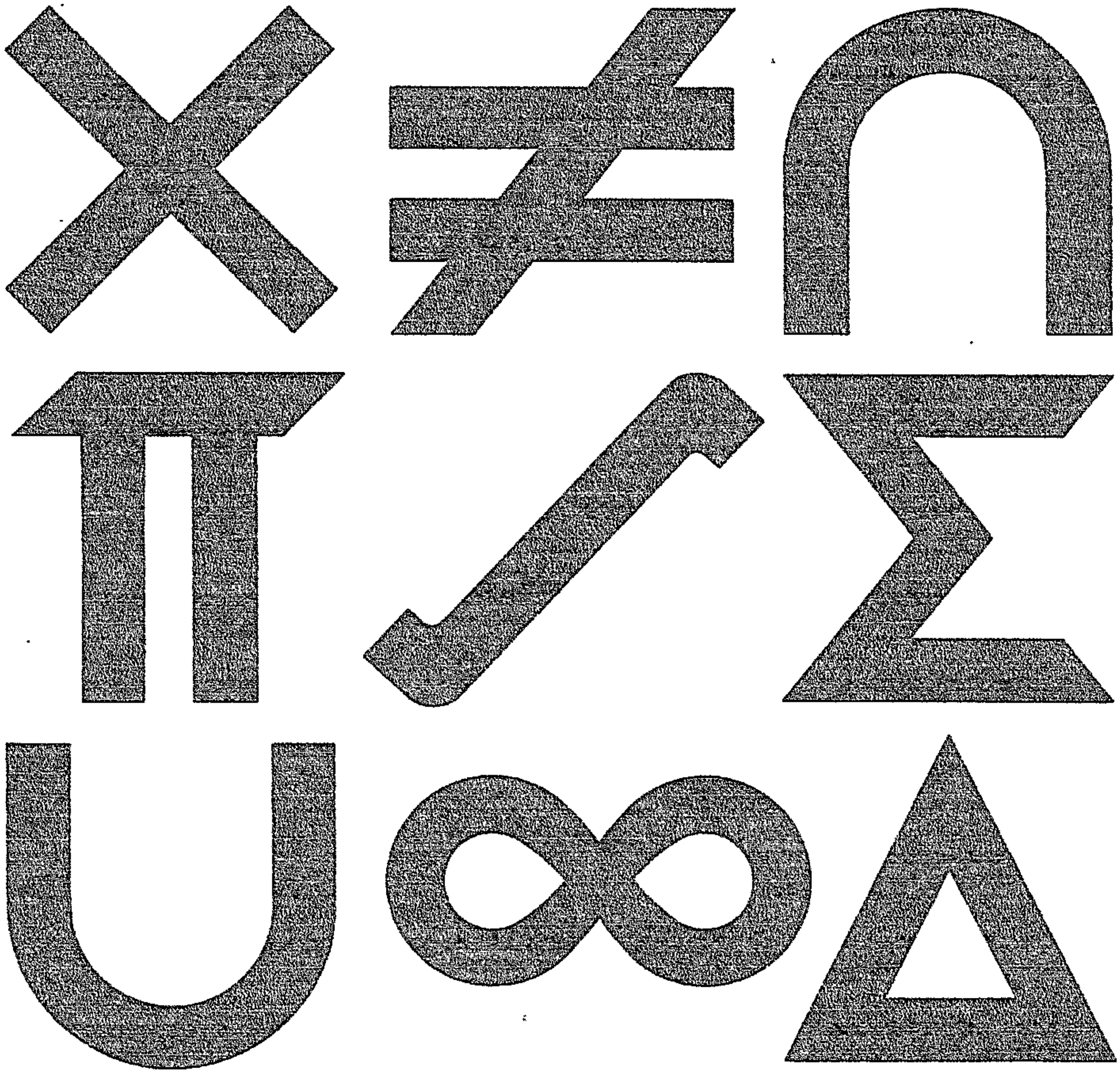
G320-2766, October 1986

NOTES ON THE SEMANTICS AND OPTIMIZATION OF A VHLL

F. ARBAB



1977 LOS ANGELES SCIENTIFIC CENTER REPORTS

G320-2689 July 1977
C. WOOD, E. B. FERNANDEZ & T. LANG, Minimization of Demand Paging for the LRU Stack Model of Program Behavior (21 p.)

G320-2690 September 1977
B. DIMSDALE, A Geometric Optimization Problem (20 p.)

+G320-2691 September 1977
B. DIMSDALE, Convex Cubic Splines (32 p.)

G320-2692 September 1977
B. DIMSDALE, Convex Cubic Splines II (18 p.)

G320-2693 September 1977
E. B. FERNANDEZ, H. KASUGA, Data Control in a Distributed Database System (26 p.)

1978 LOS ANGELES SCIENTIFIC CENTER REPORTS

G320-2694 March 1978
S. JUROVICS, An Investigation of the Minimization of Building Energy Load Through Optimization Techniques (28 p.)

G320-2695 November 1978
L. LICHTEN, An Approach to Solving Surface Connectivity Problems in Computer-Aided Design (41 p.)

G320-2696 November 1978
C. WOOD, R. C. SUMMERS, E. B. FERNANDEZ, Authorization in Multilevel Database Models (28 p.)

1979 LOS ANGELES SCIENTIFIC CENTER REPORTS

G320-2697 March 1979
P. NEISWANDER, A Review of the National Bureau of Standards Loads Determination Program (NBSLD) (12 p.)

G320-2698 March 1979
C. WOOD & E. B. FERNANDEZ, Authorization in a Decentralized Database System (35 p.)

G320-2699 June 1979
S. A. JUROVICS, Solar Radiation Data, Natural Lighting, and Building Energy Minimization (20 p.)

1979 LOS ANGELES SCIENTIFIC CENTER REPORTS

G320-2700 August 1979
A. INSELBERG, An Integral Equation Arising in a Convective Heat (Mass) Transfer Problem Through a Boundary Layer (19 p.)

G320-2701 September 1979
G. J. SILVERMAN, D. W. LOW, Construction of Optimal Synthetic Weather Data by Convex Combination (14 p.)

1980 LOS ANGELES SCIENTIFIC CENTER REPORTS

G320-2702 March 1980
K. EWUSI-MENSAH, Criteria for Decomposing an Information System Into Its Subsystems for Business Systems Planning (26 p.)

G320-2703 March 1980
K. EWUSI-MENSAH, Computer-Aided Modeling and Analysis Techniques for Determining Management Information Systems Requirements (30 p.)

G320-2704 June 1980
PAULA S. NEWMAN, An Atomic Network Programming Language (29 p.)

G320-2705 July 1980
J. G. SAKAMOTO, Use of DB/DC Data Dictionary to Support Business Systems Planning Studies: An Approach (24 p.)

G320-2707 October 1980
P. S. NEWMAN, Towards an Integrated Development Environment (29 p.)

G320-2875-5 April 1980
Compiled by KATHERINE HANSON, Abstracts of Los Angeles Scientific Center Reports (104 p.)

The availability of reports is correct as of the printing date of this report.

+ Appeared in an outside publication. Not available in Scientific Center report form. Please refer to the list of outside publications on the inside back cover for availability of reprints.

● Copies of report are no longer available from the Scientific Center.

IBM LOS ANGELES SCIENTIFIC CENTER REPORT NO. G320-2706

October 1980

Notes on The Semantics and Optimization of a VHLL

Farhad Arbab

IBM CORPORATION
LOS ANGELES SCIENTIFIC CENTER
9045 Lincoln Boulevard
Los Angeles, California 90045

ABSTRACT

This paper investigates an operational semantics model for a very high level data base manipulation language. The target machine of the operational semantics system is an applicative programming system. Use of the target machine in compilation, as the model for an implementation independent intermediate language, is considered, and its susceptibility to a class of optimization transformations is shown.

CONTENTS

1.0 INTRODUCTION

The purpose of this paper is to investigate a method of expressing the formal semantics of a very high level language. This language, which we will call "VHLL", is introduced in [1]. It is under study as the language component of an environment for the specification and development of application systems [8]. The semantic system involved is a type of operational semantic system (see below), and the notation used is a subset of LISP. The paper does not present a complete formal semantics, but rather develops the method and explores its potential.

The reasons for our interest in formal semantics are twofold. First, VHLL, a set-oriented language intended to manipulate data bases in a rather transparent fashion, involves many as yet unresolved, and some as yet unaddressed, semantic problems. The effort to associate precise meanings with the language constructs is an important means of identifying ambiguities and anomalies. The second reason for our interest is implementation-related. For VHLL to serve the purposes for which it is intended, there must be an associated language processor which is highly portable with respect to target languages and data management systems, and at the same time, provides significant optimization. These twin goals dictate the use of table-driven compilation and "back-end-independent" intermediate language. Operational semantics can be used to provide insight in the properties of the intermediate language(s) involved.

