The Evolving Algebra Semantics of COBOL

Part 1: Programs and Control

Marc Vale

26 April 1993

1 Introduction

1.1 Overview of the Method.

The following is from an earlier version of [GH].

"Evolving algebras were first proposed in [Gu1] (and more recently discussed in [Gu3]) as an improvement upon (a stronger version of) Turing's thesis. One may use an evolving algebra to model any computation. In particular, one may describe an evolving algebra which models a particular computation in "lockstep"; that is, for every step taken by the modeled computation, the evolving algebra takes one step. In addition, one may describe an evolving algebra which models a particular computation at any desired level of abstraction. This is an improvement upon the traditional Turing machine model, where the abstraction level is fixed at a low level and may require many Turing machine steps to simulate one step of an algorithm.

"An evolving algebra contains a description of a first-order logical *signature* which describes the states of an abstract machine, along with a collection of *transition rules* which describe the temporal relationships between states. Once combined with a description of the initial state (that is, a *structure* of the corresponding signature), a computation (or set of computations) is determined.

"Evolving algebras may be used to provide operational semantics for a programming language. A programming language may be viewed as a kind of universal algorithm. It takes a program and data as input and runs the program on the data. An evolving algebra for a programming language describes this type of universal algorithm, thus giving an operational semantics for the programming language.

"These types of semantic specifications may be provided on several abstraction levels for the same language. Having several such algebras is useful, for one can examine the semantics of a particular feature of a programming language at any desired level of abstraction, with unnecessary details omitted.

"Evolving algebras have been used to provide operational semantics for Modula-2 [Mor], Occam [GMs], Prolog [Bo1, Bo2, Bo3, BR1, BR2], Prolog III [BS], Smalltalk [Bl], and C [GH]."

This technical report describes a universal machine for the COBOL programming language. Descriptions of the language are taken from [ANSI], [IBM] and [PK]. This paper is modeled after a preliminary version of [GH].

1.2 Required Knowledge

A basic familiarity with evolving algebras such as that provided by [Gu3] is assumed. A brief description is provided in an appendix for those who need it. This paper is somewhat less formal. Knowledge of COBOL is not necessary for understanding (In fact, it is hoped that this paper will help to understand the language), since we explain all relevant aspects of COBOL as we proceed.

1.2.1 The COBOL Language Standard

COBOL is an acronym for COmmon Business Oriented Language. In its long history the language has been covered by many standards. The standard for use within this paper is documented in [ANSI] and is commonly referred to as COBOL-85. Due to the latitude allowed to the implementor by the standards, there are many dialects, and minor details may differ between implementations of the language.

This report was prepared as directed study for the Master of Science in Computer Science and Engineering.

[ANSI] separates the standard into various modules and for some modules, specifies low and high levels of implementation. This report will discuss the Nucleus, Sequential I-O, Relative I-O, Indexed I-O, Inter-Program Communication, and Sort-Merge modules and usually at level 2. It will not cover the Source Text Manipulation, Report Writer, Communication, Debug, or Segmentation modules.

1.3 Separation of Concerns

Because we are concerned with programming language semantics rather than syntax, we follow [GH] and assume that all syntactic information regarding a given program is available to us at the beginning of the computation through static functions of the algebra which contain that information.

However, a substantial amount of the syntax of the language will be presented because it is presumed that the average reader will not be as familiar with this language as with others.

In order to maintain the focus on semantics, we assume our algebra will evolve without regard to resource bounds. Resource management may be added to an evolving algebra without undue difficulty; see [Gu2] for further information and [Mor] for an example of resource management in an evolving algebra.

1.4 Abstraction Levels

In our report we will present a series of evolving algebras which model fragments of the COBOL programming language. Each algebra will be presented as a refinement of the previous algebra. The final revised algebra will describe significant portions of the COBOL programming language.

Our algebras will focus on the following areas of the COBOL programming language:

Programs and the transfer of control between them
Control statements and the transfer of control within a program (e.g. **IF, ADD**)
Memory allocation and initialization
Expressions and imperative statements
Input/Output

We present the first two algebras fully. For the rest, time does not permit more than an indication of where further exposition could go.

1.4.1 Language Formalities

COBOL programs always consist of four divisions in order: IDENTIFICATION, ENVIRONMENT, DATA, and PROCEDURE. The semantics of each division differs and the syntax of the DATA DIVISION is different from the others. With the exception of the IDENTIFICATION DIVISION, divisions are divided into sections although sections are optional in the PROCEDURE DIVISION. Except for the DATA DIVISION, divisions and sections contain paragraphs. There are specific section and paragraph names which may be used in the IDENTIFICATION DIVISION; the order in which they may be used is fixed, and most of them are optional. The same is true with a different set of names for the ENVIRONMENT DIVISION. It is also true of section names in the DATA DIVISION.

The syntax of COBOL is usually presented through syntax diagrams rather than context-free grammars. We will use these diagrams to illustrate each construct. We follow the conventions of [ANSI]: reserved words are written in upper case, if the word is required it is underlined (Many reserved words are optional); items which the programmer supplies are written in lower case; optional items are enclosed in square brackets; mutually exclusive choices are enclosed in braces, and potential repetition of a construct is shown as ellipsis (...). We make one change to this convention for ease of presentation. Usually mutually exclusive constructs are shown on different lines. For brevity we show them separated by the vertical bar (|), which is not part of the official convention. An example follows:

```
ADD {identifier-1 | literal-1} ... TO {identifier-2 [ROUNDED]} ...
    [ON SIZE ERROR imperative-statement-1]
    [NOT ON SIZE ERROR imperative-statement-2]
[END-ADD]
```

Sometimes it will be necessary to use more than one syntax diagram to illustrate the full range of variations of a construct. We follow the conventions of [ANSI] in referring to these as *formats* of the construct in question.

1.5 Acknowledgments

Professor Yuri Gurevich directed our research. He and Jim Huggins authored [GH] which inspired most of the algebras of the paper. We gratefully acknowledge comments by Raghu Mani.

2 Algebra Zero: The Algebra of Programs

Our initial algebra deals with programs and the passage of control among them. In [ANSI] the element which is executed is referred to as the run unit. We will be studying such a run unit. A run unit may consist of one or more COBOL programs. One program is selected, by a mechanism external to the language and specific to the implementation as the first program of the run unit to execute. This program may in turn transfer control to another program, which may return control to the original program, transfer control to another program, or terminate. We will ignore the possibility of non-termination. The standard prohibits a CALL statement that results directly or indirectly in the execution of the calling program. Control may be returned by a calling program only to the program which called it.

We regard the run unit as a rooted directed acyclic graph with the programs as nodes and calls as edges. Each call lengthens a chain through that graph and each return shortens it. The prohibition in the standard requires that each program have only one parent active in the chain.

2.1 Initial Universes and Functions.

We take from [GH] a substantial number of universes and functions. We display those of a general use before exploring an algebra in detail.

2.1.1 Run Unit Representation and Execution

We borrow from [GH] a universe *tasks* of elements representing tasks which the interpreters of the various algebras must accomplish. The concept is a general one. A task may represent execution of a program, execution of a statement within a program, initialization of a variable, or evaluation of an expression.

It is frequently necessary to indicate the nature of a task. We borrow the universe *tags* from [GH] for this purpose.

Following [GH], we define a dynamic zero-ary function (hereafter *distinguished element*) *CurTask: tasks* (that is, a function with null domain and range *tasks*) which indicates the current task being executed.

We borrow from [GH] the static function *TaskType: tasks → tags* which indicates the action to be performed by the task. We will describe the range of the *TaskType* function, and hence the contents of *tags*, as we proceed. We will often refer to the result of *TaskType* as the type of a task.

It is necessary to indicate the order in which tasks are executed. The static function *NextTask: tasks → tasks* taken from [GH] serves this purpose. For this algebra, *NextTask* is an external function with its value determined by an oracle outside the algebra.

We represent programs by a task of type *program*. We will introduce tasks to represent the COBOL constructs involved as needed.

2.2 Abbreviation: Moveto

We find it convenient to borrow some useful abbreviation from [GH]. This one, *Moveto (Task)*, accomplishes transfer of control to a particular task by modifying the *CurTask* distinguished element appropriately. Its definition is:

CurTask := Task

2.3 Interprogram Communication

The format of the PROGRAM-ID paragraph is:

```
PROGRAM-ID. program-name [IS {COMMON | INITIAL} PROGRAM].
```

COMMON indicates that a program contained within another program may be accessible to programs other than the one containing it. We will be concerned with the semantics of INITIAL in a later algebra. The PROGRAM-ID paragraph declares program-name as the name of this program and makes that name available so that the program may be called.

