PRDL -- PROCEDURAL DESCRIPTION LANGUAGE

NCI/IP Technical Report #15

October 10, 1977

Bruce Shapiro, Peter Lemkin and Lewis Lipkin

National Cancer Program

NATIONAL CANCER PROGRAM

U.S. Department of Health, Education, and Welfare / National Institutes of Health / National Cancer Institute

NCI/IP-77/04

PRDL -- PROCEDURAL DESCRIPTION LANGUAGE

NCI/IP Technical Report #15

October 10, 1977

Bruce Shapiro, Peter Lemkin and
Lewis Lipkin

Image Processing
Division of Cancer Biology and Diagnosis
National Cancer Institute
National Institutes of Health
Bethesda, Maryland 20014

"We here highly resolve . . ."

NCI/IP-77/04

PRDL

PROCEDURAL DESCRIPTION LANGUAGE

NCI/IP Technical Report #15

B. Shapiro, P. Lemkin, L. Lipkin

Image Processing Unit
National Cancer Institute, DCBD
National Institutes of Health
Bethesda, Md. 20014

October 10, 1977

ABSTRACT

The Procedural Description Language is a high level modelling language. It is designed for use in performing interactive image description and analysis. The user interactively designs models incorporating problem domain knowledge on both the low level image processing and successively higher levels of processing abstraction. PRDL achieves modelling efficiency through the use distributed special purpose processors.

T A B L E O F C O N T E N T S

SECTION PAGE

1 Introduction 1

2 Specification of PRDL 4
- 2.1 An Expression Language 5
- 2.2 Evaluation Mode of Operation 8
- 2.3 Iterative Control Structure 9
- 2.4 Function Definition and Composition 9
  - 2.4.1 Builtin procedures 12
  - 2.4.2 External processor function definitions . . 12
- 2.5 Matrix operators 13
- 2.6 String operators 15
- 2.7 Relational Data Base 17
- 2.8 Sets and Lists 19

3 Variable Classes 22
- 3.1 FUNCTION ARGUMENT variables 22
- 3.2 LOCAL variables 22
- 3.3 GLOBAL variables 23
- 3.4 PATTERN PREFIX variables 23
- 3.5 QUOTED variables 24
- 3.6 MATRIX variables 24
- 3.7 BOOLEAN variables 25
- 3.8 SET and LIST variables 25
- 3.9 STRING variables 25
- 3.10 RELATIONAL TRIPLE variables 26
- 3.11 Type of a variable 26
- 3.12 Deleting a variable from PRDL 26

4 I/O expressions 27
- 4.1 Teletype I/O 27
- 4.2 Saving and Restoring a Data Base 28
- 4.3 Control of the PDP10 by PRDL 28
- 4.4 Control of the RTPP by PRDL 29

5 Graphics Operators 33

6 EVAL and APPLY operators 35
- 6.1 Possibility Lists 36

7 Pattern Directed Invocation 38

8 Modelling Contexts 42

9 Weighting Functions 47
- 9.1 Weighting functions to implement levels of abstraction 48

10 Generator procedures 50

11 RTPP functions 51

12 PRDL Processes 54
12.1 Process control operators 54
12.2 Process scheduling 56
12.3 Safe region declarations 57

13 Modelling using composition of procedures 58

14 Top down and bottom up procedural definition 59

15 Special Grammatical Functions 60

16 Interactive command and function editors 63

17 The PRDL evaluator 65
17.1 PRDL control structure 66

18 Semantic condensation of plans - future work 68

19 References 70

A The implementation of PRDL 74
A.1 Possible graph node data structure 75

B PRDL Grammar and Symbol Table 79

C Compiling and Building PRDL 84

1. Introduction

The Procedural Description Language is being designed and implemented to allow biologists to function as biologists (without concern for detailed computer science considerations) when dealing with images from a wide variety of sources. Its flexibility is specifically directed toward permitting the biomedical scientist to construct (what are in fact operational definitions) new classes, concepts and procedures while operating an image acquisition/analysis system on line. This is without detailed concern for the underlying hardware and/or software involved. In particular, the biologist does not have to be concerned about where within the image processing network a particular facility may reside. He may invoke procedures, measurements, etc. at will and incorporate the results in the implicit logical structure which is being built by the system while he is defining and solving a particular problem. It is of special importance to note that PRDL provides for the easy incorporation of non-pictorial information about images and/or classes of images. This information may indeed be non-morphologic embodying biochemical and physiologic information not inherent in the image, but expressed by the biologist.

Alternatively, PRDL may be viewed as a high level modelling language. It is designed for use in performing interactive image description and analysis. The user interactively designs models incorporating problem domain knowledge on both the low level image processing and successively higher levels of processing abstraction. PRDL achieves modelling efficiency through the use distributed special purpose processors.

PRDL contains a subset of procedural description methods (available in PLANNER, CONNIVER, SAIL computer programming languages) as well as features oriented toward image processing and numerical analysis. It is being implemented in SAIL ([ReisJ76], [FelJ72]) as an interpreter system. It is conceptually similar to the NIH Modelling Lab (MLAB [Kno73]) in the ability to interactively define, compose, and evaluate functions in infix notation. However, it differs in being an expression language rather than being a command language as well as in the addition of new data and control structures. The data structures include scalars, matrices, sets, lists, relational data structures, patterns, weights, pictures, boundaries, and strings. The control structures include function evaluation, multiprocessing, distributed computing, context mechanisms, pattern matching, weighted function evaluation mechanisms, interprocessor function evaluation, and line drawing graphics.

PRDL may execute functions in PRDL itself or on other processors such as the Real Time Picture Processor (RTPP) functions ([Lem74], [Carm74], [Lem77a]) or other jobs running on the PDP10. The front end command parser of PRDL is generated from a BNF grammar specification using a parser generator PARGEN ([ShapB76], [ShapB77a]) while the semantic and interpreter actions are also specified in the grammar (but implemented in hand coded procedures).

2. Specification of PRDL

Several different approaches have been taken in the area of procedural description systems. In the CELWLD [Lem72a] system, procedural description was implemented as simple composition of LISP 1.6 functions [Qua68], while Tenenbaum's interactive image system [Ten74] uses composition of INTERLISP/FORTRAN functions.

The LISP embedded interpretive languages PLANNER [Hew72] and CONNIVER [McD72] perform procedural description using the composition of functions as well as pattern directed invocation. PLANNER uses goal statements which are driven by a data base consisting of assertions and pattern directed consequent and antecedent theorems (procedures) using a backtrack control structure. CONNIVER uses the manipulation of possibility lists and tree structured context mechanisms to allow user control over backtracking. These may be used to implement either a breadth-first or depth-first search. In PLANNER on the other hand, backtracking is automatic using a depth first search. In SAIL (a compiler) [FelJ72], one has facilities of associative search as well as the manipulation of sets and lists. Contexts, daemons, and processes are also available in SAIL. Most of the above mentioned systems are interpreters with the exception of SAIL.

The PRDL language contains some of the above constructions as well as others not in these languages. It includes composition of algebraic functions, pattern directed invocation, context mechanisms, associative search, set and list manipulation, multi-process communication, possibility list evaluation, generator functions, failure trace, weighting functions to facilitate heuristic evaluation of functions (in a tree type search), interprocessor procedure evaluation and line graphics. The language will be presented here by example and by formal (BNF) definition where appropriate. In the following examples, functions and PRDL operators are denoted in upper case while arguments are denoted by lower case symbols.

2.1 An Expression Language

PRDL is an expression evaluation language like LISP (without the top level printing of the evaluation) rather than a command language like ALGOL. The top level user input is an expression to be evaluated. The expression may appear to look like a command (as with an assignment statement) but it will always have a value. The BNF syntax for an expression is as follows:

```
        <expression>::= <compound expression> |
                <assignment expression> |
                <function designator> |
                <conditional expression> |
                <arithmetic expression> |
                <Boolean expression> |
                <matrix expression> |
                <string expression> |
                <list expression> |
                <set expression> |
                <pattern expression> |
                <pattern-D-I expr> |
                <pattern request  expression> |
                <context expression> |
                <WHILE loop> |
                <FOR loop> |
                <TTY expression> |
                PRINTF <function name> |
                <function definition> |
                <builtin function> |
                <relational expression> |
                <FOREACH loop> |
                <I/O expression>
                <graphics expression>
                <process expression>
```

Block structure is used as in ALGOL syntax with BEGIN/END delimiters and semicolons ";" used to delimit expressions.

```
        <compound expression>::= <compound head> END
        <compound head>::= BEGIN <expression> |
                <compound head> ; <expression>
```

Assignment of data to variables is denoted by the use of the backarrow operator "_" as in SAIL rather than with a "=" as in FORTRAN ("=" is used to denote the BOOLEAN equal test) or ":=" as in ALGOL.

```
        <assignment expression>::= <leftside> _ <expression>

        <leftside>::= identifier
```

Both IF-THEN and IF-THEN-ELSE statements are allowed in PRDL. The value of the <conditional expression> is that of the THEN expression if true (or 0 or false if not true) in the case of the IF-THEN. In the case of the IF-THEN-ELSE it is either the THEN or ELSE expressions.

```
        <conditional expression>::= <IF expression> |
                <IF expression> ELSE <expression>
        <IF expression>::= <IF clause> <expression>
        <IF clause>::= IF <Boolean expression> THEN
```

Boolean expressions are allowed over other data types than reals (which have FALSE=0, TRUE=1) by the implicit conversion of the non-real data type to real before evaluation. This point will be clarified later in the paper.

```
        <Boolean expression>::= <disjunctive expression> |
                <Boolean expression> OR <disjunctive expression>
        <disjunctive expression>::= <negative expression> |
                <disjunctive expression> AND <negative expression>
        <negative expression>::= NOT <algebraic relation> |
                ( <Boolean expression> ) |
                NOT ( <Boolean expression> |
                <algebraic relation>
        <algebraic relation>::= <algebraic expression> > <ae> |
                <algebraic expression> < <ae> |
                <algebraic expression> = <ae> |
                <algebraic expression> LEQ <ae> |
                <algebraic expression> GEQ <ae> |
                <algebraic expression> NEQ <ae> |
                TRUE | FALSE
```

Note, in a <Boolean expression>, if there is a conjunction of ANDs and one of them fails or a disjuction of ORs which succeeds, control will pass to the statement after the conjunction so that it is possible to determine why the conjunction failed. The system variable ANDFAIL contains the quoted name of the conjunction which failed, and ORSUCCEED for the quoted name which succeeded.

The basic algebraic expression is defined for the set of real numbers but carries over to matrices, strings and other data types with special semantics to be discussed later.

```
        <algebraic expression>::= <simple arithmetic expression>
        <simple arithmetic expression>::= <term> |
                <simple arithmetic expression> + <term> |
                <simple arithmetic expression> - <term>
        <term>::= <factor> |
                <term> * <factor> |
                <term> / <factor> |
        <factor>::= <primary> | <factor> ^ <factor>
        <primary>::= identifier | number | (<expression>) |
                <function designator> | + <primary> | - <primary>
```

2.2 Evaluation Mode of Operation

The user input to PRDL is an expression. As in LISP, all expressions evaluate to a value. However, unlike LISP, this value is not printed on the teletype and the value may be one of a wide variety of data structure types (real, string, matrix, set, list, procedure name etc.).

The value of an expression is printed on the users teletype by invoking the TYPE or OUTSTR operators.

```
        TYPE 1+2$
```

or

```
        OUTSTR("SUM of 3+4="&(3+4))$
```

will print "=3" and "SUM of 3+4=7" respectively. The $ denotes the altmode or escape character used to terminate teletype input.

2.3 Iterative Control Structure

The iterative control structure mimics that of ALGOL (SAIL) and many other similar languages having the WHILE-DO and FOR-DO structures.

```
        <WHILE loop>::= <WHILE clause> <expression>
        <WHILE clause>::= <Boolean expression > DO
        <FOR loop>::= <FOR assignment> <FOR control> <FOR limit>
                <expression>
        <FOR assignment>::= FOR <assignment statement> STEP
        <FOR control>::= <algebraic expression> UNTIL
        <FOR limit>::= <algebraic expression> DO
```

There are no labels and GOTOs in PRDL. Thus, function composition is encouraged as a means of expressivity. The value returned by these structures is the value of the last expression in the loop.

2.4 Function Definition and Composition

Algebraic functions may be composed using the rules of composition of functions. This includes the definition of recursive functions. A function need not be specified in advance in order to be used in a composition definition, provided that when the composed function is used (evaluated) it has been defined.