For example, more than one level of intermediate language might be used, with the first level relatively non-procedural. This level, together with a suitable abstract description of stored data organizations, would serve as a basis for gross operation sequencing decisions (major changes to program organization). The second level might be more procedural, expressing the results of those decisions, but still target-environment independent. (The third major step in the compilation would then be the translation of that language to the target HLL plus data management calls.)

One way of satisfying the intermediate language requirements of the above scheme is to use a slightly modified subset of the source language at the first level, and then an applicative language such as LISP at the second level. The latter is capable of expressing the full semantics of the language, and of expressing calculation sequences, in terms of its own, very well defined, environment. A less ambitious strategy would be to translate VHLL to the second level directly, avoiding global optimization, and utilizing storage-based criteria, expressed in abstract terms, for local alternative selection. If the latter approach is adopted, some important optimizations can be applied at the LISP (or its equivalent) level.

We thus view the development of a formal semantics for VHLL not only as a language definition method, but also as a step toward the definition of a language processor.

Sections 2 and 3 below introduce the semantic model and the basic semantic functions used in the report. Section 4 gives "translation rules" for some basic VHLL forms, and section 5 gives examples of how the model and these rules may be used to deal with VHLL expressions. In section 6, formal definitions are given for the semantic functions introduced in section 3. The definitions are constructed so as to be useful in certain optimizations. These optimizations, useful in set-oriented languages, are discussed in section 7.

(Note: A familiarity with LISP [9] is needed to follow many details of the discussion, especially in the last two sections. Also, a cursory reading of an introduction to VHLL [1] would be helpful, but is not necessary.)

2.0 THE SEMANTIC MODEL

There are three popular styles of formal definition of semantics as we summarize from [4]: operational, functional, and axiomatic.

Operational semantics associates with a program and its input a computation. A computation is a (possibly non-terminating) procedure which given an input state, returns a new state in some abstract machine, which we will refer to as the target machine. It is not uncommon to define a language using a well-understood subset of the same language as that abstract machine. In such cases, the very same semantic definitions can be used for implementation of the language by bootstrapping.

Functional semantics associates with a program a mathematical function which maps from the input domain of the program to its output domain. The result of the function for a specific input I is a special "undefined" value if the program does not halt for input I. Denotational semantics is a method of functional semantics definition which is very commonly used. The denotational method closely relates the semantics of a program to its syntactic components. In this method the syntactic primitives (terminals of the parse tree) are directly given a "meaning". The meaning of each higher level construct (in the parse tree) is then calculated from the meanings of its direct constituents. In principle, the "meanings" in a denotational definition can be in almost any domain. For example the meaning of an assignment statement is often a function from a "before" snapshot to an "after" snapshot, whereas the meaning of a program is normally a function mapping from its input domain to its domain of output.

Axiomatic semantics associates with a program a set of input and output logical assertions, usually in first order predicate calculus. A sentence like I{P}O expresses the fact that for all input states satisfying the input assertion I, if the execution of program (segment) P terminates, then the output state will satisfy the output assertion O. An axiomatic semantic definition of a language is a formal system allowing the derivation of true sentences about the input/output behavior of a program segment from the axioms.

The semantic system used here is an operational semantic system. The language used to express the semantics is a subset of LISP confined to its functional (applicative) feature. The program feature of LISP is used in section 6 to define some of the basic semantic functions, but semantic definitions themselves are in an applicative form.

The merits of applicative programming systems (of which LISP is a relative) are discussed in the excellent paper by John Backus [5]. Unfortunately, as he points out, in their current form they cannot replace conventional programming languages, in spite of their elegance and useful theoretical properties. Burge attempts to overcome the impracticality of these systems in [7] by introducing a language which incorporates some of the more conventional syntactic (and structural) constructs into an essentially applicative system.

A set-oriented language is inherently less restricted by the a-word-at-a-time pattern of assignments which Backus calls the "von Neumann bottleneck", than a conventional language. An applicative system used as the semantic back-bone of a set-oriented language may thus find it easier to radiate some of its properties through. It might also be feasible to take advantage of the reductional semantics [5] which is inherent in the nature of applicative systems for optimization purposes. In fact [2] is a clear attempt in that direction and has motivated the work presented here in section 7.

2.1 THE TARGET MACHINE

The target machine of the semantics system is a simple LISP machine. It uses the definitions given here to build a set of basic semantic functions out of standard LISP operations. (Alternatively, the model can be more sophisticated and include the semantic functions as standard operations.)

The basic objects manipulated by VHLL are sets of atoms and sets of tuples representing sets of associations. Sets of associations can be referenced as functions. A function reference in VHLL is translated either to a reference to a counterpart function or to a scan over a set of tuples in the target machine. This distinction in the model relates to the use of the semantic notation as an intermediate language. Counterpart functions represent the case where the real data organization provides easy access in the "domain to range" direction or where the functions are defined algorithmically.

In what follows, we will use some names (of sets, variables, and functions) both in VHLL expressions and in their equivalent semantic expressions. However, it should be clear that we are dealing with two disjoint domains of discourse, and this dual use of names is mainly to hint at the logical relationship between the corresponding components in these domains. (It is common to use underscored or overbarred names in the domain of semantics, but, currently, we choose not to use such conventions.)

2.2 SETS AND MULTISETS

Generally, in VHLL, a set is a multiset, i.e. it may contain some duplicate members. Thus sets and multisets are not different data types, rather, a set is a special case of a multiset. We tend to use the generic term "set" when referring to either form, except when it is important to make a distinction. A VHLL set is simply mapped into a list. However, the inherent ordering of the list representation is irrelevant in this case.

The reason for the use of multisets is performance related. Since VHLL is designed to manipulate large data bases, generally the "sets" which one would be dealing with are large. If sets were defined to be real sets, i.e. no duplicates, the burden of detection of duplicates in each and every computation result, including the intermediate results, would make the language excessively inefficient and impractical. To make matters worse, this task of deleting the duplicates is generally unnecessary in that most often, the usage of a set is such that it does not really matter if it actually contained some duplicates. Consequently, many language designers abandon the concept of "sets," and introduce into their languages an explicit form for the deletion of duplicates from a multiset, to be used only when and where it really matters.

However, multisets represent a solution to the performance problem only if no significance is associated with the multiplicities of their members (i.e., how many duplicates there are for each member). Otherwise, preservation of the implicit information that can be concluded from these multiplicities by correlating them to (the user's conception of) the execution path, may drastically limit optimization. Thus in VHLL we have defined multisets such that their intention is the same as the intention of sets, i.e. no meaningful information may be concluded from the multiplicities of their members, although their extension is different than that of sets, i.e. they include duplicates.

A set variable in our semantic domain will have a number of "properties" associated with it:

- TYPE

  This property will have the value 'SET' for all set variables.

- DUPS

  A false (NIL) value for DUPS property of a set variable guarantees that it contains no duplicate members, i.e. it is a "set." When DUPS is true, the set can be suspected to include duplicates.

2.3 TUPLES

Individual VHLL tuples are also mapped into lists, here, of course, exploiting the ordering of the list representation to reflect the order of the elements of the tuple.

A tuple variable in our semantic domain will have following properties associated with it:

- TYPE

  The TYPE property of all tuple variables have the value TUPLE.

3.0 BASIC SEMANTIC FUNCTIONS

In this section we present a set of functions which will be used as tools in construction of the semantics of VHLL. Since we will be using the lambda notation extensively, it is worthwhile to include a brief informal definition of this notation at this point. A formal treatment of the subject can be found in [10], [5], or [7].

Lambda calculus is a formal system due to Church [10] which does away with the ambiguity involved in the conventional notation for functions in mathematics. The conventional notation, e.g. f(x)=x+1, leads to ambiguity because it does not distinguish between the function and its value at an undefined point x. In other words, referring to function f above, the answer to the two questions "What is the function f?" and "What is the value of f at x?" is the same, namely, x+1. This becomes an important distinction to be made when we are dealing with functions that accept other functions as arguments and return functions as their results. In mathematics, these "functions" are sometimes called operators or transformers, and, to avoid ambiguity, special notation has been adopted in some specific cases, e.g. integration.

