THE UNIVERSITY OF MICHIGAN

Technical Report 17

A COMPILER FOR AN ASSOCIATIVE OBJECT MACHINE

William L. Ash

CONCOMP: Research in Conversational Use of Computers
ORA Project 07449
F.H. Westervelt, Director

supported by:

DEPARTMENT OF DEFENSE
ADVANCED RESEARCH PROJECTS AGENCY
WASHINGTON, D.C.

CONTRACT NO. DA-49-083 OSA-3050
ARPA ORDER NO. 716

administered through:

OFFICE OF RESEARCH ADMINISTRATION ANN ARBOR

May 1969

ABSTRACT

This paper discusses the design and construction of a compiler whose source language consists of sentences of a restricted predicate calculus, and whose output object code operates on a simulated associative target machine. The source programs are characterizations of relations, used for deriving one relation from others, and for completing relations. A program realizes the completion of a relation when it defines that relation in terms of itself, i.e., when the definition is recursive.

The most significant feature of such a compiler is that there are two levels of operators and operands. The first level operators are the logical connectives whose operands are relations, which in turn have as their "operands" relational arguments. The lower level operands determine the context-dependent interpretation of the higher level operators. The parse is further encumbered by the fact that the logical connectives in such a framework do not lend themselves to a production grammar. The situation is resolved by constructing a digraph to represent the definition of the relation. This construction proceeds using a back-up technique. The output of the compiler is an object program in the form of a macro definition, to be expanded for an interpretive associative target machine.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

An associative computer memory can effectively be employed to approximate "content addressability." By an associative memory we mean a memory that stores information in ordered n-tuples, called associations, which can be referenced by specifying any of the components of the association. We refer to an association by its contents (components), rather than by any address; indeed, it is the lack of explicit addresses that characterizes an associative machine. The term content-addressable becomes clearer when we see that we reference an association by its contents. More precisely, by a content-addressable memory, we essentially mean one in which the name of a datum contains a dynamic cue to the relevant information about that datum. Content addressability obviates table lookups, binary search, etc. An associative processor provides a useful approximation to content addressability.

The associative processor being used in the work reported herein is TRAMP [1,2], a software simulation of an associative machine. TRAMP associations are 3-tuples [3]

<A, O, V>

<Attribute> of <Object> equals <Value>: A (O) = V

This associative processor provides a semblance of content addressability and can be used to store and retrieve efficiently large amounts of data. However, any association, or

combination of associations, will in general imply many more associations. For example:

[Husband (Mary) = John] ===> [Wife (John) = Mary]

Father (Norm) = Harry
Brother (Harry) = Sam

===>

Uncle (Norm) = Sam
Nephew (Sam) = Norm
Brother (Sam) = Harry
Son (Harry) = Norm

This situation can be resolved by requiring that the user redundantly enter his information in all the various ways that he might want to access it; or, more realistically, the user can define a relation, whereby he specifies what inference rules may be used in deriving the implied associations. Now the important distinction, once we have decided to admit relations, is whether their definitions will be extensional or intensional. We can extensionally define a relation by going through the data structure and generating and storing all implied associations. This might be done with an iteration loop:

FOR HUSBAND (A) = B, LET WIFE (B) = A

That is, all the ordered pairs which comprise the (binary) relation are generated and stored. This is the approach taken by many associative processors. The result is that all the implied information can in fact be made available, but there are two serious drawbacks:

1) In general, an extensional definition will gobble up extensive amounts or core, rendering it operationally uneconomical, or, in the extreme,

infeasible.

2) Unless these iteration loops are entered frequently and regularly, their time-dependency renders the extension incomplete, and/or inaccurate.

The alternative to an extensional definition is the intensional definition: where we characterize the relation, rather than exhaustively storing all of its ordered pairs. As an example, if we were to characterize the relation "WIFE" [of] as:

WIFE = Converse of HUSBAND

and we now want to ask who is the WIFE of Harry, we could ask:

WIFE (HARRY) = ?

and the system, using the characterization given above, would expand the question to be:

WIFE (HARRY) = ? or HUSBAND (?) = HARRY

It would ask both questions since it doesn't know if the desired association appears implicitly, explicitly, neither, or both.

This is the operational strategy of TRAMP intensional relational definitions. The user enters the definition as a sentence of a modified predicate calculus, e.g.,

(WIFE = .CON. HUSBAND)

This is the source program for a compiler whose output is an object program written in the associative language. This

object program will then effect the "expansion" of questions to extract from the data those relevant associations that can be inferred from associations explicitly in core and the intensional definitions of correspondences.*

*The goal then is only to reduce the number of associative sentences necessary to represent information, rather than to provide sophisticated heuristic search procedures for a question-answering system.

2. THE ASSOCIATIVE OBJECT LANGUAGE: TRAMP

TRAMP is fully documented in [1,2], but a brief summary of it must be given here. As stated in the introduction, TRAMP works with 3-tuples, or associative triples: A, O, V, which can be read as: A(O) = V. By specifying an association with the various components of the triple being either constant or "variable," there are eight questions that can be asked:

F0	A (O) = V
F1	A (O) = ?
F2	A (?) = V
F3	A (?) = ?
F4	? (O) = V
F5	? (O) = ?
F6	? (?) = V
F7	? (?) = ?

where "?" represents a variable. Question F0 has no variables; it simply asks <u>if</u> A (O) = V, and expects a truth value. The other questions all have one or more variables, and the variable is expected to take on, as its value, the "answer set," i.e., the set which completes the association.

TRAMP is currently embedded in the UMIST* interpreter. Since the object programs of the compiler are TRAMP programs, they must obviously work within the syntax of UMIST, a macro-

*UMIST [4] is a dialect of the TRAC T-64 language [5] (registered trademark of the Rockford Research Institute, Cambridge, Mass.) locally implemented at the University of Michigan, on the IBM 360/67.

generator language. Procedures are defined by the user as UMIST macro definitions. When a "function" is called, if the function is a user-defined macro (procedure), then it is at that time expanded; if the function is a pre-defined primitive of the language, then that routine is invoked. The pound sign (#) signals the start of a function call, with the call itself enclosed in an immediately following pair of parentheses. The arguments are delimited by commas, and the first argument is the name of the function (or macro). The arguments may themselves be function calls, with nesting depth effectively unlimited. UMIST functions are evaluated recursively from the inside out. All of the arguments to a function must themselves have been evaluated, with the value, possibly null, replacing the call, before the function can be called. A UMIST function call might look like:

```
#(ds,X,Y)
```

where ds is the name of the function, and X and Y are the arguments to ds.

This paper assumes no knowledge of UMIST (TRAC), except for the syntax described above, and the recursive manner of replacing a function by its value, which will be shown more clearly in subsequent examples.

The TRAMP system is an addition to UMIST of primitives which create and manipulate an associative structure. The full facilities of TRAMP are described in [2]; following are

brief descriptions of the primitives which constitute a level 0 object language for the compiler.

NAME: RL

PROTOTYPE: #(RL,A,O,V)

DESCRIPTION:

This is the primary retrieval function in TRAMP, and as well as being used in compiler-generated programs, it is the function which when called by the user invokes the object programs of the compiler.

Variables are specified by two asterisks (*). The answer set is the value of the function. For example,

#(RL,COLOR,CAR,**)

asks "What color is the car?" and expects the answer to be the value of this function. Two-variable questions generate two answer sets, rather than a set of ordered pairs. There is one special case of the two-variable question which is significant to the compiler-produced code. This case is where one of the variables is denoted by '*@*', i.e., an at-sign between the two asterisks. The '@' signifies that although that component of the triple is to be considered arbitrary, the corresponding answer set is not desired, and should not be generated. The question #(RL,COLOR,*@*,**) would find ALL things which appear as the third component of associations in which "color" is the first component.