User defined procedures may be defined. This is done by a procedure defining operation. This operation creates a new function which is a lambda expression for the expression denoting the procedure. The lambda expression is stored as a tree consisting of a linked list of symbols composing the procedure. Thus the procedure is evaluated interpretatively. Composition of these symbolic procedure strings is performed by interpretive tree traversal as in MLAB and FORMAL [Mes70] (an algebraic string manipulation languages). For example, if we had:

```
FCT A(x) = x+(x*x);

FCT B(x) = 2*A(x);
```

then B(x) would evaluate to

```
(2*x)+(2*x*x).
```

A function is defined according to the following BNF specification.

```
<function definition>::= <funct. prefix> <remainder f-def>
<funct. prefix>::= FCT <processor declar.> <data declar.>
        <pattern declar.>
<remainder f-def>::= <function name> <f-args> = <f-body>
<processor declar.>::=  PDP10 | RTPP | GPP | GPPASM |
         null
<data declar.>::= SCALAR | MATRIX | WEIGHT | STRING | SET |
        LIST | PATTERN | CONTEXT | PICTURE | BOUNDARY |
        null
<pattern declar.>::= <pattern prefix> | null
<function name>::= identifier
<f-args>::= ( <arg list> ) | null
<arg list>::= <act-ident> | <arglist> , <act-ident>
<act-ident.>::= <data declar.> identifier
<f-body>::= <expression>
```

The RTPP, GPP and GPPASM processors are part of the RTPP.

For example a procedure is defined as follows using the FCT (function declaration):

```
FCT C(x,y)=B(x,A(y)).
```

defines function C to be the composition of functions A and B.

A function may be run on one of several processors (i.e. code is specifically written for a particular processor) by optionally specifying the processor name declaration in the function definition. An optional data declaration may be specified as part of the function arguments. The default is is determined by generic data typing i.e., use the type of the argument being passed. For example,

```
FCT PDP10 PICTURE GRADIENT(x)= .....
```

As was mentioned before, an expression may be a <function designator> or function name. There are two types of functions: builtins (already defined in the initial PRDL system) and user defined functions. The user defined functions may be any expression allowed by PRDL (as is seen from the above BNF definition).

```
<function designator>::= <builtin> |
        <weight prefix> <function name> <actual arguments> |
<builtin>::= <builtin identifier> <actual arguments>
<builtin identifier>::= identifier
<actual arguments>::= ( <actual parameter list> ) |
        null
<actual parameter list>::= <expression> |
        <actual parameter list> , <expression>
```

2.4.1 Builtin procedures

PRDL contains the usual set of algebraic Boolean operators of an algorithmic language which is available in algebraic languages. These are +, -, *, /, LOG, EXP, the radian trigonometric functions (SIN, COS, TAN, ATAN, ASIN, ACOS), the degree trigonometric functions (SIND, COSD, TAND, ATAND, ASIND, ACOSD), SQRT, TRUNC (truncate a real to its integer part).

New BUILTIN's may be added easily by specifing the appropriate class number with the BUILTIN name in the symbol table file and adding the code for the BUILTIN as part of the general interpretive case statement. No change in the grammar is necessary.

2.4.2 External processor function definitions

The external processors are to be thought of as slave processors to PRDL. Generally, PRDL will communicate with them through separate control and data I/O channels. In the case of the PDP10, the control channel is a pseudo teletype while the data channel is disk files. The RTPP has two separate teletype channels (8eTTY: and 8eHSP:) the former being the control and the latter the data channels.

```
        FCT GPP F(x)=<...>
```

defines a GPP MAINSAIL function causing it to be compiled on the PDP10 using MAINSAIL [Wil75] compiler. The assembly language output is then transmitted to the RTPP where it is then assembled with GPPASM [Lem76b]. When it is referenced, it will be evaluated on the RTPP.

```
        FCT RTPP G(x) =<...>
```

defines a RTPP function as a list of string interactions between PRDL and the RTPP. When evaluted, it sends the string interactions to the PDP8e control teletype on the RTPP.

```
        FCT GPPASM H(x)=<...>
```

defines a GPP assembly language (GPPASM [Lem76b]) program which is transmitted to the RTPP and assembled there on the PDP8e part of the RTPP. When referenced, it will be evaluated on the RTPP.

2.5 Matrix operators

Matrix operators include arithmetic as well as matrix transformation operators. Many of the matrix operators were derived from those used in MLAB [Kno73] and for that reason detailed discussion of them is left to that document. Some of the matrix operators are specified in infix notation while others are specified as built-in functions.

```
<matrix expression>::= <simple matrix expression>
<sme>::=<simple matrix expression>::= <m-term> | <term>|
        <sme> + <m-term> | <sme> + <term> |
        <sme> - <m-term> | <sme> - <term>
<m-term>::= <m-term> & <m-factor> | <m-term> &' <m-factor> |
        <m-term> * <m-factor> | <term> * <m-factor>
        <m-term> / <factor> |
        <m-factor>
<m-factor>::= <m-primary> | <m-factor> ^ <factor1>
<m-primary>::= <m-identifier> | (<matrix expression>) |
        <function designator> | + <m-primary> |
        - <m-primary> | <m-primary> '
<m-identifier>::= identifier
```

The ' operator denotes the transpose of the matrix. Thus for any element A[i,j], A[i,j]'=A[j,i].

The * operator computes the cross product of two matrices when both operands are matrices and scales the matrix otherwise.

The & operator denotes the row concatination of two matrices. Thus for C=A&B, where A and B are defined as:

```
                |a11...a1n |
                    ...
                |am1...amn |
and
                |b11...b1p |
                    ...
                |bm1...bmp |
then,
                |a11...a1n |
                    ...
                |am1...amn |
        C =         ...
                |b11...b1n |
                    ...
                |bm1...bmn |
```

.double space
where A or B are expanded so that each is the maximum of the number of columns n of the two.

The &' operator is similar to & except that it is the column concatination of the two matrices where A or B is expanded so that each is the maximum of the number of columns m of the two. For example, for A and B above:

```
          |a11...a1n b11...b1p |
C =            ...
          |am1...amn bm1...bmp |
```

Various built-in functions include:

ROW(A,n) - Return the row vector n of the matrix A.

COL(A,n) - Return the column vector n of the matrix A.

INVERSE(A) - Return the inverse matrix of A.

IDENTITY(A) - Return the identity matrix of size A (maximum dimension of A).

ABSMATRIX(A) - Return the absolute value matrix |A| computed by taking the absolute value of every element Aij.

EIGENVECTOR(A) - Return the Eigenvector of matrix A.

ARRAYSET(A,v) - Set all elements of matrix A to the specified value v and return A.

DOT(A,B) - Return the dot product of matrices A, B.

SOLVE(A,B) - Return the column vector X which is the solution to the matrix equation AX=B where B is a column vector and A is a matrix. If A is singular, then set X to the null vector.

2.6 String operators

Strings are available in PRDL and may be used in string manipulation as well as in specifying pattern names.

```
<string expression>::= <substring> | " text " |
        <string expression> & <string expression> |
        <number/str conv.> | <string/str conv.> |
<substring>::= <string expression> [ <start index> TO
                <finish index> ] |
        SUBSTR ( <string> , <start index> ,
                <finish index> )
<number/str conv.>::= <number CV> (
        <arithmetic expression> )
<string/str conv.>::= <string CV> ( <string expression> )
<number CV>::= CVS | CVOS | CVF
<string CV>::= LOP | COP | SYMBOLSCAN | INTSCAN |
              REALSCAN
<start index>::= <algebraic expression>
<finish index>::= <algebraic expression>
```

The semantics of the string operators are as follows:

& - return the concatination of two strings (s1&s2).

SUBSTR(str,i,j) - returns string str[i to j].

LOP(s) - return the first character of the string s and set s to s[2 to INF].

COP(s) - return the first character of the string s.

SYMBOLSCAN(s) - returns the next alphameric symbol or null if there is none as well as LOPing the symbol from the front of s.

INTSCAN(s) - returns an integer valued number found by scanning through a string leaving s as the string after the characters up to the end have been LOPed off.

CVS(r) - converts truncated real to string.

CVOS(r) - converts truncated real to octal string.

CVF(r) - converts real to decimal string in exponent notation.

Several functions of strings return real values. These may be used to analyze strings or substrings.

EQU(str1,str2) - Boolean returns TRUE if equal else FALSE.

LENGTH(s) - returns real valued length of the string.

CVD(s) - converts string digits to real number.

CVO(s) - converts string octal digits to real number.

```
        REALSCAN(s)   -  returns the real valued number found by
                scanning through a  string  leaving  s  as  the
                string  after  the  characters up to the end of
                the number have been LOPed off.
```

2.7 Relational Data Base

PRDL allows the creation, deletion and searching of associative triples which can be used to create a relational data base as in SAIL. An association is a 3-tuple:

```
        Attribute(Object)=Value or (A,O,V).
```

Each of the three (A, O or V) may be any PRDL identifier.

```
        <relational expression>::= <relational operator>
                ( <relational args> ) |
                <attribute> XOR <object> EQV <value>
        <relational operator>::= MAKETRIPLE | ERASETRIPLE |
                ISTRIPLE
        <relational args>::= <attribute> , <object> , <value>
        <attribute>::= identifier
        <object>::= identifier
        <value>::= identifier
```

AOV triples are created with the

```
        MAKETRIPLE(a,o,v)
```

or

```
        a XOR o EQU v;
```

operator which makes an association of any symbols in PRDL and returns the name of the triple which may be saved in a triple variable or put into a list or set. AOVs are deleted with

```
        DELETETRIPLE(a,o,v),
```

and tested with the Boolean operator

```
        ISTRIPLE(a,o,v).
```

The A variable may be tested to see whether it contains the name of a triple by

```
        ISTRPVAR(v)
```

which returns TRUE if it does exist otherwise it returns FALSE. The first, second and third components of a triple may be accessed by:

```
        FIRST(v), or SECOND(v) or, THIRD(v).
```

The triple pointed to by a triple variable may be deleted by:

```
        DELTRPVAR(v);
```

The relational data base is searched with the FOREACH statement which finds all instances of the specified triple search Boolean and executes an expression on each successful instance.

```
        <FOREACH loop>::= FOREACH <search variables> SUCHTHAT
                <element list> DO <expression>
        <search variables>::= <var1> , <var2> | <var1>
        <var1>::= identifier
        <var2>::= identifier
        <element list>::= <AOV Boolean expr.> AND <AOV expr.> |
                <AOV Boolean expr.> OR <AOV expr.>| <AOV expr.>|
                <boolean expression> | <generator procedure call>
        <AOV expression>::= <ATT> XOR <OBJ> EQV <var1> |
                <ATT> XOR <var1> EQV <VAL> |
                <var1> XOR <OBJ> EQV <VAL> |
                <ATT> XOR <var1> EQV <var2> |
                <var1> XOR <OBJ> EQV <var2> |
                <var1> XOR <var2> EQV <VAL> |
                <var1> XOR <var2> EQV <var3>
```

Note: The last rule should be used with caution since it permits an exhaustive search of the relational data base.

For example,

```
        FOREACH x SUCHTHAT INSIDE XOR x EQV CYTOPLASM DO
                PUT x INTO POSSIBLENUCLEUS;
```

2.8 Sets and Lists

Set and list variables may be defined in PRDL by setting a variable to a set or list. Various operators may then work on the set or list. Note that any PRDL identifier may be inserted as an element of a set or list.

```
        <set expression>::= <set operation> |
                SET { <actual arguments> } |
                { <actual arguments> } | <set identifier>
        <list expression>::= <list operation> |
                LIST [ <actual arguments> ] |
                [ <actual arguments> ] | <list identifier>
        <set operation>::= <set-list operation> |
                PUT <set expression> INTO <set identifier> |
                UNION ( <set expression> , <set expression> ) |
                INTERSECT ( <set expression> , <set expression> ) |
                SETDIFF ( <set expression> , <set expression> )
        <list operation.::= <set-list operation> |
                PUT <list expression> INTO <list identifier>
                        BEFORE <list expression> |
                PUT <list expression> INTO <list identifier>
                        AFTER <list expression> |
                PUT <list expression> INTO <list identifier>
                        AFTER <algebraic expression> |
                PUT <list expression> INTO <list identifier>
                        BEFORE <algebraic expression> |
                SUBLIST ( <list expression> , <algebraic expression> ,
                        <algebraic expression> ) |
                <list expression> [ <start index> to
                        <finish index> ] |
                <list expression> & <list expression>|
                SPLICE (<list expression>, <list identifier>)
        <set-list operation>::= LOP ( <set-list arg> ) |
                COP ( <set-list arg> ) |
                GETELEMENT (<set-list arg>, <algebraic expr.>)|
                REMOVE <set-list arg> FROM <set-list arg> |
                REMOVE <algebraic expr.> FROM <set-list arg>|
                REMOVE ALL <set-list arg> FROM <set-list arg> |
                LENGTH ( <set-list arg> )
        <set-list arg>::= <set identifier> | <list identifier> |
                ( <set expression> ) | ( <list expression> )
        <set identifier>::= identifier
        <list identifier>::= identifier

        LIST [a1,a2,...,an] - Create and return a list. The word
                              LIST may be omitted if [...]
```

are used.

LISTIFY(v) - Convert a set v to a list and return it.

SET{a1,a2,...,an} - Create and return a set. The word SET may be omited if {...} are used.

SETIFY(v) - Convert a list v to a set and return it eliminating duplicate elements.

LOP(x) - Remove the first element of the set or list x and return this element.

