THE DESIGN OF A PROCEDURELESS PROGRAMMING LANGUAGE

Clair W. Goldsmith
Palyn Associates, Inc.
San Jose, California

1. ABSTRACT

The programming of digital computers has been a major concern of mainframe manufacturers, academicians, computer users and software product manufacturers since the first marketable computers were produced. Most often, the machine execution order has been explicit at the level at which the machine is programmed.

This paper takes as a premise that source statement ordering does not have to describe machine execution order. It describes a specific procedureless programming language that requires no ordering of the source program. This language includes primitives for performing calculations on sets. In this language statements are not executable. They are rules for defining sets. The paper concludes with a discussion of the usefulness of the language for a typical programming application.

2. INTRODUCTION

Traditionally, programming languages are classified as either problem-oriented or procedure-oriented [13,24,29]. Since there is no general agreement, I first will establish working definitions. I follow Sammet [24], in defining a procedure-oriented language as one in which statements are taken to be executable and the flow of control is explicitly provided by the user. FORTRAN, COBOL, and PL/1 are examples of procedure-oriented languages. I follow Katzan[13], in defining a problem-oriented language as one restricted to a specialized application area. Both AMTRAN [22] and NAPSS [23] are examples of problem-oriented languages for use in the field of numerical computation.

Procedure-oriented languages are used to describe algorithms. The coded algorithm consists of a group of ordered source statements. These source statements must be translated into a machine-executable form such that the execution order corresponds to that described in the algorithm. Thus, the source statements control the order in which the machine-executable statements are performed. Additionally, the source language description of the algorithm involves bookkeeping functions which are not really part of the algorithm.

Procedure-oriented languages have considerable flexibility. However, this flexibility introduces additional programming detail into the algorithm description, such as: determining the end of the input data, using index variables, controlling interative loops, and assigning data storage. Such languages require programming expertise on the part of the user. For many applications, the power available in common procedure-oriented programming languages complicates, rather than aids, the task of obtaining results [24].

Problem-oriented languages used to perform calculations for specific application areas [22, 23, 25], usually consist of a set of functions which may be referenced explicitly or by the use of keywords. The functions perform calculations such as solving simultaneous algebraic equations or performing numerical integrations. Frequently, these languages contain procedure-oriented features so that the user may describe his own algorithms. Due to their narrow applicability, such languages do not have wide acceptance. In general, no effort is made by the designers of problem-oriented languages to reduce programming detail. However, some reduction of detail is a natural consequence of languages that contain functions for a specific application.

The reduction of programming detail is a central issue of software engineering [19]. Various approaches are taken. AMTRAN [22] and NAPSS [23], along with others, have automatic storage allocation. Homer [10] suggests a scheme, very similar to macro-facilities, for automatic statement sequencing. Dijkstra [6] proposes that explicit statement ordering through the use of the GOTO statement be eliminated. Balzer [2] suggests the concept of programming without considering data types.

These concepts address specific problems in current programming languages. A need clearly exists for programming languages designed with the goal of suppressing programming detail. The basis for such a language is to consider statements in the language as definitions. No statement must be placed before any other statement in the source language. The language is procedureless.

This paper will:

1. Define a procedureless language to solve non-trivial programming problems, based upon the concept of purely definitional statements.
2. Provide a set of primitive functions to facilitate programming in this language.
3. Describe the data structure necessary to permit useful calculations.
4. Discuss the utility of this language for practical programming problems.

3. THE LANGUAGE

3.1 Definitions

A procedureless language, which uses only definitional statements, must have a data representation that lends itself to unordered processing. A set representation of data is a logical choice for three reasons. First, a set is an unordered collection of elements (data). Second, a set representation of data is considered to be a fundamental computer data structure. Third, and most importantly, a formal set definition is procedureless in that it states rules for ascertaining set memberships rather than a procedure for selecting set elements [3].

In the context of a procedureless programming language, a program will consist of a series of set definitions. No formal definition of a set will be given since the traditional view of a set as a collection of elements is adequate. The usual set notation consisting of elements enclosed in braces will suffice. To show by enumeration that the integers 0, 1, 2, and 5 constitute a set, $\{0, 1, 2, 5\}$ will be written. This set may be named with upper case letters by stating:

$$A = \{0, 1, 2, 5\}$$

Lower case letters will be used to designate set elements. Thus defines the set of natural numbers.

$$A = \{x \mid x = 1, 2, 3...\}$$

Set elements may also be ordered n-tuples, which are written:

$$(a_1, a_2, \ldots, a_n).$$

Elements in the n-tuple are ordered by association with a particular position within the n-tuple. Lower case letters will be used for arbitrary n-tuple elements.

Additionally, a functional notation is introduced to describe attributes of set members. For example, let U be a set consisting of the real numbers. The set of integers is a proper subset of the set of reals. Thus, the elements of U may be thought of as having two attributes, a value which is a measure of the relative size of the number and a type which is either integer or real. These attributes constitute an ordered pair. U is then defined by

$$U = (u_1, u_2)$$

where u_1 is the value of the number and

u_2 is either 'integer' or 'real' according to the type of the number.

