function. The bound variable part indicates those names which are to be "substituted for" in the lambda body by the arguments to which the function is applied. Thus

11 x.x+1

denotes a function of one argument. Applying this function to an argument such as *2* implies the substitution of *2* for *x* in the lambda body, resulting in

2 + 1

A function such as that denoted by

11 (x, y) . x + y

must be applied to a 2-tuple of integers (or of reals), so that applying it to

(2, 3)

results in

2 + 3

On the other hand, a function such as that denoted by

11 x. 11 y. x + y

is different. Applying it to *2* produces ll y. 2 + y

and applying this to *3* produces '2 + 3'. That is,

```
(ll x. ll y. x + y) 2 3
            = (ll y. 2 + y) 3
            =   2 + 3
```

The lambda expression just explained may be written alternatively as

ll x y . x+y

This latter (note the syntax) has a bound variable part with two bound variable elements, while the former is a lambda expression whose body happens also to be a lambda expression. The transformational properties of the objects denoted respectively by these two lambda expressions are indistinguishable.

Rather complicated bound variable parts are possible. The function denoted by the lambda expression

ll (x, y) z . E

(where *E* is some expression presumably involving *x*, *y* and *z*) should be applied to a 2-tuple, and the result is a function to be applied to a single argument.

The appearance of empty parentheses "()" in a bound variable part defines a function that may only be applied (in that argument position) to a 0-tuple. Thus

11 () . E

is a lambda expression which, when applied to a zero-tuple, will produce the value E. Applying it to any other value results in a detected run-time error.

Consider now all variables that occur in the <expression> part of a lambda expression. Those that are listed in the <bound variable part> are called the bound variables of the lambda expression, and any other variables appearing in the <expression> are free variables of the lambda expression.

We consider now the semantics of the application to arguments of a lambda expression. The effect is as if the following steps were performed:

Step 1: The actual arguments to which the lambda expression is being applied are compared with the <bound variable part> of the lambda expression, to insure that the principle of conformality is met. (This concept is explained in appendix 1.2.) If it is not met, the effect of the application is not defined.

Step 2: Each bound variable is associated with the actual argument which occupies the same position in the argument list. (The success of step 1 insures that this can be done.)

Step 3: The <expression> is evaluated. Each bound variable in the <expression> is treated as if it had been replaced by its associated actual parameter, the association being that described in step 2. Free variables are evaluated as usual.

Step 4: The value determined in step 3 is the value of the application.

Introduction: (This section was last modified on 02/04/68 at 13:34 by Evans.)

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 x came to denote an integer. PAL provides two syntactic forms to permit the user to "create" new variables: the let expression, defined in this section, and the where expression, defined in the next section. Each of these uses the idea of a definition, as defined in Chapter 4.

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 determined to be 22.

Formulae:

```
<let expression>  ::=  let <definition> in <expression>
```

Notes:

The class <definition> used in the syntax is defined in Chapter 4. Section 4.0 contains a summary of its definition, and the rest of chapter 4 contains the details needed. In the simplest case, a <definition> is of the form

N1, N2, ... , Nk = Exp

where *k*, the number of variables on the left, may be one or more. (Chapter 4 explains in detail how the various syntactic forms possible for definitions may be recast into this form.)

Semantics:

An expression of the form

let N1, N2, ..., Nk = Exp1 in Exp2

is evaluated as if it had been written

[ll (N1, N2, ..., Nk) . Exp2] Exp1

The semantics of such an application of a lambda expression to arguments is explained in section 2.6/S. Briefly, the effect is as if the *Ni* are created with the indicated initial values and then *Exp2* is evaluated. On completion of the evaluation, the *Ni* just created go out of existence. (More precisely, they revert to what ever value -- if any -- they had before the indicated expression was encountered.) In other words, the "scope of the definition" of the variables *Ni* is confined to the expression *Exp2*.

Examples:

Consider the expression

```
let x = 5 in [ (x+1)*(x-1) ]
```

There are two ways to explain the evaluation of such an expression. Taking the less formal way first, we have

1. A new variable x is created denoting 5. The expression is square brackets is then evaluated.

2. The expression shown is replaced by the lambda expression

   ```
   [ ll x. (x+1)*(x-1) ] 5
   ```

   which is then evaluated.

The reader should be sure to understand the second explanation, reviewing section 2.6/S as much as necessary.

Consider now the expression shown in 2.7/I, here rewritten with added parentheses:

```
let    [ (x=1) and (y=2) ]
in
{     let z = (3*x + 4*y)
      in
            x * y * z
}
```

The expression in braces is evaluated with _x_ and _y_ denoting _1_ and _2_, respectively. The first part of the evaluation is to create _z_ denoting _11_, and the value denoted by the entire expression is _22_. The reader should take the trouble to satisfy himself that this expression is equivalent to

```
[11(x, y). (11 z. x*y*z) (3*x + 4*y)]  (1, 2)
```

All of the parentheses and brackets shown here are needed.

For the final example, consider the expression

```
let x = 2 in
let y = 3 in
     x + y
   + (let y = x + 10 in x + y)
   + (let x = 2 *  y in x * y)
   + x + y
```

This expression denotes

```
2 + 3 + (2 + 13) + (6 * 3) + 2 + 3
```

or _43_. Note carefully the scope of the definitions in the two parenthesized _let_-expressions.

Introduction: (This section was last modified on 02/04/68 at 13:47 by Evans.)

As suggested in section 2.7/I, there are two syntactic devices available in PAL for defining variables: the let expression and the where expression. The former is described in section 2.7 and the latter in this section.

Formulae:

```
<where expression>  ::=  <expression> where <definition>
```

Notes:

The class <definition> referred to in the syntax is defined in chapter 4. Section 4.0 contains a summary of its syntax, and the rest of the chapter contains the necessary details.

The syntax shown is inadequate, since it does not provide enough information for parsing. In particular, consider

let Def1 in Exp where Def2

The syntax does not specify whether this expression is equivalent to

(let Def1 in Exp) where Def2

or to

let Def1 in (Exp where Def2)

In fact, the latter parsing is correct. A safe rule to remember is this:

> The effect of a let extends as far to the right as possible.
>
> The effect of a where extends as little to the left as possible.

The reader should satisfy himself that this parsing issue is important. Suppose that the variable a denotes 2. Then the expression

```
(let x = a in x + a) where a = 1
```

denotes _2_, since the definition following the _where_ is in effect during the evaluation of the entire expression in parentheses. (The outer definition of _a_ is hidden.) _x_ is defined to denote _1_ also, and the sum is _1+1_. On the other hand, the expression

```
let x = a in (x + a where a = 1)
```

denotes _3_, since here the effect of the definition after the _where_ is limited to the parentheses. Thus _x_ is defined to denote the outside _a_, or _2_, and the sum is _2+1_. Removing the parentheses from each of these two expressions would cause the first to be the same as the second and would leave the second unchanged.

Chapter 3:

The Imperative Subset of PAL

(This section was last modified on 02/04/68 at 02:26 by Evans.)

In chapter 2, we were concerned with those aspects of PAL in which certain values are expressed in terms of other values. The only calculating tool available was functional application.

In the present chapter, we will introduce into PAL certain additional linguistic constructs which the user has seen in other languages, such as Fortran or PL/I. In section 3.1, we examine how such constructs may be executed sequentially, just as statements are executed sequentially in other languages. Then in section 3.2, we will consider PAL's assignment statements, the linguistic construct that permits the value denoted by a variable to be changed. In section 3.3, we will examine transfer of control ("goto" statements) and the related topic of labels. Finally in section 3.4 we consider sequences involving conditionals.

Introduction: (This section was last modified on 02/04/68 at 17:46 by Evans.)

In languages such as Fortran or PL/I, there is the notion of the sequential execution of one statement after another. Similarly in PAL, sequential execution is an important idea. The term "sequence" means a set of actions to be executed (in general) in the order written by the programmer. Section 3.3 shows how this order may be altered by the programmer with the "goto" statement. The present section discusses sequences.