The above ambiguity is due to the fact that there is no way to express a "function constant", i.e. to define a function without at the same time giving it a name. The lambda notation resolves this ambiguity by introducing an explicit notation for the concept of a function. In this notation λx.x+1 is a constant value much the same way as 2 is. They are both "constants" but of different types, the latter is of type integer whereas the former is of type function. Both constants may be assigned to variables in which case the symbolic name of the variable would stand for the assigned value. A function constant may appear anywhere a function name is used in an expression, just like the case for variables and constants. The function (constant) λx.x+1 is quite different than the expression x+1. In lambda notation λ is a special symbol which denotes a function. Following λ and before the period is a list of the arguments of the function and after the period comes the expression that evaluates the function. Now if f is defined to be the function λx.x+1, there is no ambiguity between the function (λx.x+1) and its value at x (x+1).

LISP uses a slightly modified syntax for the lambda notation. The symbol λ is represented as the atom LAMBDA which then is followed by two lists, the first of which is the argument list and the other is an expression. The example above, thus, is written as (LAMBDA (X) (PLUS X 1)) in LISP.

What follows is not an exhaustive list of the basic semantic functions, but includes a sufficient number of them to introduce the essence of the work. We introduce these functions here by informal descriptions. Later on, we will require that more properties be added. Formal definition of these functions will be provided in a later section.

(Note: While we should require all functions to check the types of their arguments, for the sake of simplicity, we have omitted these operations from our descriptions.)

3.1 MAPP

MAPP is a function of two arguments, f, a function, and X, a set. This function simply applies the function f to all of the elements of the set X, and produces a set consisting of the results. We call this function MAPP in order to make a distinction between this and the standard LISP function MAP. Notice that f must be a function of one argument.

When X is a set rather than a multiset and f is a one-to-one single valued function, the result of MAPP will be a set with no duplicates. Otherwise, the result is in general a multiset. In either case, MAPP makes no attempt to detect or delete the duplicates of X or those in its result.

3.2 FILTER

FILTER takes two arguments, p, a predicate (a function with range "true, false",) and X, a set, and returns a set as its result. The result is the set of the elements of X which satisfy the predicate p. In other words, FILTER filters out the members of X which make p true. Notice that p must be a function of one argument.

When X is a set with no duplicates, clearly, the result of FILTER will also be a set with no duplicates. Otherwise, the result is in general a multiset. FILTER never detects or eliminates duplicates of X or those of its result.

3.3 CONCAT

This function takes one argument, X, which must be a set of sets, and returns a set which is the union of the members of X. The use of this function is necessitated by the fact that in VHLL a set of sets is always implicitly replaced by a set (which is the union of the constituent sets) immediately after it is conceptually formed.

CONCAT makes no attempt to detect or delete the duplicates of its argument or those of its result.

3.4 TUPLE

This function takes an indefinite number of arguments and returns a tuple formed out of these arguments, in the given order. In fact, with our convention of representing tuples as lists, this function is very similar to the standard function LIST in LISP.

3.5 POS

POS takes two arguments, i and T, and returns the ith component of the tuple T. If i is not an integer in the correct range, the result of POS is undefined.

3.6 UNQ

UNQ takes one argument, X. It expects X to be a set and returns another set which is informationally equivalent to X, but does not contain any duplicates that X might have. Notice that duplicates in a set carry merely superfluous information, i.e. the user may not conclude any more meaningful information by looking at X, than he may do by looking at (UNQ X), although they are different in extension.

Use of UNQ is necessitated by the fact that in VHLL there are cases where the superfluous information conveyed by the duplicates in a set may hurt, although it can never be useful.

3.7 I

This is the one argument identity function. I returns as its result the very same argument that it receives. If S is a set, then the expression (MAPP I S) is semantically identical to S itself.

3.8 ALW, NVR

These two functions are in fact short hand conventions. They will be used as predicates which take a single argument and regardless of that argument, consistently return true and false, respectively. In other words, ALW is a function of one argument which always returns true, and NVR is a function of one argument which never returns true.

If S is a set, then the expression (FILTER ALW S) is semantically identical to S itself, and (FILTER NVR S) always results in the empty set.

3.9 INCR

This function acts more like a macro rather than a real function. It takes its argument uninterpreted or unevaluated and increments it by 1. For example (INCR A) increments A by 1, actually changing the value of A, and returns the new value of A. Note that the argument of INCR must be a variable.

3.10 EXISTS

This function is similar to FILTER in that it takes two parameters, P and X, a predicate and a set, respectively, and traverses the set X looking for member(s) which make P true. The result of EXISTS is not a set, but a boolean value. The result will be true if there exists at least one member of X which makes P true, and will be false otherwise.

3.11 ALL

This function is analogous to EXISTS. The only difference between ALL and EXISTS is that ALL returns true as its result if and only if all members of the set X satisfy the predicate P.

4.0 VHLL FUNCTIONS

In this section we discuss different sorts of function application in VHLL, and give formal semantic definitions for each, in the form of generic rules.

The power of VHLL, regarded as a data base query/manipulation language, stems from the fact that it includes a powerful function mechanism. The basic concepts of this mechanism are:

- All VHLL functions are treated semantically as single argument functions. A VHLL function with more than one argument is envisioned as a function which operates on an ordered list formed out of those arguments, the ordered list being its single argument. Notice that this is purely a semantic convention with no syntactic implications.

- There are two classes of functions; those that are defined to take set arguments (like the built-in function AVG which returns the average of the members of its argument set), and functions defined over element arguments. An element is either a scalar or a tuple.

- The result of a function can be a set or an element. A function that (by definition) returns a set as its result will be called a multi-valued function.

- The name of any association (relationship) can be used as a function mapping an element argument to either a set of values (a one-to-many or a many-to-many association), or a single element (a one-to-one or a many-to-one association).

- Any function defined for an element argument can be applied to a set argument, provided that the types of the argument and of the set members are compatible. VHLL defines the result of such application as the union of values returned by the function when applied to the individual members of the set.

- The result of applying any function to an argument for which it is not defined is the special value represented as UNDEF. Any function applied to UNDEF results in UNDEF. Although UNDEF may be a member of a set, this fact is never detectable, i.e. a set with or without an UNDEF as a member is the same.

4.1 SINGLE-VALUED FUNCTION, ELEMENT ARGUMENT

The simplest form of function application is to apply a single valued function to an argument which is not a set. If f is such a function and x is an element, then, remembering that f and x also represent equivalent entities in the domain of semantics, we have the following generic rule for formally expressing the meaning of such VHLL function references:

$$\boxed{\texttt{f(x)} \quad \Longrightarrow \quad \texttt{(f x)}}$$

RULE 1

Notice that we semantically do distinguish between an element and a set with a single member, even though in VHLL the latter can generally pass where an element is expected.

4.2 SINGLE-VALUED FUNCTION, SET ARGUMENT

Application of a single valued function, such as f, to a set, such as S, is defined to give a set consisting of the results of applications of f to each and every member of S. Formally, we can express this as:

$$\boxed{\texttt{f(S)} \quad \Longrightarrow \quad \texttt{(MAPP f S)}}$$

RULE 2

4.3 MULTI-VALUED FUNCTION, ELEMENT ARGUMENT

This case is similar to the first, which led to Rule 1, in that it is nothing more than application of a function to a single argument. The fact that the result of this application is a set is irrelevant at this point. So we can readily write:

$$\boxed{\texttt{F(x)} \quad \Longrightarrow \quad \texttt{(F x)}}$$

RULE 3

4.4 MULTI-VALUED FUNCTION, SET ARGUMENT

In VHLL, application of a multi-valued function F to a set S is defined to result in a set which is the union of the results of applying F to the elements of S. This can be formally stated as:

$$\boxed{\texttt{F(S)} \quad \Longrightarrow \quad \texttt{(CONCAT (MAPP F S))}}$$

RULE 4

Notice that (MAPP F S) temporarily results in a set of sets, each of which is the result of application of F to one of the members of S. The function CONCAT, then immediately flattens this set of sets out into a simple set.

When S is a singleton set, the result of (MAPP F S) will be a set whose only member is the set returned by F as the result of its application on the only member of S. The CONCAT, then, unfolds this set of a single set into a set. In other words, if we call the sole member of S "e" the two VHLL function references F(e) and F(S) will yield identical results.

5.0 VHLL EXPRESSIONS

In this section we give rules and guidelines for the definition of the semantics of VHLL expressions, and provide some examples of their use.

5.1 SIMPLE WHERE EXPRESSION

The intent of a simple where expression is to select some data elements which satisfy a particular condition. The result of a where expression is always a set. The generic form of a simple where expression in VHLL is:

```
?x WHERE where-clause
```

The ?x is a local dummy variable which must appear in the where-clause as a free variable. We refer to it as the search variable of the where expression. The where-clause provides two kinds of information:

- It indicates a set from which values for the search variable will be drawn. This set is called the **search domain**.
- It gives the condition under which any search variable value (any member of the search domain) should be included in the result of the where expression. We will call this the **search condition**.