A function V is defined on U such that

$$\text{for } x \in U,\ V(x) = V((u_1, u_2)) = u_1.$$

A second function T is defined on U such that

$$\text{for } x \in U,\ T(x) = T((u_1, u_2)) = u_2.$$

The set of all integers, I, may now be defined as

$$I = \{x \mid x \in U,\ T(x) = \text{'integer'}\}.$$

The set I so defined consists of ordered pairs containing the value and type information. To obtain the set of values of x without the redundant type information, I', it is necessary to write

$$I' = \{y \mid x \in U, y = V(x) \text{ where } T(x) = \text{'integer'}\}.$$

While the notation used in the definition of I' is consistent with the usual mathematical notation, this definition will be rewritten as

$$I' = \{y \mid x \in U;\ \text{if } T(x) = \text{'integer'} \text{ then } y = V(x)\}$$

to reflect the normal programming language notation. The semicolon is introduced to separate the definition of the formal parameters from the rules describing set members. The key word if replaces the word where in the mathematical notation. The keyword then signals the definition of the n-tuple element which is implicit in the mathematical notation. In the programming notation, sets are defined on the basis of attributes of elements within other sets.

In general, any set S may be defined, in this notation, as

$$\begin{aligned} S = \{(x_1, x_2, \ldots, x_n) \mid & y_1 \in Y_1, y_2 \in Y_2, \ldots, \\ & y_m \in Y_m; \\ & x_1 = f_1(y_1, y_2, \ldots, y_m), \\ & x_2 = f_2(y_1, y_2, \ldots, y_m), \\ & \quad \vdots \\ & x_n = f_n(y_1, y_2, \ldots, y_m)\} \end{aligned}$$

where

$x_1, x_2, \ldots, x_n$ are elements of the ordered n-tuple in the set being defined

$Y_1, Y_2, \ldots, Y_m$ are sets of interest

$y_1, y_2, \ldots, y_m$ are elements of the Y_i

$f_1, f_2, \ldots, f_n$ are functions defined on the Y_i.

The f_i are interpreted as rules for selecting set elements. Thus, the f_i are not procedures, or algorithms, for calculating set elements.

The procedureless language concept is based upon the above notation for set definition. It will be used as the fundamental statement of the procedureless programming language. Thus, the statements in the language to be designed will be definitional, rather than procedural, in nature.

To reiterate then, a set consists of n-tuples. The elements of the n-tuple are ordered by being associated with a particular position within the n-tuple. Sets are defined by using the set definition to choose elements in the new set. The following example will illustrate the use of set definitions to solve the problem frequently encountered in computer programming.

3.2 Example Problem

Consider a simple payroll calculation problem. The payroll calculation is to be done as follows:

All employees are hourly. Gross pay is calculated from the base rate for the first forty hours and time and a half for overtime. Net pay is gross pay less deductions where deductions are:

1. Income tax based on gross pay and number of exemptions,
2. Optional hospitalization based on number insured, and
3. Social Security based on gross pay.

Records are to be read which contain the employee number and the number of hours worked. A sorted payroll report is to be produced along with totals for the gross pay, net pay, and the various categories of deductions.

To begin with, the deductions are tabular in nature and are shown in Figure 1 below.

The common characteristic of these data is that they are all in tabular form. The previously defined set notation may be used to describe the tables as sets by writing:

H = {(1,1), (2,2), (3,3), (4,4), (5,5)}

SS = {(100,1), (200,2), (300,3), (400,4), (500,5)}, and

IT = {(100,0,25), (100,1,20), (100,2,15), (100,2,10), (100,4,5), (200,0,50), (200,1,40), (200,2,30), (200,3,20), (200,4,10), (300,0,75), (300,1,60), (300,2,45), (300,3,30), (300,4,15), (400,0,100), (400,1,80), (400,2,60), (400,3,40), (400,4,20), (500,0,125), (500,1,100), (500,2,75), (500,3,50), (500,4,25)}.

Additionally, there is an employee roster which contains the employee number, name, pay rate, hospitalization, exemptions, and dependents information respectively. This set is a file of records and is defined as:

ER = {(1, A.A.Jones, 1.00, no, 0, 1), (2, B.B.Smith, 1.50, yes, 2, 3), (3, C.Doe, 2.00, no, 1, 0), (4, X.Brown, 2.50, yes, 1, 0)}.

By reading a set of input records containing the employee's number and the number of hours worked, the ER file can be searched to determine the employee's name, base pay rate, and deductions. This yields net pay. The result will be a pay check for each individual, and a recapitulation of the amounts in the various categories.

For each set it is necessary to define n attribute selector functions, where n is the number of attributes (or elements in the n-tuple). For the particular problem it is necessary to define all of the possible functions. For the hospitalization set, H, define

HN(x) = the number of insured and

HP(x) = the premium.

Income Tax (IT)

	Exemptions				
Gross Pay	0	1	2	3	4
100	25	20	15	10	5
200	50	40	30	20	10
300	75	60	45	30	15
400	100	80	60	40	20
500	125	100	75	50	25

Hospitalization (H)

	Number of Insured				
	1	2	3	4	5
Premium	1	2	3	4	5

Social Security (SS)

Gross Pay	Rate
100	1
200	2
300	3
400	4
500	5

FIGURE 1

For the Social Security set, SS, define

SSGP(x) = gross pay and
SSR(x) = Social Security rate.