COP(x) - Return a copy the first element of the set or

SUBLIST(x,i,j) - Return a list consisting of the elements i to j of list x.

PUT x INTO y - Insert x into set y and return y.

PUT x INTO y BEFORE z - Insert x into list y before element z in list y. Note that currently if z evaluates to a list consisting of more than one element, the first element is used. List y is returned.

PUT x INTO y AFTER z - Insert x into list y after element z in list y. Note that currently if z evaluates to a list consisting of more than one element, the first element is used. List y is returned.

SPLICE(x,y) - splice the contents of the list x onto the end of the list y. Return y.
REMOVE x FROM y - Remove the first occurance of x from set or list y and return y.

REMOVE n FROM y - Remove the n'th element from the set or list and return y.

REMOVE ALL x FROM y - remove all occurrances of x from list y and return y.
GETELEMENT(y,i) - return the i'th element from the list y.

UNION(a,b) - Perform set union and return the result.

INTSECTION(a,b) - Perform set intersection and return the result.

SETDIFF(a,b) - Perform the set difference a-b and return the result.

LENGTH(v) - Return the number of elements in a list or set v.

Sets and list may be searched in a manner similar to relational triples with the FOREACH statement which finds all instances of elements satifying a search boolean which may be sets, lists or relational triples and executes an expression on each successful instance. Thus, the syntax for the <element list> which was defined in section 2.8 may now be expanded to include sets and lists.

```
<element list>::=<set-list arg> IN <set-list arg>
```

For example,

```
FOREACH x SUCHTHAT INSIDE XOR x EQV CYTOPLASM AND
        x IN LOBATEDSHAPE DO PUT x in POLYNUCLEUS
```

3. Variable Classes

Variables are used in ten different ways to pass argument and function name bindings. (Note: in this document "_" denotes the backarrow character.)

3.1 FUNCTION ARGUMENT variables

Function arguments may be passed through the procedure argument list as lambda variables (see [Ber64]), where there may be a null arg list. For example,

```
FCT F(x)=2*x;
```

Function arguments <f-args> may also have specific data type declarations associated with them to force checking on arguments before evaluation. For example,

```
FCT AREA(PICTURE x)=...
```

3.2 LOCAL variables

Local variables may be defined to be local within a procedure definition. In the following example, variable b is local to the function.

```
FCT F(x) = LOCAL b_2; c_1; ...
```

Note that variable c defaults to GLOBAL.

3.3 GLOBAL variables

GLOBAL variables are defined to be global within a context level (to be defined). Variables global to a context are global (unless redefined) to a context created from the parent context.

```
                a_1;
                FCT F(x)=a*x;
```

3.4 PATTERN PREFIX variables

The pattern directed invocation (PDI) of a procedure may cause one or more arguments associated with the pattern to be used as arguments in evaluating the procedure. The ? in front of a variable indicates that the variable will be supplied with an actual argument when the invocation mechanism is activated. For example, in the following procedure definition, the pattern "CAT" is associated with the pattern prefix variable ?x.

```
        FCT %"CAT",?x% F(?x)= ...
```

On invocation of procedure F by pattern "cat", the value of ?x is supplied by the pattern invocation process.

Pattern prefix ? variable bindings permits pattern directed invocation dynamic function names. For example:

```
        ... ?X(p,d,q) ...
```

3.5 QUOTED variables

Quoted variables are variables that should not be evaluated (@ prefix) when encountered in normal function evaluation. A quoted variable may hold the name of another identifier, argument, or function which may be passed between variables until evaluated using the EVAL constructions.

```
        x_@a;
        y_x               - pass the name of the variable a
        b_F(EVAL(y));     - computes F(a).
```

or, another example

```
        x_@F;
        y_x;
        APPLY(EVAL(y),a); - computes F(a).
```

3.6 MATRIX variables

Matrix variables may be addressed as subscripted arrays up to 3 dimensions using square brackets. The indices are truncated to an integer value. Both positive, 0 and negative subscripts are allowed. The array A might be addressed as:

```
        d_a[b,c];
```

If the matrix is being defined for the first time, then a matrix will be defined with size corresponding to the indices specified. If it is addressed later with larger indices, then the matrix is enlarged. For example,

```
        q[4,5]_3.416;
```

will create a matrix q of dimension 4 by 5.

3.7 BOOLEAN variables

Variables may take on the values TRUE (1) or FALSE (0) in which case they are Boolean variables and may be used in IF or WHILE constructions.

3.8 SET and LIST variables

Set and list variables are the names of sets or lists and are defined as such when they are used as sets or lists. Any variable or constant may be put into a set or list. The set-brackets "{" and "}" are used to explicitly denote sets, while square brackets are used to explicitly denote lists. The SET and LIST declarations overide the use {...} or [...].

```
        a_SET {1, var1, list3, set5, function10, ...};
```

or

```
        a_{1, var1, list3, 5, function10, ...};

        b_LIST [dog, cat, fish, cat];
```

or

```
        b_[dog, cat, fish, cat];
```

3.9 STRING variables

Strings are denoted by the use of double quotes (") and may be stored in variables and concatinated (using the & operator), and accessed (using the substring operator <string>[i TO j]. as follows:

```
        s1_"this is a string";
        s2_" and another string";
        s3_s1&s2; (the concatination of s1 and s2)
        s4_s1[i TO INF]&s2[3 TO 5];
```

3.10 RELATIONAL TRIPLE variables

Relational triples made with the MAKETRIPLE construction may be put into triple variables and thus manipulated by being put into sets or lists, etc. For example,

```
        v1_MAKETRIPLE("pattern","enzyme","reaction");
        IF ISTRPVAR(v1)
                THEN IF FIRST(v1)="pattern"
                        THEN TRUE ELSE FALSE;
```

3.11 Type of a variable

It is desirable at times to be able to determine the type of a variable. This is done by the TYPEIT(x) function which returns a number corresponding to the type of a variable. Variables are typed according to how they are used. For example,

```
        x_"abc" is a string type and
        x_MAKETRIPLE(a,b,c) is a relational triple type.
```

3.12 Deleting a variable from PRDL

It may be required to delete specific variable identifiers (eg. where an array is no longer needed). This is accomplished via the DELETE operator.

```
        DELETE(v)  - Delete variable v and its value. Care must
               be taken if the variable v occured in any
               association, set or list it must not be deleted.
```

4. I/O expressions

There are three types of I/O in PRDL: user teletype, PDP10 file, and external processor I/O. Each of these will now be discussed.

```
        <I/O expression>::= <TTY expression> |
                <file I/O expression> |
                <processor I/O>
        <processor I/O>::= <PDP10 I/O expression>
                <RTPP I/O expression>
```

4.1 Teletype I/O

The user interacts with PRDL via the teletype keyboard terminal. PRDL is able to accept input from the teletype using any of three modes.

```
        <TTY expression>::= <TTY input> | <TTY output>
        <TTY input>::= TTYLINE | TTYSYMBOL | TTYCHAR | REALIN
        <TTY output>::= TYPE <expr. list> |
                OUTSTR ( <expression> )
        <expr. list>::= <expr. list> , <expression> |
                <expression>
```

The semantics of these operators is as follows:

TTYLINE - returns a string terminated by users carriage return.

TTYSYMBOL - returns a symbol from user TTY terminated by a non-alphameric character.

TTYCHAR - return the next character typed.

REALIN - returns a real number terminated by a non-numeric character.

TYPE <expr. list> - types the value of the expr. list on separate lines as "VALUE=...". If an expression is an identifier, it will type the name of the identifier instead of VALUE.

OUTSTR(<expression>) - types the string equivalent of the <expression> but does not output extra carriage returns.

4.2 Saving and Restoring a Data Base

It is possible to setup, save and restore the PRDL data base either totally or selectively. The data files are in the form of ASCII text files such that the data is in the form of assignment or FCT <expressions>.

```
<file I/O expression>::= EXECUTE <file> | EX <file>
        SAVEFILE <I/O modifier> IN <file>
        GETFILE <I/O modifier> FROM <file>
<I/O modifier>::= <I/O modifier> , <I/O modifier> |
        ALL | <context expression> | <arg list>
<file>::= <device> <filename> <project-programmer>
<device>::= DSK | DSKA | DSKB | DSKC | SYS | null
<filename>= identifier . identifier | identifier
<project-programmer>::= [ number , number ]
```

Thus the data base may be dumped for later restoration by doing

```
SAVEFILE ALL IN data.pdl;
```

where data.pdl is the PDP10 file used. On entering PRDL at a later date, the data base may be restored completely by doing

```
EXECUTE (or EX) data.pdl;
```

or

```
GETFIIE ALL FROM data.pdl;
```

4.3 Control of the PDP10 by PRDL

PRDL may control programs that reside external to the PRDI core image. This permits the addition of separately constructed programs to become part of the PRDL environment. Thus, for example a separately built statistical package may become part of PRDL by the use of the following commands. Essentially, another job is activated which may receive or transmit data or control via a pseudo-teletype channel. The following is the set of pseudo-teletype commands available.

```
<PDP10 I/O expression>::= OUTPTY ( <string expression> ) |
                INPTY (<channel number>) | GETPTY |
                RELEASEPTY (<channel number>) |
                PTYREADY (<channel number>)
```

OUTPTY-	Output the given string over a pseud-teletype channel.
INPTY-	Receive data from a pseudo-teletype channel.
GETPTY-	Get a pseudo-teletype channel.
RELEASEPTY-	Release a pseudo-teletype channel.
PTYREADY-	Returns TRUE if it is alright to do an output over a pseudo-teletype channel and FALSE if input should be done.

4.4 Control of the RTPP by PRDL

The RTPP talks to the PRDL system through two high speed channels (via the PDP11/20 LINK [ShapB77c]). One is the PDP8e system teletype (8eTTY:) through which all commands to GPPASM/GPPLDR [Lem76b] and BMON2 [Lem77b] are issued as well as commands to OS/8 [DEC74] to run various other programs. The other PDP8e channel is called the high speed line (8eHSP:) and can be used to send and receive programs from the PDP10, and send arguments to the RTPP from the PDP10 and send the PDP10 property lists computed by the RTPP. RTPP compiled programs will reside on the PDP8e disk and would be activated by PRDL.

Strings may be sent or received on either of these two channels using the following syntax.

```
<RTPP I/O expression>::= <RTPP control I/O> |
        <RTPP data I/O>
<RTPP control I/O>::= OUT8eTTY ( <string expression> ) |
        IN8eTTY
<RTPP data I/O>::= OUT8eHSP ( <string expression> ) |
        IN8eHSP
```

The following example implements a sequence to move the microscope stage of the RTPP (x,y) microns, and then acquire a list of isolated component features with areas > 10 microns and perimeters > 25 microns.

```
FCT WAITFOR(ichar) =
         WHILE (ichar NEQ c_IN8eTTY) Do Continue ;

FCT RTPP MSTAGE(x,y,propertyliststring) = <
        OUT8eTTY("CONTROL/C");  " get OS8 monitor"
        WAITFOR (".");

        "Now  run BMON2 to acquire an image, get its
        components  property list and send the list
        back to PRDL."

        OUT8eTTY(".R BMON2"); "Run BMON2"
        WAITFOR("*");

        OUT8eTTY("INIT/8");  "Initialize Buffer memory
                               monitor to 100X
                               objective lens"
        WAITFOR("*");

        "Move the stage to relative position (x,y)"
        OUT8eTTY("MOVSTATE,SX,"&CVS(Abs(Int(x))&
                (If (x<0) Then ",/M"));
        *WAITFOR("*");
        OUT8eTTY("MOVSTATE,SY,"&CVS(Abs(Int(y))&
                (If (y<0) Then ",/M"));
        *WAITFOR("*");

        "Position the buffer memories  at  the  optical
        center of the microscope."
        OUT8eTTY("ALL384");
        *WAITFOR("*");

        OUT8eTTY("GET,BM3");  "Acquire video data in
                           central BM"
        WAITFOR("*");

        "Smooth the image beafore measuring segments
        by subtracting the Laplacian from the  average
        of the image."
        OUT8eTTY("BM2_LAPLACIAN,BM3");
```

```
WAITFOR("*");

OUT8eTTY("BM1_AVG8,BM3");
WAITFOR("*");

OUT8eTTY("BM0_BM1,SUB,BM2");
WAITFOR("*");

"Assuming the model of the material to be
segmented has two peaks, find the minimum
between the two peaks for later used in
thresholding the images"
OUT8eTTY("HISTOGRAM,BM0");
WAITFOR("*");

OUT8eTTY("SMOOTHHISTOGRAM");
WAITFOR("*");

"Now threshold the image at the histogram
minimum which is saved in Q-register variable
QRC."
OUT8eTTY("BM0_SLICE,QRC,255");
WAITFOR("*");

"Segment the sliced image with area sizing
of 10 microns and perimeter sizing of 25
microns. The /M switch specifies more input
specifications and /H fills holes in the
objects."
OUT8eTTY("BM1_SEGBND,BM0"&
                "/HOLEFILL"&
                "/INITFREESTORE"&
                "/MORESIZING");
WAITFOR("*");
OUT8eTTY("10/0"); "area sizing"
WAITFOR("*");
OUT8eTTY("25/1"); "perimeter sizing"
WAITFOR("*");
OUT8eTTY(null); "terminate the sizing list"
WAITFOR("*");

"Acquire the property list string of segmented
data."
propertyliststring_Null;
While True Do
        Begin "get data string"
        l_length(stmp_IN8eTTY);
        For i_1 Step 1 Until l Do
                If stmp[i For 1]="*"
                        Then Done;
propertyliststring_propertyliststring&stmp;
        End "get data string";
WAITFOR("*");

OUT8eTTY("EXIT");
WAITFOR(".");
```

```
                    WAITFOR(".");
                    >
```

At each step, we want to wait for a confirmation from the PDP8e that the command has been processed. The WAITFOR(character) function serves this purpose.