Once the search variable, search domain, and search condition of a where expression are determined, it is simple to formally express its semantics using our notation. There is generally a choice involved in the selection of a search domain. If the purpose of the semantic definition were restricted to the expression of meaning alone, then a general rule for the selection could be specified. However, in the context of the second implementation-related purpose, the selection should be made based on storage-organization (target machine) related considerations. The choice of search domain in our examples is somewhat arbitrary. Observe that, as suggested by the examples below, once the search domain is selected the search condition will become obvious; almost everything else in the where-clause will be the search condition.

We will maintain the VHLL search variable names, e.g. ?x above, in our rules and in the examples. However, it should be clear that none of the locally-bound variable names, such as those of search variables, are significant; they may be changed consistently, as long as this change does not cause any ambiguity through misuse of other variable names.

We can formally express the meaning of a simple where expression as:

```
?x WHERE where-clause
    ===>   (FILTER (LAMBDA (?x) (s-cond)) s-domain)
```

RULE 5

The s-cond and s-domain are the appropriate representations of the search condition and search domain components of the where-clause. The filter predicate (LAMBDA (?x) (s-cond)) is a single argument function which returns a truth value. FILTER takes every member of the set described by s-domain and pumps it through the filter predicate. Any member which makes the predicate true will be included in the result set. If a search condition is absent from the where clause, the appropriate predicate for FILTER would be the ALW function. Obviously, FILTER would then become superfluous, but for the time being, we refrain from replacing such an expression with its equivalent, i.e. the search domain.

- Example 1

?emp WHERE ?emp IS_IN EXEMPT AND SALARY(?emp)>10K

This is a very simple VHLL where expression. The search variable is ?x, we take the search domain to be the set EXEMPT, and the search condition is then SALARY(?emp)>10K. Using the generic rule of the where expressions, we can formally present the semantics of this expression as:

```
(FILTER
  (LAMBDA (?emp) (GT (SALARY ?emp) 10K))
  EXEMPT )
```

Here we have assumed that EXEMPT is the counterpart set for EXEMPT in the VHLL expression, and that SALARY is a single valued function whose semantic counterpart (also named SALARY) exists.

The second line of this expression, (LAMBDA .....), is the filter predicate, and will be true when the returned result of the function SALARY, applied to the predicate's argument, is greater than 10K. Thus, the result will be the set of EXEMPT employees who earn more than 10K.

The selection of EXEMPT as the search domain is arbitrary; other examples will illustrate additional possibilities. Also, we assume for purposes of this example that EXEMPT is a subset of the domain of function SALARY. Provisions needed when this is not the case are discussed further on.

- Example 2

```
?emp WHERE ?emp IS_IN EXEMPT
```

Here again the search variable is ?emp and the search domain is EXEMPT. The search condition, however, is not given, and as mentioned earlier, this means that the result of the where expression should be the same as the search domain EXEMPT. Using our rules and remembering that in case of absence, the search condition defaults to predicate ALW, we can formally express the semantics of this VHLL expression as:

```
(FILTER ALW EXEMPT)
```

Operationally, this means that all elements of EXEMPT are pumped through the predicate ALW, which indiscriminately returns true. Each of the members of EXEMPT which satisfy the filter predicate are included in the result set by FILTER. Hence, the result set is nothing more (or less) than EXEMPT.

- Example 3

```
?emp WHERE SALARY(?emp)>10K
```

In this example, the search domain selected is the domain of the function SALARY, which we will designate as Dsal. So the semantics of this VHLL expression can formally be written as:

```
(FILTER
  (LAMBDA (?emp) (GT (SALARY ?emp) 10K))
  Dsal )
```

This means that every member of the domain of the function SALARY will be filtered through the filter predicate, which as we saw earlier, returns true for all those ?emp's whose SALARY is greater than 10K.

Notice the implicit assumption that SALARY is a single valued function and that its semantic counterpart exists.

- Example 4

```
?emp WHERE ?emp IS_IN EXEMPT AND SALARY(?emp)>10K
```

This is in fact the first example, but we are revisiting it to discuss other alternatives for search domain selection. In general, an adequate search domain must include at least all objects in the data base which might satisfy the where expression. If we assume that there is no subset/superset relationship between EXEMPT and the domain of SALARY, then the smallest adequate search domain for this case is the intersection of the sets Dsal and EXEMPT. If this were the chosen domain, we would write:

```
(FILTER
  (LAMBDA (?emp) (GT (SALARY ?emp) 10K))
  (INTSCT Dsal EXEMPT) )
```

In general, any superset of the smallest adequate search domain may be chosen in its place. For example, if both of the sets Dsal and EXEMPT are subsets of a larger set, say EMP, then EMP might be chosen as the search domain as well. This may cause SALARY to be applied to an element for which it is not defined. According to rules for function application discussed in section 4, the result of a function (like SALARY) will be the special value UNDEF for any value (member of search domain,) say z, not in its domain. This UNDEF will get rippled through the predicate, causing FILTER to not include z in its result.

- Example 5

```
?emp WHERE SALARY(?emp)>SALARY(HAS_MGR(?emp))
```

Here the search variable is ?emp and the search condition is clearly everything after the keyword WHERE. Formally we can express the meaning of the above expression as:

```
(FILTER
  (LAMBDA (?emp) (GT
                   (SALARY ?emp)
                   (SALARY (HAS_MGR ?emp)) ))
  Dsal )
```

We have assumed here that HAS_MGR has a semantic counterpart and that it is a single valued function.

The above formal expression simply considers all employees who have a salary, and pumps them through the filter function. The result will be the set of all such employees whose salary is greater than that of their manager.

- Example 6

```
?emp WHERE SALARY(?emp)>SALARY(HAS_MGR(?emp))
   AND ?emp IS_IN EXEMPT
```

Notice the similarity between this and the previous example. Without more explanation, we can write:

```
(FILTER
  (LAMBDA (?emp) (GT
                   (SALARY ?emp)
                   (SALARY (HAS_MGR ?emp)) ))
    EXEMPT )
```

* Example 7

```
?part WHERE SUPPLIES:('PARTS_INC',?part)
```

As discussed earlier, if we can confirm that a counterpart function for SUPPLIES actually exists in the domain of semantics, then this expression can be readily written as:

```
(FILTER
  ALW
  (SUPPLIES 'PARTS_INC') )
```

But the purpose of this example is to show how a VHLL function reference with no counterpart function can be handled. The following expression represents the meaning of this example:

```
(FILTER
  ALW
  (MAPP
    (LAMBDA (u) (POS 2 u))
    (FILTER
      (LAMBDA (t) (EQ (POS 1 t) 'PARTS_INC'))
      SUPPLIES ) ) )
```

Notice that the same result would be obtained if we were dealing with the VHLL expression:

```
?part WHERE ?part IS_IN SUPPLIES('PARTS_INC')
```

provided that no counterpart function existed for SUPPLIES. Conversely, with the assumption that the counterpart function SUPPLIES exists, both VHLL expressions will result in the first expression above.

* Example 8

```
?part WHERE SUPPLIES:('PARTS_INC',?part)
   AND COST(?part)>5
```

Again, we assume that no counterpart function exists for SUPPLIES. This is clearly very similar to the previous example, the difference being that here the search condition is specified.

```
(FILTER
  (LAMBDA (?part) (GT (COST ?part) 5))
  (MAPP
    (LAMBDA (u) (POS 2 u))
    (FILTER
      (LAMBDA (t) (EQ (POS 1 t) 'PARTS_INC'))
      SUPPLIES ) ) )
```

Notice the implicit assumptions about COST.

5.2 MORE COMPLEX WHERE EXPRESSIONS

In more complex where expressions the word WHERE is preceded by a term more complex than a reference to a single dummy variable. The term can include more than one dummy variable and may call for additional processing on each filtered value or group of values (one for each dummy variable). In the domain of semantics, once the where-clause results are obtained in a set, further processing can be performed on them by MAPPing the appropriate function on that set. For example, if the expression to the left of the WHERE involves application of a function to a single variable, we can use Rule 6.

```
g(?x) WHERE where-clause
===> (MAPP
         g
         (FILTER (LAMBDA (?x) (s-cond)) s-domain) )
```

RULE 6

Here, again, the s-cond and S-domain are the appropriate representations of the search condition and search domain components of the where-clause.