For the income tax set, IT, define

ITGP(x) = gross pay,
ITE(x) = number of exemptions, and
ITR(x) = income tax rate.

For the employee roster set, ER, define

EREN(x) = employee number,
ERN(x) = employee name,
ERBR(x) = base pay rate,
ERH(x) = hospitalization,
ERE(x) = number of exemptions, and
ERNI(x) = number of insured.

There is one additional set to be defined, namely, that of the input set, which is

IN = {(4,65), (1,20), (2,40), (3,45)}.

The input set has two functions

INN(x) = the employee number, and
INH(x) = the number of hours worked.

There are three output sets, the first is CK, the set of checks with the functions

CKN(x) = employee number,
CKNA(x) = employee name, and
CKNP(x) = net pay.

The second output set to be defined is the recapitulation, RC, which has the functions

RCGPT(x) = gross pay total,
RCNPT(x) = net pay total,
RCHT(x) = hospitalization total,
RCSST(x) = Social Security total, and
RCITT(x) = income tax total.

The third output set is the sorted payroll report, SPR, consisting of the employee name, employee number, gross pay, deductions, and net pay. The functions are

SPRN(x) = employee name,
SPREN(x) = employee number,
SPRGP(x) = gross pay,
SPRD(x) = deductions, and
SPRNP(x) = net pay.

Having defined all of the data sets, it remains only to write the set definitions. The first set is

```
RC = {(gpt, npt, ht, sst, itt)|
      gpt = SUM(GP), npt = SUM(NP),
      sst = SUM(SSS), ht = SUM(HS),
      itt = SUM(ITS)}.
```

This requires the definition of the function SUM, and the sets GP, NP, SSS, HS, and ITS. The set operator SUM simply sums the elements of the set named as its argument. The second set to be defined is

```
SPR = {SORT((n,en,gp,D,np),1)|
       y∈IN, x∈ER;
       if INN(y) = EREN(x) then
       n =ERN(x),
       en = INN(y),
       if INN(Y) = EREN(x) then
       if INH(y)≤40 then gp = INH(y)*
                              ERBR(x)
                    else gp = ERBR(x)*
                              (1.5*INH
                              (y)-20),
       D = {(ss, h, it)|
            z∈SS; if FLOOR(SSGP(z), gp)
                  then ss = SSR(z),
            z∈IT; if INN(y) = EREN(x)
                  and ERE(x) = ITE(z)
                  and FLOOR(ITGP(z), gp)
                  then it = ITR(z),
            z∈H;  if INN(y) = EREN(x)
                  and ERH(x) = 'yes'
                  and ERNI(x) = HN(z)
                  THEN h = HP(z)},
       np = GP - (h + ss + it)}.
```

The SORT function simply sorts a set by the given n-tuple index. The general form is SORT(X, n) where

X is the name of the set to be sorted and

n is the n-tuple index of the entry to be sorted.

The FLOOR function has the general form FLOOR(X(y), z) where

X is the attribute field of interest,

y is a typical element of the set to be searched, and

z is the argument used in the determination of the result.

The invocation of FLOOR(X(y), z) causes the set pointed to by y to be searched for the largest X entry which divides z yielding a number greater than 1. If there is such an entry then y will point to that particular entry in the named set. If there is not such entry, then y is set to zero.

The remaining sets may be defined as

```
GP  = EXT(SPR, 3),
NP  = EXT(SPR, 5),
SSS = EXT(D, 2),
HS  = EXT(D, 3), and
ITS = EXT(D, 4).
```

The function EXT extracts an entry from every n-tuple in the argument set. The function is written EXT(X, n) where

X is the set name and

n is the n-tuple index of the entry to be extracted.

The remaining output set, CK, may be defined as

```
CK = {(x, y, z)|u∈SPR;
      x = EXT(SPR, 2), y = EXT(SPR, 1),
      z = EXT(SPR, 3)}.
```

In the above notation, the set definition is similar to the ALGOL FOR statement. The phrase y∈IN in the definition of SPR is a declaration that defines y to be an element of the IN set. The interpretation of the set definition is, "for every element in the input set, IN, perform the following calculations".

The definition of D within SPR is simply a loop within a loop. However, the formal parameter z is redefined for every entry in the n-tuple.

The statement

if INN(y) = EREN(x) then n = ERN(x)

causes the ER set to be searched until an employee number is found which matches the employee number in the input set, IN. The declaration x∈ER established x as a formal parameter of the ER set. For every occurrence of the match INN(y) = EREN(x), an n-tuple in SPR is defined. The statement

n = ERN(x)

causes the name field for each x to be extracted and assigned to an n, a formal parameter for the first entry in each n-tuple in the SPR set.

The goal of the procedureless language is to reduce the amount of programming detail necessary to specify the computer solution of a problem. The illustrative problem shows that a somewhat formal notation can be used to define sets which form the solution of a posed problem. The example clearly shows that there are several areas in which the reduction of programming detail can occur.