5. Graphics Operators

Line drawing graphics functions are available as builtin procedures. These basic set of graphics functions mimic the basic set of graphics procedures in the PDP10 Omnigraph system [CCB76].

Pictures in OMNI are denoted by picture numbers which are the set of positive integers greater than 0. The graphics display may be one of several different types: DEC-340, DEC-GT40, Tektronics 4012, 4010, 4014, etc.

```
<graphics expression>::= <Omni builtin> <actual arguments>
<Omni builtin>::= DAPPEND | DCLOSE | DCROSS  |  DGET  |
      DDONE  |  DDONE1 | DDRAW | DINI | DINT | DKILL |
      DMOVE | DOPEN | DPLOT | DPOST | DREL |   DTEXT  |
      DTSCALE  |  DUNPOST  |  FASTA  | FASTD | SLOWA |
      SLOWD | DWIND
```

The Semantics of these functions is given as follows.

DAPPEND(<omni number>) - append DDRAWs etc to specified picture.

DCLOSE(<omni number>) - close DOPENed or DAPPENDed picture.

DCROSS(op,xyarray,inchar) - when the crosshair is enabled on the Tektronix TK4012 class of terminals and a character is typed, return the value of the x,y crosshair in the array xyarray and the character typed in inchar.

DGET - get DINI specified display

DDONE - erase the screen and execute set of commands for storage display.

DDONE1 - do not erase the screen and execute set of commands for storage display.

DDRAW(x,y) - draw at DINT specified density to x,y.

DINI(<display number>,<I/O channel>,<display buff>,<buff size>) - initialize OMNIGRAPH.

DINT(<intensity>) - set the drawing intensity (function of display characteristics).

DKILL(<omni number> - delete omni picture.

DMOVE(x,y) - move the cursor (without drawing) to x,y.

DOPEN(<omni number>) - open a new omni picture.

DPOST(<omni number>) - post the omni picture.

DPLOT(<display buffer>,<file>) - PLOT the omni picture composed in the display buffer.

DREL - release the display previously gotten with DGET.

DTEXT(<string expression>) - display text at current cursor.

DTSCALE(<text size>) - set the text size (see [CCB76]).

DUNPOST(<omni number>) - unpost but do not delete the specified omni picture.

FASTA - turn on the Tektronix 4012 "fast ASCII" text mode where text does not get stored on the screen.

FASTD - turn on the Tektronix 4012 "fast vector" line drawing mode where line drawings do not get stored on the screen.

SLOWA - turn off the fast text mode if on.

SLOWD - turn off the fast line drawing mode if on.

DWIND - specifies the viewing window.

The following example briefly illustrates how these commands may be used with composition of functions to draw a SIN wave of varying sampling distances.

```
*DINI(4,0,0,0)$
*DWIND(0,TWOPI,-1,1)$
*DGET$
*FCT DRAWSIN(x)=BEGIN DMOVE(0,0); FOR i_0 STEP x
           UNTIL TWOPI DO DDRAW(i,SIN(i)) END$
*DOPEN(1)$
*DRAWSIN(.1)$
*DPOST(1)$
*DDONE$
```

6. EVAL and APPLY operators

It is possible to indirectly specify the names of identifiers to be used as variables (arguments) or procedure names. The names of the indirect identifiers may be assigned with the quote (@) operation. These variables may then be manipulated by the following operators.

```
<builtin>::= EVAL ( <expression> ) |
        APPLY ( <f-expr> , <a-expr> , <ordering> ) |
        PROG ( <p-list> , <a-expr> , <ordering> )
<f-expr>::= <list expression>
<a-expr>::= <list expression>
<p-expr>::= ( <p-expr> , <f-expr> ) | <f-expr>
<ordering>::= "A"|"P"
```

In its most general form the PROG and APPLY function (see below), may contain a list of procedures and a list of lists of arguments. This essentially permits a convolution of the procedures with the arguments. A question arises as to the order in which the convolution is accomplished. Under some circumstances it may be desirable to let the procedure list vary most quickly while in others it may be more desirable to let the argument list vary the quickest. The user can specify the ordering in the third argument of the function. "A" indicates that the arguments are to be scanned most rapidly while "P" indicates that the procedures are to be scanned most rapidly. If either the procedure list or the argument list contains one element then the third argument is ignored.

```
EVAL(x) - Return the value of the evaluation of quoted
                    variable x.

APPLY(F,x,) - Apply function F to a list of arguments
                    arguments contained in list x
                    and return F(x).
```

PROG(plist,arglist,) - apply the list of functions plist to the list of arguments in arglist and do not stop until the plist is empty. i.e. whereas possibilities list evaluation [see below](using the APPLY) stops on evaluating a FALSE. Returns FALSE when done.

CONTINUE - may be placed in either the argument or procedure list and has the effect of stopping the current level of convolution and evaluating the next element at the top level of convolution.

NOMORE - is a procedure which may be SPLICED into the argument or procedure list. It has the effect of halting the APPLY or PROG and returning FALSE.

The following are some examples of how EVAL, APPLY, and PROG may be used:

```
        q_@F;
        EVAL(q);
```

would evaluate to F.

```
        APPLY(F,37);
```
and
```
        APPLY(EVAL(q),37);
```
would both evaluate F(37).

```
        q_{@F,@G,@H};
        PROG(q,37);
```

would evaluate F(37), G(37), and H(37) and would return H(37).

It should be noted that the list operator SPLICE may be used at any time to dynamically expand the lengths of the procedure or argument lists.

6.1 Possibility Lists

There are times when the ability to evaluate sequentially a list of procedures on a specified list of arguments or to evaluate a procedure on a list of lists of arguments is useful when a boolean TRUE or FALSE is returned. This corresponds to a success or failure of a method in trying to solve a subproblem. These types of operations are called possibility list operations since each element either in the procedure list or argument list may be considered to be a new possibility to be tried in the attainment of a solution to a problem. The concept of a possibility list is taken from CONNIVER. If a procedure requires fewer arguments than is given to it, then it takes the required number. If there are two few arguments, then the function can not be evaluated and returns either FALSE or an error message.

The procedures in the list are tried one at a time until either a procedure returns TRUE or the list is empty (in which case it returns FALSE). Similarly, argument lists may be applied to a single procedure until TRUE or FALSE is returned. Therefore, the evaluation of a possibilities list is the evaluation of a disjunction of procedures or the same procedure with different arguments. The user can determine what procedure and/or arguments suceeded by looking at the system variable "LASTPOSSIBILITY". The interesting aspect of these possibilities lists is that they may be used dynamically, whereas normally procedures and arguments once created are static.

The following example illustrates the interpretation of a possibility list.

```
plist_[F1,F2,...,Fn];
arglist_[a1,a2,...,am];
WHILE arglist NEQ null DO
        IF APPLY(plist,arglist,)
              THEN PUT lastpossibility INTO thingstodo;
```

The possibilities list may also specify procedures and arguments in different contexts as

```
a_[a1,<alpha>a2,<beta>a3,...,an].
```

Parentheses may in general be used to delimit the range of a context specification. For example,

```
a_[a1,<alpha>(a2,a3),...,an].
```

7. Pattern Directed Invocation

As is mentioned in [ShapB74], there are times when implicit relationships should exist between data types. One could have data driven invocation rather than have explicit calls to procedures. This can be especially useful when unexpected events occur, e.g. by-products are produced. In CELMOD this can be useful for invoking high level routines from low level procedures. To accomplish this a pattern may be specified as a prefix (between two "%"s) to a functional definition.

A pattern is an expression which evaluates to a string or Boolean expression of strings which has the internally created associative triple:

"PATTERN" XOR <pattern Boolean> EQV <list of procedures>.

This triple is created whenever a function is defined that specifies a pattern. Procedures associated with the pattern may be selectively activated by the procedure ASSERT(pattern, args). The pattern Boolean and an optional pattern variable list (? variable instantiation list) are embedded in a pair of "%"'s. The optional pattern variable list allows arguments to be passed to procedures that are being activated by the pattern invocation. The syntax for the pattern prefix is:

```
        <pattern prefix>::= % <pattern Boolean> ,
                <pattern ?v-list> %
        <pattern Boolean>::= <disj. pattern Boolean> |
                <pattern Boolean> OR <disj. pattern Boolean>
        <disj. pattern Boolean>::= <neg. pattern Boolean> |
                <disj. pattern Boolean> AND <neg. pattern Boolean>
        <neg. pattern Boolean>::= NOT <string expression> |
                ( <pattern Boolean> ) |
                NOT ( <pattern Boolean> ) |
                <string expression>
        <pattern ?v-list>::= <pattern ?v-list> , <?-ident> |
                <?-ident>
        <?ident>::= ? identifier
```

A pattern Boolean might be:

```
        %(("lobated" AND "nucleated") OR ("nucleated" AND
                "granulocytic"))% - implies polymorph.
```

Negative expressions are very useful, as for example

```
        %"NUCLEUS" AND "CELL"% - implies white cells.

        %NOT "NUCLEUS" AND "CELL"% - implies non-nucleated cells.
```

where the above are pattern directed invocation prefixes used in procedure definitions.

A pattern prefix variable list might be:

```
        (?x,?y,?z).
```

Arguments to be supplied by the invocation mechanism (see ASSERT below) are denoted by the use of the "?x" construction discussed above.

```
        FCT  %<pattern Boolean>,?x,?y% F(?x,?y)  = <f-body(?x,?y)>;
```

Those procedure names are then associated with a pattern and are entered into a system pattern list so that whenever the pattern is ASSERTed, an internal APPLY routine may (if, for example, their value is TRUE) evaluate the set of procedures which have that pattern. Argument lists are instantiated with pattern data through the ASSERT construction.

A pattern may be a pattern name or a Boolean expression of pattern names. The ASSERT procedure causes the pattern mentioned to be made TRUE while DEASSERTed patterns are assumed FALSE.

```
<pattern-D-I expr>::= <pattern request expr> |
        ASSERT ( <pattern Boolean>, <pattern ?v-list>) |
        DEASSERT ( <pattern Boolean> ) |
        ERASE ( <pattern Boolean> )
```

The pattern invocation functions are:

ASSERT(pattern,?args)	-	which makes the specified pattern TRUE and associates the arguments with it. If a pattern Boolean is logically true, then the associated procedures are evaluated on the associated arguments (see PROG).
DEASSERT(pattern)	-	set a pattern to FALSE so that the pattern may no longer hold true in later Boolean patterns. This will not back up previous pattern directed invocations, but will prevent future ones. It is also useful in algorithms for proving the negation of patterns.

ERASE(pattern) - Remove pattern from data base.

The ASSERT and conjuctive boolean pattern may be illustrated by the following example:

```
%"LOBATED" AND "NUCLEATED",?x% POLY(?x)
```

Here, if a bottom-up shape procedure finds a lobated nucleus, and labels it as such, a proof of a poly will be attempted on the argument instantiated by the one of the calls to ASSERT when all pattern prefixes that make the pattern Boolean true are ASSERTed. This may be accomplished by

```
ASSERT("LOBATED", blob9);
```

and

```
ASSERT("NUCLEATED", blob9);
```

In addition, primitive pattern directed invocation may be invoked if specific patterns or relational triples are added or deleted from the data base. This is effected with the following commands:

```
<pattern request expression>::= <IF-PDI>
        <pattern expression> THEN <expression>
<IF-PDI>::= IFADDED | IFERASED
```

For example,

```
IFADDED <pattern expression> THEN <expression>>;

IFERASED <pattern expression> THEN <expression>>;
```

Note that these forms are really special cases of the general pattern function definition.

8. Modelling Contexts

The addition, deletion or replacement of a rule or of data may be viewed as a change in the axioms for which a model holds. Generally, this is viewed as an iteration toward a more informed model. In PRDL, this is performed by labelling all functions (rules) and data as to a particular "context" to which they are said to belong. If the contexts are represented by nodes, then manipulating contexts corresponds to inserting, deleting or replacing nodes or group of nodes in the graph world model. The ability to segment this graph structure is imperative. Without a context graph segmenting mechanism the graph structure becomes too cumbersome for a user. Thus, concepts must be placed into their appropriate context blocks, isolated to some degree from other blocks. This accomplishes two needs of the user. First it reduces the amount of interaction among elements of the data base. Second it makes it easier for the user to grasp the contents of the world model.