The CALL statement provides the capability to execute another program, then continue execution in the current program. The formats are:

```
CALL {identifier-1 | literal-1}
    [USING {[BY REFERENCE] {identifier-2} ... |
        BY CONTENT {identifier-2} ...} ...]
    [ON EXCEPTION imperative-statement-1]
    [NOT ON EXCEPTION imperative-statement-2]
[END-CALL]
```

and

```
CALL {identifier-1 | literal-1}
    [USING {[BY REFERENCE] {identifier-2} ... |
        BY CONTENT {identifier-2} ...} ...]
    [ON OVERFLOW imperative-statement-1]
[END-CALL]
```

We will discuss the role of EXCEPTION/NOT EXCEPTION or OVERFLOW in a later algebra.

The statement to return control from a called program to a calling program is:

```
EXIT PROGRAM
```

If the statement occurs in a program which has been called, control returns to the statement in the calling program following the CALL statement which caused control to pass to the called program. If the program has not been called, control passes to the next statement in this program.

The statement to terminate execution is:

```
STOP RUN
```

This statement may be executed in any program in the run unit. It terminates execution of the run unit.

Let us consider the execution of the call. First if the USING clause is present, the expressions in the clause are evaluated. The specifics of this will be discussed in another algebra. Then control is passed to the first task of the program identified by literal-1 or by the value in identifier-1. Execution proceeds in that program until it executes an EXIT PROGRAM statement, at which time control is returned to the statement following the CALL; another CALL which transfers control into another program or STOP RUN.

2.4 Tasks and Functions

Tasks of type *call*, implement the CALL behavior. They are provided with a dynamic partial function *CalledTask: tasks → tasks* which returns a task of type *program*. Tasks of type *program* have a dynamic function *EndTask: tasks → tasks* which contains the task to which to return control when the program ends. Tasks of type *exitprog* implement the exit program behavior. In addition to those tasks created for the EXIT PROGRAM statement, the compiler inserts an *exitprog* task at the physical end of the program. We define the function *Prog: tasks → tasks* as a complete static function for use with *exitprog* and anticipating that it will see further utility. It returns the *program* task in which this task occurs. *Prog* of a *program* task returns the original task when the program is not contained within another program. We represent STOP RUN with a task of type *stop*.

2.5 Transition Rules

The transition rules are:

```
if TaskType(CurTask) = call then
    Moveto(CalledTask(CurTask))
    EndTask(CalledTask(CurTask)) := NextTask(CurTask)
endif
if TaskType(CurTask) = program then
    Moveto(NextTask(CurTask))
endif
if TaskType(CurTask) = exitprog then
    if EndTask(Prog(CurTask)) = ⊥ then
        Moveto(NextTask(CurTask))
    else
        Moveto(EndTask(Prog(CurTask)))
    endif
endif
if TaskType(CurTask) = stop then
    Moveto(⊥)
endif
```

2.6 Initial State

Initially, *CurTask* indicates the first task of type *program* of the run unit. The initial value of *EndTask* for this task is ⊥

3 Algebra One: The Algebra of Control.

This algebra is concerned solely with the PROCEDURE DIVISION.

3.1Initial Universes and Functions.

For this algebra, we add more universes and functions from [GH].

3.1.1 Program Values

One of those is the universe *results* which is the universe of values which may appear as the "result" of a computation. We will specify more precisely the constituents of this universe in later algebras.

3.1.2 Program Representation and Execution

"At various times, we will need certain internal information to describe the nature of a given task or computational process. We accordingly will define a universe *internals*, whose elements will be used to represent this information. We will specify the elements of *internals* as we proceed.

"We define a function *TestValue: task* → *results* which indicates the value of certain expression tasks. For the purposes of this algebra, we will assume that the *TestValue* function is an external function, whose values are determined by an oracle external to the evolving algebra." This will allow us to show how computed values are used by control statements in COBOL while delaying our discussion of how those values are computed.

NextTask now becomes a static internal function linking tasks in the order in which they (normally) would be executed. We intend to supply transition rules for those cases where

NextTask does not indicate the next task (i.e. statement or expression) to be performed once the specified task has been completed. We will further constrain *NextTask* as we proceed.

3.2 Statement Classification

A COBOL statement is recognized by its first word which is commonly referred to as the *verb*. Statements are frequently called sentences and parts thereof are called clauses or phrases. In [ANSI] and [IBM], statements are divided into four categories: imperative, conditional, delimited scope, and compiler directing. We will not discuss compiler directing statements; some of them are concerned with source or listing management, and the rest implement little-used features which we will not have time to explain. Conditional statements determine the subsequent progress of the program based upon evaluation of the conditional expression. Delimited scope statements are conditional statements followed by a scope terminator, usually END- concatenated with the verb. The syntax diagram of page 4 allows the following examples:
An imperative statement.

```
ADD 1 TO A
```

A conditional statement with another imperative statement within its implicitly delimited scope:

```
ADD 1 TO A
    ON SIZE ERROR
        ADD 1 TO B.
```

Note that the addition of the ON SIZE ERROR changes the first statement from imperative to conditional.

A delimited scope statement with another within its scope:

```
ADD 1 TO A
    ON SIZE ERROR
        ADD 1 TO B
END-ADD
```

The utility of this will become more apparent when we discuss compound statements later.

The list of imperative verbs with the clause which converts them into conditionals follows:

verb	clause	verb	clause
ADD	ON SIZE ERROR	DELETE	INVALID KEY
COMPUTE	ON SIZE ERROR	READ	AT END
DIVIDE	ON SIZE ERROR	READ	INVALID KEY
MULTIPLY	ON SIZE ERROR	REWRITE	INVALID KEY
SUBTRACT	ON SIZE ERROR	START	INVALID KEY
STRING	ON OVERFLOW	WRITE	AT END-OF-PAGE
UNSTRING	ON OVERFLOW	WRITE	INVALID KEY
CALL	ON OVERFLOW	RETURN	AT END
CALL	ON EXCEPTION		

Another classification of statements is into arithmetic, data manipulation, input/output, and procedure branching. It is this last category that will supply most of the constructs to be considered in this algebra.

3.3 IMPERATIVE STATEMENTS

In this algebra, in which we are concerned only with the flow of control, the transition rules for imperative statements are:

if *TaskType (CurTask) = imperative-statement* ***then***
Moveto(NextTask(CurTask))
endif

3.4 COMPOUND STATEMENTS

The notion of scope is similar to that of compound statements in languages of the ALGOL family such as Pascal or C. Wherever one imperative statement may be specified, a series of imperative or delimited scope statements may be specified. In the absence of a scope terminator, the period (.) terminates the scope of all previous conditional statements. Note that this precludes the appearance of a period within a compound statement.

3.5 SELECTION STATEMENTS

We can treat the conditional statements described above and the IF statement in the same fashion. We then consider the EVALUATE statement which allows the selection of more than two alternatives. We follow that with consideration of the conditional GO TO which transfers control based upon the value contained.

3.5.1 Conditional Selection Statements

The syntax diagram for the IF statement is:

```
IF condition-1 THEN
    {imperative-statement-1 | NEXT SENTENCE}
[ELSE
    {imperative-statement-2 | NEXT SENTENCE}]
[END-IF]
```

where condition-1 is a logical or relational expression. For the control algebra, this IF statement and the conditional statements discussed before are similar. Both evaluate an expression, and select the next task based upon the result of that evaluation.

There is one possible surprise. For historical reasons, NEXT SENTENCE exits all enclosing scopes, i. e. skips to the next period. This preserves the behavior of programs written to prior standards which did not possess explicit scope delimiters. (Recall that in the absence of scope delimiters the scope of a conditional continues to the next period.)

We will represent an IF statement by the graph shown in Figure 1 using a diagram and the conventions of [GH], where the ovals represent tasks, the arcs represent unary functions, and the boxes represent subgraphs. For example, given:

```
IF A EQUAL 'B'
    ADD 1 TO C
ELSE
    ADD 1 TO D
END-IF
```

The guard *expression* is A EQUAL 'B', the leftmost *stmt* is ADD 1 TO C, and the rightmost *stmt* is ADD 1 TO D.

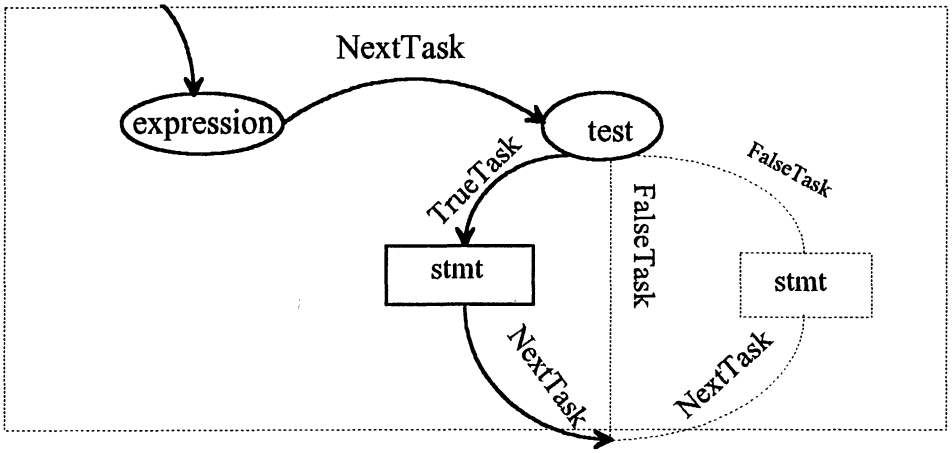

Figure 1: Pictorial description of the IF statement.