First, the procedureless language should not require that sets be defined in any order. In the example, the sets RC and SPR are defined without consideration of the fact that the elements of SPR must be defined before the elements of RC can be defined.

Second, the language does not need to provide methods for explicitly indexing through the elements of a set. Definitions simply apply to all elements.

Third, there need be no explicit method for performing iterative calculations. The set definitions are automatically interative in that the set definition is performed as many times as necessary to completely define the set.

Fourth, there need not be a method for explicitly specifying the number of elements a set contains.

Fifth, no method exists for explicitly associating the elements within a n-tuple with the correct n-tuple. For example, the definition of SPR does not make clear that the employee name, en, will be associated with the correct employee number, n. The implication is that the language compiler must keep account of these variables and accomplish the correct association.

Thus, the procedureless language concept is that a description of the problem solution is all that is necessary to define a computer solution for that problem. The procedure for the computer solution need not be specified. In effect, all that must be stated is a prototype solution.

3.3 Syntax Specification

The sample problem in the Section 3.2 illustrated the procedureless language concept. The language provides a means for describing a computer solution of a problem without specifying the detailed calculative procedure for that problem. The salient features of the procedureless language concept must be relected in the programming language design. In addition, there are certain arbitrary choices that must be made. For example, the language is required to process both numeric data and character data rather than being limited to numeric data. To clarify the formal description of the programming language, the general requirements of the language are described.

The procedureless programming language is to be used to solve problems in diverse areas of the computing field. Therefore, both numeric and character string constants will be valid representations of data. The language deals only with sets, thus there is a requirement for a statement in the programming language that defines sets consisting of constants.

The language must provide the usual arithmetic operators for performing calculations on numeric data. The usual relational operators and Boolean operators are included to facilitate decision making. The relational operators must provide for comparison of character string data in addition to comparison of numeric data.

There must be a method for referencing elements within an n-tuple which corresponds to the functional notation used in the procedureless language concept. Thus, the language must provide a facility for defining a prototype set element. This facility must establish the name of each element within the n-tuple and its position within the n-tuple. Further, the prototype n-tuple definition must identify the set for which the definition is valid.

Since a set is an unordered data structure, element indices must be implicit. Thus, a particular set element may not be accessed by direct reference. The element may only be accessed by searching for that element. Calculations involving set elements imply using all elements in the set. Consequently, the language has no provision for performing explicit loop calculations. Along these same lines, the number of elements in a given set is implicit in the set definition.

In the procedureless programming language, set definitions are interpreted as rules for selecting set elements. The procedure for calculating the set elements is not provided to the compiler. The compiler accepts a description of the solution and generates a procedure for accomplishing the required results. In doing this, the compiler must insure that the elements within a particular n-tuple belong to that n-tuple for that set.

The language must permit statements to appear in any order. A set need not be defined before it is referenced. The

prototype element of a set need not be defined before the set is defined.

3.3.1 Metalanguage

The language description technique chosen is based on that of the IBM Vienna Laboratories. The syntax description was developed to describe the concrete syntax of PL/1 [16]. The description language is an extended Backus Normal Form (BNF) notation, which is considerably more compact than BNF. In brief, the description introduces the ellipse, "...", to indicate the repetition of syntactic signs; braces, { } , to indicate mandatory choice; and brackets, [ ], to indicate optional choice (including none).

The character set chosen for the language includes both upper case and lower case letters as well as special symbols. In the following description, lower case letters will be used in a metalinguistic variable names. The metalinguistic variable names will be hyphenated or terminate with a hyphen. These variable names will be chosen so as to indicate the object for which the variable stands. In case of conflict between metalinguistic symbols and syntactic signs, the symbols in the language being described will be underlined.

The description of the language will be done in sections following the traditional methods. Thus, constants, variables, statements, etc. are described in a hierarchical order to facilitate understanding the language.

3.3.2 Basic Symbols

The procedureless programming language is built up from the following basic symbols:

basic-symbol ::= alpha-numeric delimiter-

alpha-numeric ::= letter- digit-

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

letter- ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|
P|Q|R|S|T|U|V|W|X|Y|Z|a|b|c|d|
e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|
t|u|v|w|x|y|z

delimiter- ::= DEFINE|PROTOTYPE|END|DO|
operator-|;|,|.|∈blank

operator- ::= arithmetic-operator|
comparison-operator|
Boolean-operator

3.3.3 Constants

1. Numbers

integer- ::= digit-...

signed-integer :: =[+ -]
integer-

numberic-constant ::=
signed-integer[.]|
[signed-integer|+|-].
integer-

Thus, a integer- is any number of digits, while a signed-integer is simply an algebraic sign followed by any number of digits. The numeric constant allows for signed numbers in the algebraic sense.

Examples:

5 +5 -.6 8.9

2. Strings

string-character ::= alpha-
numeric|;|.|,|''|blank

This rule defines the available string-characters as all of the alpha-numerics plus the semicolon, comma, period and double quite (succession of two single quotes). The double quote introduced as a string character is used to place a single quote in the character string.

character-string ::=
'[string-character...]'