The context mechanism permits a segmentation of the data base into logical components called contexts as in [McD72]. The contexts may contain static and procedural data peculiar tc a particular part or way of looking at the model. By partioning the data in this fashion, interaction among the data base elements and consequently search time are reduced. Also, from a conceptual point of view a user is able to keep track of the data base by compartmentalizing his ideas.

The context mechanism is organized in a tree structure.

The root of the tree contains the initial information while its offspring contain newer information. Thus, if one is looking at a particular node of the tree, the model contents of that node are visible, plus the model contents of all nodes in the branch leading from the node to the root except for those items in the ancestor nodes which are different from the items in the descendent nodes.

Thus, one may define the current context as a the result of a merging of the context blocks from the one currently being examined to the root context block. Conflicting information contained in a descendent overrides the information in an ancestor. By permitting the tree to branch out one may have several contexts active at one time, but only one is currently visible, namely the branch which is pointed to.

The merging operation may be further described in the following way:

```
Let C(i) be the i'th node in the branch of a tree.
Let C(i-1) be the immediate ancestor of the i'th node.
Let R(i) be the set of all objects in C(i-1) that  were
                redefined in C(i-1).
Let A(i) be the set of all new objects added to C(i-1).
Let D(i) be the set of all objects deleted from C(i-1).
Then a merge between two levels is defined recursively:

C(0) = [initial data base]
C(i) = A(i) UNION [C(i-1) - R(i) - D(i)]
```

This mechanism goes under the assumption that relations with the same name have been updated as well as excluding older functions which are not found in the new context. When a merge occurs a temporary context is produced which contains the updated information. The context then gets merged with its ancestral context and continues until the root of the tree is reached. Copies of all the properties of the relations should be made as the merging proceeds. These temporary contexts should not necessarily be destroyed, because of the work expended to create them. They can become roots of other context trees.

The tree formalism eliminates some of the overhead to the user in keeping track of his data base, e.g., his most recent assertions and denials.

A model context contains user defined procedures, static constants, variables, patterns and associative triple relations. These are said to hold in the context they are defined in and in other contexts inside of their range Thus a procedure (or data object) name may actually correspond to two different procedures (or data objects) in different contexts.

On creating a new context, with the CREATE operator, functicns (or data objects) not redefined are carried over from the last context visited. Functions (or data objects) which are redefined are used in the new context and may be fixed in it with the REMEMBER operator. A context has a name and may thus be referenced.

```
<context expression>::= CREATE <context identifier> |
    REMEMBER <list expression> IN <context identifier> |
    RESTORE <context identifier> |
    FORGET  <list expression> IN <context identifier> |
    MERGE  <list expression> INTO <context identifier> |
    LASTCONTEXT | CURRENTCONTEXT
<context identifier>::= identifier
```

The semantics is given as follows:

CREATE c - create a new context called c from the current context. Return name of the new context.

REMEMBER {a1,a2,...,am} IN c - creates a new context c, if it does not already. exist, and "preserve" a1 through am in it.

LASTCONTEXT - restore the last context accessed, erasing those procedures, etc. which were not REMEMBERED in the current context.

RESTORE c - go to an old context c from the current context. Functions defined in the current context are deleted unless they are "fixed" by doing a REMEMBER operation.

FORGET {a1,a2,...,am} FROM c - removes a1,a2,...,am from context c.

MERGE {a,...,y} INTO z - merges (and deletes) contexts a through y into context z.

CURRENTCONTEXT - Return the name of the current context as a quoted (@) variable.

A context may also be used in the partial evaluation of a function or data object where the definition of the subexpression is gotten from the specified context, but the current context is returned to for the remainder of the evaluation. A pattern context is specified in "<", ">" brackets preceding the function or data object. For example,

<alpha>A(x,y) - context alpha applies to whole function,

or

A(<beta>x,y) - context beta applies only to x,

or

B(x,<gamma>A(y)) - context gamma applies only to A(y).

Context names themselves may be used as context variables in a recursive manner. For example, let normal be an existing context name then

```
        c1_@normal;
or
        REMEMBER c1 IN alpha;
        RESTORE alpha;
        <EVAL(c1)>Q - the context of Q is specified by variable
                               c1.
```

This permits dynamic model domains by allowing the shifting of the contexts under which the models hold.

9. Weighting Functions

PRDL syntax also allows the specification of dynamic weighting functions for use in determining the order in which procedures in a Boolean expression should be evaluated - or if they should be evaluated at all. The weights are evaluated before the associated functions. Then the functions are evaluated according to the results of the weighting function evaluations. If a weighting function is FALSE (0) then the associated function is not evaluated. If two weighting functions W1 and W2 evaluate to w1 and w2 with w2>w1, then the function associated with W2 will be evaluated before that associated with W1. The weighting function expression is either disjunction or conjunction of functions.

```
<weight prefix>::= ! <expression> !
```

This can be illustrated as follows. Let F(y), G(y), and H(y) be weighting functions (FCT with the FCT construction); then the phrase

```
!F(y)!SHAPE(x) OR !G(y)!AREA(x) OR !H(y)!TEXTURE(x)
```

might evaluate SHAPE, AREA or TEXTURE first depending on which has a higher weight. Weighting functions are useful only when the associated procedure to be evaluated is much more costly to compute than the weighting function. In the above example, the weighting functions (F,G,H) are computed first and then the function associated with the highest weight is selected.

For example, this might be the case in the above example if the object y being looked at had a simple predicate test for correct AREA, a more complex predicate for SHAPE, and a most complex predicate for TEXTURE. As the weighting functions are really a function of the top level object being described, the weights are usually heuristically adjusted to the global description rather than being local property values.

Weighting functions may also evaluate to Boolean values. If a weighting function returns TRUE it will always cause its associated function to be evaluated. If the weight returns FALSE, its associated function is never evaluated.

Conjunctions of weighted procedures are also allowed, in which case the weighting function determines only which function to try first since all functions must be satisfied.

In a sense, the use of weighting functions is an efficient mechanism for the coding of multiple models using the same basic model viewed in different global (weighting function) contexts.

9.1 Weighting functions to implement levels of abstraction

Sacerdoti [Sac74] uses a concept similar to dynamic weighting functions he calls "planning in an abstraction space". The problem is expressed as a goal. with "criticality values" associated with the various parts of the goal. Thus on solving the problem, one starts by attempting to prove those parts with only the highest criticality values, ignoring temporarily the components with lower criticality values. The problem is then solved recursively at successively lower hierarchies of abstraction taking into account those components with successively lower values of criticality.

The use of weighting functions in goal evaluation corresponds to the evaluation in an abstraction search space. For example: Given the criticality function C and subgoals AREA, SHAPE, and TEXTURE,

```
where: C(y,n) = if y>n then TRUE else FALSE;
```

then,

```
!C(y,2)!SHAPE(x) AND !C(y,1)!AREA(x) AND !C(y,3)!TEXTURE(x);
```

Depending on the criticality value y of the level of abstraction, the conjunction would in order of highest criticality consider

1. texture
2. texture, shape
3. texture, shape, area.

10. Generator procedures

As in CONNIVER and SAIL, in PRDL one has the ability to search the data base using generator procedures where one maintains the state of the generator in between calls to it. For example, the generator could supply (one per call) a set of procedure or variable names. Such operators as CONNIVER's AU-REVIOR/ADIEU, or SUCCEED/FAIL with matching procedures in SAIL allow the generator procedure to be reactivated where it last left off. This is useful when doing a FOREACH search where a generator function may be used as in the <element list>. The function can be restablished for entry at the beginning by either executing ADIEU within the body of the generator function or by specifing the function reset external to the generator by ADIEU <function name>.

PRDL implements generator procedures explicitly by using a construction similar to CONNIVER. For example,

```
FCT GEN1 =
FOREACH x SUCHTHAT  (NUCLEUS XOR x EQV LARGE)  DO
        BEGIN
        <SOME PROCESS PRODUCING A VALUE OR FALSE>;

        AU-REVOIR;

        Comment ------------------------
                returns here when FOREACH next
                requested. Control returns just
                after this point, but within the
                range of the DO. The function
                can be reset when ADIEU is executed.
                ------------------------;
                Control is put here after an AU-REVOIR;
        END;
```

11. RTPP functions

The ability to define and invoke RTPP [Lem77a] functions is required for image processing to be useful in such a interactive computing environment. These RTPP functions are written in MAINSAIL [Wil75]. The output of the MAINSAIL compiler (on the PDP10) is sent to the RTPP where the RTPP assembler GPPASM [Lem76b] assembles a load module for the RTPP. See ([Lem77a], [Lem76b]) for the definition of RTPP operators and assembly language. RTPP functions will either affect the RTPP system state of the RTPP/microscope system or request a function be computed by the RTPP. The resultant value (be it a scalar or list) is then returned to PRDL to be used as with any other PRDL procedure.

The following is an example of a GPP function. When evaluated, it returns a numeric value for the area of the specified object.

```
FCT GPP AREA(BMi,BMj)=

        Begin "Compute area"
<       "..............................................."
        BEGIN "area"
        "Compute area of BMi under mask BMj"
        INTEGER ioword1,ioword2,x,y,area;

        "[1] Reset the area counter"
                area_0;
"               setup line buffer channel information"
        ioword1_(256 words,256 lines,horiz,8-bit,read,I1,BMi);
        ioword2_(256 words,256 lines,horiz,8-bit,read,I2,BMj);

        "[2] Get the lines for the image and its mask"
        For y_0 Step 1 Until 255 Do
                Begin "Process line"

                STARTCODE "get lines"
        BMIO ioword1,y,LINE
        BMIO ioword2,y,LINE
                END "get lines";

        "[2.1] Position the line buffers' dynamic address
        vectors at left edge of the screen"
        XRST_($I1 Lor $I2 Lor $X);

        "[2.2] Compute masked area for central neighborhood"
        For x_0 step 1 Until 255 Do
                Begin "Process pixel"
                If (I10 Land I20)
                        Then area_area+1;
                End "Process pixel";

        End "Process line";

        "[3] Send data back to PDP8e via 8e/GPP channel
             and forward it to PRDL"
                OUT(8eHSP:,("area="&CVS(area)&CRLF);

        "Signal Mainsail runtime that the function is done"
                HALT_Done;

        END "area";
        "..............................................."
>;

        "PRDL now reads in ASCII data from the high speed
         channel."
        Intscan(8eHSP);
        End "Compute area";
        return a real number"
```

The AREA procedure, once defined, could then be used to compose other PRDL functions as:

```
        FCT NUCLEARAREA(BMi)=
                AREA(BMi,NUCLEARBLOB(BMi));
```

where the RTPP function NUCLEARBLOB(BMi) computes a mask and stores it in BMj and returns the name of BMj. That is,

```
        FCT NUCLEARBLOB(BMi) =
                            IF (BMj_FINDNUCBLOB(BMi))
                                AND  NUCAREA(BMj))
                                    THEN RETURN BMj ELSE RETURN NIL;
```

Given a number of such RTPP functions and the ability to easily add new ones, one can compose more complex modelling functions such as:

```
        FCT MONOCYTE(x)=
                NUCLEUS(X)<(b1 GE NAREA(X) LT a1)
                         AND (b2 GE NSHAPE(x) LT a2)
                                .
                                .
                                .
                         AND (bn GE NTXTUR5(x) LT an)>
           AND CYTOPLASM(X)<(d1 GE CAREA(X) LT c1)
                         AND (d2 GE CSHAPE(x) LT c2)
                                .
                                .
                                .
                         AND (dn GE CTXTUR5(x) LT cn)>
```

12. PRDL Processes

A PRDL process is a user defined function that may 'appear' to run in parallel with other PRDL functions. Since in reality these functions will be able to run only on a single CPU, actual parallelism is not possible. A schedular within PRDL will produce the parallel effect. The syntax for processes was derived from that of SAIL.

A process can be in any one of four possible states:

1. RUNNING - the process has control over the CPU.

2. READY - the process is waiting for a chance to get control of the CPU. The process would run currently if another process was not in control of the CPU.

3. SUSPENDED - The process has been halted and is waiting for an external signal from another process to begin execution.

4. TERMINATED - the process has been removed from existence. It can no longer run.

12.1 Process control operators

Several primitives are required to manipulate processes. These are defined in the following BNF syntax and discussed below.

```
<process expression>::= SPROUT ( <process name> ,
                <process function> , <options> ) |
        TERMINATE ( <process name> ) |
        SUSPEND  ( <process name> ) |
        RESUME ( <process name> , <return name> ,
                <options> ) |
        JOIN ( <set of process names> )
```

1. SPROUT(<process name>,<process function>,<options>) - <process name> is assigned to <process function>. This process name is the name of the created process (and associated arguments in <process function>). Options include:

a. Time quantum to be run based on clock.
b. Priority level.
c. Switch indicating whether or not newly created processes should be suspended.
d. A switch indicating whether or not the process in which the SPROUT occurred should be suspended.
e. A switch indicating whether or not to continue to run the process in which the SPROUT occurred.