Formulae:

<sequence> ::= <sequence element> $\{$; <sequence element> $\}_{0}^{\infty}$

<sequence element> ::=

<assignment statement>

| <goto statement>

| <conditional sequence element>

| <expression>

| (<sequence>)

| <label> <sequence element>

| dummy

<label> ::= <variable> :

Notes:

Assignment statements are discussed in section 3.2, goto statements and labels in section 3.3, and conditional statements in section 3.4.

The literal dummy may stand alone as a sequence element, denoting the same value as does an an assignment statement. Its purpose is similar to that of the CONTINUE statement in Fortran, providing a place not otherwise available to place a label.

The syntax shown is ambiguous, since the class <expression> includes <sequence>s. Nonetheless, it is suggestive of the truth of the situation.

Note that according to the syntax a label may be placed on any element of a sequence. This fact is true, although the syntax shown slightly misrepresents PAL's actual syntax.

Introduction: (This section was last modified on 02/04/68 at 02:32 by Evans.)

In PAL as in most conventional programming languages, the purpose of an assignment statement is to make it possible to alter the value denoted by a variable. Thus for example, executing

```
x  :=  E
```

causes *x* to denote the value denoted by *E*, regardless of what value *x* denoted previously. Indeed, *x* may previously have denoted a string and *E* a label, or any other types.

A second example of assignment is

```
f x  :=  1
```

Here the result of applying *f* to *x* is (roughly) the location into which to store the value *1*. For example, suppose that *f* denotes a 3-tuple and that *x* denotes *2*. Then the effect of the above assignment statement is to assign the value *1* to the second component of *f*.

A third type of assignment is illustrated by

```
(x > y  ->  x !  y)  :=  0
```

Here the effect is that whichever of *x* or *y* denotes the larger value is to be assigned the value *0*.

A final type of assignment is illustrated by

```
x, y, z  :=  4, 5, 6
```

The intent here is the simultaneous assignment of the three values on the right to the three variables on the left.

Formulae:

<assignment statement> ::=

<left element> $\{ , \text{<left element>} \}_0^\infty$:= <expression>

<left element> ::=

<variable> | <combination> | <conditional expression>

Notes:

Strictly speaking, the syntax of assignment statements is

<expression> := <expression>

since statements with this syntax are accepted by the PAL translator. This syntax admits statements such as

1 := 2

or

1 + 2 := 8

The effect of either of these statements is nugatory: In both cases, a new cell is created which does not share with anything else. (In the first case, this cell will contain 1 and in the second case it will contain 3.) Then the contents of that cell will be altered. That is, there will be no detectable effect from the execution of either of these statements. Similarly, executing a statement such as

Sqrt 8 := 55

(where "Sqrt" is the square root function) also is nugatory. A new cell is created to hold the square root of eight, and this cell is then updated to hold 55.

Semantics:

If the left side of an assignment statement is a variable, the intent is that the value denoted by that variable is to be changed. If the left side is a combination, the result of its evaluation is taken to designate a location and the contents of that location is to be changed. Similarly, a conditional on the left side is evaluated to select the location to be changed. If the left side is a tuple, the value denoted by the expression on the right must be a tuple of the same length. The effect is the simultaneous assignment of the values on the right to the elements named on the left. It is important to realize that "simultaneous" is the key word in the preceeding sentence. The next part of this section contains several examples to emphasize this point.

The effect of sharing becomes of interest solely in the assignment statement. If variables _x_ and _y_ share, the effect of assigning to either of them is to change the value denoted by both of them. This subject is discussed further in the Examples and Advice parts of this section. Sharing is discussed at length in appendix 1.5.

Examples:

The effect of the statement

```
x, y  :=  3, 4
```

is the same as that of the two statements

```
x := 3;  y := 4
```

in that in each case *x* is set to *3* and *y* is set to *4*. However, the assignment statement

```
x, y  :=  y, x
```

is more complex, since it is the intent that both assignments be done simultaneously. In other words, the values denoted by *x* and *y* are interchanged.

Now suppose that *i* denotes *2* and that *f* denotes a 5-tuple. Then executing

```
i, f i  :=  i + 1, 37
```

changes the second element of *f* to *37* and changes *i* to *3*. Since the two assignments are simultaneous, it is the "old" value of *i* that is used on the left.

Assume that the following examples have been preceeded by suitable definitions such that *a* denotes *1*, *b* denotes *2* and *c* denotes *3*. Assume further that x denotes 1, 2, 3

y denote the object denoted by a, b, c

z denotes 4, 5, 6

Then the execution of

```
x 1  :=  4
```

will result in *x* denoting the value

4, 2, 3

Here the left side is a combination which denotes the first element of the 3-tuple *x*, and it is this element that is changed. Executing

```
a, b, c  :=  7, 8, 9
```

will cause new values to be assigned to *a*, *b* and *c*, so that the value denoted by *y* will be

7, 8, 9

This is because the elements of *y* share with *a*, *b* and *c*. Note however that executing

```
y  :=  11, 12, 13
```

will have no effect on *a*, *b* or *c* but will change only *y*. Finally, executing

```
z 1, z 3  :=  z 3, z 1
```

will result in $\underline{z}$ denoting the 3-tuple

6, 5, 4

Advice:

One of the most common errors in PAL is to neglect the effect of sharing. Consider the following:

```
let i = 1 in
let t = 1, 2, i, 4 in
Print t;
i := 4;
Print t
```

The effect of the first "Print" is to print

(1, 2, 1, 4)

and that of the second is

(1, 2, 4, 4)

since i shares with t 3.

Introduction: (This section was last modified on 02/04/68 at 02:34 by Evans.)

The concept of executing one statement after another is an important one in PAL, just as it is in other languages. As in many languages, execution normally proceeds from one statement to the next in the order written, but this order may be altered by the programmer as he desires. Two notions are needed: the "goto" statement, execution of which effects "transfer of control"; and the label, which marks a point in the program as a possible candidate for being the "target" of a goto statement.

Formulae:

```
<goto statement>  ::=  goto <expression>

<label>  ::=  <variable> :
```

Notes:

The syntax on the preceeding page is not particularly helpful, in that it gives no clue as to where in a PAL program a label may be placed. This topic will be discussed in more detail in section 3.1 above, on sequences. For the purposes of the present discussion, it suffices to say that labels may be placed in a program at points to which control may meaningfully be transferred. For a complete discussion, see appendix 2.

In a goto statement, the expression must be one which denotes a label.

Semantics:

The effect of labelling a statement is to define a new variable whose name is that of the label and which denotes "that point in the program where the label stands". Such a value may (in the last analysis) only be used as the "operand" of a goto, although it may be assigned, passed as an argument, incorporated into a tuple, etc., beforehand. The scope of a label is the smallest enclosing block in which it exists. (See appendix 3.1 for a discussion of scope.)

The effect of executing a goto statement is to "transfer control" to that statement labeled by the value of the <expression>.

Note that the identifier used as a label is a variable and not a constant, so that it possible to update it with an assignment. An example below illustrates this point.

For a complete description of the scope of a label, see Appendix 1.3. For a description of goto's in terms of the operator ii, see section 5.4.

Examples:

Consider the following segment of PAL code:

```
L:   Print 1;
     goto M;
M:   Print 2;
     M := N;
     goto L;
N:   Print 3
```

The effect of executing this segment will be to print "1213". Note that M is a variable (not a constant) that initially is associated with the "Print 2" statement but which later (as a result of assignment) becomes associated with the "Print 3" statement.

Introduction: (This section was last modified on 02/04/68 at 02:35 by Evans.)

Just as the course of evaluation of an expression may be made dependent on the truth value of some proposition, so also may the sequential execution of a sequence be made dependent on a proposition.