Thus, a character-string is any sequence of characters enclosed in single quotes.

Examples:

'A≤B' '9X.=' ''''

3.3.4 Identifiers

identifier- ::=
letter-[alpha-numeric...]

An identifier must begin with a letter and is followed by any number, including none, of alpha-numerics.

Examples:

a B CC X4 V3c

3.3.5 Set Constants

Sets which are composed entirely of constants are defined by the rules:

define-statement ::={DEFINE ,.set-
constant...}.

set-constant ::= set-name={{},.n-tuple...}}

n-tuple ::= ({},.constant-...})

constant- ::= numeric-constant|
character-string

Thus, a single define-statement can declare any number of set-constants. Each set-constant may have any number of n-tuples. Each n-tuple is required to have the same number of fields as all the other n-tuples in the set.

Examples:

DEFINE X = {(2,4),(5,6)}.

DEFINE Y = {('yes'),('no'),('maybe')},
Z = {(1,'true'),(0,'false')}.

3.3.6 Data Description

A notation for describing the data to be processed must be provided. Since the data is thought of as a set this may be organized in whatever fashion the user deems appropriate, flexible rules for describing sets must be available. A set may be thought of as a file. The n-tuples in a set may then

be thought of as a record in a file. The composition of the records in the file will be defined by the prototype statement.

Since each record has to have a length, this length will be specified by the sum of the lengths of the fields in the record. The field lengths will be specified in characters, so that the language description is machine independent. The number of bits necessary to represent a character for a particular machine will then define the record length for a given implementation. Thus, the statement defining prototype set elements is governed by the rules:

```
prototype-statement ::= PROTOTYPE
                        {,. set-prototype...}.

set-prototype ::= set-name =
                  ({,.element-list...})

element-list ::= field-specification|
                 set name

field-specification ::= field-name
                        (integer-)

field-name ::= identifier-

identifier ::= identifier-
```

The set-name is the name by which the set may be referenced. A field-name is the name of that particular field in all of the elements of the set. The prototype n-tuple has the ordering imposed by the order in which the fields are named. The field-names are of particular importance since they play a key role in subset selection. These names will be required to act as functions which will allow accessing that field in a given element in a set. The element-list may alternatively contain set-names. A set so used is a convenient way of referencing a group of elements in the current n-tuple by one name. Each element in the current n-tuple is associated with only one element in the referenced set.

Examples:

```
PROTOTYPE X = (A(6),B(1)).

PROTOTYPE Y = (C(2),D(3)),
              Z = (E(4),Y).
```

3.3.7 Set Definition

Set definitions are the analogs of the executable statements in the usual programming languages. They require that a prototype statement be declared in order to specify the element length and the field identifiers, field-names. All calculations that are performed upon set elements as well as all comparison and decision making capabilities are part of the set definition. The rules pertaining to set definition are below:

```
set-definition ::= set-name={n-tuple-reference|{,.selector-function...}}
n-tuple-reference ::= n-tuple-variable|function-call
n-tuple-variable ::= ({,.identifier-...})
selector-function ::= [declaration-part]selector-part
declaration-part ::= {,.declaration-...};
declaration- ::= formal-parameter∈set-name
formal-parameter ::= identifier-
selector-part ::= if-statement|assignment-
assignment- ::= variable-=expression-
variable- ::= set-name-|formal-parameter
if-statement ::= IF expression THEN block-[ELSEblock-]
block- ::= DO block-body
block-body ::= {,.selector-part...} end-clause
end-clause ::= END
expression- ::= expression-four|expression-Boolean-operator expression-four
Boolean-operator ::= and|or
expression-four ::= expression-three|expression-four comparison-operator expression-three
comparison-operator ::= >|≥|=|<|≤|≮|≠|≯
expression-three ::= expression-two|expression-three{+|-} expression-two
expression-two ::= expression-one|expression-two{*|/} expression-one
expression-one ::= basic-expression|{+|-} expression-one
basic-expression ::= formal-parameter|field-name (formal-parameter)|simple-constant|
                     function-call
simple-constant ::= character-string|unsigned-number
unsigned-number ::= integer [.]|[integer]. integer
```

The declaration- is a preamble which defines a formal-parameter which stands for some element in the named set. Once a declaration for a formal-parameter occurs, it remains so defined until it is redefined in a subsequent declaration. Thus, any reference to the formal-parameter in the selector-function automatically references the set for which it is defined. The selector-part performs set definition. It contains the usual assignment statements as well as IF statements for decision making.

The semantics of the comparison-operator can best be illustrated by an example:

```
PROTOTYPE U = {A(6)}
          V = {B(6),C(4)}.
          Z = {(z)|x∈U,y∈V;
               IF A(x) = B(y) THEN
               z = C(y)}
```

The declaraction x∈U essentially establishes an iterative loop which will consider each element of U paired with each element of V. The IF statement has the interpretation of a search. Thus, for each element in U, the entire V set is searched for equality on the B field. The result is a set of pairs of the form (x,Y). The THEN clause then operates on these ordered pairs to create the elements of Z by extracting the C field in every element of V pointed to by an (x,y). If no match is found, then the Z set is not created. In all cases the formal-parameter determines which set is referenced.