There are several variations of this function and the compiler uses some of them. The function named RL@ is the entry point which specifies that any programs previously put out by the compiler are NOT to be executed, i.e., only explicitly stored associations are to be retrieved. This function is always used by the compiler. The other variations that the compiler uses do not differ from each other in ways significant to the discussion in this paper and can be considered implementational details. In all examples, only these two forms will be shown, although in practice others are used. There are certain other minor points which will differ in actuality from what is shown in the examples, but they are details whose explanations are not warranted.

NAME: INT

PROTOTYPE: #(INT,SET1,SET2)

DESCRIPTION:

This function forms the set intersection of its two arguments, and returns the intersection as its value. TRAMP sets are unordered collections of things, separated by semicolons(;) (necessitated by the crucial role that the comma plays in UMIST) and possibly containing redundancies, i.e., the same element may appear several times in the same TRAMP "set."

NAME: RCOM

PROTOTYPE: #(RCOM,SET1,SET2)

DESCRIPTION:

This function similarly forms the relative complement of its two arguments. The second argument is logically subtracted from the first, with the result being the value of the function.

NAME: @@

PROTOTYPE: #(@@,STRING,NAME)

DESCRIPTION:

This function can be considered an internal function, since it was written specifically for the compiler, is not generally available to the user of TRAMP, and does not appear in the TRAMP reference manual.

This is the "clean-up" function. The programs generated by the compiler are always the first argument, STRING. Because of the program structure, as we shall see, several "paths" specified by the definition may lead to the same point, causing redundancies; and some paths may lead to dead ends, resulting in dangling set element delimiters. The function @@ checks for these two conditions and corrects them if necessary. The second argument, NAME, specifies the disposition of the result. If the argument is given, then the cleaned-up set is stored, labeled by the NAME, and the function itself is null-valued. If this argument

is omitted, the cleaned-up set is the value of the function.

3. OPERATIONAL BEHAVIOR WITH THE COMPILER

Retrieval functions are handled as follows. All calls to RL and its variations, with the exception of RL@, are calls to a preprocessor. This preprocessor looks to see if the relation (first, or A component) has been defined. If not, the preprocessor simply passes on the call to the associative structure, as if the original call had been RL@. If the name is found to have an active definition, an interpreter is called. This interpreter (not to be confused with the UMIST host interpreter) expands the programs output by the compiler, filling in items specific to the call, like a macro expander. The actual expansion is really quite short, fast, and simple, with the bulk of the work being performed first by the compiler at definition time, and later by the preprocessor, which must appropriately manipulate the expanded programs to effect the intent of the original retrieval call. The result of the preprocessor is returned to the UMIST interpreter to be recursively expanded. This quite significant, since at each stage the preprocessor need only be concerned with one level of complexity. If a relation is defined in terms of other relations, which in turn are defined, etc., the UMIST recursion will take care of it. We will follow a very simple example to demonstrate how this happens, at the same time exhibiting the recursive replacement behavior of UMIST. Suppose that WIFE

has been defined to be the converse of HUSBAND, and the user now makes the function call:

```
#(RL,WIFE,HARRY,**)      [Who is Harry's wife?]
```

Then the value of this function, as returned by the preprocessor, would be:

```
#(@@,#(RL@,WIFE,HARRY,**);#(RL,HUSBAND,**,HARRY))
```

Notice that the call for WIFE will bypass the preprocessor and hence avoid an infinite recursion. That call must be made, since the desired association may very well appear explicitly. We might expect that one of the two calls will result in a null value, and hence a dangling semicolon. @@ will then delete it. If HUSBAND has not been given a definition, then that call, though it will not bypass the preprocessor, will not be affected by it and will result in a straight associative retrieval. Had HUSBAND been defined, say as a Male Spouse, then that program would be the value of the call for HUSBAND. Specifically, after making the call for HUSBAND, depending on the exact definition used, the string being processed by UMIST might look like:

```
#(@@,wifeofharry;#(@@,#(RL@,HUSBAND,**HARRY);
#(RL,SPOUSE,**,#(INT,HARRY,#(RL,MALE,**))))
```

where "wifeofharry" represents the explicit retrieval specified by the first call, since this will have already been evaluated. Now, if SPOUSE has been defined, say to be symmetric, this process continues recursively until there is nothing left to evaluate. Note that we are guaranteed at

least one more level of expansion, since the unary relation MALE appears. The associative structure only deals with triples, indicating that a unary relation is either defined, or the call is a syntactic error.

3.1 COMPILER SOURCE LANGUAGE

The compiler itself is called with the function DDR. The syntax of the call is:

```
#(DDR,(R = exp))
```

where R is the relation being defined, and "exp" is a sentence of a restricted predicate calculus, without explicit quantifiers, which defines the relation. The relations on the right side of the equation are joined by the normal logical connectives:

```
AND (.A.);  OR (.V.);  NOT (.N.).
```

In addition, there are two relational operators: Composition, or relative product, denoted by a slash (/); and Converse (.CON.). Finally, equality or inequality may be specified (.EQ.; .NE.). The precedence of these operators is as shown below.

```
/
.CON.
.EQ., .NE.
.N.
.A.
.V.
```

The above precedence (descending order) ordering may be altered in the usual way by the appropriate use of parentheses.

The relational notation used is the R(x,y) format, where R is the relation and x and y its arguments. This can be read as: y stands in relation R to x. The arguments are set off by parentheses. This is a slight distortion of the associative format, A (O) = V, but the ordering is preserved: R(x,y) corresponds to R (x) = y. With this as the source language, some typical definitions are given in Figure 1.

Figure 1. Sample Source Programs

```
#(DDR,(BIGGER = BIGGER / BIGGER))   BIGGER is transitive

#(DDR,(BIGGER(a,b) = BIGGER(a,q) .A. BIGGER(q,b)))
                    exact same definition using expanded
                    format--specifying the dummy arguments.

#(DDR,(SIB = BRO .V. SIS .V. .CON.SIB))
                    a sibling is a brother or a sister, and
                    it is symmetric.

#(DDR,(BRO(cain,abel) = SIB(cain,abel) .A. SEX(abel,"male")))
                    a brother is a male sibling.  Note that
                    constants are denoted by enclosing them
                    within double quotes.

#(DDR,(MALE(x) = SEX(x,"male")))
                    defines the unary relation MALE.

#(DDR,(BRO(x,y) = FATHER(x,z) .A. FATHER(y,z) .A. MALE(y)
   .A. x .NE. y))
                    a brother is a male offspring of the same
                    father, other than oneself.

#(DDR,(STEPMOTHER = FATHER / SPOUSE .A. .N.MOTHER))
                    a stepmother is the spouse of the father
                    who is not the mother.

#(DDR,(NEPHEW = SIBLING / SON))
                    a nephew is the composition of sibling
                    and son.
#(DDR,(UNCLE = .CON.(SIBLING / SON)))
                    in a male world, uncle is the converse
                    of nephew and may be defined as the con-
                    verse of the definition of nephew ...

#(DDR,(UNCLE = .CON.NEPHEW))
                    or simply as the converse of nephew.
```

The first two examples of Figure 1 show the difference between the abbreviated and expanded formats. The problem of the compiler would be reduced to trivia if all definitions were abbreviated (this would be analogous to propositional calculus). The expanded format is necessary for several reasons: it is an important means of implicit existential quantification (explained below); the equality operators EQ and NE take as their operands the dummy variables of the expansion; it adds an important facet to the TRAMP language--easing the mathematical formality with which the programmer must view his relations. Many relations would be much more difficult, and some impossible (e.g., unary relations), to define without this feature, as exemplified by three of the four expanded definitions of Figure 1.