Formulae:

```
<conditional sequence element>  ::=                        element
       <expression> -> <sequence element> ! <sequence>
                                                      ^
```

Semantics:

A conditional sequence element is evaluated by evaluating first the <expression>. If it denotes *true*, the value is that denoted by the element between the "->" and the "!"; if it denotes *false*, the value is that denoted by the element to the right of the "!"; and otherwise the value is undefined.

(This section was last modified on 02/04/68 at 14:12 by Evans.)

In sections 2.7 and 2.8 the idea of definitions was mentioned, in connection with let-expressions and where-expressions. In this chapter we describe such *definitions*, through the use of which the programmer may introduce names of his own choosing into his program, at the same time associating "initial values" with those names. Section 4.2 discusses function form definitions, which permit the programmer to define his own functions. Section 4.3 is on simultaneous definitions, section 4.4 on recursive definitions, and section 4.5 on *within* definitions.

For convenience, the complete syntax of <definition>s is now given:

<definition> ::=

<simple definition> $\{$ and <simple definition> $\}_0^\infty$

| rec <simple definition>

| <simple definition> within <definition>

<simple definition> ::=

<variable> $\{$, <variable> $\}_0^\infty$ = <expression>

| <variable> <bound variable part> = <expression>

| (<definition>)

Introduction: (This section was last modified on 02/04/68 at 14:17 by Evans.)

The simplest possible type of definition is of the form

Var = Exp

where *Var* is a variable and *Exp* is any expression. The effect is the creation of *Var*, initialized to denote the value denoted by *Exp*. A more complex definition is one of the form

N1, N2, ..., Nk = Expression

In this case *Expression* must denote a *k*-tuple, and each of the *Ni* is associated with the corresponding element of that *k*-tuple. This section contains a discussion of both types of definition.

Formulae:

<simple definition> ::=

$\langle\text{variable}\rangle \; \{ \; , \; \langle\text{variable}\rangle \; \}_0^\infty \; = \; \langle\text{expression}\rangle$

Semantics:

The evaluation of a definition involves the creation of one or more new variables, initialized to denote certain values. A <simple definition> as described in the syntax part of this section is processed as follows:

Step 1: Evaluate the expression to the right of the equal sign.

Step 2: Let _k_ be the number of names to the left of the equal sign. If _k_ is one, go to step 5 while otherwise continue at step 3.

Step 3: The value determined in step 1 must be a _k_-tuple. If not, the effect of the definition is undefined.

Step 4: Each of the _k_ names to the left of the equal sign is created as a new variable, initialized to denote the value of the corresponding component of the _k_-tuple denoted by the right side. The new names share with the corresponding component. The processing of the definition is complete.

Step 5: The name to the left of the equal sign is created as a new variable, initialized to denote the value denoted by the right side. The new name shares with the right side.

It follows from this description that the entire right side is evaluated before any new variables are created. Thus it is not possible for the expression on the right to refer to any of the variables being defined. (Section 4.4 discusses recursive definitions, the facility that permits circumventing this restriction.)

Examples:

The effect of the definition

```
x = 3
```

is the creation of the new variable *x* initialized to denote *3*. Because *3* is a constant, *x* does not share with anything.

Suppose that *T* denotes a tuple of order three or more. Then the definition

```
z = T 3
```

creates the new variable *z* which shares with the third component of *T*.

The definition

```
x = x + 1
```

creates a new instance of a variable *x*, initialized to be one greater than the *x* in the block in which this definition appears. (See appendix 1.3 on scope rules.)

Suppose that *U* denotes a 2-tuple. Then the definition

```
x, y = U
```

creates new *x* and *y*, sharing with the first and second components of *U* respectively. The effect is undefined if it is not the case that *U* denotes a 2-tuple.

Introduction: (This section was last modified on 02/04/68 at 14:18 by Evans.)

In section 2.6/1, it was suggested that (mathematically) the definition

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

and

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

are equivalent. Since a lambda-expression is an expression, it is clear that the second line is an acceptable instance of a <definition> in PAL (providing of course that the "λ" is replaced by "ll"). Because the first form is convenient for people, it has also been provided by PAL's designers. Such a definition is called a function form definition.

Formulae:

<function form definition> ::=

<variable> <bound variable part> = <expression>

<bound variable part> ::= <bound variable element>$_{1}^{\infty}$

<bound variable element> ::= <variable> | ()

| (<variable> $\{$, <variable> $\}_{0}^{\infty}$)

Notes:

The definitions of <bound variable part> and <bound variable element> just given are identical to the definitions given in section 2.6 in connection with lambda-expressions. (The definitions are repeated here for convenience.)

Semantics:

A function form definition can most easily be described by showing the simple definition to which it is equivalent. The definition

<variable> <bound variable part> = <expression>

may be replaced by the definition

<variable> = ll <bound variable part> . <expression>

The object to the right of the equal sign is a lambda expression (the syntax of <bound variable part> in this section and in section 2.6/F is identical), so this definition is in the form of a simple definition and explained by the discussion in section 4.1.

The application to arguments of a function defined by a function form definition is explained in section 2.6.

Examples:

The definition

f x = 1 + x

defines a function *f* whose value is one greater than its argument. The bound variable of this definition is *x*, and there are no free variables. The value of "f 4" is determined by replacing *x* by *4* in the expression "1 + x", yielding "1 + 4" or *5*.

The definition

Add x y = x + y

is equivalent to the definition

Add = ll x y.x + y

and therefore to

Add = ll x.(ll y. x + y)

The function denoted by *Add* is to be applied to a number, yielding a function which upon application to a second number yields their sum. For example, having defined *Add* this way, the function *f* defined above could alternatively have been defined as

f = Add 1

A definition with equivalent effect is

```
f y = Add 1 y
```

More examples of function form definitions can be found in appendix 4.

Introduction: (This section was last modified on 02/04/68 at 14:23 by Evans.)

It is frequently convenient to be able to define several variables simultaneously rather than to define them sequentially. For example, suppose it is desired to interchange the roles of *x* and *y* in a block. Either the simultaneous definition

$$x = y \quad \text{and} \quad y = x$$

or the simple definition

$$x, y = y, x$$

could be used, with equivalent effect. In each case, the right side is evaluated completely before new variables *x* and *y* are created.

Formulae:

<simultaneous definition> ::=

<simple definition> $\{$ and <simple definition> $\}_0^\infty$

Semantics:

The effect of a simultaneous definition is the following:

Step 1: Replace each individual definition by an equivalent simple definition as follows: If the definition is a function form definition, replace it as described in section 4.2. If the definition is a recursive definition, replace it as described in section 4.4. If the definition is a *within* definition, replace it as described in section 4.5.

Step 2: The definition is now in the form

N1 = E1 and N2 = E2 and ... and Nk = Ek

where each of the *Ni* is a variable. Now replace it by the form

N1, N2, ... , Nk = E1, E2, ... , Ek

Step 3: The definition is now in the form described in section 4.1, and the explanation of 4.1/S may be followed.

Note that the evaluation is recursive, since some one of the <basic definition>s in one of the alternates may itself be a <definition> in parentheses.

Examples:

Introduction: (This section was last modified on 02/04/68 at 14:33 by Evans.)

In evaluating a definition, the right side is evaluated before the name on the left side is created, so that any instance of that same name which appears on the right is evaluated as it would be in the block in which the definition appears. Equivalently, the body of a definition may not refer to the variable being defined. Although this situation is usually the one desired, there are sometimes cases in which it is important that it be possible to refer to the variable being defined. The usual such case is the recursive function, in which the function being defined is used as part of its own definition. (Many of the syntax equations of this manual are recursive, in that the class being defined is used as part of its own definition.) The usual example given of a recursive function is factorial. In English, we might say

> The factorial of _n_ is defined to be _1_ if _n_ is _0_ and is defined to be _n_ times the factorial of _n-1_, otherwise.