If an ELSE clause is not present in an IF statement or only one of the two options of a conditional statement is present, the corresponding task graph omits the right-hand side of Figure 1, with the *FalseTask* function connecting the *test* node to the next task outside of the statement.

"We define static functions *TrueTask: tasks* → *tasks* and *FalseTask: tasks* → *tasks* which indicate the task to be performed if the guard of the selection statement evaluates to true (or false). (We will use these functions in other contexts in our algebra.)

"We will represent the branching decision made in the selection statement by an element of the *tasks* universe for which the *TaskType* function returns *test*.

The transition rule for *expression* is almost the same as that for *imperative-statement*.

if *TaskType (CurTask) = expression* ***then***
  Moveto(NextTask(CurTask))
endif

The transition rules for *test* are:

if *TaskType(CurTask) = test* ***then***
  if *TestValue(CurTask) = TRUE* ***then***
    Moveto(TrueTask(CurTask))
  endif
  if *TestValue(CurTask) = FALSE* ***then***
    Moveto(FalseTask(CurTask))
  endif
endif

3.5.2 Multiple Choice Selection Statement

The syntax diagram for EVALUATE is:

```
EVALUATE {identifier-1 | literal-1 | expression-1 | TRUE | FALSE}
    [ALSO {identifier-2 | literal-2 | expression-2 | TRUE |
        FALSE}] ...
{{WHEN {ANY | condition-1 | TRUE | FALSE |
    [NOT] {identifier-3 | literal-3 | arithmetic-expression-1}
        [{THROUGH | THRU} {identifier-4 | literal-4 |
            arithmetic-expression-2}]}
[ALSO {ANY | condition-2 | TRUE | FALSE |
    [NOT] {identifier-5 | literal-5 | arithmetic-expression-3}
        [{THROUGH | THRU} {identifier-6 | literal-6 |
            arithmetic-expression-4}]}] ...} ...
imperative-statement-1} ...
[WHEN OTHER imperative-statement-2]
[END-EVALUATE]
```

The expressions between the EVALUATE and the first WHEN are referred to individually as selection subjects and collectively as the set of selection subjects. The expressions in each WHEN phrase are referred to as selection objects and collectively as the set of selection objects for that WHEN. The syntax requires that there be exactly as many ALSOs in each set of selection objects as there are in the set of selection subjects and that selection objects match the class (numeric or alphabetic) of the corresponding selection subjects. ALSO binds less strongly than any of the logical operators allowing logical expressions as selection subjects. ANY is allowed for any class and indicates that any value of the corresponding selection subject will result in a successful match. In other words, it indicates a "don't care'" (or wildcard) position in matching the set of selection objects to the set of selection subjects.

The behavior of the EVALUATE statement is: the expression(s) in the main clause are evaluated and saved in a list representing the subject set, then those of the first WHEN clause are evaluated and saved in a list representing the object set which is compared with those of the first list, ignoring positions marked with ANY. If they are equal (or in the case of the THROUGH phrase, the selection subject is within the range of the corresponding selection object), the next imperative statement is executed and control proceeds to the statement following the EVALUATE. If not, the next WHEN is evaluated in a similar fashion. Notice that several WHENs may select the same imperative statement. If none of the WHEN clauses match and a WHEN OTHER clause is present, the imperative statement following it is executed.

Some examples may help to explain the syntax diagram:

```
EVALUATE A
    WHEN 'C'
        ADD 1 TO B
END-EVALUATE
```

A pictorial representation is in Figure 2.

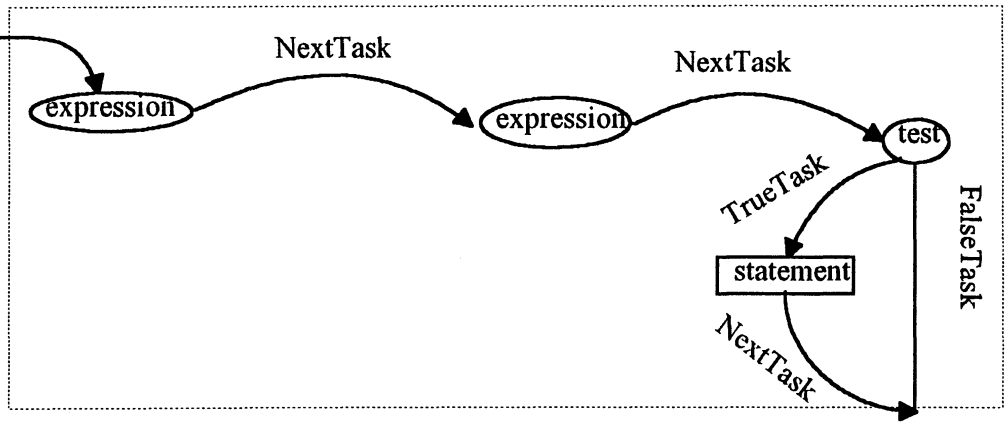

Figure 2: Pictorial representation of an EVALUATE with a single WHEN

Where the leftmost expression is A and the rightmost A = 'C'.

This has the same effect as:

```
IF A EQUAL 'C'
    ADD 1 TO B
END-IF
```

but might be used because the programmer expects the code will need to be modified later to check additional values of A.
A example which is a little more complex is:

```
EVALUATE A ALSO TRUE
    WHEN 'C' ALSO B LESS 99999
        ADD 1 TO B
    WHEN OTHER
        ADD 1 TO UNCLASSIFIABLE
END-EVALUATE
```

(represented in Figure 3.),

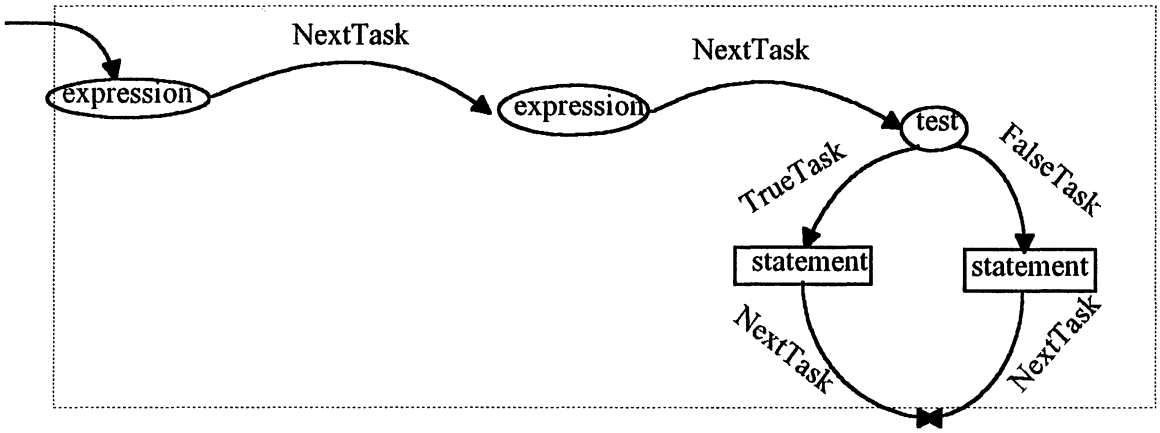

Figure 3: Pictorial description of EVALUATE with WHEN and WHEN OTHERWISE

but this is just equivalent to:

```
IF A EQUAL 'C'
AND B LESS 99999
    ADD 1 TO B
ELSE
    ADD 1 TO UNCLASSIFIABLE
END-IF
```

and both have a logical defect in that they count items as unclassifiable when they are merely uncountable. More specific code for this is:

```
EVALUATE A ALSO TRUE
    WHEN 'C' ALSO B LESS 99999
        ADD 1 TO B
    WHEN 'C' ALSO ANY
        ADD 1 TO OVERFLOWS
    WHEN OTHER
        ADD 1 TO UNCLASSIFIABLE
END-EVALUATE
```

which gives the graph in Figure 4. The equivalent IF statements should be obvious.