The problem posed by expanded definitions is that we have two levels of operators and operands. We have the "normal" operators, such as conjunction, which would have for its operands, relations. But now relations themselves are a kind of operator, with the dummy arguments of the expansion as their operands. Ultimately, it is these dummy arguments that determine the context, and hence the semantic interpretation of the higher level (normal) operators.

There are no explicit quantifiers. Quantification is handled in the following manner. On the left side of the defining "equation" are two dummy relational arguments.

They are free variables. Any other dummy argument is existentially quantified. Existential quantification is used in two different ways. The first is composition, e.g.,

GRANDFATHER(x,y) = PARENT(x,a) .A. FATHER(a,y)

where the implicit quantification is:

$\exists$ a[PARENT(x,a) $\wedge$ FATHER(a,y)] ==> GRANDFATHER(x,y)

"a," the intermediary (link) is a bound existential variable. The second case is where the existential variable is used to require only that the free variable lies in the domain (or range) of some other relation. We will demonstrate this by using the unary relation "AUTHOR," defined in English as "AUTHOR(x) if x WROTE something," or formally as:

$\exists$ z[WROTE(z,x)] ==> AUTHOR(x)

which in TRAMP would be written:

AUTHOR(x) = WROTE(z,x).

In short, the two (one) dummy arguments that appear on the left-hand side of the equation are free variables, and any other dummy argument is existentially quantified.

4. THE TRAMP INFERENCE COMPILER

The compiler that takes these definitions and generates programs for an associative target machine is a many-pass compiler. The actual number of "passes" or scans of the original source program is variable, depending on the particular program, and is not a meaningful way to talk about the algorithm. Rather, the compiler is said to consist of three distinct "phases."

The first phase does little for the case where the expanded definition is used, but it has a slightly tricky, if straightforward, job for abbreviated definitions. The first phase cleans up the definition, i.e., does a reasonably thorough syntax check and converts to canonical form. Canonical form means that the definition is fully expanded and is in disjunctive normal form. In addition, the first phase does all of the list processing (Figure 2). The programs output by the compiler are coded into a string of bits, which are kept in normal TRAMP lists, and there is a fair amount of list processing required for linkage and compatibility with the rest of TRAMP.

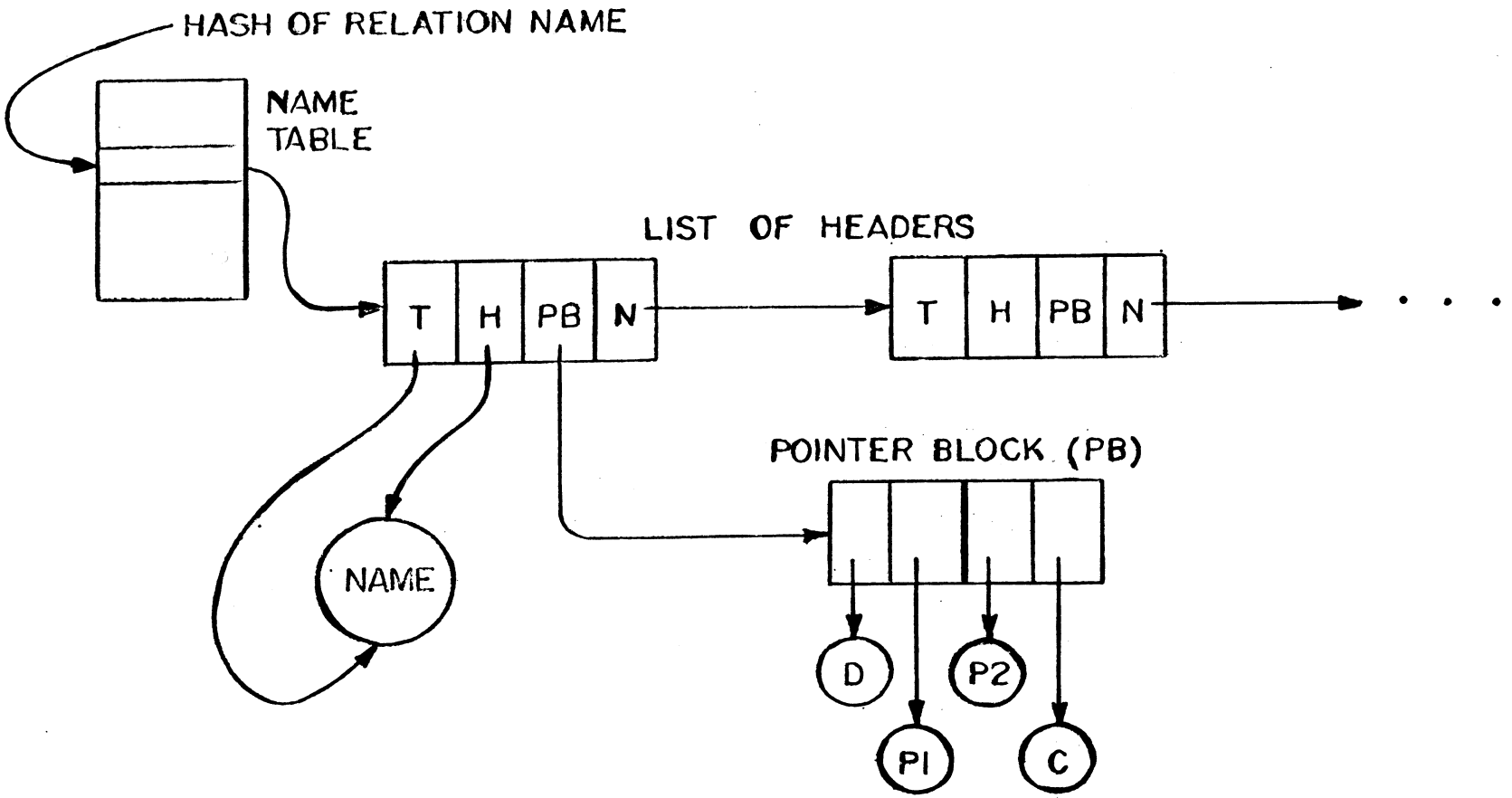


Figure 2. List Processing for Relations

The name of the relation is "hashed" to designate an entry in the defined relation name table. That entry points to a list of headers. Each Header has four pointers: H and T are the pointers to the Head and Tail respectively of the sublist holding the name; PB points to the Pointer Block, described below; and N is the pointer to the next element on the list.

The Pointer Block also contains four pointers: the first, D, is the pointer to the EBCDIC representation of the definition as entered by the user (for editing and displaying); P1 and P2 are the pointers to the two programs output by the compiler; C is the list of relations that this relation is defined in terms of, for circularity checks.

The second phase is the largest (physically) and most complicated. Its job is to build up a series of tables. These source programs are heavily context-dependent, and the tables are designed so that they actually specify precisely the program that must result. The tables constructed during the second phase make the job of phase 3 reasonably simple, in that it can easily translate these tables into object code recursively. In this section, each of the three phases will be explored, hopefully clarifying the preceding paragraphs.

4.1 AN OVERVIEW

Recall the eight questions that can be asked of the associative structure (page 5). Of these eight questions, the compiler is concerned only with F1 and F2. The other six can all be answered by appropriate manipulation of the two programs that would directly answer F1 and F2 (recall that that is the job of the preprocessor). Therefore the compiler addresses itself to:

F1:	A (O) = ?	Ξ	R(x,?)
F2:	A (?) = V	Ξ	R(?,y)

i.e., it assumes that the relation and one of its arguments is given, and the object program must find the other relational argument. The compiler strategy is: given a relation and its first argument, build a relational chain to the second argument. This will be the program P1, which answers

question F1. Then the situation is reversed and phase 3 is repeated, with the result that P2 is output to answer question F2. The difference between the two passes is in the position of the variables, and the difference in the programs ouput is usually a difference in the ordering of the chain, and always a difference in the position of the argument.