It should be clear that this definition will work if _n_ is a non-negative integer, and that it will "loop forever" if _n_ is a non-integer or is negative.

The naive programmer might attempt to express the above definition in PAL somewhat as follows:

```
Factorial n  =  (n = 0)  ->  1  !  n * Factorial(n - 1)
```

Of course this will not work, since the instance of "Factorial" on the right is *not* the function being defined but some "Factorial" defined in an outer block. (If, as might be expected, there is no "Factorial" in any outer block, the effect of the definition is undefined.) The syntactic device provided in PAL to meet this need is the punctuation *rec*. In a definition such as

```
rec f x  =  E
```

the intent is that any *f* appearing in *E* be the *f* being defined.

Formulae:

<recursive definition> ::= rec <definition>

simple(?)

Semantics:

The effect of a recursive definition is as if the following were performed:

Step 1: The <definition> to the right of the *rec* is replaced by a simple definition.

Step 2: The effect of step 1 is a definition of the form

rec (N1, ... , Nk = E)

(where *k* may be *1*.) The expression *E* is evaluated as if the *Ni* had already been defined by this equation, and the *Ni* are then created and associated with the components of *E* as described in section 4.1.

The discussion of step 2 is not a particularly satisfactory one, since it is not at all clear how the stated effect can be achieved. It is beyond the scope of this manual to explain further the semantics of a recursive definition, since a detailed understanding of the evaluating mechanism of PAL is required. However, it is worth while to observe that the next step in the processing of the form exhibited in step 2 is to replace that form by

N1, ..., Nk = yy [ll(N1, ..., Nk) . E]

Here *yy* can only be thought of as a magic operator that does all of the right things. (*yy* is not available directly to the PAL programmer -- only indirectly as above through the use of *rec*.)

The reader should not feel that this discussion enhances understanding of the semantics of a recursive definition. It is included here only to show how a recursive definition is replaced by a simple definition. This topic is discussed further in appendix 1.4.

Examples:

The factorial function may be defined as

```
rec Factorial n  =  (n = 0) -> 1 ! n*Factorial(n-1)
```

Of course, the equivalent definition

```
rec Factorial  =  ll n.[ (n = 0) -> 1  !  n*Factorial(n-1) ]
```

could also be used.

We give now a function Sum which may be applied to a tuple each of whose components is an integer and which will return the sum of those integers. We might write

```
Sum T  =
        f(1, 0, Order T)
        where rec f(i, s, n) =
                (i > n)  ->  s  !  f(i+1, s + T i, n)
```

The three parameters to f represent the count through the tuple, the sum so far and the order of the tuple, respectively. (The order of a tuple is the number of components it has at the top level. For a more precise definition, see the definition of "Order" in appendix 3.3.) The reader should satisfy himself that "Sum nil" denotes zero, as it presumably should.

Introduction: (This section was last modified on 02/04/68 at 14:48 by Evans.)

It is on occasion desireable that a function be able to maintain within itself a record of values calculated on previous calls. The within clause provides the linguistic facility that meets this need. For example, the function Next defined by

```
          let
                    n = 0 within
               Next ()  =  (n := n + 1;  $ n)
```

is a function of no arguments that returns 1 on its first call, 2 on its second, etc. The unsharing functor "$" protects the n of Next from being updated from outside.

Formulae:

```
<within definition>  ::=  <definition> within <definition>
```

Semantics:

A completely satisfactory explanation of the semantics of a within definition is beyond the scope of this primer, since such an explanation requires reference to PAL's evaluating mechanism. Instead, two alternate explanations are provided: one expressed informally in English prose and the other a precise discussion showing the equivalent lambda expression. The former suffers from being inadequately precise, and the latter suffers from being so excessively precise as to be rather incomprehensible.

An informal approach to within. Consider a definition of the form

a = b within f L = E

When such a definition is processed by the PAL evaluator, the definition "a = b" is processed (in the usual way) but so that the scope of a is limited to E. Then whenever f is applied, the evaluation of E takes place with a properly defined. The value of a is maintained from one call of f to the next.

Upon reentry to the block in which the above definition appears, the definition will again be processed by the PAL evaluator. It therefore follows that such reentry will create a new a.

A more formal approach to within. Consider again the definition given above. It is equivalent to the definition

$$f = (ll\ a.\ ll\ L.\ E)\ b$$

The reader should satisfy himself that, other than by using <u>within</u>, there is no convenient way to write the above definition without using lambda.

Examples:

One example may be found in the introduction to this section.

Consider a definition in which PI is used frequently, and in which the programmer does not want to write the value very many times. Writing

```
PI = 3.1415926536  within  f x = E
```

defines the variable PI with the indicated value and with scope limited to E. (Presumably E is some expression involving, at least, PI and x.)

A more involved example using within is given in appendix 4.2.

Features Dependent on the Current Implementation

The term PAL refers to two different entities: the language defined in this manual and the implementation of that language on a particular computer. To use the latter, it is not adequate that one just understand the former -- additional information is needed. There are two aspects to this "additional information":

- The mechanics of using the PAL implementation must be understood. This includes issues of obtaining access to the computer, preparation of source text, invoking the compiler, etc.

- The language of the implementation differs in some ways from the language as described in the manual.

The first point is not covered at all in this manual. The remainder of this section addresses itself to the second point. This section was last modified on 02/13/68 at 10:20 by Evans.

For convenience in what follows, we use the term PALI to refer to the language as it is actually implemented. A piece of text in PALI, suitable for input to the computer, obeys the following syntax:

<PALI program> ::=

$$\{ \text{ def } \langle\text{definition}\rangle \}_1^\infty \mid \langle\text{expression}\rangle$$

(Here <definition> and <expression> are as described elsewhere in this manual.) The usual way to use PAL is to prepare several PAL programs, all but the last being the definition type. Then the programs are loaded sequentially. The definitions are processed in order, and then the expression does the useful work.

Certain languages issues are relevant here. Because some characters of PAL's alphabet are not available on some consoles, the current implementation permits the user to do without them. The characters at issue are less than, greater than and vertical bar. The following correspondences exist:

<table>
<tr><td>></td><td>gr</td></tr>
<tr><td><</td><td>ls</td></tr>
<tr><td>-></td><td>-*</td></tr>
<tr><td>|</td><td>logor</td></tr>
<tr><td>&</td><td>logand</td></tr>
</table>

(*logand* is not really needed, but it is provided for symmetry.) Whenever a word of the left column is required, the corresponding word in the right column may be substituted. Note that "-gr" may not be used for "->" but that "-*" must be used instead.

A final set of implementation issues has to do with the fact that precisions and ranges are limited.

Integers are limited in magnitude to 2^{35}. That is, any integer *i* in PALl satisfies

$$-2^{35} < i < 2^{35}$$

Any computation in which the result or any partial result is out of this range will produce an incorrect answer. There will be no run time diagnostic.

Reals are approximated by rational quantities of limited precision and limited magnitude. The approximate magnitude limits are $\pm 10^{38}$. Complete details can be found in a reference manual for the IBM 7094. (The representation of PAL's reals is that used by the floating point hardware on the 7094.)

String constants are limited to be no longer than 511 characters. Strings produced during computation are not limited, other than by the limited memory size.

Memory size is a limitation in two different ways: There is a maximum length of PAL program that can be compiled, and there is a limitation on data space available to the running program. Neither limitation should affect 6.231 students doing assigned problems.

(Last modified on 02/12/68 at 14:19 by 4620.)

Library Functions

Introduction

A number of variables in this implementation of PAL have been given pre-defined meanings. This section lists those variables, and gives their semantics.

Some of these pre-defined variables are functions which make certain checks on their arguments. The legal arguments of such functions will be listed here. When these functions are applied to other arguments, they print a message beginning "Run time error: " and call an internal error function with an error argument. The error value for each such function will also be listed here. The internal error function will be described at the end of this section.