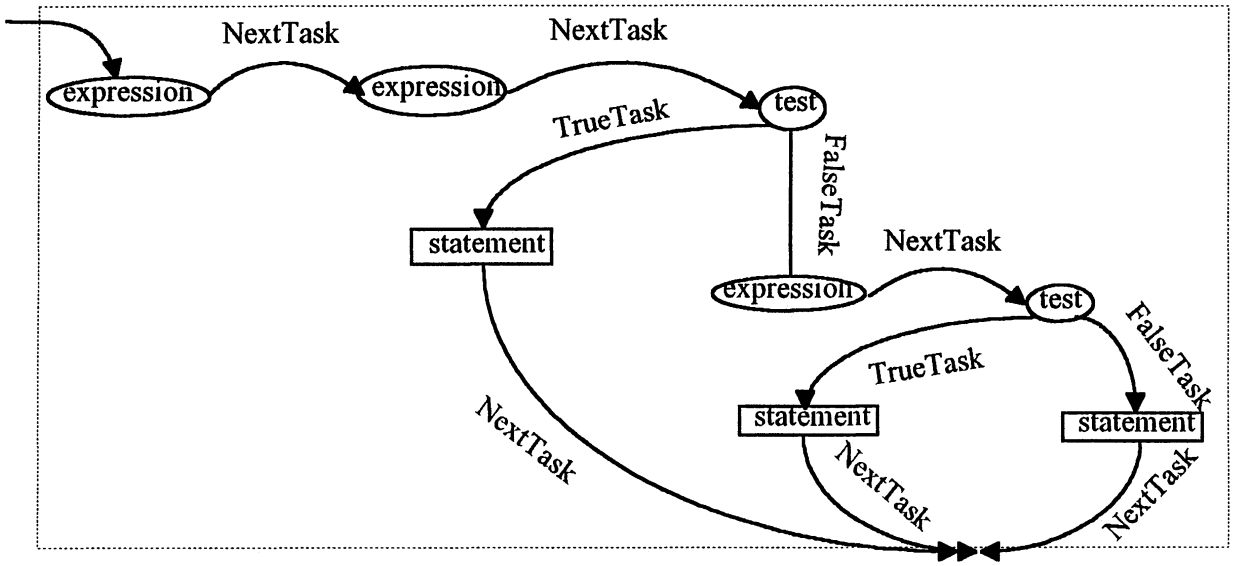

Figure 4. EVALUATE with two WHENs and WHEN OTHERWISE

Because EVALUATE uses no new types of tasks, we need no new transition rules.

3.5.3 Conditional GO TO

The syntax diagram for the conditional GO TO is:

```
GO TO {procedure-name-1} ... DEPENDING ON identifier-1
```

Identifier-1 must have an integer type. The result is to transfer control to the procedure name whose position in the list corresponds to the value of identifier-1. If the value contained in identifier-1 is less than one or greater than the number of paragraph names in the list, control goes to the statement following the GO TO.

To support this, we define a function *SwitchTask: tasks* x *results* → *tasks* which indicates the task to be executed by the given GO TO statement and expression value. Note that control always transfers outside the statement.

The conditional GO TO is depicted in Figure 6.

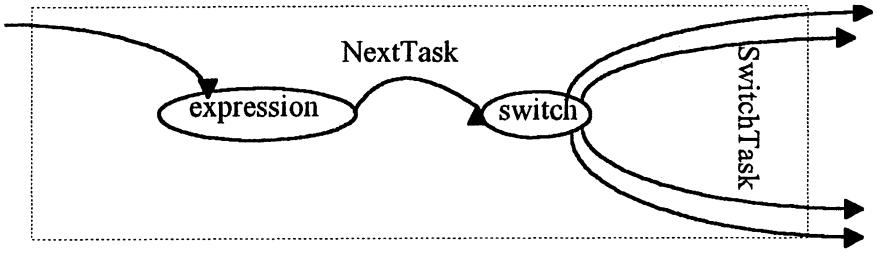

Figure 6: Pictorial description of the conditional GO TO

This introduces a new task type *switch* with the following rules:

if *TaskType(CurTask) = switch* ***then***
  Moveto(SwitchTask(CurTask, TestValue(CurTask)))
endif

3.6 ITERATION STATEMENTS

The primary iteration statement in COBOL is PERFORM. We will here consider the "in-line" PERFORM. The term will become clearer later when we discuss other perform formats.

We will consider several different syntax diagrams of the in-line PERFORM. The first is:

```
PERFORM {identifier-1 | integer-1} TIMES
    imperative-statement
END-PERFORM
```

which is expected execute the imperative statement the number of times specified in identifier-1 or integer-1. If the contents of identifier-1 are not greater than zero, the imperative statement is not executed.

The pictorial representation is in figure 7. This shows an expression to set up a variable not visible to the programmer, another expression to check that variable for the termination of iteration, test to select either execution of the statement or the task following the PERFORM, and another expression to increment (or decrement, depending upon the implementation) the hidden variable, whose *NextTask* is the comparison for termination. Note that all this detail is hidden by the compiler.

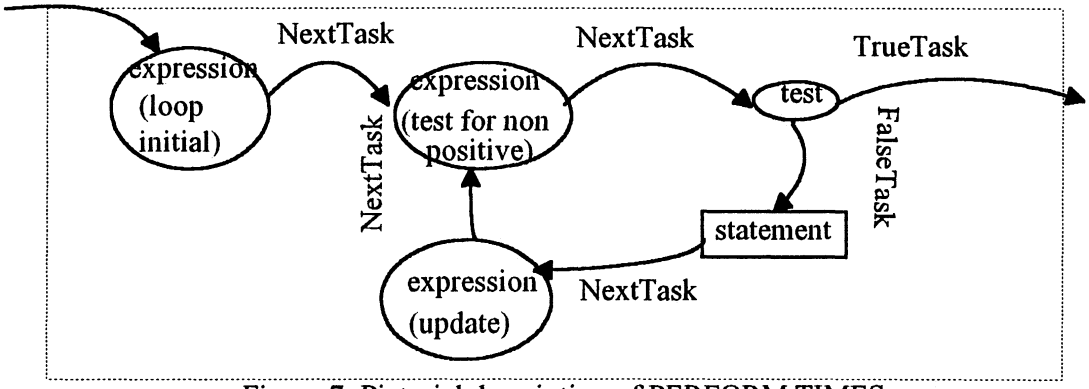

Figure 7: Pictorial description of PERFORM TIMES

The second format is:

```
PERFORM [WITH TEST {BEFORE | AFTER}] UNTIL condition-1
    imperative-statement
END-PERFORM
```

TEST BEFORE is the default, and gives the representation in figure 8. The expected behavior is the execution of the imperative-statement while condition-1 is not TRUE.

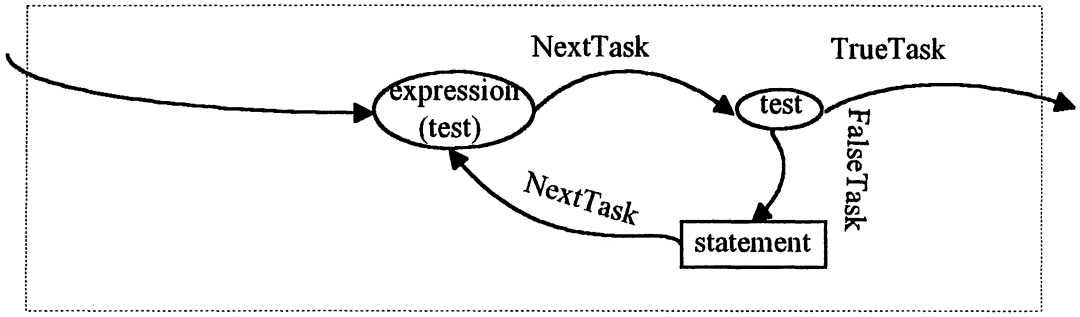

Figure 8: Pictorial Description of PERFORM UNTIL

TEST AFTER causes the statement to be executed before the condition is checked and is depicted in figure 9. Once again the imperative-statement is executed while condition-1 is FALSE.

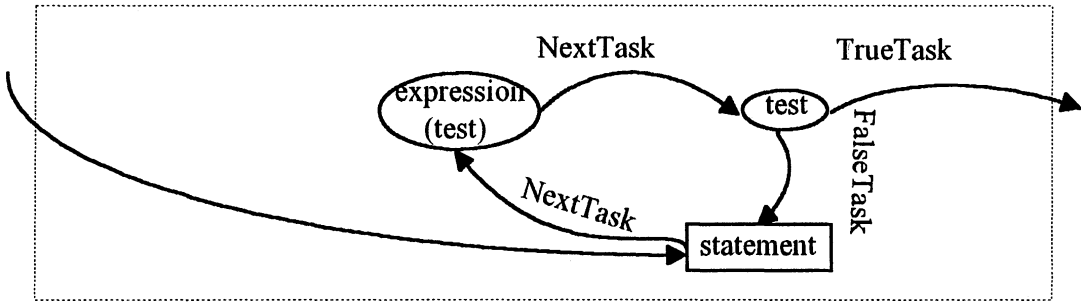

Figure 9: Pictorial description of PERFORM TEST AFTER UNTIL

The third format is:

```
PERFORM [WITH TEST {BEFORE | AFTER}]
VARYING {identifier-1 |index-name-1}
FROM {literal-1 | identifier-2 | index-name-2}
BY {literal-2 | identifier-3}
UNTIL condition-1
    [AFTER {identifier-4 | index-name-3}
    FROM {literal-3 | identifier-5 | index-name-4}
    BY {literal-4 | identifier-6}
    UNTIL condition-2] ...
    imperative-statement
END-PERFORM
```

(The AFTER phrase is not allowed with the in-line PERFORM in [IBM]) The behavior is as follows: First the variables in the VARYING and AFTER clauses are initialized to the values specified by the corresponding FROM clauses. Then, unless TEST AFTER has been specified, all the conditions are tested. If they are all true, the imperative statement is executed and the variable of the innermost loop is augmented by its associated BY value. If the condition of the outermost loop (condition 1 in the syntax diagram) is FALSE, control passes to the statement passing the perform. In any other case, the variable of the innermost loop whose condition is FALSE, is augmented, and the variables of deeper loops are reinitialized. See figure 10 for the depiction of a 3-deep loop.