The concept of the relational chain is fundamental to the compiler. The construction of the relational chain amounts to building a directed graph, or network, where x is the source, y the sink, and each relation is a directed line. Such a network will often be trivial, and usually simple. Complexity is introduced when there is more than one source or sink. Such a situation can arise when an argument is implicitly existentially quantified. The principal table constructed during phase 2 is designed to represent the digraph.

With this representation of the definition of a relation, it is seen that the compiler can think of each relation as having an "input" and an "output" (the end points of the directed line). If it is building the chain from x to y, then x is always the initial input, and y is the final output, though there may be any number, and any level of intermediary inputs and outputs (generally introduced by composition, though they may arise in other ways). For the

second pass, to generate P2, the situation is reversed in that y is now the source, and x the sink, i.e., all arrows of the digraph are reversed. The relational chains are uniquely specified by the tables constructed in phase 2, as we shall see, and the fact that one relation will in general be the input to another relation (more exactly, the output of one is the input of the next) allows phase 3 to operate recursively.

4.2 PHASE 1

Phase 1 first does an initial cleanup of the definition string (source program) and prepares the necessary list processing. The cleanup includes a preliminary, but reasonably thorough, syntax check. The list processing is complicated by the fact that it is perfectly valid, and often desirable, that a relation have more than one definition. We might define sibling:

```
#(DDR,(SIB = BRO .V. SIS .V. .CON.SIB)),
```

and then realize that this is not complete and append to it:

```
#(DDR,(SIB(x,y) = SIB(x,z) .A. SIB(y,z) .A. x.NE.y)).
```

Changes of this type can be made either by entering a new definition, which will be ORd with the old, or by editing a definition, i.e., altering, deleting, adding to a program. Of course a definition may be destroyed to start fresh, too. In any case the compiler is called, and if the relation is not new, must set up for continuation, including making

provisions for saving the old definition in case the new one is not valid (to the compiler). If the relation is new, table entries must be made and lists set up to hold the necessary information. Finally, all lists and tables are part of the TRAMP machine and must maintain compatibility and respect conventions.

Now, if the definition is the expanded type, then the compiler skips the next section, otherwise the definition is abbreviated and phase 1 must expand it, i.e., fill in the relational arguments. For example,

```
R = Q / (C .V. D) .A. .N.P .V. .CON. A/D
```

must be expanded to:

```
R(x,y) = Q(x,a).A.(C(a,y).V.D(a,y)).A..N.P(x,y).V.A(a,x)
           .A.D(a,y).
```

The job here is to decide what the arguments are. The compiler uses the convention that the definition is in terms of x and y, and assigns other arguments different letters of the alphabet. We will use the convention that lowercase Latins denote arguments, and capital Latins denote relations.

The first thing is to remove the slashes. This is, or would be, very straightforward, except for the fact that no limit is placed on the depth or the complexity of composition. An example of the problem is definition (1):

```
A/B/C .V. D/E ---> A(x,a).A.B(a,b).A.C(b,y) .V. D(x,a).A.E(a,y)
```

(1)

while

(A .V. B/D/C)/E —> (A(x,c).V.B(x,a).A.D(a,b).A.C(b,c)).A.E(c,y)

In this example, we really have to make a complete pass before we can be sure of what the output of relation A will be, since the context is essentially unbounded.

At this point we must introduce some terminology. Everywhere in this paper, the word clause will refer to a matched left and right parenthesis and whatever is between them, with the exception of parentheses used to set off relational arguments, which are not considered to be clauses. The individual relations within a clause will be called terms. Thus the expression

R(x,y) .A. .N.((R1(x,y).V.R2(x,y)).A.R3(x,y))

contains four terms and two clauses.

The routine which removes slashes makes one complete pass over the string, counting nesting "levels" of slashes. This level is determined in terms of clauses and intervening operators. For example, two slashes separated by an OR account for a depth of 1, while two slashes with no intervening operator account for a depth of 2. On this depth-counting pass, each term and each clause is assigned a level. The level is then used to index a pool of dummy variable names, which when inserted, as in (1), will effect the proper expansion of the indicated compositions. The level of the clause is necessary to designate the proper output of a relation, since the level of the term is

sufficient only for designating its input.

The next step is to remove the converse operator. Essentially this amounts to nothing more than reversing the two relational arguments. Thus if the source program had been:

HUSBAND = .CON. WIFE

then the first step will have trivially expanded it to:

HUSBAND(x,y) = .CON. WIFE(x,y)

and the next step should yield:

HUSBAND(x,y) = WIFE(y,x).

Complications arise when the converse operator is applied to a clause, since the clause can be arbitrarily complex in terms of expanded compositions. One cannot simply walk through the clause reversing arguments indiscriminantly, as might be expected. The solution, simple, but not necessarily obvious, is to find the highest and lowest "level" arguments (as determined in the first step) and substitute each for the other, thus reversing local source and sink--and nothing else.

Throughout the list processing and expansion, phase 1 is checking syntax. After any expansion, all parentheses must be removed, leaving the definition in disjunctive normal form. This step is fairly simple, though quite awkward and bulky.* The main problem is that, in general,

*In fact, the compiler is presently being run without this section of code, requiring the user to put the definition in disjunctive normal form. This is not a serious drawback, since 99% of user definitions are extremely simple, unlike most of the examples in this paper.

a clause can not be reduced to a term as it always can in an algebraic language. In algebraic compilers reductions can always be made: the operations specified within a clause will elicit triples, with as many triples as necessary being put together in telescopic fashion until the entire clause is reduced to a single triple, which then can act like a term in place of the clause. Such is not the case with relations. Consider the source program:

R(x,y) = (A(x,a) .V. B(x,b)) .A. (C(a,y) .V. D(b,y))

[which is equivalent to: R = A/C .V. B/D].

This definition consists of two clauses, neither of which can be operated on by itself, since out of context it is meaningless. Furthermore, it is not clear that the clauses can be "distributed," since that results in conjuncts such as: A(x,a).A.D(b,y) where, of the four relational arguments, no two are alike.

On the surface, it might seem that disjunctive normal form might result in inefficient object code. Such an inefficiency could arise from the fact that certain retrieval calls will be made redundantly. Remember that each term that the compiler is working with is ultimately an associative retrieval call, and while we believe the associative machine to be efficient, we don't want to have to go to it more than necessary. As an example, consider the program:

R = (A .V. B) / C

which will be expanded to:

$$R(x,y) = (A(x,a) \ .V. \ B(x,a)) \ .A. \ C(a,y). \qquad (2)$$

Now here is a case where the clause can indeed be reduced to a single term, and it seems far more desirable to do so than to put it into disjunctive normal form:

$$R(x,y) = A(x,a).A.C(a,y) \ .V. \ B(x,a).A.C(a,y)$$

Here, the inefficiency is the calling of the relation C twice from the associative store, when it needs to be called only once.

However, it was decided that the generality gained was worth the slight cost (the overhead incurred is really more of an aesthetic rub than a physical inefficiency, because of the way the associative machine operates; the extra cost is very slight). The main advantages of going to disjunctive normal form are: generality--since all programs can be handled in the same way without having to see when "distribution" of a clause is applicable, etc.; this form yields each of the conjuncts a complete program unto itself, greatly simplifying the final phase of the compiler.

One final transformation that occurs in phase 1 is for equality operators. They have the form "x.EQ.y" and are transformed to have the syntax of normal relations: "@EQ(x,y)", where @ signifies a non-graphic character to avoid interference with the user's names, and also serves as a flag to phase 3 that this is a pseudo-relation, not a retrieval call.