3.3.8 Functions

function-call ::= function-identifier argument-list

function-identifier ::= identifier-

argument-list ::= ({,.argument-...})

argument- ::= basic-expression| set-name|n-tuple-variable

Since the language is set oriented, there is a requirement to operate upon sets. The nature of the elements of the sets are sufficiently complicated that a function subroutine will have to be used in order to perform the required evaluation. Some functions are defined to accept a variable number of arguments. If the function receives only one argument, then it must be a field name and a formal parameter defining the set to be referenced. If the function receives two arguments, then the first is a set name and the second is an ordinal index specifying the field within the elements of the set which is to be used in the evaluation of the function. For functions which require additional input arguments, these arguments are simply added to the argument-list.

The following functions are defined as part of the language.

1. SUM — For one argument, sum the field specified in all the elements of the set. For two arguments, in the named set, sum the field whose index corresponds to the second argument. The result of SUM is a single number.
2. EXT — For one argument, extract the named field from all the elements in the set. For two arguments, extract the field whose index corresponds to the second argument from all elements. The result of EXT is a set.
3. SORT — For one argument, sort the set on the named field. For two arguments, sort the set on the field whose index corresponds to the second argument. The result is the sorted set.
4. MAX — For one argument, find the largest of the named field in all the elements. For two arguments, find the largest of the fields in all the elements based on the second argument. The result is a single number.
5. MIN — For one argument, find the smallest of the named field in all the elements of the set. For two arguments, find the smallest of the field in all the elements based on the second argument. The result is a single number.
6. FLOOR — For two arguments, the first is the field name and formal parameter. The second is an argument to be used in the calculation of the result. The set is searched for the largest field entry which divides the second argument giving a number greater than or equal to 1. If there is such an entry, then on exit the formal parameter points to that entry. The value of the function is then true. If there is no such entry, the value is false. For three arguments, the first is a set name and the second is the ordinal index of the set to be searched. The third is the argument to be used in the calculation. The calculation proceeds as described in the two argument case. The result is a single number.
7. CEILING — For two arguments, the first is a field name and formal parameter. The second is the argument to be used in the evaluation of the function. The set is searched for the smallest number which when divided into the second argument yields a result less than 1. If there is such an entry,

then on exit the formal parameter points to that entry and the value of the function is true. If there is no such entry, the value of the function is false. For three arguments, the first is the set name, the second is the ordinal index of the field to be searched, and the third is the argument to be used in the calculation. The evaluation of the function proceeds as described for the two argument case. The result is a single number.

8. CARDINAL The function accepts only one argument which is the name of a set. The function counts the number of elements of the named set and returns this number. The result of the function is a single number.

3.3.9 Descriptions-

The composite group of statements which describes a problem solution is called a description. Statements within a description may appear in any order with the exception of the END statement. Naturally, the END statement must be the last statement in the description. The rules defining a description are:

```
description- ::= {,.statement-...}
                 end-clause

statement- ::= assignment-|set-definition|
               prototype-statement|
               define-statement
```

Through these definitions assignment-statements can occur outside of a set-definition. IF statements, however, must be imbedded in a set-definition.

3.3.10 Example

To illustrate the usefulness and to further clarify the sematics of the language, an example program will be exhibited. The purpose of this program is to show that programming can be done on a definitional, rather than procedural, basis.

For this example, the problem from Section 3.2 will be programmed. The same problem definition, set names, field names, and formal parameters will be used.

Some coding conventions are observed to make reading the program easier. These are listed below;