The default option is that the process in which the SPROUT statement occurred will revert to ready status and the newly sprouted process will become a running process.

2. TERMINATE(<process name>) - terminate process.

3. SUSPEND(<process name>) - set the state of the indicated process to suspended. If the process suspended is currently running then suspend it and the schedular will find another process to run. A message will be sent to the user indicating an error if the process suspended has already been terminated.

4. RESUME(<process name>,<return name>,<options>) - provides a means for one process to restore a suspended process

to ready/running status while at the same time communicating a <return name> variable to the awakened process. The process doing the resuming may specify what its own state should be by the use of <options>. Note that RESUME returns a value. This value is the <return name> of the process doing the resuming. Thus, the only way a suspended process will receive a message upon resumption is if it was suspended with the RESUME primitive. The options are:

a. Current process will not be suspended but made ready.
b. Current process will be terminated.
c. Current process will not be suspended but will be made running. The newly resumed process will be made ready.
d. Newly resumed process will be made ready instead of running and if (c) is not true, then rescheduling occurs.

The default is that the current process is suspended and the newly resumed process will be made running.

5. JOIN(<set expression of process names>) - the current process is suspended until all of the processes of a set are terminated. One must be careful not to cause infinite loops.

12.2 Process scheduling

Whenever the currently running process performs some action that causes its status to change (to ready, suspended, or terminated) without specifying which process to run next, the schedular will be ASSERTed. It chooses a process from the pool of Ready processes using a round-robin ordered priority list. The one chosen becomes a running process. If there is nothing to run then an error message is printed. Normally, the teletype listen loop is the initial process.

12.3 Safe region declarations

It may be desirable to have certain sections of function code have safe regions where operations such as SUSPEND, TERMINATE or schedule (quantum timeout) would not be operative until the safe regions are passed. This corresponds to turning off interrupts during critical operations of a system. The operators are SAFE and UNSAFE with UNSAFE being the default.

13. Modelling using composition of procedures

Modelling may be performed using composition. For example, theorems about groups of cells may be created using previous definitions of parts of cells:

```
        FCT CELLTYPE(x)=
                IF( NOT NUCLEUS(x))
                        THEN ASSERT(nonnucleated,x)
                        ELSE ASSERT(nucleated,x);
or it could return
                        THEN FALSE ELSE TRUE;
```

CELLTYPE(x) could evaluate either to a pattern directed invocation or tc TRUE/FALSE depending on the definition. One might want to ASSERT the pattern "nucleated" or "nonnucleated" so that procedures looking for these patterns might be ASSERTed.

Some functions are for effect while others are (as in the above examples) for computation. For example, to post an OMNI picture DPOST is evaluated. To control the RTPP microscope stage, one might want to move the x stage direction by +5 stage steps. Then the DDTG function MOVESTAGE(+5,SX,'RELative') would be evaluated. This would call on the RTPP to move the stage by executing a command in DDTG.

14. Top down and bottom up procedural definition

The system has two modes of defining procedures. The normal way of defining procedures in an interactive system is bottom up. That is procedures which are used in the composition of new procedures must be defined in the system before the new procedures are evaluated.

By declaring TOP-DOWN-MODE, one waves this type of protection with the advantage that top down description may proceed. In bottom up mode reentered by typing BOTTOM-UP-MODE if one tries to compose a procedure from one not defined, an error message will be returned.

Top down mode will let you define the procedure, flagging any undefined procedures as such in the symbol table. The message

"PROCEDURE --- UNDEFINED!"

may be turned off by typing NO-TOP-DOWN-MESSAGES.

Note that neither bottom up nor top down will evaluate an undefined procedure. Top down mode will, however, on detecting an undefined procedure in the pushdown stack, save the stack, request a definition from the user, then restore the stack with control continuing from the procedure (or variable) just defined. This form of operation faciliates incremental model building.

15. Special Grammatical Functions

Since PRDL's syntactic front end is constructed from a parser generator ([ShapB76], [ShapB77a]), special subgrammars may be included which function on totally different syntactic elements than the PRDL language. In particular, a subgrammar may be defined which describes the shape of an object. Thus, when a string describing the shape of RNA strands, leukocytes or red blood cells for example, is scanned the parse will note special features and via the semantic mechanisms of PRDL indicate where these features lie on the contour. Since the grammar describes the object, it becomes possible to alter the description or note other features by altering the grammatical rules.

The following is a subgrammar that detects feature vertices and vertex bases on RNA contours. The numbers preceding the rules indicate semantic actions that should be taken when a reduction occurs, e.g. mark the current segment as a vertex point. The numbers that appear after a rule specify token numbers. The grammar shown here produces a list of critical points indicating in what segments the vertices or vertex bases appear. It should be noted that the primitive elements for the descriptive string were derived from the circle transform descriptor [ShapB77b].

```
52$<EXPRESSION>--> <RNA>$
<RNA>--><SEGMENT>$
<SEGMENT>--><VERTEX>$
<SEGMENT>--> <SIDE>$
<SEGMENT>--> <SIDE> <VERTEX>$
55$<SIDE>--> <VERTEX> <SIDE>$
```

```
<SEGMENT>--> XCORNER$5
<SEGMENT>--> <SIDE> <SEGMENT>$
<SEGMENT>--> <SEGMENT> <SIDE>$
<SEGMENT>--> <SEGMENT> <VERTEX>$
<SEGMENT>--> <VERTEX> <SEGMENT>$
55$<SIDE>--> <VERTEX> <BREAK>$
<VERTEX>--><VERTEX> <A>$
55$<SIDE>--><VERTEX> <B>$
55$<SIDE>--> <VERTEX> <C>$
55$<SIDE>--> <VBASE>$
<A>--><A> SHORT$6
<BREAK>--> <B> SHORT$6
50$<VERTEX>--> <BREAK> <A>$
55$<SIDE>--> <BREAK> <C>$
<LXCORNER>--> XCORNER <BREAK>$5
50$<VERTEX>--> <LXCORNER> XCORNER$5
55$<SIDE>--> <LXCORNER> <BREAK>$
55$<SIDE>--> XCORNER <VERTEX>$5
55$<SIDE>--> <LXCORNER> <C>$
55$<SIDE>--> <SIDE> <BREAK>$
55$<SIDE>--> XCORNER <B>$5
55$<SIDE>--> <SIDE> <C>$
55$<SIDE>--> <SIDE> <B>$
50$<VERTEX>--><A>$
55$<SIDE>--> <B>$
55$<SIDE>--> <C>$
54$<VBASE>--><D>$
<VBASE>--> <VBASE> <D>$
<D>--> <D> SHORT$6
51$<A>--> XSHARPCURVE$3
51$<A>--> XSHARPCORNER$4
51$<B>--> STRAIGHT$7
51$<B>--> XMODCURVE$17
51$<B>--> VMODCURVE$15
51$<B>--> VMODSHARPKNEE$19
51$<B>--> XSHARPKNEE$14
51$<C>--> VCORNER$13
51$<C>--> SMOOTH$11
51$<D>--> VSHARPCORNER$12
51$<D>--> VMODSHARPCURVE$18
51$<D>--> VSHARPCURVE$16
```

A file containing a string shape description of an object may be executed via the EX command of PRDL. The result is a parse of the shape and in the current implementation the number of features that were found in an RNA contour is the value of the string.

```
*EX("RNA4.DES")$
= 7
```

Higher level PRDL functions may be used with this technique. For example, objects or parts of objects may be placed in sets denoting various shape catagories. These may be further manipulated by PRDL functions. Pattern directed invocation may also be employed when a specific shape feature is detected.

16. Interactive command and function editors

The normal (non-edit) top level teletype input is normally terminated with an escape character. A low level editing capability is built into the top level teletype interface for command entry.

```
^A - advance line pointer and print out the edit line.
^B - backup the line pointer and print out the edit
        line.
^D - delete the current edit line if it exists.
^E - print this message.
^F - print the front part of the buffer up to but not
        including the current line.
^K - delete the entire edit buffer.
^N - print the current line.
^P - print the entire edit buffer.
rubout - delete the previous character (works across
        crlfs).
backspace - same as rubout.
escape - terminate input and cause PRDL to evaluate it.
```

As commands are entered into the PRDL system, parsed, and executed, one has occasion to redo the last command or previously entered function. Therefore, the last command entered will be saved in the string variable LASTCMD. To edit and reexecute it, a simple string editor may be used to edit, save and execute different strings. The editor is entered by evaluating

```
EDIT <function  name>  or <string variable>.
```

If no string name is given, then LASTCMD is assumed. This allows functions to be edited and new functions to be created from old without constant retyping.

The editing commands for the EDIT operator given below in table 1 are similar to a subset of those in the TECO editor and the ALTER function in the PDP10 SOS text editor (both editors are used on the PDP10). In the following table $ denotes the escape character.

Table 1. EDIT commands

B	Move the pointer to the beginning of the string.
nC	Space forward(+n)/backward(-n) characters, default 1.
nD	Delete next(+n)/last(-n) characters.
E<file>$	Read in the string contained in the specified file.
F<text>$	Find the next occurance of <text>.
I<text>$	Insert text up to the escape character.
J	Break line and stick rest at front of next line.
nL	Move the pointer n lines forward, (-n backward), 0 current line.
nP	Print n lines forward (n>0) or n lines backward, (n<0). If n=0 then print the current line.
Q	Quit the EDIT and return to PRDL.
nR<t1>$<t2>$	Replace next n forward occurances of text t1 with text t2.
nS<char>	Skip forward(+n)/backward(-n) occurances of <ch>, default 1.
nT<n1>$<n2>$	Transfer n lines starting at line n1 to after line n2.
nW	Skip forward n words (or backward -n words), default 1.
X<name>$	Exit the EDIT, saving the string in the function or string being used or optionally specified.
Z	Goto end of string.
Ctrl/U	Erase current line being input in I<text>$ mode.
.	The value of the current line.

The PRDL command EXECUTE may be used to evaluate a command string previously edited with EDIT.

17. The PRDL evaluator

As can be seen, the PRDL evaluator differs somewhat in form and in function from previously existing high level languages such as LISP, PLANNER, MLAB etc. These differences, although not major, prevent the use (without major modification) of one of these existing languages for a real time multiprocessor interactive system. The special features of PRDL are listed below:

(a) Procedure definition is in infix form which does exist in MLAB but not in LISP or the PLANNER like languages.

(b) Pattern matching techniques are used with argument and procedure variables to implement pattern directed invocation of procedures and data. This does not exist in MLAB or explicitly in LISP, but does exist in PLANNER and CONNIVER.

(c) Contexts, which may themselves be functions which return a context name, serve as filters in model building. CONNIVER has explicit mechanisms for performing this while the other languages do not.

(d) The definition and evaluation of procedures on special purpose processors means that PRDL must maintain a multiprocessor envirnment. Such an envirnment must ensure that the connection between these processors and PRDL is a minimum burden on the user.

(e) Interactive graphics functions (at a lower implementation level than in MLAB) permits the user complete flexibility in designing graphical structures. The other languages do not have built-in interactive graphics.

(f) Because the language has a very flexible front end i.e. parser, special forms may be defined and incorporated in the language. For example, a shape grammar.

17.1 PRDL control structure

In discussing the PRDL control structure it is assumed that the special purpose processors are slave processors of the PDP10.

The PRDL control structure is a graph. Each node within the control structure graph can be one of several possible data types. They are:

a. Object label (name of an entity)
b. Static fact such as value(s) or procedure body.
c. Pointer to a special purpose processor program.
d. Pointer to context blocks (which subgraphs are active).
e. Pointer to other nodes in the graph (as lists etc.).

The graph is both implicit and explicit. An explicit relation is defined to be a set of nodes where the arcs joining the nodes are specified directly in the graph structure itself. An implicit relation is defined to be a set of nodes which exist because of pattern directed invocation, associative search, or function computation. Thus, explicitly defined functions determine explicit relations within the graph, whereas relations which are determined by the data and through associative search form the implicit relations.

The ability to have implicit relations allows the defining of relational models which are not complete in the sense of 1st order predicate calculus. During interactive problem definition one tries to make these relations complete by adding new relations and assertions to the data base. The ability to implicitly define relations allows a user to concern himself minimally with the effects that the addition of a new concept into a world model will have. Of course, such inattention to completeness may cause problems regarding the validity of the model due to the introduction of inconsistancies.

The graph structure of the modelling process forces the user into thinking in a structured programming framework [Dij71] because the syntax of formal function composition definition must be followed. The formalism of the graph structure imposes a formalism on the definition facilities of the system, thereby reducing the tendency one might have to produce a set of inconsistant ad-hoc programs.

18. Semantic condensation of plans - future work

Eventually, as more experience is gained with the definitions, a subgraph may be combined into what might be called a "fat" node. This process is called "semantic condensation" whereby subgraphs are condensed into algorithmic plans. Program synthesis through procedures written in PRDL using the transformation of implicit relations to explicit relations might be used to perform the condensation and create single special processing procedurs. Such procedures previously would have been invoked through much interaction with PRDL. This subgraph embodies those procedures which were previously invoked separately, The creation of such a "fat" node implies that the user has sufficient confidence in the procedures contained in that node that the "fat" node might be invoked instead of the subgraph. This may be considered to be a form of plan saving. Thus implicit recommendations are made explicit.