4.3 PHASE 2

Phase 2 begins the actual compilation, since if the program was in canonical form, phase 1 performed only list processing initializations and did not otherwise process the source program at all. The heart of the compilation is in the table construction. The problem is to decide which of the two arguments is the "input" and which is the "output." The approach is to use certain strategies to construct a main table, and then, by means of backtracking, adjust the table until it is correct.

For the purpose of discussion, and without loss of generality, we will assume that in the program being compiled, the left side of the equation is "R(x,y)"; i.e., "x" is the source --the given argument, and "y" is the sink--what our object program is to be designed to derive from the associative structure.

The main table referred to above, and the one that phase 3 will translate into a program, will go by the name CHAIN. One other table of interest is called ARG and is used in the construction of CHAIN.

The ARG table contains entries for each relational argument encountered, other than x or y. The entry consists of EBCDIC name of the argument (up to nine characters are allowed, though if phase 1 generated the names they will be single characters), and the associated term, i.e., the relation that "output" this argument. Of course, if an argument

is come across which has not yet been output by any term, then ARG assigns the term index, and the first term that does output this argument will then be assigned that index in CHAIN.

CHAIN table entries correspond to terms (relations) and contain the indices relative to CHAIN and to ARG of the input of the term. In either case, if the input is "x", then the index is zero. Otherwise the CHAIN index is the index of the term that output this argument, and the AGR index is just the corresponding ARG entry number. The output index, relative to ARG is contained in CHAIN. The last byte of the table entry contains a "count" and several flags. The count is the number of times that the output of this term is used as the input to some other term. There is a flag denoting whether or not the output of this relation is "y". Another flag specifies whether or not this is a real relation or a pseudo-relation (@EQ); this flag is mainly for phase 3, but it is also used during phase 2, since when the table is being shuffled, pseudo-relations are not considered to have outputs. The last flag of interest denotes whether or not this relation has been negated. The negation operator (.N.) is thus used as an attribute of its operand, rather than as an actual operator. This works out rather nicely, again because we are always working with conjuncts--the output code always intersects the operands unless the negation attribute is flagged,

in which case the relative complement (RCOM) is formed.

4.3.1 Initial Table Construction

First a left-to-right scan of the entire program is made (phase 2 handles the entire program at one time--and does not require disjunctive normal form--whereas phase 3 will only process individual conjuncts), and as each term is encountered its arguments are examined. If the arguments are x and y, then there is no decision to be made, since x is always an input and y is always an output. If only one of [x,y] is present, then the other argument is unambiguously specified as being the input or output, depending on which one is present. If neither x nor y is present, then ARG will hopefully give insight into the matter, and if not, then a pure guess is made. To see how ARG is capable of giving such insight, consider the following example:

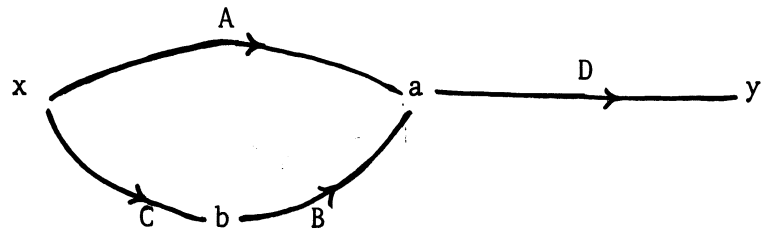
R = A/B/C ⟶ R(x,y) = A(x,a) .A. B(a,b) .A. C(b,y) (3)

First the relation A(x,a) is processed and since x must be the input, "a" is placed on ARG and is noted as being the output of the term A. Now when B(a,b) is encountered, neither of the arguments is x or y, but the calculated guess is made that "a" is the input, on the basis of its presence on ARG and corresponding information. The input to B is therefore the term A. This is only a guess at this point, and it will be checked later. To see where this guess might

be wrong, suppose that the definition had been:

R(x,y) = A(x,a) .A. B(a,b) .A. C(x,b) .A. D(a,y) (4)

[a "backward" way of saying: R = (A .A. C/.CON.B) /D]
This would correspond to the following relational chain where A is intersected with B, but is definitely not the input to B.



It should be noted that the operator .A. can really have two distinct meanings: intersection and composition. In (4), the first .A. operator denotes set intersection, while the other two are composition. We will see that CHAIN does fully specify the program, in that the compiler does not have to worry about the intent of .A., the inputs and outputs will implicitly and unambiguously resolve the matter.

The full details of how the two tables are constructed are too laborious to go into here. We will try, by means of examples, to give some insight into the manipulation (back-up) and function of the table CHAIN. The final contents of CHAIN for program (3) are shown in Table 1.

#	NAME	INPUT	OUTPUT	Y	CNT
1	A	0-0	1	0	1
2	B	1-1	2	0	1
3	C	2-2	FF	1	0

Table 1. CHAIN for Program (3).

In Table 1 and subsequent listings of CHAIN, the following conventions are used. Under INPUT there are two indices, corresponding to the CHAIN index and the ARG index. Though the two will often be identical, they need not be. Any term that puts out "y" will have the Y flag set, and also will show an output of "FF". The NAME column is for the reader and is information indexed by another table.

Table 1 thus represents the input to phase 3. The third phase will work only with entries that are flagged as outputting y. Thus, it will process directly only entry #3 of this example. But the processing of any term means using an input to get an output. The "output" will always be represented in the object program as a TRAMP variable "**". The input must be constructed. If the input is "x" then that is the "given" argument of the call and will be inserted by the interpreter. If not x, then it is another term, and phase 3 must recursively evaluate that term, thereby generating nested function calls in the object program,

i.e., the "chain" is followed, and all links are accounted for. This example shows how the composition form of .A. is handled. The actual program that would be output from Table 1 is:

#(RL,C,#(RL,B,#(RL,A,X,**),**),**)

where "X" represents the given argument of the call. The exact external form that the program would take, of course, is the job of the preprocessor. All programs finally returned by the preprocessor will also contain the prefix: '#(@@,#(RL@,R,X,**);' which retrieves explicit associations as well as disposing of the result of the entire program. In all object programs shown in this section, the prefix is not included. The compiler does not concern itself with the prefix anyway--it is the responsibility of the preprocessor.

The table corresponding to program (4) entails complications we haven't described yet, but since it illustrates two forms of .A., we will display the final program for it:

#(RL,D,#(INT,#(RL,A,X,**),#(RL,B,**,#(RL,C,X,**))),**). (5)

Program (4) also serves as a good example that the two chains built by the compiler need not be especially similar. When x and y are interchanged in program (4) for the second pass, the result is:

#(@@,#(RL,D,**,X),@IS)#(INT,#(RL,A,**,#(@IS)),#(RL,C,**,
#(RL,B,#(@IS),**)))

where "@IS" is a temporary form, described in Section 4.4.

Example 3 showed where the "guess" made concerning the input/output of the arguments was correct, and (4) showed it to be incorrect, i.e., the data gave misleading information to the algorithm. There is one type of program for which the guess has exactly a 50-50 chance of being correct. An example of this type of program is:

R(x,y) = A(a,b) .A. B(x,a) .A. C(b,y)

[a valid, if awkward, way of saying: R = B/A/C],

i.e., the first term that is encountered does not have x or y as an argument. Since ARG at this point is empty, a guess will be made and will be correct depending on the direction of the chain.