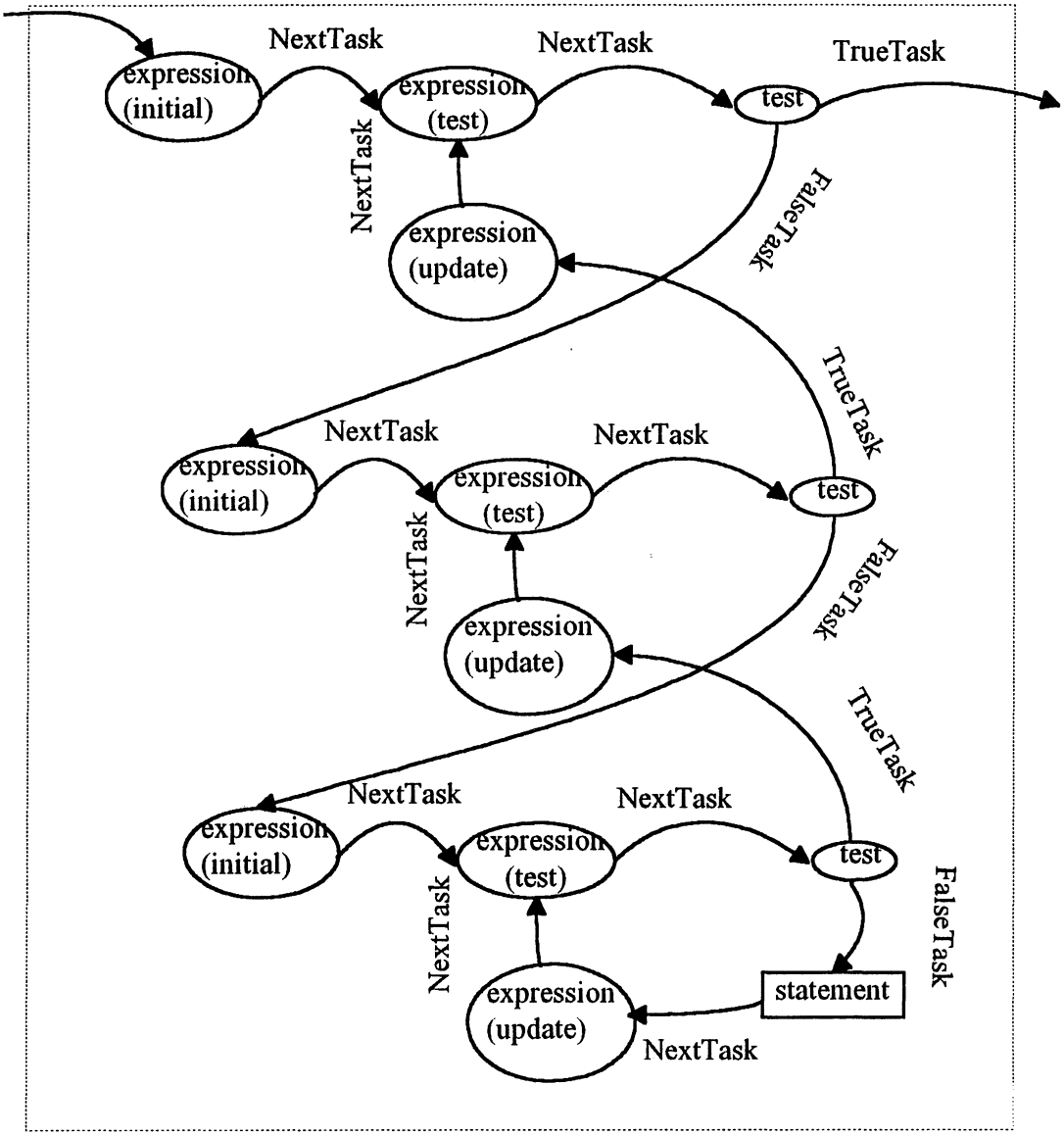

Figure 10: Pictorial description of a 3-deep PERFORM loop

3.7 INTERNAL SUBROUTINES

For consistency, the other PERFORM formats should be called out-of-line PERFORMs, but are usually just called PERFORMs (because these were there first). The simplest format is:

```
PERFORM procedure-name-1 [{THROUGH | THRU} procedure-name-2]
```

This implements a parameterless internal subroutine whose behavior is:

Control passes to the first statement of procedure-name-1 (which may be either a paragraph or a section). When the last statement of procedure-name-1 (or procedure-name-2 if the THROUGH option is used) has executed, control returns to the statement immediately after the PERFORM rather than into the next paragraph or section.

It is clear that the compiler should make the *NextTask* of the PERFORM the first statement of procedure-name-1. What is less clear is implementing the return of control. The obvious way to model this is to make *NextTask* a dynamic function and this might be consistent with some very early implementations of COBOL. But most modern versions of the language run in some form of reentrant environment in which program code is not modified. To preserve this behavior, we define new functions (some of which we would need in either case): *AfterTask: tasks → tasks* a static function locating the statement immediately following the PERFORM, *ExitTask: tasks → tasks* a static function giving the compiler-generated task for exiting a paragraph or section, and *ReturnTask: tasks → tasks*, a dynamic function, which receives and returns the task to which the performed routine will return; and new tasks *perform* and *exit* named for the verbs they implement. (Actually, we will see later that *exit* does not implement the EXIT verb.)

A pictorial representation is shown in Figure 11. We have not enclosed this diagram within a dotted box as it represents more than one language construct.

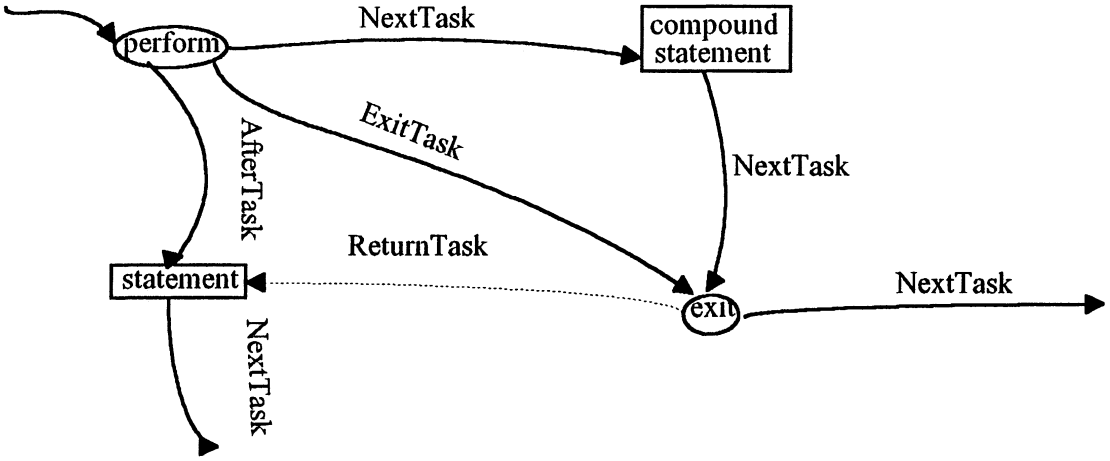

Figure 11 Pictorial representation of PERFORM and associated exit

The transition rules for a *perform* task are:

if *TaskType(CurTask) = perform* ***then***
  Moveto(NextTask(CurTask))
  ReturnTask(ExitTask(CurTask)) := AfterTask(CurTask)
endif

The transition rules for an *exit* task are:

if *TaskType(CurTask) = exit* ***then***
  if *ReturnTask(ReturnAddress(CurTask)) = ⊥* ***then***
    Moveto(NextTask(CurTask))
  else
    Moveto(ReturnTask(ReturnAddress(CurTask)))
    ReturnTask(CurTask) :=⊥
  endif
endif

We must note that one method of optimization for speed rather than space, replaces the PERFORM with the code performed (invisibly, of course) making as many copies of the code as there are PERFORMs.