1. Set names and field names are upper case.
2. Formal parameters are lower case.
3. Set names are acronymns of the set description.
4. Field names begin with the set name.
5. Set constants are defined with their elements arranged in a readable order; however, the programs take no advantage of this.

```
PROTOTYPE D   = (DR(6),DSS(6),DH(6),DTT(6)),
          H   = (HN(1),HP(1)),
          SS  = (SSGP(3),SSR(1)),
          IT  = (ITGP(3),ITE(1),ITR(3)),
          ER  = (EREN(1),ERN(20),ERBR(4),ERH(3),ERE(1),ERNI(1)),
          GP  = (GPI(6)),
          NP  = (NPI(6)),
          SSS = (SSSI(6)),
          HS  = (HSI(6)),
          ITS = (ITSI(6)),
          CK  = (CKN(1),CKNA(20),CKNP(6)),
          SPR = (SPRN(20),SPREN(1),SPRGP(6),SPRD(6),SPRNP(6)).,
DEFINE    IN  = {(4,65),(1,20),(2,40),(3,45)},
          ER  = {(1,'A.A.JONES',1.00,'no',0,1), (2,'B.B.SMITH',1.50,'yes',2,3),
                 (3,'C.DOE',2.00,'no',1,0), (4,'X.BROWN',2.50,'yes',1,0)},
          H   = {(1,1),(2,2),(3,3),(4,4)},
          SS  = {(100,1),(200,2),(300,3),(400,4),(500,5)},
          IT  = {(100,0,25),(100,1,20),(100,2,15),(100,3,10),(100,3,4),(200,0,50),(200,1,40),
                 (200,3,20),(200,4,10),(300,0,75),(300,1,60),(300,2,45),(300,3,30),(300,4,15),
                 (400,0,100),(400,1,80),(400,2,60),(400,3,40),(400,4,20),(500,0,125),(500,1,100),
                 (500,2,75),(500,3,50),(500,4,25)}.,
          RC  = {(gpt,npt,ht,sst,itt)| gpt = SUM(GP,1), npt = SUM(NP,1), sst = SUM(SSS,1),
                 ht = SUM(HSS,1), itt = SUM(ITS,1)},
```

```
SPR = {SORT((n,en,gp,D,np),1)| y∈IN,x∈ER; IF INN(y) = EREN(x) THEN DO n = ERN(x),
      en = INN(y) END
      IF INH(y)≤40 THEN gp = INH(y)*ERBR(x) ELSE gp = ERBR(x)*(1.5*INH(y)-20),
      D = {(ss,h,it)|
            z∈SS; IF FLOOR (SSGP(z),gp) THEN ss = SSR(z), z∈IT; IF ERE(x) = ITE(z) and
            FLOOR (ITGP(z),gp) THEN it = ITR(z), z  H; IF ERH(x) = 'yes' and ERNI = HN(z)
            THEN h = HP(z)}
      hp = gp - D
      END},
GP  = EXT (SPR,3),
NP  = EXT (SPR,5),
SSS = EXT (D,2),
HS  = EXT (D,3),
ITS = EXT (D,4),
CK  = {(x,y,z)|u∈SPR; x = EXT (SPR,2), y = EXT (SPR,1), z = EXT (SPR,3)}
END
```

The statement

```
IF INN(y) = EREN(x)
```

causes an ordered triple to be defined. This triple associates one element of SPR with an element of IN and ER. This insures that the elements of the SPR set are properly constructed. The second IF statement generates two sets which are acted upon by the THEN and ELSE clauses.

4. CONCLUSIONS

The procedureless programming language which I will call PPL, draws its basis from set theory. This is a reasonable starting point since much of the data used in programming applications can be described in terms of sets. Set theory is also attractive since the usual notation for defining a set has procedureless aspects. The example description provides proof that a procedureless language can solve a non-trivial programming problem.

The language is described as procedureless because all statements, with the exception of END, can occur in any order. The actual execution of set definitions occurs in the correct order regardless of the order in which the definitions occur in the source description. As a convenience in expressing problem descriptions, some procedural attributes are included in the language. In particular, n-tuple variable assignments must occur within the scope of the braces which enclose the set definition. The statement parentheses, DO and END, require that statements belonging to a particular DO-END pair be located within the scope of that DO-END. Finally, the END statement is included in the language to allow the user to indicate the termination of the source language description.

Certainly, the END statement is not necessary. The compiler can always determine whether or not a set definition is complete. For an interactive environment, the language could be implemented as an interpreter. The interpreter would accept input statements and perform set definitions as information became available. For undefined sets, the interpreter would ask the user for the required definition. In a standard batch processing system, the compiler would simply terminate with the diagnostic that a certain set is undefined. Hence, the END statement is a convenience to the user and the compiler.

The DO-END pair likewise is not necessary. Its introduction allows elimination of redundant statements by the user. This economy of notation brings some degree of procedural organization into the language. To remove DO-END from the language requires that every n-tuple variable assignment be preceeded by an IF statement which binds the formal parameters in that statement. The DO-END pair frees the user from repeating the IF statement. Nothing in the language itself requires the use of the DO-END pair; it is a user convenience. Thus, within the set definition, the statements may be unordered, if desired.

The requirement that set definition statements occur within a set of braces introduces another element of procedural organization. Again, this would not be required in a completely procedureless language. The requirement could have been stated in another manner: all variable and set names must be unique. Then a sort routine would determine statement ordering. Since a goal of the language is to improve man-machine communication, this is an unnecessary requirement. The generality provided by such a tool may be easily misused by accident.

Thus, a completely procedureless programming language can be designed. However, compromising the language design to include some procedural features is necessary to obtain a more readily usable language. In PPL, the user must group his statements; but neither the statements within a group nor the groups themselves must be ordered.

Clearly, a language such as FORTRAN cannot be made procedureless by simply sorting the statements into the order in which the variables are defined. Two statements prohibit this: GO TO and assignment. The GO TO statement provides no clue as to where it might belong in the program. The assignment which redefines a variable may provide no information as to the ordering of the definitions. Thus, any language which has explicit transfer of control statements cannot be procedureless nor can any language which allows arbitrary redefinition of variables.

An interesting phenomenon of the language is that all elements of the result set could be calculated in parallel. Hence, PPL may yield some clues as to methods for isolating parallelism in programs.

The writing of descriptions does require some orientation for one who is used to procedure-oriented languages. Primarily, the thought process must be more clearly focused on the data whereas in the usual procedure-oriented language it is focused on the logic necessary to solve the problem. Thus, a procedureless language moves the user a level further away from the machine code solution to his problem. The compiler itself constructs a larger portion of the algorithm than a procedure-oriented language compiler. Therefore, the amount of programming detail provided by the user is appreciably reduced. This is the most significant improvement PPL offers over the traditional programming languages.