A plan may be defined to be a sequence of procedure names which are given to the special purpose processor. This sequence may contain iteration mechanisms and recursive mechanisms to be evaluated by the special purpose processor. Thus, a plan is effectively a high level command program to the special purpose processor which is synthesized at run time by PRDL based upon the internal and external influences. Parts of a plan sit in the control structure nodes waiting to be synthesized with the proper instantiations.

Control of the special purpose processors is really determined by the paths that the PRDL chooses through interpretively traversing the graph structure. As each node is encountered either a program will be activated on the special purpose processors or local processing will take place.

19. References

Ber64. Berkeley E, Bobrow D:The Programming Language LISP: Its Operation and Applications. M.I.T. Press, Cambridge, Mass., 1964.

Carm74. Carman G, Lemkin P, Lipkin L, Shapiro B, Schultz M, Kaiser P:A real time picture processor for use in biological cell identification - II hardware implementation. J. Hist. Cyto. Vol 22, 1974, 732:740.

CCB76. Computer Center Branch:DECsystem-10 Omnigraph Display Manual. NIH, Bethesda, Md., 20014, April 1976.

DEC74. Digital Equipment Corp.:OS/8 handbook. DEC, Maynard, Mass., 1974, DEC-S8-OSHBA-A-D.

Dij71. Dijkstra E:Hierarchical ordering of sequential processes. Acta Informatica 1, 1971, 115:138.

FelJ72. Feldman J, Low J, Swineart D, Taylor R:Recent developments in SAIL. SAI memo June 5, 1972.

Hew72. Hewitt C:Description and theoretical analysis (using schemata) of PLANNER:A language for proving theorems and manipulating models of a robot. Thesis, MIT, AI-TR-258, 1972.

Kno73. Knott, G, Reese, D:MLAB - An on-line modeling laboratory. NIH, DCRT, Dec., 1973.

Lem72a. Lemkin P:A Simplified Biological Cell World Model for Question-Answering using Functional Description. Univ. Maryland

Scholary paper 75, May 16, 1972.

Lem72b. Lemkin P F:An extended LISP 1.6 system with the ability to page procedures using a working set model.Unpublished report for Univ. Md. course CMSC 838b, Dec. 1972.

Lem74. Lemkin P, Carman G, Lipkin L, Shapiro B, Schultz M, Kaiser P:A real time picture processor for use in biological cell identification - I systems design. J. Hist. Cyto. Vol 22, 1974, 725:731.

Lem76a. Lemkin P:Functional specifications for the RTPP monitor and debugger - DDTG running on the PDP8e/RTPP. NCI/IP-76/02, Technical Report #2, NTIS PB250726, Feb, 1976.

Lem76b. Lemkin P, Shapiro B, Lipkin L, Schultz M, Carman G:A PDP8e Assembler for the General Picture Processor. NCI/IP Technical Report #16, Dec. 1976.

Lem77a. Lemkin P, Carman G, Lipkin L, Shapiro B, Schultz M:The Real Time Picture Processor- Description and Specification. NCI/IP-76/03, Technical Report #7, NTIS PB252268/AS, March 1976. (Revised TR-7a, June, 1977).

Lem77b. Lemkin P:Buffer Memory Monitor System for Interactive Image Processing. NCI/IP Technical Report #21, Dec. 1976. (Revised TR-21a, June, 1977).

McD72. McDermott, D, Sussman, G:Conniver Reference Manual", MIT AI memo 259, May, 1972.

Mes70. Mesztenyi C:FORMAL - a formula manipulation language.

Univ. Md. report TR70-133, Sept. 1970.

Qua68. Quam L:Standford LISP 1.6 manual. S.A.I 28.2 Dec. 31, 1968.

ReisJ76. Reiser J: SAIL User Manual. Stanford Artifical Intelligence Laboratory memo AIM-289, August 1976.

Sac74. Sacerdoti E D:Planing in a hierarchy of abstraction spaces. Artificial Intelligence. Vol 5, 1974, 115:135.

ShapB73. Shapiro B:A Survey of Problem Solving Languages and Systems. Univ. Md. report TR-235, March, 1973.

ShapB74. Shapiro, B, Lemkin, P, Lipkin, L:Application of artificial intelligence techniques to cell identification. J. Hist. Cyto Vol 22, 1974,741:750.

ShapB76. Shapiro B:A SLR(1) parser generator. NCI/IP-76/01, Technical Report #9, NTIS PD249127/AS, Feb 1976.

ShapB77a. Shapiro B: Language processor generation with BNF inputs: methods and implementation. Comp. Prog. in Biomed. Vol 7, 1977,85:98.

ShapB77b. Shapiro B. and Lipkin L: The circle transform: an articulable shape descriptor. Comp. in Biomed. Res. In Press. 1977.

ShapB77c. Shapiro B:The PDP11/20 Message switcher. NCI/IP Technical Report #17, In prep.

Ten74. Tenenbaum J M, Garvey T D, Weyl S, Wolf H:An

Interactive Facility for Scene Analysis Research. Stanford Research Institute Tech note 87, Jan. 1974.

Wil75. Wilcox C:MAINSAIL - MAchine INdepent SAIL. DECUS meeting, Languages in Review Session, 1975.

APPENDIX A

The implementation of PRDL

Although much of the PRDL language appears to be similar in syntax to MLAB and uses stack evaluation facilities similar to that of MLAB, the actual implementation is very different from that of MLAB. PRDL uses a SLR(1) parser program generator [Shap76]. The parser generator takes a BNF grammar specification PRDL.GRM and generates a SAIL source code parser PRDL.SAI (the main procedure of PRDL).

Additional procedures are used to support the parser. SEM.SAI is the set of semantic routines required by the parser which defines how the polish stack is to be built. Calls to this module occur when semantic numbers are attached to the grammatical rules in the grammar file. ACCEPT.SAI contains the user teletype interface and editor, and the symbol token lexical scanner and symbol table. PROCES.SAI is the recursive interpreter for operations on the parser produced stack. Unlike MLAB, PRDL waits until the complete input string has been parsed into a polish stack before interpretation of the stack has begun. This not only permits more efficient interpretation, but prevents partial evaluation when the input contains errors. This reduces the number of side-effects that can occur. The PROCES interpreter deals with the arithmetic, function evaluation, context, pattern directed invocation, relational search operations, as well as multiprocessing and alternate processor execution.

The existing MLAB evaluator accesses data structures pointed to by a hash coded symbol table. Auxillary tables use the symbol table index to point to a data structure type and datum by symbol table index. The MLAB function space is a set of nodes connected as a forest of tree structures. The new data structures in PRDL are similar to that of MLAB but every identifier and procedure is an item which may enter into a relational triple thus yielding much greater flexibility.

SAIL [ReisJ76] allows only 4096 items which are used in creating the elements of associations, sets, lists and matrix data. For reasonably sized models, this should be enough since not every identifier would have a corresponding item or be used in a triple, set, list or matrix.

The pattern matching is most effective if the context item <...> is stored on each symbol item in a special context field (similar to PROPS) with the last context used and its pointer being readily available from the symbol table. Thus it would be easy to filter contexts while interpreting. Pattern directed invocation is implemented through the use of lists of function names associated with the patterns.

A.1 Possible graph node data structure

The following 5-tuple data structure is used for PRDL entries in the global symbol table. This structure contains the pointer to the current value (whether real, string, set, etc.) in the current context as well as a pointer to the context pointer array used for accessing other structures in other contexts with the same symbol name.

```
                     ----------------------------------
                     | Symbol name                    |
                     ----------------------------------
                     | Context name of last access    |
                     ----------------------------------
                     | Data Type of last access       |
                     ----------------------------------
                     | Idx pointer of last access     |
                     ----------------------------------
                     | Pointer to Context array       |
(a) Symbol node      ----------------------------------
```

The context array is a n column by 2 row array.

```
        C1      ...     -----------------------  ...     Cn
                ...     | Type for context Ci |  ...
                ...     -----------------------  ...
                ...     | Idx pointer for Ci  |  ...
                ...     -----------------------  ...
```

Figure 1. Node graph structure of PRDL

The Idx pointer is the same for two contexts in which the value of the data structure is the same. Note that the Idx pointer points to the actual data structure (via either a LEAP item or function tree) in all cases except real identifiers and constants. The latter use the Idx value as the actual value.

Pointers to such symbol table nodes are then used in the constuction of tree structure PRDL functions.

Data type is either (constant, real, list, set, bracketed triple, string, function, builtin, function, pattern or context). Contexts point to the data or procedure locations by a pointer to static or procedural data (tree structures); the print name item allows the easy access of a datum by print name or item number (eg. for use in lists, FOREACH triple search etc.).

Contexts may contain pointers to older contexts which include procedural or static data, or may contain new or redefined data themselves. Allowing contexts to contain pointers permits the dynamic updating of recommendations and static data. A global symbol table will be used which will act as a directory for all named procedures, static data areas, and context blocks. Contexts may be set up in a fashion similar to CONNIVER [McD72] (i.e., tree-like) so that all contexts do not have to be specified. Contexts will be opened implicitly based upon an access point specified in a tree. That is, all ancestors in the tree are visible but descendents are not.

When a data type is defined, the context block in which it is to be incorporated is specified by the current context. A possible bookkeeping mechanism for contexts which allows the extension of the current MLAB symbol table system would associate a dynamic 1-D array with each symbol. The size of the array is that of the highest numbered context in which that symbol appeared before the context was shifted to one in which it did not appear. If it is the current context, then the size of the array corresponds to the current context number (ordinal numbers). The entry in the array corresponds to the pointer to be used by the symbol table entries at the current context. Thus if a new context was created and a procedure F was to be carried to the new context, its array must be lengthend by 1 and the contents of the previous pointer carried forward. When intermediate contexts are deleted, the corresponding array entries must be marked (with 0) to note this fact. Thus to access any object in any context, a lookup is made to the array to see if an entry exists and then to use it.

We are also considering the necessity for swapping contexts to the PDP10 disk. An algorithm was written by Lemkin for the swapping of procedures as determined by a working set model of procedure activity for LISP functions [Lem72b]. If the size of a multicontext model is extremely large, then such a mechanism might have to be considered.