Unless otherwise specified, the value of each of these library functions does not share with any other object in the environment.

The pre-defined variables are listed below in alphabetical order, along with the section of this appendix in which they are described.

Atom	3.2.
Conc	1.3.
Cy	9.3.
Isboolean	2.1.
Isinteger	2.2.
Isfunction	2.6.
Islabel	2.8.
Isprogramclosure	2.7.
Isreal	2.3.
Istuple	2.5.
Isstring	2.4.
ItoR	4.2.
LookupinJ	7.1.
Null	3.1.
Order	5.1.
Pr	6.2.
Print	6.1.
Readch	6.3.
RtoI	4.3.
Share	3.3.
Stem	1.1.
Stern	1.2.
StoI	4.1.
Swing	5.2.
SYSTEMERROR	8.1.
Tuple	9.2.
Write	9.1.

Three pre-defined variables are provided for manipulating strings:

1.1. Stem

Legal Arguments: All strings of length greater than zero.
Error Value: ''
Value: That string of length one which contains the first character of the argument.
Examples: The value of Stem 'abc' is 'a'.

The value of Stem '4' is '4'.
The application of Stem to 76 is an error.
The application of Stem to '' is an error.

1.2. Stern

Legal Arguments: All strings of length greater than zero.
Error Value: ''
Value: That string which is obtained by deleting the first character of the argument.
Examples: The value of Stern 'abc' is 'bc'.
The value of Stern '4' is ''
The application of Stern to 76 is an error.
The application of Stern to '' is an error.

1.3. Conc

Legal arguments: Any 2-tuple both of whose elements are strings.
Error Value: ''
Value: That string obtained by concatenating the first component of the argument onto the left end of the second component of the argument.
Examples: The value of Conc('ab','c') is 'abc'.
The value of Conc('4','') is '4'.
The value of Conc('','why?') is 'why?'.
The value of Conc('','') is ''.
The application of Conc to (7,'6') is an error.
The application of Conc to ('a','b','c') is an error.

Because a PAL variable may have as its value an object of any of a large number of data types, eight functions which test the type of their argument have been pre-defined.

2.1. Isboolean

Legal Arguments: Any object.
Value: "true" if and only if the value of the argument is either "true" or "false".

2.2. Isinteger

Legal Arguments: Any object.
Value: "true" if and only if the value of the argument is an integer.

2.3. Isreal

Legal Arguments: Any object.
Value: "true" if and only if the value of the argument is a real number.

2.4. Isstring

Legal Arguments: Any object.
Value: "true" if and only if the value of the argument is a string.

2.5. Istuple

Legal Arguments: Any object.
Value: "true" if and only if the value of the argument is a tuple, including the 0-tuple.

2.6. Isfunction

Legal Arguments: Any object.
Value: "true" if and only if the value of the argument is a closure, a basic function (e.g., these library routines), or the result of evaluating jj.

2.7. Isprogramclosure

Legal Arguments: Any object.
Value: "true" if and only if the value of the argument is a programclosure.

2.8. Islabel

Legal Arguments: Any object.
Value: "true" if and only if the value of the argument is a label.

Three additional functions which always have either "true" or "false" as their value have been pre-defined.

3.1. Null

Legal Arguments: Any object.

Value: "true" if and only if its argument is the 0-tuple.

3.2. Atom

Legal Arguments: Any object.

Value: "true" if and only if its argument is "true", "false", an integer, a real number, or a string. Note that these are the same objects to which "=" can be applied.

3.3. Share

Legal Arguments: Any 2-tuple.

Error Value: false

Value: "true" if and only if the elements of the 2-tuple share.

Three functions are provided for converting data from one type to another.

4.1. StoI

Legal Arguments: Any string of length greater than zero.

Error Value: 0

Value: If the string consists of digits (e.g., '12345'), the value is an integer whose decimal representation consists of those digits (e.g., 12345). If the string contains characters which are not digits, or if the integer represented exceeds the capacity of the machine on which PAL is implemented, the value is an integer which depends upon the implementation.

4.2. ItoR

Legal Arguments: Any integer.

Error Value: 0.0

Value: That real number which has the same "numeric" value as the argument, to within the accuracy obtainable from the implementation's representation of real numbers.

4.3. RtoI

Legal Arguments: All real numbers of magnitude less than or equal to the largest integer which can be represented by the implementation.

Error Value: 0

Value: That integer which has the same "numeric" value as the integral part of the argument.

Two functions are provided for use with tuples.

5.1. Order

Legal Arguments: Any tuple.

Error Value: 0

Value: The largest integer to which the tuple may be applied.

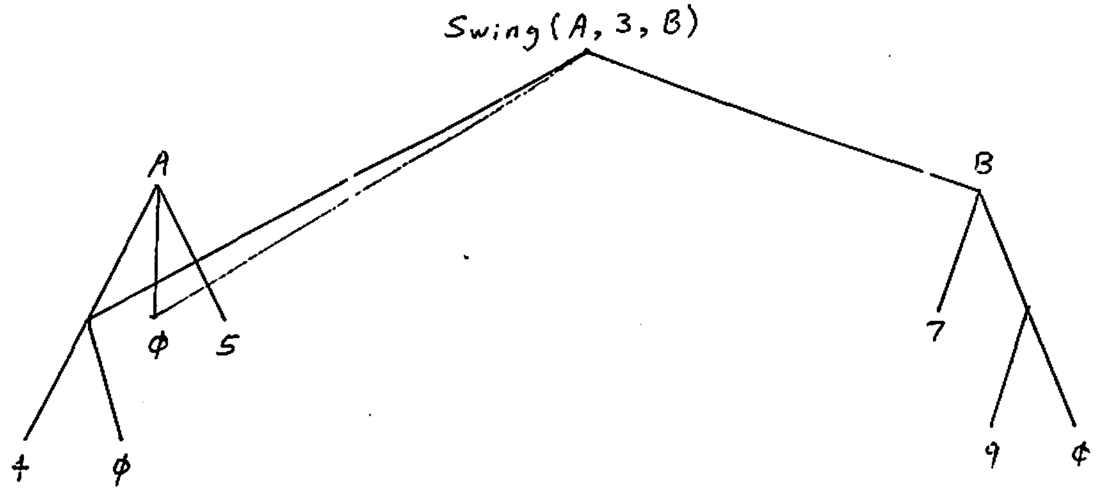
5.2. Swing

Legal Arguments: All 3-tuples satisfying the following conditions: 1. the first component is a tuple; 2. the second component is an integer; 3. the first component may be applied to the second component.

Error Value: nil

Value: Let the first component of the argument be A, the second component of the argument be N, and the third component of the argument be B. The value is a tuple identical to A except that its Nth component is B. Note that, although the value of Swing does not share with any object on the environment, the components of the value of Swing may share with other objects.

Example: The tuples A, B, and Swing(A, 3, B) are diagrammed below.


Swing(A, 3, B)
A
B
φ
5
7
4
φ
9
¢

Three functions for console input-output have been pre-defined.

6.1. Print

Legal Arguments: Any object.

Value: The value of "dummy"

Side Effects: Print has the side effect of printing a representation of its argument on the console. These representations are:

OBJECT	REPRESENTATION
true	true
false	false
integers	one to eleven decimal digits
real numbers	-d.ddddE+dd where d stands for a digit from 0 to 9. The minus sign will be replaced by a blank for positive numbers. The plus sign will be replaced by a minus sign for numbers whose magnitude is less than 1.0.
strings	zero or more characters. The string will not be placed in quotation marks. The application of Print to the string of length zero therefore has no effect.
tuples	a left parenthesis will preceed the first element of the tuple. A right parenthesis will follow the last element. Elements of tuples of length greater than one will be separated by a comma followed by a space. Each element will be printed in a format determined by its type, as described in this section. The application of Print to a re-entrant tuple, that is, a tuple one of whose components is itself, will cause the evaluator to loop. As a special case, the representation of the 0-tuple is

	nil
closure	closure
basic function	basic function
program closure	program closure
the result of evaluating jj	jj
all other objects	$$$

Note that Print does not automatically insert spaces or carriage returns. Thus the sequence:

```
     Print 47;
     Print 19;
     Print '*n'
```

will cause the following line to be printed on the console:

```
     4719
```

Note also that no printing is actually done until a new-line character is printed out. A new-line character is automatically put out in three cases:

1. When the length of a line exceeds 69 characters;
2. Before an error message;
3. At the end of the program, before the "Execution finished" message.