4.3.2 Checks and Revisions to CHAIN

After the initial construction of CHAIN it must be checked for validity. Aside from having inputs mixed up with outputs, the construction of CHAIN from ARG is capable of specifying inputs that do not exist, that is, the corresponding entry of CHAIN is blank (the two types of errors often occur together). Sometimes this is valid, if the argument under consideration is a spurious source, as in the next example. Otherwise, it is an error in the table construction and will be corrected. An example of a valid instance of a non-existent input would be:

R(x,y) = A(x,y) .A. B(a,y),

where the argument "a" is implicitly existentially quantified. This program would make use of the special two-variable question mentioned in Section 2, "*@*". The final external form of the corresponding object program would be:

#(INT,#(RL,A,X,**),#(RL,B,*@*,**)).

The process of correcting invalid instances of non-existent inputs is a matter of backtracking and tracing out chains. If a chain terminates before arriving at x (from y), then the last input is looked for as an input to some other term. If it cannot be found as such, then the chain is existential; if it is found, then the term on which it is found is considered to be backwards, and therefore inverted if it is feasible to do so (output not y).

For the program (4) this checking procedure is not enough. After this first check the CHAIN table will appear

#	NAME	INPUT	OUTPUT	Y	CNT
1	A	0-0	1	0	2
2	B	1-1	2	0	0
3	C	0-0	2	0	0
4	D	1-1	FF	1	0

Table 2. CHAIN for Program (4).

as in Table 2. The only chain found was $x \xrightarrow{A} a \xrightarrow{D} y$, which completes a path from source to sink. But there are two terms whose output is index #2, and neither of them is

used, i.e., CNT = 0. The next check looks for precisely this situation: an argument that is found on ARG that is the output of two or more terms, all of whose counts are zero. This is taken to mean that one of the chains is backwards. Which one is backwards is decided by tracing back the non-terminal chains (from their local sink) to see which one was guessed at. In the present example, the trivial chain specified by the relation C was not guessed at and must be correct. Therefore, the input and output of relation B are interchanged, giving C a use-count of 1. The table must be further altered, but we will put that off for a moment. The result of this step is shown in Table 3. Of course, if neither term was guessed at, then no changes are made and the chains remain existential.

It should be noted that for all of this chain tracing and searching for arguments, the procedure is quite efficiently handled by the CHAIN table itself--there is no need to

#	NAME	INPUT	OUTPUT	Y	CNT
1	A	0-0	1	0	1
2	B	2-3	1	0	0
3	C	0-0	2	0	1
4	D	1-1	FF	1	0

Table 3. CHAIN for Program (4);
First Revision.

laboriously scan through the source program for any of this information. The CHAIN table represents a sufficient but very concise reduction of the data.

The fact that phase 2 acknowledges its fallibility by closely checking its results should not mislead one to thinking that the initial table construction is arbitrary or haphazard. In fact, the correcting routines rely heavily on knowing the kinds of decisions that were made in the initial construction; they could not possibly work properly without there being careful order in the construction of the table.

4.4 PHASE 3

The input to phase 3 is the CHAIN table, built and revised as described above. It is not finished yet. The final alterations are now made, after which it is translated into object code, recursively. Phase 2 was programmed to accept arbitrary definitions (programs) while phase 3 will only work with conjuncts, and therefore requires disjunctive normal form. The tables were constructed and revised thus far considering the entire program. From here on, only conjuncts will be considered, as phase 3 iteratively processes all conjuncts individually.

The basic approach will be to find all terms in CHAIN that output y, and form nested intersections (calls to INT or RCOM). When the argument to INT is put out, at

that time the linking information in CHAIN is employed to generate the calls to terms that do not output y. Thus, for the simple case where all terms put out y, e.g.,

$$R(x,y) = A(x,y) \ .A. \ B(x,y) \ .A. \ C(x,y),$$

there will be only three entries in CHAIN, each of which will be flagged as putting out y. The object code will then simply intersect them, as accomplished by the following program:

```
#(INT,#(INT,#(RL,A,X,**),#(RL,B,X,**)),#(RL,C,X,**)).
```

This handles completely all compositions and intersections of terms which output y. But going back to source program (4), we must there intersect two terms whose output is not y. It is for this reason that phase 3 must do one last scan of the CHAIN table to make necessary changes and additions.

We will refer to the intersection of two terms that output y as being <u>global</u> intersection, and intersection of the lower level terms as <u>local</u> intersection. It is apparent that local intersection will have to be handled quite differently from global, since the global are nicely taken care of implicitly by restricting the data units to conjuncts. Local intersection is resolved by <u>attaching</u> one of the operands to the other. Referring to Table 3, there are two terms, both of which output #1 (or "a"). At this point we are reasonably certain that the inputs and outputs are correct,

and we are working with conjuncts, so these two terms must therefore be intersected. The two terms in question are A and B (#1 and #2). We see that term B has a use count (CNT) of zero, which dictates that B must be attached to A rather than the other way round. If they both have counts of zero then there will be a third term. By the algorithm of phase 2, if they all had counts of zero, then this would be an "existential chain" (a spurious sink), and we will see how that is handled subsequently. The process of "attaching" means that the term that is referenced is moved to a new location in CHAIN, and its previous location is flagged (shown in Table 4 as "FFFF" for both input and output), and in place of the other information are the two indices of the operands being attached. In this case, #2 is one of the operands (B), and #5 is the new location of term A, the second operand. Thus, attaching is a binary operation, but clearly we can intersect locally as many terms as we like. For example, if there is yet another term that puts out output #1, then CHAIN term #5 can also be flagged to point to that other term and the second new location for A, etc.

To see that the table for program (4) is now in the proper form to dictate object program (5), we will follow the interpretation of the table that phase 3 will ultimately perform. The only term that outputs y is term #4. Thus the entire program will be a retrieval call for relation D.

#	NAME	INPUT	OUTPUT	Y	CNT
1	-	FFFF	FFFF	2	5
2	B	2-3	1	0	1
3	C	0-0	2	0	1
4	D	1-1	FF	1	0
5	A	0-0	1	0	1

TAble 4. CHAIN for Program (4);
B attached to A.

Now the input to D is CHAIN term #1 which contains a flag meaning that term #1 has been replaced by the local intersection of two terms: #2 and #5. Neither #2 nor #5 is flagged, so the local intersection stops here. Now when #2 is evaluated, its input is #3 and so #3 will be evaluated. The result is the object program (5) reproduced below.

```
#(RL,D,#(INT,#(RL,A,X,**),#(RL,B,**,#(RL,C,X,**))),**)     (5)
```

The remaining table manipulation to be done concerns existential chains, where one or more terms output an ARG index, and all have counts of zero. An example of a program in which this situation would arise is program (6), whose CHAIN is Table 5.

```
R(x,y) = A(x,a) .A. B(a,b) .A. C(a,b) .A. D(a,y)          (6)
```

The procedure is to look for terms that still have use counts of zero (when we say we are looking for counts of zero, if the Y flag is set, then the count is automatically non-zero). Note that in Table 4, when B was attached to A its use count was incremented. Therefore, at this point, any terms with zero counts are sinks of (possibly trivial) existential chains.

#	NAME	INPUT	OUTPUT	Y	CNT
1	A	0-0	1	0	3
2	B	1-1	2	0	0
3	C	1-1	2	0	0
4	D	1-1	FF	1	0

Table 5. CHAIN for Program (6).