5. ACKNOWLEDGMENT

I would like to thank Robert M. McClure for his comments and suggestions concerning PPL.

BIBLIOGRAPHY

1. "An Information Algebra." Communications of the ACM, vol. 5, no. 4, April 1962, pp. 190-204.
2. Balzer, R.M. "Dataless Programming." AFIPS Proceedings of the FJCC, 1967, pp. 535-544.
3. Berztiss, A.T. Data Structures. Theory and Practice. New York: Academic Press. 1971.
4. Burge, W. H. "Combinatory Programming and Combinatorial Analysis." IBM Journal of Research and Development, vol. 16, no. 5, September 1972, pp. 450-461.
5. Cocke, John, and Schwartz, J. T. Programming Languages and Their Compilers. 2nd rev. ed. New York: Courant Institute of Mathematical Sciences. 1970.
6. Dijkstra, E. W. "Letter to the Editor." Communications of the ACM, vol. 11, no. 3, March 1968, pp. 147-148.
7. Dodd, G. G. "APL--A Language for Associative Data Handling in PL/I." AFIPS Proceedings of the FJCC, 1966, pp. 677-684.
8. Feldman, Jerome A., and Rovner, Paul D. "An Algol-Based Association Language." Communications of the ACM, vol. 12, no. 8, August 1969, pp. 439-449.
9. Harrison, Malcolm. Data-Structures and Programming. rev. ed. New York: Courant Institute of Mathematical Sciences. 1971.
10. Homer, E. D. "An Algorithm for Selecting and Sequencing Statements as a Basis for a Problem-Oriented Programming System." Proceedings of the ACM 21st National Conference, 1966, pp. 305-312.
11. Iverson, Kenneth E. A Programming Language. New York: John Wiley and Sons, Inc. 1962.
12. Katz, Jesse H., and McGee, William C. "An Experiment in Non-Procedural Programming." AFIPS Proceedings of the FJCC, 1963, pp. 1-13.
13. Katzan, Harry, Jr. Advanced Programming. New York: Van Norstrand Reinhold Co. 1970.
14. Klerer, Melvin, and May, Jack. "Automatic Dimensioning." Communications of the ACM, vol. 10, no. 3, March 1967, pp. 165-166.
15. Knuth, Donald E. The Art of Computer Programming, Vol. 1: Fundamental Algorithms. Reading, Massachusetts: Addison-Wesley Publishing Company. 1968.
16. Lucas, P., Lauer, P., and Stigleitner, H. Method and Notation for the Journal Definition of Programming Languages. IBM TR25,087. IBM Laboratory: Vienna, Austria. 1968.
17. McCarthy, J. "Recursive Functions of Symbolic Expressions and Their Computation by Machine." Communications of the ACM, vol. 3, no. 4, April 1960, pp. 184-195.
18. Naur, Peter, ed. "Revised Report on the Algorithmic Language Algol 60." Communications of the ACM, vol. 6, no. 1, January 1963, pp. 1-17.
19. Naur, Peter, and Randell, Brian, eds. Software Engineering. Scientific Affairs Division NATO, Brussels, Belgium. 1969.
20. Neuhold, E. J. "The Formal Description of Programming Languages." IBM Systems Journal, vol. 10, no. 2, 1971, pp. 86-112.

21. PICL Users Guide. Telpar, Inc., Dallas, Texas. 1970.

22. Reinfelds, J., et al. "AMTRAN--An Interactive Computing System." AFIPS Proceedings of the SJCC, 1970, pp. 537-542.

23. Rice, John R., and Rosen, Saul. "NAPSS-A Numerical Analysis Problem Solving System." Proceedings of the ACM 21st National Conference, 1966, pp. 51-56.

24. Sammet, Jean E. Programming Languages: History and Fundamentals. Englewood Cliffs, New Jersey: Prentice-Hall. 1969.

25. Schlesinger, I., and Sashkin, L. "POSE: A Language for Posing Problems to a Computer." Communications of the ACM, vol. 10, no. 5, May 1967, pp. 279-285.

26. Schwantz, Jacob T. On Programming An Interim Report on the SETL Project Installment 1. New York: Courant Institute of Mathematical Sciences. 1973.

27. Swinehart, Dan, and Sproull, Bob. Sail. Stanford, California: Stanford Artificial Intelligence Project. 1970.

28. Symes, Lawrence R. "Manipulation of Data Structures in a Numerical Analysis Problem Solving System--NAPSS." AFIPS Proceedings of the SJCC, 1970, pp. 157-164.

29. Wegner, Peter. Programming Languages, Information Structures, and Machine Organization. New York: McGraw-Hill Book Company. 1968.

30. Whiteman, I. R. "New Computer Languages." International Science and Technology, April 1966, pp. 62-68.

31. Young, J. W., Jr. "Non-Procedural Languages." Presented at ACM Southern California Chapter's Seventh Annual Technical Symposium, March 23, 1965.