6.2. Pr

Pr is an alternate name for the function Print. It is exactly defined by the PAL program

```
Pr = Print
```

6.3. Readch

Legal Arguments: Any object.

Value: A string of length one, consisting of the next character in the input stream.

Side Effects: When a program begins execution, the input stream is empty. Characters are added to the input stream by typing them at the console. No characters are actually added until a new-line character is typed. Readch removes characters, including the new-line character, from the input stream in the order in which they were typed. If Readch finds the input stream empty, it causes the execution of the program to pause until some characters are added.

The erase character, "#", will remove the last character added to the input stream, unless that character was a new-line, in which case the erase character will have no effect. The kill character, "@", will remove from the input stream any and all characters added after the last new-line.

One function has been pre-defined to convert a string into the value of the name which is the string with its quotation marks removed.

7.1. LookupinJ

Legal Arguments: Any 2-tuple satisfying the following constraints:

1. the first element is a string;
2. the second element is the result of evaluating jj;
3. the name obtained by removing the quotation marks from the string is defined in the environment in which the jj was evaluated.

Error Value: nil

Value: The object whose name in the environment in which the jj was evaluated is obtained by removing the quotation marks from the string. This value will share with other objects in the environment.

8.1. SYSTEMERROR

This function is called indirectly when a library function is applied to an illegal argument; it may also be called directly by applying SYSTEMERROR to any argument. Whenever SYSTEMERROR is called, it prints the following message:

```
Do you wish to continue?
```

Unless the user types

```
yes
```

evaluation is terminated. If the user types

```
yes
```

the evaluator continues, returning, as the value of SYSTEMERROR, the argument. That is, SYSTEMERROR is the identity function.

Some variables have been pre-defined by PAL programs.

9.1. Write

Write may be applied to any object. The value of Write is the value of "dummy." If the argument is not a tuple of length greater than zero, the side effect of Write is identical to that of Print. Otherwise, the side effect of Write is to type out the element(s) of the tuple which is its argument without the parentheses and commas which would be inserted for this tuple by Print. For example,

```
Write('H', 'e', 'l', 'p', '*n')
```

causes the message

```
Help
```

to be printed, while

```
Print('H', 'e', 'l', 'p', '*n')
```

results in the printing of

```
(H, e, l, p,
)
```

```
Write((1, 2), '   ', (3, 4), '*n')
```

results in the printing of

```
(1, 2)   (3, 4)
```

Write is exactly defined by the PAL program

```
Write x = Istuple x -> W(1, Length x) !
          Print x

          where rec W(i, n) = n=0    -> Print nil !
                              i > n  -> dummy     !
                              ( Print(x i);
                                W(i+1, n)  )
```

9.2. Tuple

The argument of Tuple must be an integer. If this integer is less than or equal to zero, the value of Tuple is the 0-tuple. If this integer is n, n greater than 0, the value of Tuple is a Curried function of n arguments. The value of the application of this function to its n arguments is an n-tuple. Neither this n-tuple, nor the value of Tuple applied to an integer, nor any of the functions produced as Tuple is applied to its arguments share with any other object in the environment. The elements of the n-tuple produced may share with other objects however.

The value of

```
Tuple 3 'a' 4 17.2
```

is the same as the value of

```
( 'a', 4, 17.2 )
```

The value of

```
Tuple 0
```

is the same as the value of

```
nil
```

Tuple is exactly described by the PAL program

```
                     rec Extpl n x = n < 1 -> x !
                          ll y . Extpl (n-1) (x aug y)

within Tuple N = N < 1 -> nil !
                     ll z . Extpl (N-1) (nil aug z)
```

9.3. Cy

Cy may be applied to any object. The value of Cy is an object which is a duplicate of the argument. Neither the value of Cy nor any of the components of Cy (assuming that the argument is a tuple) share with the argument or with any of the components of the argument.

Cy is exactly described by the PAL program

```
Cy S = let List = nil in Copy S

       where Lookup Nde =
             let rec Lkp L = Null L ->
                                  ( let NewN = nil in
                                    List := ((Nde, NewN), $ List);
                                    (false, NewN)             ) !
                             Share(Nde, L 1 1) -> (true, L 1 2) !
                             Lkp(L 2)
             in Lkp List

       within rec Copy Node =
              let Fnd, CpyN = Lookup Node
              in  Fnd -> CpyN         !
                  not Istuple Node ->
                      ( CpyN := Node;
                        CpyN         ) !
                  let j, Size = 1, Order Node
                  in
            Copyloop: j > Size -> CpyN !
                      ( CpyN := CpyN aug Copy(Node j);
                        j    := J + 1
                        goto Copyloop                )
```

Some Examples of PAL

In this section there is a rather simple correctly written example of PAL programming. In the next sections of this appendix are some more complex programs. This section was last modified on 02/11/68 at 12:45 by Evans.

For our example, we write a PAL program which prints, for numbers from zero to ten, the number and its square root. The main part of the program will be the following:

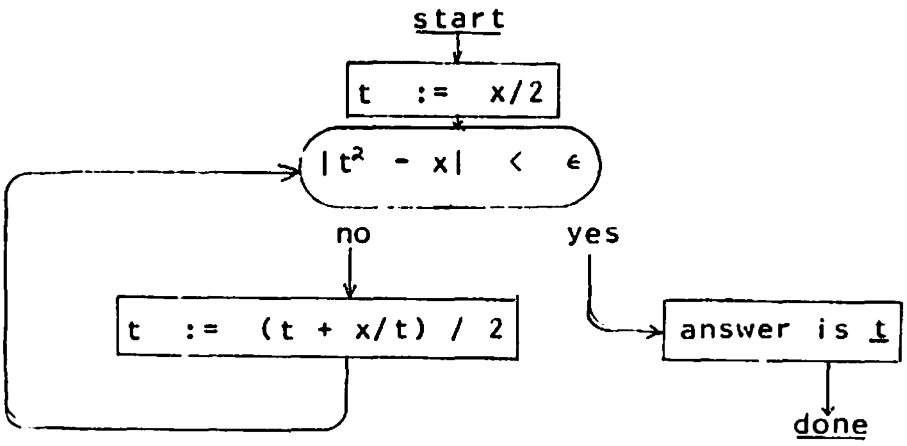
```
let i = 0
in

L:
     Write (i, '*t', Sqrt (ItoR i), '*n');
     i := i + 1;
     i < 11
     ->
          goto L
     !
     Write '*nAll done.*n'
```

This little program creates an integer counter _i_, which counts from zero to 10. The statement in this program that does all the useful work is

```
Write (i, '*t', Sqrt (ItoR i), '*n')
```

The effect of executing this statement is to print the value of _i_, a "tab" character, the square root of _i_, and a "new line" character. Since the function _Sqrt_ requires an argument of type _real_, the built in function _ItoR_ is invoked to do the necessary type conversion.

This program will not run as it is, since PAL's designers have provided no square root function in the library. Thus we must provide our own such function. A method used frequently is that shown in the accompanying flow chart.



Flow Chart for Square Root

We choose to implement this flow chart with the followinr PAL definition:

```
let Sqrt x =
     f(x/2.0)
     where rec f t =
          Abs( t*t - x ) < 0.005
          ->   t
          !
               f ( 0.5*(t + x/t) )
```

Here f is a recursive function of one argument. If that argument is close enough to the square root of x, it is returned as the value of the function; while otherwise f calls itself with the next approximation as argument. All that Sqrt does is to call f with the first approximation as argument.