- Example 9

```
SOC_SEC(?emp) WHERE ?emp IS_IN EXEMPT AND SALARY(?emp)>10K
```

We can simply follow the rules and derive the formal meaning of this expression as:

```
(MAPP
  SOC_SEC
  (FILTER
    (LAMBDA (?emp) (GT (SALARY ?emp) 10K))
    EXEMPT ) )
```

Of course, we have assumed that SOC_SEC and SALARY are single valued functions with counterparts.

- Example 10

```
              <?emp,?mgr> WHERE SALARY(?emp)>SALARY(?mgr)
                      AND HAS_MGR:(?emp,?mgr)
```

Here we assume that a counterpart function exists for HAS_MGR. We further assume that this function is a single valued function. The next example considers the case where either of these assumptions does not hold.

```
    (MAPP
      (LAMBDA (e) (TUPLE e (HAS_MGR e)))
      (FILTER
        (LAMBDA (?emp)
          (GT (SALARY ?emp) (SALARY (HAS_MGR ?emp))) )
        EMP ) )
```

In effect, we have taken the HAS_MGR:(?emp,?mgr) phrase to be identical to ?mgr=HAS_MGR(?emp) and then replaced the local variable ?mgr with its value.

Notice also the implicit assumption that the domain EMP is a superset of the domains of the two functions SALARY and HAS_MGR.

- Example 11

```
              <?emp,?mgr> WHERE SALARY(?emp)>SALARY(?mgr)
                        AND HAS_MGR:(?emp,?mgr)
```

Here we assume that no counterpart function exists for HAS_MGR. As a result, the HAS_MGR:(?emp,?mgr) is interpreted as <?emp,?mgr> IS_IN HAS_MGR. By creating a new search variable, say t, on the search domain HAS_MGR, we can derive the relationships between the original search variables and the new one. Clearly, ?emp is identical to (POS 1 t) and, likewise, ?mgr is identical to (POS 2 t). Hence we derive:

```
    (FILTER
      (LAMBDA (t)
        (GT
          (SALARY (POS 1 t))
          (SALARY (POS 2 t)) ) )
      HAS_MGR )
```

- Example 12

```
         ?emp WHERE EXISTS(?mgr WHERE SALARY(?emp)>SALARY(?mgr)
                              AND ?mgr IS_IN HAS_MGR(EMP))
```

Given that the variable ?emp is to range over the domain of the function SALARY, Dsal, and assuming that a counterpart function for HAS_MGR exists, we may write:

```
(FILTER
  (LAMBDA (?emp)
    (EXISTS
     (LAMBDA (?mgr) (GT (SALARY ?emp) (SALARY ?mgr)))
     (MAPP HAS_MGR EMP) ) )
  Dsal )
```

In the above expression, the filter predicate forms the set of all managers by MAPPing the function HAS_MGR onto the set EMP, and then searches the resulting set for a ?mgr whose SALARY is less than the SALARY of a given ?emp. FILTER simply loops over the set Dsal, selecting the ?emp's which make the filter predicate true.

6.0 SEMANTIC FUNCTIONS (REVISITED)

Mechanical application of the rules discussed in sections 4 and 5 will generally yield long, "complex" semantic expressions for most VHLL expressions. (The "complexity" is in fact due to the relatively large number of functions involved, which generally are applied to relatively simple arguments and potentially large sets.) These expressions can often be transformed into equivalent "simpler" expressions which are computationally less expensive.

We first observe that, from an operational point of view, it is less expensive to traverse a set once, applying n functions to each of its members, than to traverse the set (or ones of equivalent length) n times, each time applying one of the functions.

A second observation is that the amount of intermediate result which has to be held affects the cost of computation. An intermediate result is any result which is produced by a part of an expression and is destined to be used in another part of the same expression. In a set oriented language, often these intermediate results are large sets and any tricks that can be used to avoid the necessity of their formation would result in a significant cost reduction.

We will see that the first observation will lead to the equivalent of "loop paralleling" in classical optimization; even though in our case the "loops" are very much implicit and imbedded within the functions. The second (and first) observation leads to another simplification guideline: the equivalent of "loop jamming" in classical optimization, i.e. composition of functions.

The rest of this section develops some formalism that can be used to handle these two optimization techniques in particular. We intend to apply this formalism to "simplify" the expressions we obtain from our operational semantics system. To be able to do so we will require that the semantic functions introduced in section 3 display some specific structural properties. We will discuss these properties, and then give formal definitions for our semantic functions satisfying both the required functional, and structural properties.

This method is inspired by the work of W. Burge in [2], but there are differences (other than notation) between our functions and those in [2]. One difference is due to our use of currying which permits a more explicit notation than in [2]. The major difference, however, is that we introduce an additional argument, C, for the looping function L, which is the termination condition (see below).

6.1 LOOPING FUNCTION

Any function that works on a set contains, somewhere in its logic, a mechanism which enables it to loop over the members of that set. If we had a generalized looping function which we could use to define other set manipulator functions, then, intuitively, the latter would be structurally more homogeneous and thus more susceptible to the loop paralleling and loop jamming transformations discussed above. The following is a definition for such a generalized looping function. This definition uses the program feature of LISP, i.e., it uses explicit control statements, for practical reasons; this point, however, is irrelevant and a recursive functional definition can replace the present one.

```
(DE L (C A G X)
   (PROG (Y Z)
     (SETQ Y A)
     (SETQ Z X)
LOOP (COND
      ( (OR (C Y) (NULL Z)) (RETURN Y))
      (T
        (SEQ
          (SETQ Y ( (G (CAR Z)) Y))
          (SETQ Z (CDR Z)) ) ) )
     (GO LOOP) ) )
```

DE is a function which we shall use to define other functions with. It takes three arguments, a name, an argument list, and a function body. Referring to the definition above, L is a function of four arguments, C, A, G, and X. X is the set (a list in LISP) over whose elements L is to loop. G is a functional (a function whose result is yet another function) and A is an initial value for the result. L loops over X and applies G to each of its members. The resulting function is then applied to the partial result, accumulated in Y, whose initial value is A. The loop ends when either all members of X are considered or when predicate C becomes true for Y, the partial result. For reasons which will become clear later, some limitations must be put on C. Essentially we do not permit C to loop over the partial result, if in fact the latter is a set.

Notice that both C and G, as well as the result of G, must be functions of only one argument. This restriction, particularly on G, is very important and may make some function definitions more difficult. To overcome this difficulty we use "currying", discussed in the next section.

6.2 CURRYING

Any function of more than one argument can always be defined using a sequence of single argument functionals. This latter form is called the curried form of the function. As an example, consider the normal addition operation on two integers represented by the function PLUS. Thus (PLUS i j) returns the sum of i and j, two integers. Now, assume we have the infinite family of single argument functions ADD1, ADD2, ..., ADDn available with a one to one correspondence to the set of integers. Each ADDi takes only one integer argument and increments that argument by i. It is easy to see that (PLUS i j) is always equal to (ADDi J), i.e. for all i we can find a member of the ADD family of functions which satisfies the equality. (We have conveniently named the members of this family such that the i itself designates the correct member.) The curried form of PLUS now can be understood as a function which using (may be implicitly and very indirectly) the ADD family of functions, expresses the same intention as that of PLUS.

The difference between PLUS and its curried form is that PLUS maps from the domain INTEGER X INTEGER to the codomain (range) INTEGER, whereas its curried form maps from the domain of INTEGERs to the codomain of the ADD family of functions, each member of which is, in itself, a mapping from INTEGERs to INTEGERs. Thus:

```
          PLUS: INTEGER X INTEGER ---> INTEGER

 Curried form of PLUS: INTEGER ---> (INTEGER ---> INTEGER)
           or simply: INTEGER ---> INTEGER ---> INTEGER
```

To restate more formally, we know that a function f with range D, which takes n arguments from domains D1, D2, ..., Dn, respectively, can be defined as the mapping:

```
f: D1 X D2 X ... X Dn ---> D
```

There always exists a compatible function g defined as:

```
g: D1 ---> D2 ---> ... ---> Dn ---> D
```

(where right-nested parenthesis are deleted for simplicity) with the same intention as f; that is:

```
(f d1,d2, ...,dn) = ((...((g d1) d2) ...) dn)
```

for all d1, d2, ..., dn in D1, D2, ..., Dn, respectively. The function g is called the curried form of f, after professor H. B. Curry of Penn State [3,6]. The difference between f and g is that f is a mapping from a domain which is the cartesian product of D1 through Dn, to the range D. On the other hand, g is a mapping from domain D1 to a range which itself is a set of mappings. Each particular member of this range in turn is a mapping from domain D2 to yet another set of mappings, etc. The final range of these series of mappings is D.