APPENDIX B

PRDL Grammar and Symbol Table

The following depicts the PRDL grammar in its current form of implementation. The number preceding a grammatical rule indicates a semantic action and the number(s) following a rule is the token number of the terminal symbol(s) appearing in that rule. These token numbers are used to both identify a given symbol and to indicate which interpretive block to enter in the recursive interpreter.

```
<EXPRESSION>--><COMPOUNDSTATEMENT>$
<EXPRESSION>--> <ASSIGNMENT>$
<EXPRESSION>--> <FUNC!DESIGNATOR>$
<EXPRESSION>--> <CONDITIONAL>$
<EXPRESSION>--> <AE>$
<EXPRESSION>--> <WHILELOOP>$
<EXPRESSION>--> <FORSTATE>$
<EXPRESSION>--> <BOOLEXP>$
7$<EXPRESSION>--> TYPE <EXPRESSION>$8
41$<EXPRESSION>--> PRINTF <FUNC!NAME>$34
<EXPRESSION>--> <FUNCTION!DEF>$
<EXPRESSION>--> <BUILTIN>$
44$<COMPOUNDSTATEMENT>--> <COMPOUNDHEAD> END$41
43$<COMPOUNDHEAD>--> BEGIN <EXPRESSION>$42
45$<COMPOUNDHEAD>--> <COMPOUNDHEAD> ; <EXPRESSION>$43
33$<BUILTIN>--> <ONEARGNAME> <LPAREN> <ONEARG> )$10
33$<BUILTIN>--> <TWOARGNAME> <LPAREN> <TWOARG> )$10
33$<BUILTIN>--> <THREEARGNAME> <LPAREN> <THREEARG> )$10
33$<BUILTIN>--> <FOURARGNAME> <LPAREN> <FOURARG> )$10
39$<ONEARG>--> <EXPRESSION>$
5$<BUILTIN>--> ZERONAME$35
42$<ONEARGNAME>--> ONENAME$36
39$<TWOARG>--> <ONEARG> <COMMA> <EXPRESSION>$
42$<TWOARGNAME>--> TWONAME$37
39$<THREEARG>--> <TWOARG> <COMMA> <EXPRESSION>$
42$<THREEARGNAME>--> THREENAME$38
39$<FOURARG>--> <THREEARG> <COMMA> <EXPRESSION>$
42$<FOURARGNAME>--> FOURNAME$39
37$<FUNCTION!DEF>--> <FUNC!WORD> <REMAIN!FDEF>$
37$<REMAIN!FDEF>--> <FUNC!NAME> <FUNC!ARG> = <FUNC!BODY>$19
31$<FUNC!WORD>--> FCT$31
32$<FUNC!NAME>--> FID$0
40$<FUNC!ARG>--> <LPAREN> <ARGLIST> )$10
34$<FUNC!ARG>--> EPSILON$
35$<ARGLIST>--> ID$1
```

```
35$<ARGLIST>--> <ARGLIST> <COMMA>  ID$ 1
36$<LPAREN>--> ($9
<FUNC!BODY>--> <EXPRESSION>$
<FUNC!DESIGNATOR>--> <BUILTIN>$
<FUNC!DESIGNATOR>--> <FUNC!NAME> <ACT!ARG>$
33$<ACT!ARG>--> <LPAREN> <ACT!PARAM!LIST> )$ 10
<ACT!ARG>--> EPSILON$
39$<ACT!PARAM!LIST>--> <EXPRESSION>$
39$<ACT!PARAM!LIST>--> <ACT!PARAM!LIST> <COMMA> <EXPRESSION>$
30$<EXPRESSION>--> PSTACKSW$33
21$<WHILELOOP>--> <WHILECLAUSE> <EXPRESSION>$
20$<WHILECLAUSE>--> WHILE <BOOLEXP> DO$24,25
29$<FORSTATE>--> <FORASSIGN> <FORCONTROL> <FORLIMIT> <EXPRESSION>$
26$<FORASSIGN>--> FOR <ASSIGNMENT> STEP$28,29
27$<FORCONTROL>-->  <AE> UNTIL$30
28$<FORLIMIT>--> <AE> DO$25
8$<CONDITIONAL>--> <IFSTATE> ELSE <EXPRESSION>$13
9$<IFSTATE>--> <IFCLAUSE> <EXPRESSION>$
10$<IFCLAUSE>--> IF <BOOLEXP> THEN$14,15
6$<ASSIGNMENT>--> <LEFTSIDE> _ <EXPRESSION>$ 11
<BOOLEXP>--> <DISJUNCEXP>$
19$<BOOLEXP>--> <BOOLEXP> OR <DISJUNCEXP>$23
<DISJUNCEXP>--> <NEGEXP>$
18$<DISJUNCEXP>--> <DISJUNCEXP> AND <NEGEXP>$22
17$<NEGEXP>--> NOT <ALGREL>$21
<NEGEXP>--> ( <BOOLEXP> )$9,10
17$<NEGEXP>--> NOT ( <BOOLEXP> )$21,9,10
<NEGEXP>--> <ALGREL>$
11$<ALGREL>--> <AE> > <AE1>$ 12
12$<ALGREL>--> <AE> < <AE1>$ 16
13$<ALGREL>--> <AE> LEQ <AE1>$17
14$<ALGREL>--> <AE> GEQ <AE1>$18
15$<ALGREL>--> <AE> = <AE1>$19
16$<ALGREL>--> <AE> NEQ <AE1>$20
22$<ALGREL>--> TRUE$26
23$<ALGREL>--> FALSE$27
47$<ARRAY>--> <ARRAY!NAME> <LBRACKET> <ACT!PARAM!LIST> ]$44
<ARRAY>--> LEFTARRAY$47
46$<LBRACKET>--> [$45
42$<ARRAY!NAME>-->ARID$46
<AE1>--> <SAE>$
<AE>--><SAE>$
<SAE>--><TERM>$
0$<SAE>--><SAE> + <TERM>$6
1$<SAE>--><SAE> - <TERM>$7
<TERM>--> <FACTOR>$
<TERM>--> <FACTOR1>$
3$<TERM>--><TERM> * <FACTOR>$4
4$<TERM>--><TERM> / <FACTOR>$5
49$<TERM>--><TERM> & <FACTOR>$49
<FACTOR>--><PRIMARY>$
<FACTOR1>--><PRIMARY>$
2$<FACTOR>--><FACTOR> ^ <FACTOR1>$ 3
5$<PRIMARY>-->ID$  1
5$<PRIMARY>--> STRCONST$40
5$<PRIMARY>--> STRID$48
```

```
5$<LEFTSIDE>-->ID$ 1
5$<LEFTSIDE>-->STRID$48
48$<LEFTSIDE>--> <ARRAY>$
5$<PRIMARY>-->INT$ 2
<PRIMARY>-->( <EXPRESSION> )$       9,10
<PRIMARY>--> <ARRAY>$
<PRIMARY>--> <FUNC!DESIGNATOR>$
25$<PRIMARY>--> + <PRIMARY>$6
24$<PRIMARY>--> - <PRIMARY>$7
38$<COMMA>--> ,$32
```

The following depicts the symbol table file that is used to initialize the symbol table of PRDL in its present form. The numbers in the right most field indicate the token numbers of the terminal symbols. In some cases it will be noted that the token field is broken into two parts (comma separates the parts). This indicates a builtin function. The number on the left indicates a class e.g. one argument builtins and the number to the right indicates a function number within that class.

```
ID$         1
INT$        2
^$ 3
*$ 4
/$ 5
+$ 6
-$ 7
TYPE$     8
($ 9
)$ 10
_$ 11
>$ 12
IF$         14
THEN$       15
ELSE$       13
<$ 16
LEQ$        17
GEQ$        18
=$ 19
NEQ$        20
NOT$        21
AND$        22
OR$         23
WHILE$      24
DO$         25
TRUE$       26
```

```
FALSE$     27
#UPLUS#$6
#UMINUS#$7
FOR$       28
STEP$      29
UNTIL$     30
FCT$       31
,$ 32
PSTACKSW$ 33
PRINTF$34
DREL$      35,0
DGET$      35,-1
DCLOSE$    35,-2
DDONE$     35,-3
DDONE1$    35,-4
FASTA$ 35,-5
SLOWA$ 35,-6
FASTD$ 35,-7
SLOWD$ 35,-8
REALIN$ 35,-9
TTYLINE$ 35,-10
TTYSYMBOL$ 35,-11
TTYCHAR$ 35,-12
SIN$       36,0
COS$       36,-1
TAN$       36,-2
SQRT$      36,-3
LOG$       36,-4
EXP$       36,-5
DPOST$     36,-6
DUNPOST$36,-7
DOPEN$     36,-8
SIND$      36,-9
COSD$      36,-10
TAND$      36,-11
DAPPEND$36,-12
DKILL$     36,-13
DTEXT$     36,-14
EXECUTE$36,-15
EX$        36,-15
ASIN$      36,-16
ACOS$      36,-17
ATAN$      36,-18
OUTSTR$ 36,-19
SIGN$      36,-20
ABS$       36,-21
ASIND$     36,-22
ACOSD$     36,-23
ATAND$     36,-24
TRUNC$     36,-25
CVS$       36,-26
CVOS$      36,-27
CVF$       36,-28
TYPEIT$    36,-29
CVD$       36,-30
CVO$       36,-31
```

```
LENGTH$     36,-32
LOP$        36,-33
COP$        36,-34
REALSCAN$ 36,-35
SYMBOLSCAN$ 36,-36
DTSCALE$ 36,-37
LISTIFY$ 36,-38
SETIFY$ 36,-39
DELETE$ 36,-40
EVAL$ 36,-41
FIRST$ 36,-42
SECOND$ 36,-43
THIRD$ 36,-44
ISTRPVAR$ 36,-45
DELTRPVAR$ 36,-46
DMOVE$      37,0
DDRAW$      37,-1
DCURSOR$37,-2
DDOT$       37,-3
EQU$        37,-4
UNION$      37,-5
INTERSECTION$ 37,-6
SETDIFF$ 37,-7
APPLY$      37,-8
DPLOT$ 37,-9
DCROSS$ 37,-10
SUBSTR$     38,0
MAKETRIPLE$ 38,-1
DELETETRIPLE$ 38,-2
ISTRIPLE$ 38,-3
SUBLIST$ 38,-4
DINI$       39,0
DWIND$      39,-1
DVECT$      39,-2
STRCONST$   40
END$        41
BEGIN$      42
;$ 43
]$ 44
[$ 45
ARID$       46
LARID$      47
STRID$      48
&$ 49
SETCON$ 50
LISTCON$    51
SETID$      52
LISTID$     53
TRIPLECON$ 54
TRIPLEID$ 55
ZGARBAGE$ 56
```

APPENDIX C

Compiling and Building PRDL

PRDL is compiled with SAIL and loaded as follows including the use of BAIL as a runtime debugger.

```
.R SAIL
*PRDINV.REL/27b_PRDINV.SAI

.R SAIL
*FREEST.REL/27b_FREEST.SAI

.R SAIL
*FUNPAK.REL/27b_FUNPAK.SAI

.R SAIL
*PROCES.REL/27b_PROCES.SAI

.R SAIL
*ACCEPT.REL/27b_ACCEPT.SAI

.R SAIL
*SEM.REL/27b_SEM.SAI

.R SAIL
*CVT.REL/27b_CVT.SAI

.R SAIL
*BOUND.REL/27b_BOUND.SAI

.R SAIL
*GETABL.REL/27b_GETABL.SAI

.PARGEN
*PRDL.SAI
*PRDL.GRM
*<EXPRESSION>

.R SAIL
*PRDL.REL/27b_PRDL.SAI

.LOAD SYS:SAILOW/REL,DSK:PRDL/REL
.SAVE PRDL
```

When PRDL is started, it will request the symbol table name with PRDL.SYM being the default.

INDEX

<?-ident> 39
<a-expr> 35
<act-ident> 10
<actual arguments> 11
<actual parameter list> 11
<ae> 8
<algebraic expression> 8
<algebraic relation> 7
<AOV expr.> 18
<arg list> 10
<assignment expression> 6
<attribute> 17
<Boolean expression> 7
<builtin identifier> 11
<builtin> 11, 35
<compound expression> 6
<compound head> 6
<conditional expression> 7
<context expression> 44
<context identifier> 44
<data declar.> 10
<device> 28
<disj. pattern Boolean> 39
<disjunctive expression> 7
<element list> 18
<expr. list> 27
<expression> 6
<f-args> 10
<f-body> 10
<f-expr> 35
<factor> 8
<file I/O expression> 28
<file> 28
<finish index> 16
<FOR assignment> 9
<FOR control> 9
<FOR limit> 9
<FOR loop> 9
<FOREACH loop> 18
<funct. prefix> 10
<function definition> 10
<function designator> 11
<function name> 10
<graphics expression> 33
<I/O expression> 27
<I/O modifier> 28
<IF clause> 7
<IF expression> 7
<IF-PDI> 41

<leftside> 6
<list expression> 19
<list identifier> 19
<list operation> 19
<m-factor> 14
<m-identifier> 14
<m-primary> 14
<m-term> 14
<matrix expression> 14
<neg. pattern Boolean> 39
<negative expression> 7
<number/str conv.> 16
<object> 17
<Omni builtin> 33
<p-expr> 35
<pattern ?v-list> 39
<pattern Boolean> 39
<pattern declar.> 10
<pattern prefix> 39
<pattern request expression> 41
<PDP10 I/O expression> 29
<primary> 8
<process expression> 55
<processor declar.> 10
<processor I/O> 27
<project-programmer> 28
<relational args> 17
<relational expression> 17
<relational operator> 17
<remainder f-def> 10
<RTPP control I/O> 30
<RTPP data I/O> 30
<RTPP I/O expression> 30
<search variables> 18
<set expression> 19
<set identifier> 19
<set opearation> 19
<set-list arg> 19
<set-list operation> 19
<sirple arithmetic expression> 8
<simple matrix expression> 14
<sme> 14
<start index> 16
<string expression> 16
<string/str conv.> 16
<substring> 16
<term> 8
<TTY expression> 27
<TTY input> 27
<TTY output> 27
<value> 17
<weight prefix> 47
<WHILE clause> 9
<WHILE loop> 9

An Expression Language 5

BOOLEAN variables 25
Builtin procedures 12

Compiling and Building PRDL 84
Control of the PDP10 by PRDL 28

Deleting a variable from PRDL 26

EVAL and APPLY operators 35
Evaluation Mode of Operation 8
External processor function definitions 12

FUNCTION ARGUMENT variables 22
Function Definition and Composition 9

Generator procedures 50
GLOBAL variables 23
Graphics Operators 33

I/O expressions 27
Interactive command and function editors 63
Introduction 1
Iterative Control Structure 9

LOCAL variables 22

Matrix operators 13
MATRIX variables 24
Modelling Contexts 42
Modelling using composition of procedures 58

OUTSTR 9

Pattern Directed Invocation 38
PATTERN PREFIX variables 23

pattern-D-I expr> 40

PDI - pattern directed invocation 38
Possibility Lists 36
Possible graph node data structure 75
PRDL Grammar and Symbol Table 79

QUOTED variables 24

Relational Data Base 17
RELATIONAL TRIPLE variables 26
RTPP functions 51

Saving and Restoring a Data Base 28
SET and LIST variables 25
Sets and Lists 19
Special Grammatical Functions 60
Specification of PRDL 4
String operators 15
STRING variables 25

Teletype I/O 27
The implementation of PRDL 74
Top down and bottom up procedural definition 59
TYPE 9
Type of a variable 26

Variable Classes 22

Weighting Functions 47