Once a term with a zero count has been found, again we trace back through the chain until either the source is encountered, (x), or a link in the chain is used more than once. Here, A is used three times. Thus B is traced back only as far as A, at which point it is attached to A. In the event that an existential chain is trivial, then that term is attached to any other term whose input is the source, x, that has a use count greater than zero. If all sources have counts of zero, then that is a semantic error, a diagnostic is printed, and the program rejected. For non-trivial chains, once they have been attached to some other term, the chain must be inverted, since what is really wanted is for the sink of that chain to be arbitrary, and the local source will be made a local sink to be intersected at the common node of the terminal chain. In the case of program (6), the existential chain is of length one, and it is easily seen that by inverting the I/O of term B, its output will now be the same as term A, making the attachment of B to A

meaningful. The input to B is now non-existent, as it should be since this chain is existential. The same thing happens to term C and the final result is Table 6, which dictates object program (7):

```
#(RL,D,#(INT,#(INT,#(RL,A,X,**),#(RL,B,**,*@*)),
                #(RL,C,**,*@*)),**)                    (7)
```

#	NAME	INPUT	OUTPUT	Y	CNT
1	-	FFFF	FFFF	2	5
2	B	2-6	1	0	1
3	C	2-6	1	0	1
4	D	1-1	FF	1	0
5	-	FFFF	FFFF	3	7
6					
7	A	0-0	1	0	1

Table 6. Final Table for Program (6).

The CHAIN table is now completed, and phase 3 is ready to produce code. The first step is to avoid redundant calls to the associative structure. We said that some redundant calls will be made, but some can be averted, and these are the ones where the calls would be purely redundant since all arguments to RL would be identical. For example, in the program (8):

```
R(x,y) = C(x,a).A.A(a,y) .V. C(x,a).A.B(a,y)           (8)
```

[or R = C / (A .V. B), (8) is equivalent to the

second pass, y-x chain, that would result from (2)]. We don't want to, and don't have to, call the relation C twice. Remember that phase 2 handled the entire program, not just conjuncts, when it was building CHAIN. Therefore, the entry corresponding to relation C will have a use count of 2. Any term whose count is greater than one will be called only once and placed in a temporary UMIST string, from whence it can be called whenever it is needed. Thus the first step in code production is to see if any term has a count greater than one, and if so, to generate the call:

#(@@,#(RL,R,X,**),@IS).

Recall that if the function @@ has a second argument, then that is used as a label, and the first argument is stored in UMIST form storage labeled by the second argument. The form is then callable as if it were a UMIST macro; a macro that consists of a simple character string will result in no "expansion" taking place, but the string itself is the value of the "macro" call.

Here the temporary form is labeled "@IS". @ again is a non-graphic character so as not to interfere with form names of the user; I is the CHAIN index number of the term in question; S is a sequentially generated number used to insure that the names used at various depths of recursion will be unique. The table entry of the pre-generated term is now flagged so that when it is used as an input to another

term, instead of expanding the term anew, the temp form @IS will be called. Thus the object program for (8) would be:

```
#(@@,#(RL,C,X,**),@IS)#(RL,A,#(@IS),**);#(RL,B,#(@IS),
                    **)#(DD,@IS)                    (9)
```

where DD is the UMIST function delete definition, which will destroy the temporary form created, when it is no longer needed.

Perhaps this is a good time to point out another advantage of disjunctive normal form, namely the natural way in which disjunctions are formed by placing a semicolon (set element delimiter) between the conjuncts. Because of the way that UMIST operates, this simply concatenates into a single string the first conjunct, followed by the semicolon, followed by the second conjunct, etc. The The clean-up function @@ will remove dangling semicolons caused by any conjunct that generates a null set. Thus, in (9), each of the two composed calls to RL represents a conjunct, and the two calls are concatenated with an intervening semicolon.

To see how the negation "attribute" is handled, consider the source program:

```
R = A .A. .N.B ——————>  R(x,y) = A(x,y) .A. .N.B(x,y).  (10)
```

During the translation from table to object code, there are actually two scans of the CHAIN table made. In the first scan only those terms that have not been flagged as being negated are considered. Then a second pass is made in which

only those terms that have been flagged are processed. Thus, the first pass generates nested calls to INT and the second pass generates a single call to RCOM. INT is a binary operator and can only intersect two sets at a time, hence the nesting of calls to INT, as in (7). RCOM is likewise a binary operator, but the nature of the operation is such that one compound operand can replace the otherwise nested calls. That is, the second operand is just the union (concatenation) of all negated terms, by DeMorgan's law. Thus the object program resulting from (10) is:

#(RCOM,#(RL,A,X,**),#(RL,B,X,**)).

Had there been more terms negated, they would be concatenated with the call for B in the second argument to RCOM. In very much the same way, the equality operators, or pseudo-relations, are taken care of by calling INT or RCOM as required.

Since phase 3 is recursive, this double scan is performed at each level of recursion before popping up. Thus there is no problem when an intermediate level term is negated. The "input" to any term is processed by recursively treating that input as a "program," recursing as deeply as necessary, and then popping back up. At each level of the recursion two passes of the applicable section of CHAIN are made. As a final example which might better demonstrate the generality of this technique, we have the following definition and its corresponding object program.

```
R = A / (B .A. .N.C) / D .A. E .A. .N.(F .V. G)
#(@@,#(RL,A,X,**),@IS)#(RCOM,#(INT,#(RL,E,X,**),#(RL,D,(RCOM,
#(RL,B,#(@IS),**),#(RL,C,#(@IS),**)),**)),#(RL,F,X,**);
#(RL,G,X,**))#(DD,@IS).
```

At each step of the recursion, a copy of the relevant section of CHAIN is what is being processed. The cost of making the copy and the storage that it uses is very slight* and is compensated for by the generality with which it can be processed. The initial step, then, is to find all entries in CHAIN whose Y flag is set, and put those entries into the first copy of CHAIN. After this has been done the recursion can be entered.

The recursive section of code then goes through the current copy of CHAIN, going through the copy once looking for non-negated terms, and then looking for negated terms. When a term is processed, code is produced signaling for a call to RL, and then the signal for a constant (the name of the relation--strictly speaking, this is redundant). A subroutine is then entered to process the relation name. A check is made of TRAMP name tables to see if the name is already present. If not, it is inserted in the proper table, and in either case, the pointer to it is returned and will be part of the object code. This subroutine then adds the

*Programs are always single sentences, and even a "very large" (20 relations) program is compiled in a few milliseconds.

name to a list of all relations found on the right side of the defining equation. This list will be used later for a "circularity check." Also, the name is compared with the name of the relation being defined. If this matches, then the name is not put on the circularity list, but instead another subroutine is entered to determine if the current definition specifies the relation as being transitive, symmetric, or both.

4.4.1 Transitive and Symmetric Relations

The rules for deciding on these attributes are as follows:

If an entire conjunct consists of just this one relation, namely the relation being defined, then: if the order of the relational arguments is reversed on one side of the equation from what it is on the other, then this relation is flagged as being symmetric and no code is generated for this conjunct; if the arguments were not reversed, then this conjunct is just noise and is completely skipped by the compiler.

If the input to this relation is itself, then it is transitive and is so flagged. This part of the conjunct produces code only for the inner nested function call (composition always results in nested calls to RL, see program (5)). The transitive closure will be formed by a special function discussed below. If, in addition to the relation being its own input, the composition is specified "backwards":

R(x,y) = R(x,z) .A. R(y,z)

instead of the "normal" way:

R(x,y) = R(x,z) .A. R(z,y)

then, by the commutativity of the conjunction operator (.A.), it follows that the relation is both transitive and symmetric.

Finally, if the relation name appears on both sides of the equation, but is neither transitive or symmetric, then the function call that it generates is like any other except that RL@ will be called, rather than RL, thus avoiding an infinite recursion for the interpreter. (This is not really correct, see Section V.)