Now, we define a new function called CURRY which transforms a function to its equivalent curried form. CURRY takes a function name and a list of dummy arguments and returns the curried form of that function. As an example consider PLUS, a function of two arguments. The expression:

```
                    (CURRY PLUS A B)
```

is the curried form of this function, i.e., a functional which when applied twice gives the sum of its two arguments. In the above expression, A and B are merely dummy arguments. Assuming that CPLUS is defined to be the curried form of PLUS, as the above expression, then we have:

```
              ((CPLUS 6) 8) = (PLUS 6 8) = 14
```

Notice the requirement that, in case of PLUS, both arguments must be available before the function is applied; a restriction that is removed in case of CPLUS.

6.3 FUNCTIONS BASED ON L

We have, to this point, defined a looping function L which can be used to give a homogeneous structure to the definition of set manipulator functions. We have also described "currying", for transforming functions to a homogeneous single argument form. We will now illustrate how L is actually used to define other functions. Consider a function SUM which is to take a set (list) of numbers as its argument and is to return their sum as its result. We can define this function as:

```
          (DE SUM (X)
              (L NVR 0 (CURRY PLUS A B) X) )
```

Here L loops over the set X and will not terminate unless the entire set is considered (due to NVR.) The initial value for the (partial) result is 0 and (CURRY PLUS A B) keeps adding the members of X to this partial result until the loop completes.

As a second example, consider COUNT, a close relative of SUM, which is supposed to count the members of its argument set (list). COUNT is very much like SUM, the only difference being that rather than adding the very member of the set, the function in COUNT's definition should add 1 to the partial result in lieu of each member. This can be simply achieved by forming a composite function out of the function used in the definition of SUM and the constant function 1, i.e. (LAMBDA (Y) 1) which indiscriminately returns 1 when applied to any argument. Thus we have:

```
  (DE COUNT (X)
      (L NVR 0 (COMB (CURRY PLUS A B) (LAMBDA (Y) 1)) X) )
```

COMB is the function composition operator, i.e., if h and k are single argument functions, then (COMB h k) is our equivalent of the composite function h•k.

6.4 FORMAL DEFINITION OF BASIC SEMANTIC FUNCTIONS

We are now ready to present formal definitions for some of the basic semantic functions introduced in section 3. For reasons which will become clear later when we discuss a certain class of optimization in section 7, we prefer to define a more generalized version for some of these functions first, and then use these auxiliary functions to define the original ones as their special cases.

6.4.1 EXISTS

The following definition for the function EXISTS (of section 3.10) illustrates how we may take advantage of the termination predicate C in L. Observe that as soon as a member is found in X which satisfies P, there is no reason to continue the loop.

```
(DE EXISTS (P X)
    (L I NIL (COMB (CURRY OR A B) P) X) )
```

Because the terminating predicate is I, the loop will terminate immediately after the (partial) result, whose initial value is NIL, becomes true. The composite function (COMB (CURRY OR A B) P) computes (P x), a boolean, for every x member of X, and this will be then ORed to the partial result. The first x encountered during the traversal of X which makes P true, will naturally make the partial result true too. Consequently, the terminating predicate, I, will become true for the partial result and L will terminate.

6.4.2 ALL

The function ALL need not continue with examination of the rest of the members of X, after it finds the first member which makes P false. Thus we may define ALL in a manner analogous to EXISTS as:

```
(DE ALL (P X)
    (L NOT T (COMB (CURRY AND A B) P) X) )
```

6.4.3 MAPP

The first auxiliary function we introduce is CMAP, a generalized version of MAPP.

```
(DE CMAP (C A G F X)
    (L C A (COMB G F) X) )
```

The intention of CMAP should be clear; it keeps applying the composite function G•F to the members of set X, applying the resulting function to the partial result (whose initial value is A) each time, until either condition C becomes true for the partial result or all members of X are considered.

It is a trivial task now to define MAPP in terms of CMAP:

```
(DE MAPP (F X)
    (CMAP NVR NIL INS F X) )
```

The function INS is the curried form of a function which inserts an element (1st argument) into a set (2nd argument.) For the time being, since the ordering of set members is not an issue, this latter function can be the standard LISP function CONS. Thus INS can be equivalent to:

```
(CURRY CONS A B)
```

6.4.4 FILTER

The generalized form for FILTER is defined as:

```
(DE CFILTER (C A G F X)
    (L C A ( (FILTH G) F) X) )
```

The function FILTH plays a role very much similar to the composition operator in the definition of CMAP. FILTH is in fact the curried form of a function which implements the notion of conditional selection of the members of X. It is easier to first consider its uncurried version FLT, defined below:

```
(DE FLT (G P X Y)
    (COND
      ( (P X) ( (G X) Y))
      (T Y) ) )
```

FLT takes four arguments, G, P, X, and Y, which are, respectively, a function of two arguments in its curried form, a predicate, a new set member, and a partial result. The result of FLT will be Y, the partial result, if the member X does not satisfy the predicate P. If (P X) is true however, the result will be that returned by applying G to X and Y. FILTH is simply the curried form of FLT. Notice that ((FILTH G) F) in the definition of CFILTER is itself a curried function of two arguments, as it should be; the remaining two arguments will be supplied to this function by L.

CFILTER, in essence, loops over the members of X, terminating if the partial result satisfies condition C, and modifies the partial result using function G only when the member under consideration makes predicate F true.

Our FILTER function is only a special case of CFILTER where the loop is to run to its completion, the initial value for the partial result is NIL, and function G is INS. Thus:

```
(DE FILTER (P X)
    (CFILTER NVR NIL INS P X) )
```

6.4.5 CONCAT

Another function we used earlier is CONCAT, whose generalized version CCONCAT, can be defined as:

```
(DE CCONCAT (C A G X)
    (L C A ( (CONCH C) G) X) )
```

Function CONCH is the curried form of a variant of L named CNCH. The only difference between L and CNCH, as can be seen, is the order of their arguments.

```
(DE CNCH (C G X A) (L C A G X))
```

Our CONCAT function is a simple case of CCONCAT, where the terminating condition is NVR, the initial value is NIL, and the function to be applied is INS:

```
(DE CONCAT (X)
    (CCONCAT NVR NIL INS X) )
```

7.0 CODE IMPROVEMENTS

Suppose that somehow we obtained the expression below as the meaning of a peculiar VHLL expression which evaluates the average of a set of integers, say INTS:

```
(QUOTIENT (SUM INTS) (COUNT INTS))
```

There is another form for this expression which is computationally less expensive. We need not apply the two functions in any particular sequential order; hence we need not loop over the set INTS twice. We may instead define a function which, when applied to INTS, would produce a list of the two values yielded by SUM and COUNT. We can then extract the individual results and apply the function QUOTIENT (or anything else) to them. This is an example of what is classically called "loop paralleling". Notice that we may be able to parallel more than two functions (loops) in a complex expression. Such transformations are obviously most important where the size of the "input" set is such that it must be kept on secondary storage.

Now suppose we have an expression of the form:

```
(f (g X))
```

where f and g are two functions and X is a set. If both f and g are such that they somehow traverse the entire set that is passed to them as their arguments, and the result of g itself is a set produced during the traversal, then again we can simplify this expression. As it is, this expression applies g to the set X and then applies f to the resulting set, thus requiring two function applications (two loops), and operationally speaking, also builds a (potentially large) set as an intermediate result. We can transform the above expression to one which applies an equivalent of the composite function f•g to the set X directly. Such expression involves only one function application and does not build an intermediate result (not a set intermediate result anyway). This transformation is called jamming (of loops or functions). The composition operator (•) will be represented by the function COMB, as noted above.

To apply such transformations, a process would scan through the semantic expressions looking for specific patterns, and replace such patterns with more efficient equivalents. In this section we discuss the kinds of rules needed. The material in this section is based on work presented in [2] which is more detailed in some respects. Our termination condition (the first argument of looping function L) has advantages over using the "exit function" of [2] in that when the exit function is used, it restricts use of jamming and paralleling rules (see below). The way that paralleling is handled is quite different here due to presence of a termination condition.

7.1 JAMMING

The function L, presented earlier, has the property that it can "absorb" other set manipulator functions which can be expressed in terms of L [2]. The way each such function is absorbed depends on the particular function involved, and is specified by a "jamming rule".

Each jamming rule permits an SMSP (set-manipulating set-producing) function to be absorbed by a preceding L, i.e., if F is an SMSP function, then a sequence like:

```
                    (L C A G (F ... S ...)),
```

can be transformed using the jamming rule associated with F. In the above expression F traverses the set S and somehow produces a second set. This latter set is then in turn traversed by L. The jamming rule associated with F permits the same result to be obtained through only one traversal of S, in its worst case. The nature of the rule depends on the intention of F.