The out-of-line PERFORM may be combined with the full set of iterative options. The result is the same as enclosing the simple PERFORM within the iterative form (although the full range of VARYING with AFTER is allowed only with this form in [IBM]). Because the format varies slightly, we present only the syntax diagrams here.

```
PERFORM procedure-name-1 [{THROUGH | THRU} procedure-name-2]
{identifier-1 | integer-1} TIMES

PERFORM procedure-name-1 [{THROUGH | THRU} procedure-name-2]
[WITH TEST {BEFORE | AFTER}]
UNTIL condition-1

PERFORM procedure-name-1 [{THROUGH | THRU} procedure-name-2]
[WITH TEST {BEFORE | AFTER}]
VARYING {identifier-1 |index-name-1}
FROM {literal-1 | identifier-2 | index-name-2}
BY {literal-2 | identifier-3}
UNTIL condition-1
    [AFTER {identifier-4 | index-name-3}
    FROM {literal-3 | identifier-5 | index-name-4}
    BY {literal-4 | identifier-6}
    UNTIL condition-2] ...
```

3.7.1 The SORT and MERGE statements

The syntax diagram for the SORT statement is:

```
SORT file-name-1
    {ON {ASCENDING | DESCENDING} KEY {data-name-1} ...} ...
    [WITH DUPLICATES IN ORDER]
    [COLLATING SEQUENCE IS alphabet-name-1]
    {INPUT PROCEDURE IS procedure-name-1
        [{THROUGH | THRU} procedure-name-2] |
    USING {file-name-2} ...}
    {OUTPUT PROCEDURE IS procedure-name-3
        [{THROUGH | THRU} procedure-name-4] |
    GIVING {file-name-3} ...}
```

We will defer discussion of the USING and GIVING clauses until later. A SORT with INPUT PROCEDURE and OUTPUT PROCEDURE begins with the transfer of control to the first statement of procedure-name-1 which must be the name of a section, not a paragraph. The entire section is executed (and the section procedure-name-2, if the THROUGH phrase is used), then control passes to an external function which sorts records in order by the specified keys. Control then proceeds into the first statement of procedure-name-3, through all the statements of that section (and that of procedure-name-4, when present) and finally to the statement immediately after the SORT statement.

We will use the mechanisms already described for the PERFORM to model the behavior of entering and exiting the INPUT and OUTPUT procedures. The SORT statement appears as a *perform*, follow by a new task *sort*, followed by another *perform*. The new task invokes a new external function *DoSort: internals* → *internals*, which does the ordering. We will not expand

further on this function. Typically sorting is done by a package external to the compiler, sometimes supplied by a different software manufacturer than the compiler.

A pictorial representation is shown in Figure 12

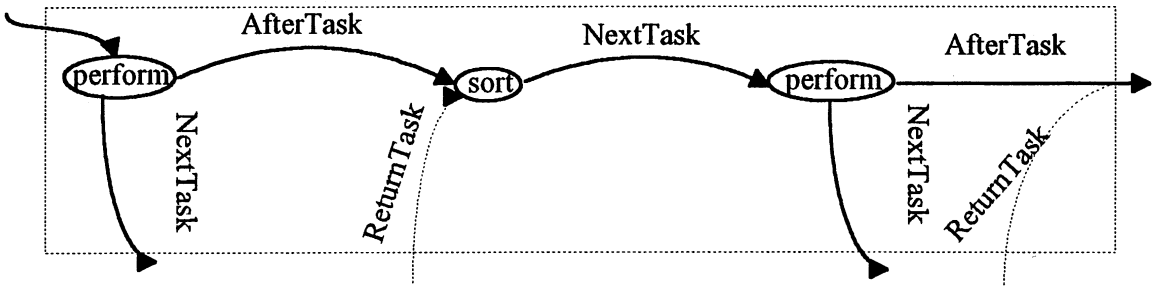

Figure 12: Pictorial representation of the SORT statement

The syntax diagram for the MERGE statement is:

```
MERGE file-name-1
    {ON {ASCENDING | DESCENDING} KEY {data-name-1} ...} ...
    [COLLATING SEQUENCE IS alphabet-name-1]
    USING file-name-2 {file-name-3} ...
    {OUTPUT PROCEDURE IS procedure-name-1
        [{THROUGH | THRU} procedure-name-2] |
    GIVING {file-name-4} ...
```

Once again, we defer discussion of the GIVING clause. The semantics of the MERGE with an OUTPUT PROCEDURE is that an external function merges the input files upon the specified keys, then control is transferred to the first statement of procedure-name-1; after the last statement of procedure-name-1 (or procedure-name-2 for the THROUGH phrase) is executed, control returns to the statement following the MERGE. Once again, the PERFORM mechanism suffices to model this behavior.

For completeness, we define a task *merge* which invokes an external function *DoMerge: internals → internals*. Most sort packages provide access to the same merge function used internally, so as with the SORT, we will not consider *DoMerge* further.

The USING and GIVING clauses can be regarded as performing procedures created by the compiler to read the files and release the records into the SORT, or to return the records from the SORT/MERGE and write them to the file. We will see what could be in these procedures when we discuss Sequential I/O.

3.8 NULL STATEMENTS

COBOL has two null statements. CONTINUE serves those situations where an imperative statement is needed, but nothing should be done. An example of a reason for its use is in a WHEN clause to exempt a specialized condition from the processing implied by a following WHEN, for example:

```
EVALUATE TRUE
    WHEN A EQUAL 'C' AND B EQUAL 99999
        CONTINUE
    WHEN A EQUAL 'C'
        ADD 1 TO B
END-EVALUATE
```

Another is in the popular practice of restating a condition involving a negation (NOT) as the condition without the NOT followed by CONTINUE ELSE and the statement to be executed. The practitioners of this feel the resulting code is clearer.

The other null statement is EXIT. The syntax requires that the statement inhabit a paragraph by itself. If that paragraph is the end of a PERFORM range and was entered by the PERFORM, control is returned to the statement following the PERFORM. But the compiler inserts an *exit* at the end of any paragraph which is at the end of a PERFORM range, without consideration of what statements are in the paragraph. The default action of an EXIT in the absence of PERFORM is to go to the next paragraph, which is just what is expected of a null statement.

3.8.1 Task Types and Transition Rules

We represent the continue statement by a task of type *continue* with the following transition rule:

if TaskType(CurTask) = continue **then**
  Moveto(NextTask(CurTask))
endif

As noted above the EXIT statement needs no task to represent it.

3.9 UNCONDITIONAL TRANSFER OF CONTROL

There are two cases where control is transferred unconditionally. The GO statement explicitly transfers control to a paragraph or section. The NEXT SENTENCE clause transfers control to the next statement following the period terminating this sentence.

3.9.1 The GO statement

The format of the unconditional GO statement is:

```
GO TO [procedure-name-1]
```

when procedure-name-1 is supplied, this effects an unconditional transfer of control to the first statement of the paragraph or section named by procedure-name-1. When procedure-name-1 is omitted, the statement must be the only one in its paragraph, and the target of an ALTER statement. If the GO statement is the only one in its paragraph, it may be ALTERed, whether procedure-name-1 is specified or not. Nonetheless, the compiler is able to determine through static analysis of the program whether a paragraph containing a GO is ALTERed or not, so we will consider two cases. An unALTERed GO will always transfer control to procedure-name-1, so the *NextTask* function is all we need. We represent this unconditional transfer of control with the *jump* task taken from [GH] which has the following transition rule:

if TaskType(CurTask) = jump **then**
  Moveto(NextTask(CurTask))
endif

To support the altered GO TO we define two new task types, *go* and *alter*. Once again, it would be possible to accomplish our goal by making *NextTask* dynamic, and once again we will

reject that approach as being incompatible with a reentrant environment. Thus we need the following functions: *GoTask: task → task*, a dynamic function which receives and returns the task to which the GO will go, *GoTarget: tasks → tasks*, a static function returning the task to which *alter* will direct the GO, and *GoSubject tasks → tasks*, a static function giving *alter* the task containing the *go* task.

The syntax diagram for the ALTER statement is:

```
ALTER {procedure-name-1 TO [PROCEED TO] procedure-name-2} ...
```

It will be convenient and loses no generality to regard repetition of the TO clause as separate instances of the ALTER statement.

The transition rules for *alter* are:

if *TaskType(CurTask) = alter* ***then***
  Moveto(NextTask)
  GoTask(GoSubject(CurTask)) := GoTarget(CurTask)
endif

The transition rules for *go* are:

if *TaskType(CurTask) = go* ***then***
  Moveto(GoTask(CurTask))
endif

3.9.2 The NEXT SENTENCE clause

The NEXT SENTENCE within the scope of an IF or the WHEN clause of the SEARCH verb will transfer unconditionally to the next statement following a period. We can represent these transfers of control using the *jump* task, although often the function which leads to this task can return the *NextTask* of the *jump* allowing it to be eliminated.

3.10 PARAGRAPH AND SECTION LABELS

Paragraph and section labels provide the targets for GO TO, the out-of-line PERFORM, and ALTER statements. The compiler may place *exit* tasks at the end of paragraphs as necessary. The compiler recognizes a paragraph by a non-reserved word starting in Area A, preceded and followed by a period. A section is recognized by a non-reserved word in Area A, preceded by a period, followed by the keyword SECTION, optionally followed by a number (used by the Segmentation Module, one of the parts of the language we do not discuss, partly because virtual memory systems have made it unnecessary), and terminated by a period. (There is a style of structured programming in which these are the only periods used within the procedure division.)