When the relation is symmetric, it is so flagged, but no code was produced for the conjunct that determined it as being symmetric. At interpretation time, any other code specified by the program is generated, and in addition to the normal prefix that the interpreter always puts out: '#(@@,#(RL@,R,X,**);', it will now also put out: '#(RL,R,**,X,@)' where it has reversed the arguments. Also, there is another argument in the call to RL, namely the argument '@'. As usual, '@' denotes a non-graphic character. Here it is the signal to the interpreter to expand the program for this relation as if it was <u>not</u> symmetric. This is necessary so that this reversal will happen only once, instead of an infinite number of times. To see this a little better, we will follow the simple example of a

relation that is symmetric, and symmetry is its complete definition:

SPOUSE = .CON. SPOUSE

The complete program compiled from that definition would be just the flag for symmetry. Now, if the interpreter were invoked with the retrieval call: #(RL,SPOUSE,PETER,**), then the value of that call would be:

#(@@,#(RL@,SPOUSE,PETER,**):#(RL,SPOUSE,**,PETER,@)).

The first call to SPOUSE, made via RL@, will bypass the interpreter completely. The second call will go to the interpreter, but the extra argument in the call will permit nothing but the prefix to be generated (since this definition has no other information), and then the recursion will cease.

Transitivity is not as simple and requires a special function. The algorithm used by this funtion to form the transitive closure is to start with the given of the problem, X, and generate an answer set. This answer set then becomes the new "X", and the process is recursively repeated until, at some level, nothing new is added to the combined answer set.

There is a subtle difference between relations that are "created" by their definitions, and those that are "filled out." Relations defined in terms of themselves are filled out: it must be assumed that a kernel of the relation is explicitly stored (or derivable) and the definition

gives the inference rule to form the closure. The example of the two definitions for SIB (page 21) demonstrates both types: the first definition creates SIB, and both definitions fill it out.

With these special cases taken care of, the more mundane code generation is quite straightforward. Code is generated for each of the terms in the current copy of CHAIN. Then the stack is popped and the last copy is resumed. Whenever the "input" for a term is nonzero, i.e., another term, then the stack is pushed and a new level of recursion is entered.

4.4.2 Further Semantics

Besides the syntactic check made during phase 1, there are several semantic checks carried out in the final phase. Principally, these checks concern "circular" definitions, and definitions which specify global complements. Circular definitions are seen to be invalid, since if such a program were ever executed, the result would be an infinite recursion (at the UMIST level of recursion). An example would be:

```
R1 = R2 ...
R2 = R1 ...
```

Now a retrieval call to R1 would generate a call to R2, which in turn would generate a call to R1, ad infinitum.

This situation is checked for by means of the lists

constructed by the subroutine that processes constants (see Fig. 2). At the conclusion of the first pass (the x-y chain), this list is checked. The list is scanned, and each element is compared with the name of the relation being defined. If it matches, then the definition is circular and is rejected. If it does not match, it is looked for on the defined relation name table. If it is found there, then it will also have such a list, which is now appended to the current one. The process continues until the list, however long it might grow, is exhausted.

Global complements are not so obviously incorrect. The problem arises with unary relations (where global complements might most likely be used). In this case, the compiler has no way of knowing with respect to which universe the complement is to be taken, and it could not know without drastically changing the entire design and mode of operation.

Global complements are easily checked for, again because of the disjunctive normal form and the fact that phase 3 makes two passes over each segment of the CHAIN table. If the first pass ever results in no code being written, then a global complement was specified.

5. CONCLUDING REMARKS

Admittedly, most of the examples in this paper have been of relations for which one would be very hard pressed to come up with real life interpretations, except for the examples in Figure 1. Given that UMIST is an awkward (but very powerful) language to begin with, it has also been demonstrated that even simple definitions will quickly lead to unwieldy code. To this extent it is important to relieve the programmer of this burden. Indeed, this is the function of any compiler. The TRAMP system with the compiler has been successfully used to this end in applications programming.

Although the compiler is currently in use, with good results, it cannot be said to be completely debugged. There are certain logical inadequacies, such as: in Section 4.4.1 it was stated that if a relation is defined recursively but is neither transitive or symmetric, then it is not treated specially. To show that this is not proper, consider

```
B = P / B.
```

The logical implications of this definition demand that we compose <u>not</u> P with B, but the transitive closure of P with B.

When the system is finally deemed to be debugged, work needs to be done on the routine which converts to disjunctive normal form, making it less bulky and more practical.

This item is not too pressing since the great majority of definitions actually used are already in this form, and the burden on the programmer almost non-existent. (In Fig. 1, only one of the ten definitions even contains a clause!) After this, there is an obvious extension to the system that should be worked on.

This would be to allow the user to specify whether the right side of the equation implies the left, or is equivalent to it. Presently, the "equation" specifies one-way implication, to the left. One would very much like to have "if and only if" declarations. Such a feature would be more than a convenience. For example, if one defines husband to be the converse of wife, he cannot now define wife to be the converse of husband, since that would be a circular definition! The solution would be to allow him to say: HUSBAND(x,y) <===> WIFE(y,x). The code for such an extension has already been partially written, but unfortunately is far from working, due to other priorities. The new format will be that the equal sign will denote equivalence, whereas if one-way implication is desired, the assignment symbol (:=) would be used. Of course, such a format allows the user to specify equivalence incorrectly, i.e., when the left side of the equation cannot logically give any information about the terms on the right side.

REFERENCES

1. Ash, W.L. and Sibley, E.H., TRAMP: An Interpretive Associative Processor with Deductive Capabilities, ACM National Conference, Las Vegas, August 1968, pp. 143-156.

2. Ash, W.L. and Sibley, E.H. TRAMP: A Relational Memory with an Associative Base, Technical Report 5, Concomp Project, University of Michigan, Ann Arbor, June 1968.

3. Feldman, J.A., Aspects of Associative Processing, Technical Note 1965-13, Lincoln Laboratory, Massachusetts Institute of Technology, Lexington, April 1965.

4. Pinkerton, T.B., "UMIST Manual," The University of Michigan Terminal System Manual, 2nd edition, Vol. II, The University of Michigan Computing Center, Ann Arbor, December 1967, pp. 717-764.

5. Mooers, C.N., "TRAC, a Procedure-Describing Language for the Reactive Typewriter," Comm. ACM, 9, March 1966, pp. 215-219.

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*
THE UNIVERSITY OF MICHIGAN
CONCOMP PROJECT

2a. REPORT SECURITY CLASSIFICATION
Unclassified

2b. GROUP

3. REPORT TITLE

A COMPILER FOR AN ASSOCIATIVE OBJECT MACHINE

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*.
Technical Report 17

5. AUTHOR(S) *(First name, middle initial, last name)*

WILLIAM L. ASH

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
May 1969	53	5

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
DA-49-083 OSA-3050	Technical Report 17
b. PROJECT NO.	
c.	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
d.	

10. DISTRIBUTION STATEMENT

Qualified requesters may obtain copies of this report from DDC

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
	Advanced Research Projects Agency

13. ABSTRACT

This paper discusses the design and construction of a compiler whose source language consists of sentences of a restricted predicate calculus, and whose output object code operates on a simulated associative target machine. The source programs are characterizations of relations, used for deriving one relation from others, and for completing relations. A program realizes the completion of a relation when it defines that relation in terms of itself, i.e., when the definition is recursive.

The most significant feature of such a compiler is that there are two levels of operators and operands. The first level operators are the logical connectives whose operands are relations, which in turn have as their "operands" relational arguments. The lower level operands determine the context-dependent interpretation of the higher level operators. The parse is further encumbered by the fact that the logical connectives in such a framework do not lend themselves to a production grammar. The situation is resolved by constructing a digraph to represent the definition of the relation. This construction proceeds using a back-up technique. The output of the compiler is an object program in the form of a macro definition, to be expanded for an interpretive associative target machine.

DD FORM 1 NOV 68 1473

14 KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
associative processor						
compiler						
content addressable						
inference						
interpreter						
list processor						
macro generator						
relational memory						
relations						