We have defined the basic semantic functions in such a way as to allow the expression of jamming rules. For example, we have defined the functions MAPP and CMAP such that a sequence like:

```
                    (L C A G (MAPP F S))
```

is equivalent to:

```
                    (CMAP C A G F S)
```

To verify this, consider the original expression. It first builds up a set out of the results of F applied to the members of S, and then loops over this set, applying G to its members, terminating either when the whole set has been traversed, or when the partial result satisfies C. The exact same thing happens in case of CMAP, except that, rather than applying F to the entire set first and building an intermediate set, CMAP simply traverses S once and applies G•F to its members. We use this equivalence as the jamming rule for MAPP. Note that CMAP, as well as other functions like it which are not absorbed by L, can themselves be replaced by their definitions which, in our case, will be in terms of L. This guarantees jamming of a sequence of jammable functions. For example:

```
   (MAPP F1 (MAPP F2 (MAPP F3 S)))

can be transformed:

by the definition of MAPP, into:
   (CMAP NVR NIL INS F1 (MAPP F2 (MAPP F3 S)))

by the definition of CMAP, into:
   (L NVR NIL (COMB INS F1) (MAPP F2 (MAPP F 3 S)))

by the jamming rule for MAPP, into:
   (CMAP NVR NIL (COMB INS F1) F2 (MAPP F3 S))

by the definition of CMAPP, into:
   (L NVR NIL (COMB (COMB INS F1) F2) (MAPP F3 S))

by the jamming rule for MAPP, into:
   (CMAP NVR NIL (COMB (COMB INS F1) F2) F3 S)

and finally, by the definition of CMAP, into:
   (L NVR NIL (COMB (COMB (COMB INS F1) F2) F3) S)
```

The other SMSP functions which we used are FILTER, CONCAT, and APEND. The jamming rule for FILTER specifies that the sequence:

```
(L C A G (FILTER P S))
```

can be replaced by:

```
(CFILTER C A G P S)
```

The validity of this rule can be verified by checking the definition of CFILTER. Notice that with these two rules we are able to convert any sequence of MAPP and FILTER, applied in a series, to a single application of L.

The rules for jamming of CONCAT and APEND given below can be verified similarly. For CONCAT we have

```
(L C A G (CONCAT S))
```

is replaceable by

```
(CCONCAT C A G S),
```

and for APEND, we have

```
(L C A G (APEND S1 S2))
```

can be replaced by

```
(CAPEND C A G S1 S2).
```

7.2 PARALLELING

Whenever a set is traversed more than once to produce several results, we can parallel the production of these results into a single traversal of the set. Assume that we have the following two applications of L somewhere in an expression:

```
... (L C1 A1 G1 X) ... (L C2 A2 G2 X) ...
```

Provided that the two references to L are independent, we may wish to replace the above expression with the following sequence:

```
(SETQ TEMP (L C A G X))
(SETQ M (EX1 TEMP))
(SETQ N (EX2 TEMP))
... M ... N ...
```

where TEMP, M, and N are temporary unique identifiers and SETQ is the standard LISP's assignment operator which assigns the value of its second argument to the first. The two references to L are independent if C2, A2, and G2 are in no way dependent on the result of the first L, and the expression does not modify X (at least not in between the two L's.)

The latter sequence replaces the two references to L by a single reference which produces a list that includes the original results. EX1 and EX2 are two functions which extract these results from the list returned by the parallel L. The exact form of EX1 and EX2 depends on the exact format of the result of the parallel L and will be given later. These extracted results are then assigned to (unique) variables which substitute their corresponding references to L in the original expression. Notice that this last expression sequence involves only one loop over the set X, and that if there is yet another reference to L involving X in the expression, again these two could be paralleled in a similar fashion. Our target now is to determine the form of the functions C and G, and also that of A. We already know that C must be a single argument function, G must be a function of two arguments in curried form, and A must be an initial value having the same format as the (partial) result. We also know that the intention of C is to terminate L when both C1 and C2 become true for their corresponding partial results.

The effect of G must be equivalent to taking the partial result of the parallel L, extracting each of the partial results expected by G1 and G2, applying G1 and G2 to the given member of the set and then to their corresponding partial results, and, finally, forming the new composite partial result by combining the two new partial results returned by G1 and G2.

Observe that the parallel L can, in general, run beyond the point where either of the two conditions C1 and C2 become true for their corresponding partial results. We may wish to prevent further application of G1 or G2 after the corresponding Ci becomes true, even though it is not necessary to do so. The following function will be used to implement this conditional prevention:

```
(DE GF (C G X Y)
    (COND
      ( (C Y) (LIST T Y))
      (T (LIST NIL ((G X) Y))) ) )
```

GF takes a predicate, C, a two argument curried function, G, a partial result, Y, and a set member X, and returns a result, say Z, which is a list that includes a new partial result. GF first tests to see if the partial result Y satisfies the predicate C. If so, then the new partial result will be the same as Y. Otherwise, the new partial result is obtained by applying G to X and then to Y. The returned result of GF, Z, is a two element list whose second element is the new partial result. The first element in Z is a truth value indicating whether (C Y) was true or false.

Using GF and JY, which will be discussed below, GP1 implements the intention of G (but, of course, is not in curried form).

```
(DE GP1 (C1 G1 C2 G2 X Y)
    (JY
      (GF C1 G1 X (EX1 Y))
      (GF C2 G2 X (EX2 Y)) ) )
```

Given the two predicates C1 and C2, functions G1 and G2, the set member X, and the composite partial result Y, GP1 extracts each of the individual partial results using functions EX1 and EX2. It then passes them together with their corresponding Ci and Gi, to GF to obtain the two new partial results. GP1 then uses JY to combine these results into a new composite partial result for the parallel L.

The format of the arguments of JY is dictated by GF. The format of the result of JY dictates the format of the (partial) result of the parallel L, and thus, determines the form of A as well as the two functions EX1 and EX2, and also that of the predicate C.

```
(DE JY (Z1 Z2)
    (LIST
      (AND (CAR Z1) (CAR Z2))
      (CADR Z1)
      (CADR Z2) ) )
```

With the above definition, JY returns a list of three elements, the second and third of which are the two partial results and the first being a conjunction of two truth values. Because these truth values (computed by GF) each determine whether their corresponding Ci is satisfied, the first element in the result of JY determines whether the parallel L should halt.

C then, simply needs to examine the first element of the partial result of the parallel L; C is thus identical to CAR.

EX1 and EX2, similarly, merely need to extract the second and the third elements of the partial result, and are identical to LISP functions CADR and CADDR, respectively.

The initial value for the partial result of the parallel L, A, is to combine the two initial values A1 and A2, together with a truth value of false, into a list. A is thus (LIST NIL A1 A2).

Assuming that GP is the curried form of GP1, G is exactly GP applied to the first four arguments, i.e., ((((GP C1) G1) C2) G2). The parallel L is thus composed as:

```
(L
  CAR
  (LIST NIL A1 A2)
  ((((GP C1) G1) C2) G2)
  X )
```

7.3 OTHER OPTIMIZATIONS

In addition to the two types of improvements discussed in the previous sections, there are other important transformations that must be considered. Some, such as constant folding, removal of common subexpressions, and relocation of costly invariant expressions to the outside of loops are orthogonal to the ones discussed above. There are other issues directly related to optimizations of the prior sections. Two of these are the problem of transformation ordering and the necessity for transformation criteria based on the information contents of the data base. To illustrate these issues, consider Example 12 of section 5.2 again. The expression:

```
(FILTER
  (LAMBDA (?emp)
    (EXISTS
     (LAMBDA (?mgr) (GT (SALARY ?emp) (SALARY ?mgr)))
     (MAPP HAS_MGR EMP) ) )
  Dsal ),
```

derived as the meaning of:

```
?emp WHERE EXISTS(?mgr WHERE SALARY(?emp)>SALARY(?mgr)
                           AND ?mgr IS_IN HAS_MGR(EMP))
```

does serve the purpose of formal semantics derivation. But we would like to convert it to a more efficient form for execution. It is clear that the FILTER will be replaced by its definition and implies one occurrence of the looping function L. The filter predicate itself involves a reference to MAPP and one to EXISTS, both of which are defined in terms of L. For simplicity, let us call the predicate (first argument) of EXISTS, P. The EXISTS and MAPP can be jammed as follows.

```
   By replacing the EXISTS with its definition:
      (L I NIL (COMB (CURRY OR A B) P) (MAPP HAS_MGR EMP))

   By the jamming rule for MAPP:
      (CMAP I NIL (COMB (CURRY OR A B) P) HAS_MGR EMP)

   By the definition of CMAP:
      (L
        I
        NIL
        (COMB (COMB (CURRY OR A B) P) HAS_MGR)
        EMP )
```

```
   The whole expression can then be written as:

      (FILTER
        (LAMBDA (?emp)
          (L
            I
            NIL
            (COMB (COMB (CURRY OR A B) P) HAS_MGR)
            EMP ) )
        Dsal ),
```

where P, as mentioned above, is a short-hand for (LAMBDA (?mgr) (GT (SALARY ?emp) (SALARY ?mgr))) and FILTER itself can be rewritten in terms of L in the obvious fashion.