We have already discussed all the task types related to these statement labels.

3.11 INITIAL STATE

Initially, *CurTask* indicates the first statement of the PROCEDURE DIVISION (following END DECLARATIVES, one of the topics which we did not discuss, if it is present). In addition, each *ReturnTask* and *GoTask* has an initial value of $\perp$.

3.12 SEARCH, a composite control structure.

The syntax diagram for (the serial option of) SEARCH is:

```
SEARCH identifier-1 [VARYING {index-name-1 | identifier-2}]
[AT END imperative-statement-1]
WHEN condition-1 {imperative-statement-2 | NEXT SENTENCE}
[WHEN condition-2 {imperative-statement-3 | NEXT SENTENCE}] ...
[END-SEARCH]
```

The SEARCH provides a method of locating an entry in an table (tables are array structures which will be discussed in later algebras) which matches one of several conditions. The syntax requires that identifier-1 be the name of a table which has one or more names (called the index(es) of the table) assigned to select an entry of the table. Note that the VARYING phrase is optional. If index-name-1 is one of the indexes of the table, it is used in the search. If it is not, the first index of the table is used in the search and index-name-1 or identifier-1 is incremented at the same time as the search index. The search first checks to insure that the index is within the range of the entries of the table. If so, it checks each of the conditions in turn until one is found which is TRUE. If none are found, search points the index to the next entry in the table, and resumes its checks. SEARCH combines multiple selection with iteration.

We present a pictorial representation of a SEARCH with two WHEN clauses in Figure 13:

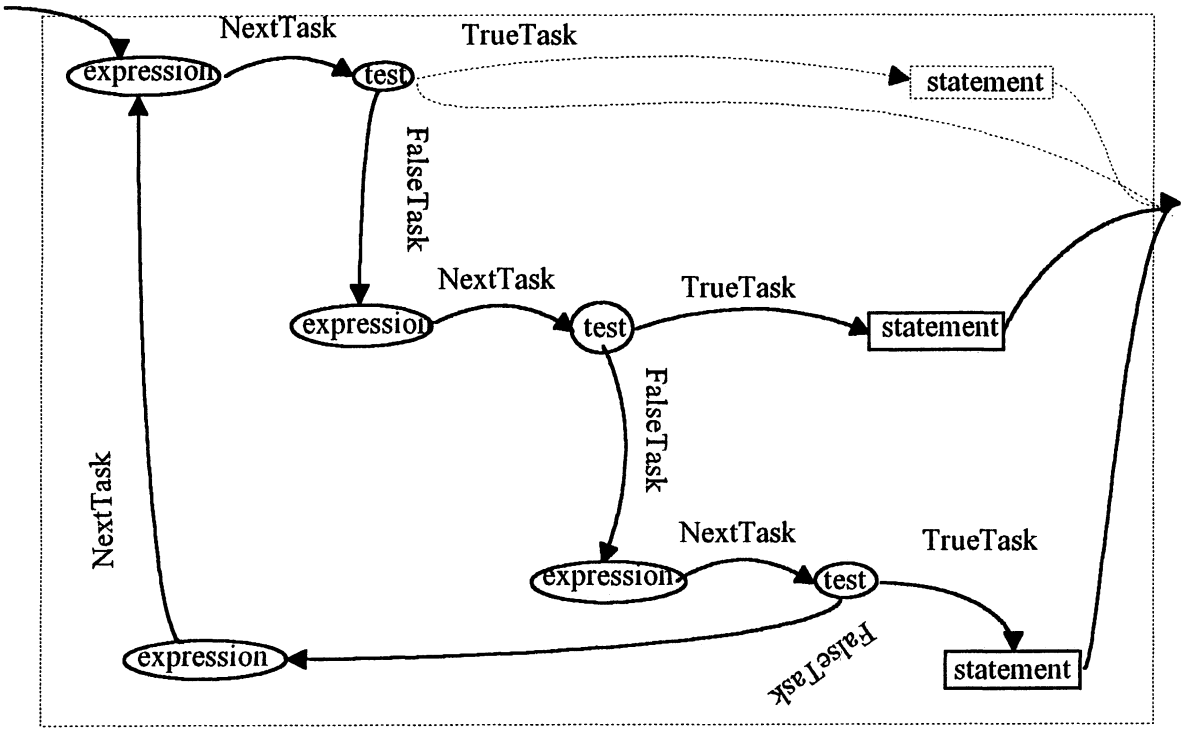

Figure 13: Serial SEARCH with two WHEN clauses

4 The Algebra of Memory Allocation and Initialization.

This algebra is concerned primarily with the DATA DIVISION.

4.1 Declarations

Following [GH], we represent declarations as elements of the *tasks* universe, linked by the *NextTask* function.

The semantics of declarations differ with the section in which they occur. Declarations in the FILE SECTION provide the location of an identifier relative to the current record of the file, the size and the usage, a term which will receive further explanation. Those of the WORKING STORAGE SECTION, provide location, size, usage and possibly initial value for an identifier. Working storage for a program may be static or dynamic, although it is far more common to utilize static storage. In the LINKAGE section, declarations describe items passed to this program by another program (or in many cases, the Operating System).

One classification of data items is elementary vs. group. The distinction is simple. A group item contains other elementary items. An elementary item cannot contain another item.

4.2 Syntax Diagrams

The formats for data descriptions are:

```
level-number [data-name-1 | FILLER]
    [REDEFINES data-name-2]
    [IS EXTERNAL]
    [IS GLOBAL]
    [{PICTURE | PIC} IS character-string]
    [[USAGE IS] {BINARY | COMPUTATIONAL | COMP | DISPLAY | INDEX |
       PACKED-DECIMAL}]
    [[SIGN IS] {LEADING | TRAILING} [SEPARATE CHARACTER]]
    [OCCURS integer-2 TIMES
       [{ASCENDING | DESCENDING} KEY IS {data-name-3} ... ] ...
       [INDEXED BY {index-name-1} ...] |
     OCCURS integer-1 TO integer-2 TIMES DEPENDING ON data-name-4
       [{ASCENDING | DESCENDING} KEY IS {data-name-3} ... ] ...
       [INDEXED BY {index-name-1} ...]]
    [{SYNCHRONIZED | SYNC} [LEFT | RIGHT]]
    [{JUSTIFIED | JUST} RIGHT]
    [BLANK WHEN ZERO]
    [VALUE IS literal-1].

66  data-name-1 RENAMES data-name-2 [{THROUGH | THRU} data-name-3].
```

and

```
88  condition-name-1 {VALUE IS | VALUES ARE}
    {literal-1 [{THROUGH | THRU} literal-2]} ... .
```

Level numbers 01 through 50 are used to indicate hierarchical structuring. A group item contains all the items declared following it with higher level numbers until the end of the section or a level number of the same or lower value is encountered. 01 level numbers designate records. 77 level numbers indicate an independent item. 66 level numbers allow an alternate name for the area containing an item or from the start of one item through the end of a second without reference to the original hierarchical structure. (This construct is rarely used. The syntax rules require these to be placed at the end the record in which the fields to which it refers are defined. Other constructs exist which can produce the same effect and are placed near the items affected, so they better document the structure.) 88 level numbers do not define any data area. Instead, they allow the programmer to declare a Name for a comparison of the field prior to the 88 level to a value, comparison of the prior field to a range of values, or a list of such comparisons. The result will be an expression in one of our later algebras.

4.3 Representations

We represent each data item with a task of type *decl*. These tasks have several partial functions associated with them. The partial function *Size: task → integer* returns the number of characters used for the data. This function is dynamic as the size of certain fields may vary during execution.

Another function *address: task → internals* returns the location in memory where the value is stored. We are leaving the details of this algebra for a later paper, but will mention some of the issues involved with this dynamic function. An *address* may change as the size of preceding fields is changed. The *address* of a task representing a field in the FILE SECTION will be undefined until the file is opened and may change with each READ or WRITE on the file. The *address* of a task representing a field in the LINKAGE SECTION is calculated upon program entry from the addresses passed from the calling program. The *address* of a task related to a field in the WORKING-STORAGE SECTION may be calculated on each entry to the program if the INITIAL phrase was included within the PROGRAM-ID paragraph. This phrase also controls whether the initial value of a field in the WORKING-STORAGE SECTION is reset upon each entry or not, and whether files must be opened.

Yet another function *usage: task → internals* is required to describe further details of how the data will be represented in memory.

5 The Algebra of Expressions

Following the example of [GH], we replace each occurrence of a task of type *expression* and many of those of type *imperative-statement* with a collection of tasks reflecting the structure of the expression. We now make *TestValue* an internal, dynamic function.