One more thing is needed: PAL's library does not include an absolute value function Abs, so we need

```
let Abs x =  x < 0.0  ->  -x  !  x
```

The last page of this appendix shows the entire program, along with a run of it on the computer.

```
print sqrty pal
W

 SQRTY    PAL     02/11/68  1151.8

// Sample PAL program for the PAL Manual.
// This program was written by A. Evans on 11 Feb 68.
// It was last modified on 02/11/68 at 11:50 by Evans.

let Abs x =  // compute absolute value of argument
             // works for reals only - not integers
          x < 0.0  ->  -x  !  x
in

let Sqrt x =  // compute square root of x (reals only)
     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.
          !         // No, so keep trying...
               f ( 0.5*(t + x/t) )
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
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
R
```

(Last modified on 02/12/68 at 12:50 by 4620.)

The Abstract Syntax Tree

One step in the evaluation of a PAL program involves determining which operands go with which operators. For example, before executing the program of Fig. 1, it is necessary to decide that the addition is done before the function application.

```
Print(4 + 5)
```

Fig. 1

The result of this analysis may be expressed in a tree format. The tree for the program of Fig. 1 is shown in Fig. 2. The tree indicates that "Print" is being applied to the sum of 4 and 5.

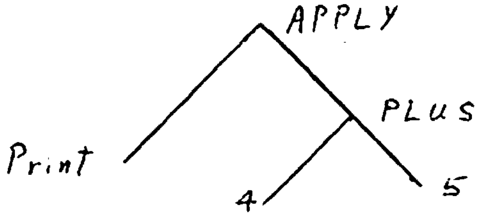


Fig. 2

The time-sharing consoles available at M.I.T. are poorly adapted for drawing trees such as that of Fig. 2. Therefore we have chosen an alternate representation, which is exemplified by Fig. 3. This representation is derived from that of Fig. 2 in the following way:

1. The node at the top of the tree structure of Fig. 2 is printed at the left margin of Fig. 3;
2. A node n levels from the top of the tree in Fig. 2 is preceeded by n dots in the representation of Fig. 3;
3. The left-to-right ordering of nodes in Fig. 2 is converted to a top-to-bottom ordering in Fig. 3.

```
APPLY
. * Name12345  Print
. PLUS
. . * Number 4
. . * Number 5
```

Fig. 3

This representation is called an "abstract" syntax tree because the labels on each node need not be the same as the corresponding symbols in the PAL language. For example, we use the label "PLUS" to correspond to the functor "+", and the label "APPLY" to correspond to the operation denoted by juxtaposition in PAL. We have also added the tags "* Name" and "* Number" in the abstract representation. The digits after the "* Name" tag, by the way, tell something about the internal representation of the name "Print" in the machine. They are of no importance to the user.

In the example above, the abstract syntax was reasonably close to the concrete syntax of the PAL program. The user should be aware that this is not always the case, as illustrated by the programs of Fig. 4. These two programs both have the same abstract syntax tree, which is shown in Fig. 5. Both programs have the same abstract syntax because the "where" construction in PAL is not really a new idea, but is simply an alternate concrete representation for the idea behind the let...in construction.

```
let y = f 4                    Print(y + y)
in Print(y + y)                where y = f 4
```

Fig. 4

```
LET
. VALDEF
. . * Name12345   y
. . APPLY
. . . * Name23456   f
. . . * Number 4
. APPLY
. . * Name34567   Print
. . . PLUS
. . . . * Name12345   y
. . . . * Name12345   y
```

Fig. 5

The abstract syntax tree is often useful for checking PAL's interpretation of a program. Consider the segment of program shown in Fig. 6. Someone reading this program would probably assume that it meant that either the assignment to b or the assignments to c and d should be executed, depending upon the value of a. However, the abstract syntax tree for this program, shown in Fig. 7, indicates that PAL interprets this segment as meaning that either the assignment to b or the assignment to c should be executed, followed in any case by the assignment to d. Incidently, this segment is a good example of the fact that the PAL compiler is not influenced by indentation or spacing. For the benefit of the humans involved, the author probably should have typed the segment as shown in Fig. 8.

```
a < 100 -> b := 90 !
           c := 80;
           d := 70
```

Fig. 6

```
SEQ
. COND
. . LS
. . . * Name12345  a
. . . * Number 100
. . ASS
. . . * Name23456  b
. . . * Number 90
. . ASS
. . . * Name34567  c
. . . * Number 80
. ASS
. . * Name45678  d
. . * Number 70
```

Fig. 7

```
    a < 100 -> b := 90 !
               c := 80 ;
    d := 70
```

Fig. 8

To limit the amount of printed output obtained when the /tree/ option is used, PAL has been designed to print only five levels of the abstract syntax tree. Therefore, the program of Fig. 9 would give rise to the printing of Fig. 10. In general, if the user is interested in seeing the abstract syntax tree for a segment more than five levels down in the tree for the complete program, he can submit just this segment to the PAL compiler. The result may not be a complete program, so that it may not compile or run successfully; nevertheless; the compiler will print a tree for it.

```
f( g( h( u( v( w x ) ) ) ) )
```

Fig. 9

```
APPLY
. * Name12345   f
. APPLY
. . * Name23456   g
. . APPLY
. . . * Name34567   h
. . . APPLY
. . . . * Name45678   u
. . . . APPLY
. . . . . * Name56789   v
. . . . . APPLY
. . . . . . * Name67890   w
. . . . . . Etc
```

Fig. 10

Sharing

An important idea in PAL is that two variables may "share the same storage location". In this section that idea is explained. This section was last modified on 02/19/68 at 11:14 by Evans.

Introduction

One of the simplest statements that one can make about sharing in PAL is the following:

> Two variables are said to share if updating either of them causes the other to be updated also.

As an example of sharing, consider the following PAL program:

```
let a = 1 in           // create a new a with value 1
Write a;               // write it
let b = a in           // create new b, sharing with a
Write b;               // write it
a := 3;                // update a
Write b;               // b will now be 3 also
b := 4;                // update b
Write a                // a will be 4
```

The effect of executing this program is to print "1134".

Unfortunately, sharing is a more complex issue than this example indicates. For example, if T denotes a tuple, then

```
let a = T 2 in
```

results in a sharing with the second component of T. Similarly,

```
let U = a, b, T 1 in
```

results in a sharing with U 1, b with U 2 and T 1 with U 3, so that executing

```
U 2  :=  4
```

will change *b*. However, executing

```
T  :=  7, 8, 9, 10
```

changes *I* but does not change *U*.

Lvalues and Rvalues

Each variable in PAL is associated with a *cell* in the computer's memory, the association being made at the time the variable is defined (by a definition). This storage cell is called the *Lvalue* of the variable. An Lvalue in PAL has the following three properties:

- An Lvalue contains a value, called an *Rvalue*. (That is, the memory cell which is the Lvalue has a contents which we call an Rvalue.)
- The *Rvalue* contained can be replaced, but only by an assignment that updates the cell.
- An Lvalue remains in existence as long as there is a reference to it.

Before proceeding, it is perhaps worth while to mention the source of the terms Lvalue and Rvalue. In an assignment statement such as

```
x  :=  x + 1
```

it should be clear that the *x* on the left side occupies a role essentially different from the *x* on the right. On the right, it is the *value* associated with *x* that we are concerned with, while on the left we are concerned with the *location* in the computer where we are to store a new value. Thus on the *r*ight side of the assignment statement we are concerned with the Rvalue associated with *x* and on the *l*eft side of the assignment statement we are concerned with *x*'s Lvalue.