This result is more efficient because it involves one less reference to L, i.e., one less loop, than the original one. However, it is possible to derive another expression by application of a rule for relocation of costly invariant expression to the outside of the FILTER "loop":

```
  (SETQ X (UNQ (MAPP HAS_MGR EMP)))
  (FILTER
    (LAMBDA (?emp)
      (EXISTS
        (LAMBDA (?mgr) (GT (SALARY ?emp) (SALARY ?mgr)))
        X ) )
    Dsal )
```

This last expression could be yet more efficient, even though it involves three references to L and creation of an intermediate result set. If there are roughly as many managers as there are employees, then the previous derivation, with two references to L, is more efficient. But, if there are much fewer managers than employees, then the above expression will be preferable, especially if Dsal is large, because a much smaller set (of managers) will be traversed for each ?emp member of Dsal, and the saving can easily offset the overhead of creating the intermediate result set (X).

This illustrates the significance of the order in which transformation rules are applied to optimize an expression. One can see that application of jamming rules can hinder applicability of a relocation rule, which (depending on the data base contents) could result in a more efficient expression. It also illustrates the necessity for transformation rules sensitive, not only to the semantics of the program, but also to those characteristics of the data base definition and instantiations which influence the statistical attributes of the data base contents.

A third, unavoidable issue is that some optimizations are even less easily classifiable. These seem to involve additional understanding and reconstruction of program "intent", as opposed to mere semantics. One class of such optimizations may be addressable at the VHLL level, while a VHLL expression is being translated to a normalized version, by rephrasing. For example, in the same VHLL expression of Example 12, it is easy to see that the only significance of ?mgr is to link the set EMP to a set of values (SALARY's) which is then used for comparison with SALARY(?emp). Thus, the VHLL expression can be rewritten as:

```
?emp WHERE EXISTS(?s WHERE SALARY(?emp)>?s
                         AND ?s IS_IN SALARY(HAS_MGR(EMP)))
```

which then will mean:

```
(FILTER
  (LAMBDA (?emp)
    (EXISTS
      (LAMBDA (?s) (GT (SALARY ?emp) ?s))
      (MAPP SALARY (MAPP HAS_MGR EMP)) ) )
  Dsal )
```

Again, subject to the knowledge that there are far fewer SALARY's for managers than there are members in EMP, the two MAPPs can be removed from within the loop, as in the previous case; otherwise, the two MAPPs and the EXISTS will be jammed into a single L which would traverse the set EMP.

8.0 ACKNOWLEDGEMENTS

I wish to thank Paula Newman for her constructive remarks and her reading and editing of the (sometimes unreadable) drafts. I also would like to thank Simon Lin for his careful review of the final version.

REFERENCES

[1] Newman, P. S.,
An Atomic Network Programming Language,
IBM Scientific Center Report,
G320-2704, June 1980

[2] Burge, W. H., An Optimizing Technique
for High Level Programming Languages,
IBM T. J. Watson Research Center,
RC 5834 (#25271), February 1976

[3] Gordon, M. J. C., The Denotational
Description of Programming Languages,
Springer-Verlag 1979

[4] Berry, D. M. and Schwartz, R. L.,
Formal Definition of ADA:
Issues and Recommendations,
Computer Science Department, UCLA,
September 1979

[5] Curry, H. B. and Feys, R.,
Combinatory Logic Vol. 1,
North Holland Publications 1958

[6] Backus, J., Can Programming be
Liberated from the Von Neumann Style?,
CACM vol. 21 No. 8, Aug. 1978

[7] Burge, W. H.,
Recursive Programming Techniques,
Addison-Wesley 1975

[8] Newman, P. S.,
Towards an Integrated Development Environment,
(In Preparation)

[9] Mc Carthy, J., et al,
LISP 1.5 Programmers Manual,
MIT Press 1962

[10] Church, A.,
The Calculi of Lambda-Conversion,
Princeton University Press, 1941, 1951

SCIENTIFIC CENTER REPORT INDEXING INFORMATION

1. AUTHOR(S) :
Farhad Arbab

2. TITLE :
Notes on the Semantics and Optimization of a VHLL

3. ORIGINATING DEPARTMENT
Los Angles Scientific Center

4. REPORT NUMBER
G320-2706

5a. NUMBER OF PAGES
38

5b. NUMBER OF REFERENCES
10

9. SUBJECT INDEX TERMS
Formal Semantics
Operational Semantics
Set-Oriented Language
High-Level Optimization
Data Base Manipulation Language

6a. DATE COMPLETED
September, 1980

6b. DATE OF INITIAL PRINTING
December, 1980

6c. DATE OF LAST PRINTING

7. ABSTRACT :

This paper investigates an operational semantics model for a very level data base manipulation language. The target machine of the operational semantics system is an applicative programming system. Use of the target machine in compilation, as the model for an implemetation independent intermediate language, is considered, and its susceptibility to a class of optimization transformations is shown.

8. REMARKS :

1977 IBM LOS ANGELES SCIENTIFIC CENTER OUTSIDE PUBLICATIONS

T. LANG & E. B. FERNANDEZ, Improving the Computation of Lower Bounds for Optimal Schedules, IBM Journal of Research & Development, Vol. 21, No. 3, May 1977, 273-280.

A. INSELBERG, (G320-2684) Variable Geometry Cochlear Model at Low Input Frequencies: A Basis for Compensating Morphological Disorders, IBM Journal of Research & Development, Vol. 21, No. 5, September 1977, 461-478.

T. LANG, E. NAHOURAII, K. KASUGA, E. B. FERNANDEZ, An Architectural Extension for a Large Database System Incorporating a Processor for Disk Search, Proceedings of the 3rd International Conference on Very Large Data Bases, IEEE Computer Society, or ACM, Tokyo, 1977, 204-210.

T. LANG, E. B. FERNANDEZ, R. C. SUMMERS, A System Architecture for Compile-Time Actions in Databases, ACM 77 Proceedings of the Annual Conference, Seattle, Washington, October 17-19, 1977, 11-15.

E. B. FERNANDEZ & C. WOOD, (G320-2685) The Relationship Between Operating System and Database System Security: A Survey, Proceedings of COMPSAC 77, 1st International Computer Software Applications Conference, IEEE Computer Society, Chicago, Ill., November 8-11, 1977, 453-462.

T. LANG, C. WOOD & E. B. FERNANDEZ (G320-2686), Database Buffer Paging in Virtual Storage Systems, ACM Transactions on Database Systems, Vol. 2, No. 4, December 1977, 339-351.

1978 IBM LOS ANGELES SCIENTIFIC CENTER OUTSIDE PUBLICATIONS

B. DIMSDALE, (G320-2692) Convex Cubic Splines, IBM J. Res. Develop., Vol. 22, No. 2, March 1978, 168-178.

E. B. FERNANDEZ, T. LANG, C. WOOD, Effect of Replacement Algorithms on a Paged Buffer Database System, IBM J. Res. Develop., Vol. 22 No. 2, March 1978, 185-196.

A. INSELBERG, (G320-2669) Cochlear Dynamics: the Evolution of a Mathematical Model, Siam Review, Vol. 20, No. 2, April 1978, 301-351.

S. A. JUROVICS, Optimization Applied to the Design of an Energy Efficient Building, IBM Journal of Research and Development, Vol. 22, No. 4, July 1978, 378-385.

E. B. FERNANDEZ, R. C. SUMMERS, T. LANG, & C. D. COLEMAN, (G320-2683) Architectural Support for System Protection and Database Security, IEEE Transactions on Computers, C-27, No. 8, August 1978, 767-771.

R. C. SUMMERS & E. B. FERNANDEZ, An Approach to Data Security, Proceedings of the 8th Australian Computer Conference, September 1, 1978.

D. W. LOW, A Directed Weather Data Filter, IBM Journal of Research & Development, Vol. 22, No. 5, September 1978, 487-497.

STEPHAN A. JUROVICS & DAVID W. LOW, Optimizing the Passive Solar Characteristics of Buildings, Presented at the Winter Annual Meeting of ASME, San Francisco, California, December 10-15, 1978, 43-51.