We will also follow the usual practice of describing an expression by a *parse tree*, although this is more consistent with the use of context-free grammars than the syntax diagrams of [ANSI].

5.1 Basic functions for computations.

We borrow from [GH] the following functions:

A static partial function *Parent: tasks → tasks* to locate the closest enclosing expression for a given expression. For expressions not contained in other expressions, *Parent* returns the *test* task which uses the result of the expression. The static partial function *WhichChild: tasks → {left, right, only, test}*, a subset of *internals*, indicates the relationship between a task and its parent. Dynamic partial functions *LeftValue, RightValue: tasks → results* indicate the results of evaluating the left and right operands of binary operators as does *OnlyValue: tasks → results* for a unary operator. *ConstValue: tasks → results* indicates the values of program constants.

5.2 The ReportValue macro

We assign the result of evaluating an expression to the appropriate storage function of the parent expression with the *ReportValue(value)* macro abbreviation from [GH]. Its definition is:

```
if WhichChild(CurTask) = left then
   LeftValue(Parent(CurTask)) := value
elesif WhichChild(CurTask) = right then
   RightValue(Parent(CurTask)) := value
elseif WhichChild(CurTask) = only then
   OnlyTask(Parent(CurTask)) := value
elseif WhichChild(CurTask) = test then
   TestValue(Parent(CurTask)) := value
endif
```

6 Input/Output

One of the distinguishing features of COBOL is the richness of its input/output commands. The standard contains three modules, each for a different model. The Sequential I-O module provides access to records each of which has a defined successor, except the last in the file. The Random I-O module provides for direct access to records by specification of an integer which is the relative number of the record within the file. The Indexed I-O module provides for access to records based upon specific data within the record itself. An indexed file has a primary key field in which the values for individual records must be unique and may have alternate key fields in which duplicate values may be allowed. Records in such a file may be accessed either directly or sequentially in order of the key

7 ACKNOWLEDGMENT

It is customary to include the following acknowledgment in any work concerned with the language:

Any organization interested in reproducing the COBOL standard and specifications in whole or in part, using the ideas from this document [the standard] as the basis for an instruction manual or for any other purpose, is free to do so. However, all such organizations are requested to reproduce the following acknowledgment paragraphs in their entirety as part of the preface to any such publication (any organization using a short passage from this document, such as a book review, is requested to mention 'COBOL' in acknowledgment of the source, but need not quote the acknowledgment).

COBOL is an industry language and is not the property of any company or group of companies, or of any organization of group of organizations.

No warranty, expressed or implied, is made by any contributor or by the CODAYSL Programming Language Committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by any contributor, or by the committee, in connection therewith.

The authors and copyright holders of the copyrighted material used herein

FLOW-MATIC (trademark of Sperry Rand Corporation), Programming for the UNIVAC I and II, Data Automation Systems copyrighted 1958, 1959, by Sperry Rand Corporation; IBM Commercial Translator Form No. F28-8013, copyrighted 1959 by IBM, FACT, DSI 27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell

have specifically authorized the use of this material in whole or in part, in the COBOL specifications. Such authorization extends to the reproduction and use of COBOL specifications in programming manuals or similar publications.

8 REFERENCES

[ANSI] American National Standards Institute, "American National Standard for Information Systems Programming Language COBOL", 1985, New York, NY, ANSI X3.23-1985, ISO 1989-1985

[Bl] George Robert Blakley, "Building Interpreters Using Algebraic Operational Semantics", Ph.D. Thesis, The University of Michigan, 1992.

[Bo1] Egon Börger, "A Logical Operational Semantics for Full Prolog. Part I: Selection Core and Control" in Proc. of CSL'89, 3rd Workshop on Computer Science Login (eds. E. Börger, H Kleine Büning and M Richter), Springer LNCS 440 (1990), 36-64.

[Bo2] Egon Börger, "A Logical Operational Semantics for Full Prolog. Part II: Built-in Predicates for Database Manipulations", in Proc. of Mathematical Foundations of Computer Science 1990 (ed. B. Rovan), Springer LCNS 452, 1-14.

[Bo3] Egon Börger, "A Logical Operational Semantics for Full Prolog. Part III: Builtin Predicates for Files, Terms, Arithmetic and Input-Output", IWBS Report no. 117, IBM Germany, Institut Für Wissensbasierte Systeme, April 1990, pp25; to appear in Proc. of Workshop on Logic from Computer Science (Berkeley, 1989), MSRI Proceedings Series, ed. Y. Moschovakis, Springer Verlag.

[BR1] Egon Börger and Dean Rosenzweig, "A Formal Analysis of Prolog Database Views and their Uniform Implementation", University of Michigan Technical Report CSE-TR-89-91, 1991.

[BR2] Egon Börger and Dean Rosenzweig, "A Formal Specification of Prolog by Tree Algebras", Proceedings of ITI, Cavtat, 1991.

[BS] Egon Börger and Peter Schmitt, "A Formal Operational Semantics for Languages of Type Prolog III", in Proc. of CSL'90, 4th Workshop on Computer Science Logic (eds. E. Börger, H. Kliene Bünning and M. Richter), Springer LNCS, 1991.

[Gu1] Yuri Gurevich, "Logic and the Challenge of Computer Science", in Current Trends in Theoretical Computer Science (ed. E. Börger), Computer Science Press, 1987, 1057.

[Gu2] Yuri Gurevich, "Algorithms in the World of Bounded Resources", in "The Universal Turing Machine: A Half-Century Story" (ed. R. Herken), Oxford University Press, 1988, 407-416.

[Gu3] Yuri Gurevich, "Evolving Algebras: An introductory Tutorial", Bulletin of European Association for Theoretical Computer Science, February 1991.

[Gu4] Yuri Gurevich and James Morris, "Algebraic Operational Semantics and Modula 2", Lecture Notes in Computer Science, Proceedings of Logik und Informatik (Karlsruhe, October 1987) Springer-Verlag, Berlin.

[GH] Yuri Gurevich and James K. Huggins, "The Semantics of the C Programming Language"

[GMs] Yuri Gurevich and Larry Moss, "Algebraic Operational Semantics and Occam", Springer LNCS 440, 176-192

[IBM] International Business Machines, "VS COBOL II Application Programming: Language Reference", GC26-4047-6, International Business Machines, Seventh edition, 1990, San Jose, Ca.

[Mor] James Morris, "Algebraic Operational Semantics for Modula2", Ph.D. thesis, The University of Michigan, 1988.

[PK] Andreas S. Philippakis and Leonard J. Kazmier, "Advanced COBOL", McGraw-Hill, 2nd edition, 1987, New York, NY

9 Appendix: Evolving Algebras.

A *static algebra* consists of a set called the *superuniverse* of the algebra, the *signature*, a collection of function names, each with a fixed arity, and *basic functions*, interpretations of the function names in the signature. A function name of arity n is interpreted at a n-ary operation on the superuniverse. A function name of arity zero is called a *distinguished element*. A static algebra is regarded as a *state* of an evolving algebra. The superuniverse does not change as the algebra evolves, the basic functions may.

But some functions may not be defined for the whole superuniverse. The superuniverse contains the element *undef* which is the value returned when the function is undefined. It is frequently useful to collect elements of the superuniverse into groups. The superuniverse contains elements *true* and *false* which allow us to deal with relations. A *universe* U is a basic function which identifies a set $\{x : U(x)\}$. When we speak of a function from one universe to another, we mean that the function returns a value in the set defined by the second universe for each value in the first universe and is undefined otherwise. The notation used is self-explanatory; $f : U \rightarrow V$, $f : U_1 \times U_2 \rightarrow V$, and $f : V$.

A function is *dynamic* if its interpretation may be changed as the algebra evolves. Otherwise functions are *static*. An *external* function has its values determined by something (usually called an oracle) outside the algebra.

A program of an algebra is a set of transition rules of the form:

if t_0 **then** $f(t_1,...,t_r) = t_{r+1}$ **endif**

where t_o, $f(t_1,...,t_r)$, and t_{r+1} are terms containing no free variables (i.e. are closed terms). This means: Evaluate all the t_i, if t_0 evaluates to true, change the value of f at $(t_1,...,t_r)$ to t_{r+1}, otherwise do nothing. Rules are actually defined in a more liberal way. Let $b_0,...,b_k$ be terms and let $C_0,...,C_{k+1}$ be sets of rules for some natural number k. Then

if b_0 **then** C_0
elseif b_1 **then** C_1
.
.
.
elseif b_k **then** C_k
else C_{k+1}
endif
and
if b_0 **then** C_0
elseif b_1 **then** C_1
.
.
.
elseif b_k then C_k
endif

are both rules.

Nesting of the transition rules is allowed and occurs frequently. The more liberal rules can be transformed into the stricter form when necessary. We use the more liberal forms and textual abbreviations (called macros) for brevity and clarity.

The algebra evolves through the updates of dynamic functions in the transition rules. The updates are applied in parallel and if several updates contradict each other one is chosen nondeterministically.