Mode of Evaluation

Any expression in PAL can be evaluated in either *Lmode*, to yield an Lvalue, or in *Rmode* to yield an Rvalue. In the assignment statement shown above, the *x* on the right is evaluated in Rmode and its Rvalue is added to *1*. The *x* on the left is evaluated in Lmode to yield *x*'s Lvalue, so that the Rvalue associated with that Lvalue can be updated to hold the Rvalue computed on the right. In the next part of this appendix we provide detailed rules so that the reader can work out for himself modes of evaluation and sharing. There are three points to make: First, the *context* of an expression indicates what mode it must be made to yield. For example, an expression to be an operand of "+" must yield an Rvalue. Second, the *form* of an expression determines what mode it will actually yield. Thus any expression of the form "E+E" always yields an Rvalue. Finally, transfer functions are provided to convert between modes when the wrong one is available.

The Mode Context Table

The mode of evaluation of any particular expression appearing in a PAL program is determined by the syntactic context of the expression. For example, the two operands of the functor "+" are always evaluated in Rmode, since addition takes place on Rvalues. Table 1 below defines the mode of evaluation for every context in PAL. The symbols *E*, *L* and *R* are used in this table as follows:

L denotes a left-hand context, one in which the evaluation is in Lmode yielding an Lvalue.

R denotes a right-hand context, one in which evaluation is in Rmode yielding an Rvalue.

E denotes a context in which the mode of evaluation is the same as that of the expression of which the context is a part.

As an example of *E* context, let us consider the arms of a conditional. The evaluation scheme for a conditional is

R -> E ! E

indicating that the Rvalue of the boolean is needed but that the mode of evaluation of the arms is dependent on where the conditional expression is used. For example, in

(x > y -> x ! y) := 5

the arms of the conditional would be evaluated in Lmode, while in

1 + (x > y -> x ! y)

they would be evaluated in Rmode.

We now give the table. Here $\underline{\alpha}$ stands for any one of the marks

+ - * / ** & | < = >

and $\underline{\beta}$ for any of

+ - not

The notation <bv part> is short for <bound variable part> as defined in section 4.2/F.

Table 1: The Mode Context Table

R α R β R   \$ R   R % <variable> R

R aug L   L $\{ \ , L \}_1^\infty$

R L

R -> E ! E

goto R   R ; E   L := R

E where <definition>   let <definition> in E

val L   res L

(E)

ll <bv part> . L

<variable> $\{ \ , \text{<variable>} \}_0^\infty$ = L

<variable> <bv part> = L

The last two lines of this table refer to definitions.

The Functions $\mathcal{S}$ and $\mathcal{C}$

The functions $\mathcal{S}$ and $\mathcal{C}$ are of supreme importance in understanding PAL, even though they are not available directly to the PAL programmer. $\mathcal{S}$ takes an Rvalue as argument and yields an Lvalue containing the Rvalue as its result, the Lvalue thus supplied being distinct from all Lvalues previously in existence. It is an important feature of PAL that Lvalues can be created only by applying this function. It is this fact that is crucial to the present discussion of sharing.

$\mathcal{C}$ is the inverse function: Taking an Lvalue as argument it returns the corresponding Rvalue as result.

R-Type Expressions and L-Type Expressions

The expressions shown in Table 2 are called Rtype expressions in that they always yield Rvalues when they are evaluated. Here $\mathcal{E}$ stands for any expression, and α and β are as above.

Table 2: R-Type Expressions

<quotation> <numeric> <literal>

$\$\ E \quad E\ \alpha\ E \quad \beta\ E \quad \$\ E$

$E\ \{\ ,\ E\ \}_1^{\infty} \quad E \text{ aug } E$

ll <bv part> . E

$E := \mathcal{E}$

If such an expression appears in a left hand context, then PAL automatically invokes the function $\mathcal{S}$ to supply the needed Lvalue. Thus the evaluation of one of these expressions in an Lmode context will produce an Lvalue guaranteed not to share with anything else.

The expressions shown in Table 3 are called Ltype expressions, always yielding an Lvalue when evaluated:

Table 3: L-Type Expressions

E E	<variable>
E % <variable> E	val E

The function $\underline{\mathcal{C}}$ is automatically invoked if such an expression appears in a right hand context.

Sharing

We are now able to give an accurate definition of sharing:

> Two components are said to *share* if they have the same Lvalue.

Here a *component* is either a variable or an element of a tuple.

We also make one further observation: A *tuple* is a set of Lvalues, and the result of applying a tuple to an integer (within the proper range) will always be the relevant Lvalue. When the programmer writes a listing (i.e., the representation of a tuple written with commas), it is the Lvalues of the elements that are put together to make the tuple.

The unsharing operator "$" causes its operand to be evaluated in Rmode. If the expression appears in a left hand context, PAL then applies $\underline{\mathcal{L}}$ to get a new Lvalue that shares with nothing. Note that "$" is only useful if its insertion causes the application of both $\underline{\mathcal{C}}$ and $\underline{\mathcal{L}}$.

Examples

The reader now has all the information needed to understand sharing. We present now a few examples to illustrate how to use that information. Let us examine the line

```
let b = a in
```

Here we have an instance of a definition. On the right we have a <variable>, and we find <variable> in Table 3 of Ltype expressions. That means that the evaluation of a variable produces an Lvalue. In Table 1, we find that the right side of a definition is a left hand context. Thus the Lvalue supplied is what is wanted, and neither $\mathcal{L}$ nor $\mathcal{C}$ is needed. The effect of the definition is to establish a new variable named b, with the same Lvalue as that of a. Thus a and b share, having the same Lvalue.

Had the definition been

```
let b = 2 in
```

we would note that 2, a <numeric>, is an Rtype expression. Since it appears in a left hand context, the function $\mathcal{L}$ is invoked automatically by the PAL system to return an Lvalue guaranteed to be distinct from all other existing Lvalues. b is then created with this Lvalue, resulting in a b that shares with nothing.

Now we consider

```
let U = a, b, T 1 in
```

a and b are Ltype expressions, as is T. 1 is an Rtype expression. In Table 1 we find the scheme "R L", so we must apply $\mathcal{C}$ to T and $\mathcal{L}$ to 1 and then apply the first result to the second. The result is an Ltype expression. Table 1 shows that elements of a listing are to be evaluated in Lmode, so the proper mode exists here. The listing itself is an Rmode expression, so $\mathcal{L}$ is invoked to create a new Lvalue. U is then created with this Lvalue. Thus U shares with nothing, but the three components of U share as indicated.

As another example, let us consider again

```
x  :=  x + 1
```

The operands of "+" are to be evaluated in Rmode. 1 is already an Rvalue and we apply $\mathcal{C}$ to the Lvalue of x to obtain the Rvalue. The sum is an Rtype expression according to the Table 2. The x on the left is evaluated in Lmode, and the Rvalue determined on the right is then assigned to be the new Rvalue associated with the Lvalue of x.

Finally, we can now see why statements such as

$$1 := 2$$

are nugatory, as suggested in section 3.2/N. The $\underline{1}$ on the left side is an Rtype expression appearing in an Lmode context, so $\underline{\mathcal{L}}$ is invoked as a transfer function to produce a new Lvalue. It is the Rvalue associated with this Lvalue that is changed to $\underline{2}$.

Introduction: (This section was last modified on 02/04/68 at 03:30 by Evans.)

In this section, we are concerned with the application of a function to arguments. In the simplest case, both the function and its argument are given. For example, the expression

```
Sqrt 8
```

denotes the result of applying Sqrt to 8. Either the function or the argument may be the result of arbitrarily complex calculations. Thus the expression

```
(x y) (z w)
```

denotes the result of applying (the result of applying x to y) to (the result of applying z to w).

Note that PAL differs from conventional notation in not requiring parentheses to surround arguments: Juxtaposition is adequate to indicate application. Thus we permit either

```
x y
```

or

```
x(y)
```

rather than require the latter.

Formulae:

```
<combination>  ::=  <expression> <expression>
      | <combination> <expression>
```