MIT/LCS/TM-143

AN INTERMEDIATE FORM FOR DATA FLOW PROGRAMS

James William Leth

November 1979

AN INTERMEDIATE FORM FOR DATA FLOW PROGRAMS

by

James William Leth

October 1979

Massachusetts Institute of Technology

Laboratory for Computer Science

Cambridge Massachusetts 02139

An Intermediate Form for Data Flow Programs

by

James William Leth

Submitted to the Department of Electrical Engineering and Computer Science
on 13 July 1979, in partial fulfillment of the requirements for the Degree of
Master of Science.

ABSTRACT

A data flow program, often represented as a data flow graph, is a program that expresses a computation by indicating the data dependencies among operators. A data flow computer is a machine designed to take advantage of *concurrency* in data flow graphs by executing data-independent operations in parallel (that is, a sequential ordering exists only between operations for which the result of one operation is an operand of the other). This thesis presents a form of computer representation of data flow programs (based on data flow graphs) that can serve as an intermediate form in the translation of source language code into machine code for a data flow computer. The proposed intermediate representation is implemented in the structured programming language CLU, and is designed to allow analysis and transformation of programs (for optimization purposes) to be performed either automatically or with programmer interaction.

Thesis Supervisor: Jack B. Dennis
Title: Professor of Computer Science and Engineering

Key words: data flow graphs, data flow language, data flow computers, VAL, applicative programming, parallel programming, graph representation

Acknowledgements

I would like to thank my thesis supervisor, Jack Dennis, for advice and encouragement from the beginning to the end of this endeavor. I gratefully acknowledge the support of Bell Telephone Laboratories, Inc., without which this thesis would not have been written. I also thank the members of the data flow project for an interesting and enjoyable working environment. I owe much to Prof. Barbara Liskov and the CLU group, especially Bob Scheifler and Russ Atkinson, for the software foundations of this thesis, and for major contributions to my philosophy of software engineering. I thank Alan Snyder and Eliot Moss for the text formatter used in the preparation of this thesis. I am very grateful for the personal friendship and moral support of Christine Comins, Larry Crume, Martha Buck, Sally Kornfeld, Don Aoki, Andy and Donna Boughton, and Glen Miranker. Lastly, I wish to thank my parents, Charles and Harriet Leth, for making all of this possible.

CONTENTS

FIGURES

1. Introduction

This thesis presents a computer representation of data flow programs that can serve as an intermediate form in the translation of high-level source code into machine code for a data flow computer. This section describes the *data flow computer* designed at MIT, and presents the data flow *graph* (or data flow *schema*), which is the most usual manner of specifying a data flow program. The data flow program source language VAL is also discussed in this section.

Section 2 presents the *operator* cluster, which is a CLU abstract data object that implements the representation proposed. (The programs used in the implementation are presented in the appendix.)

Section 3 presents a scheme for translating a subset of VAL programs into their equivalent data flow graphs represented as *operators*. This translation scheme can be used by a VAL compiler. Compound expressions are translated by first deriving the *operator* form of their subexpressions, then using these *operators* as subgraphs in building the graphs for the larger expression. The final subsection of section 3 discusses some of the VAL constructs for which a satisfactory form of data flow graph has not been derived, specifically the **forall** expression and procedure invocation.

Section 4 briefly discusses transformations and optimizations of data flow programs in the context of the *operator* representation scheme, and shows that this representation scheme offers a sufficient means of performing such transformations.

Section 5 summarizes the results of the thesis and the conclusions of the author.

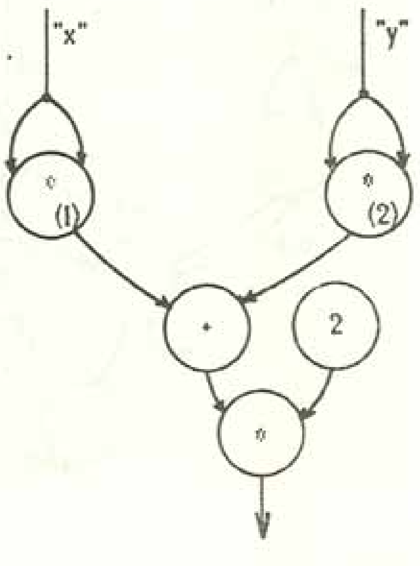
The appendix presents the actual CLU programs that implement the *operator* cluster and its support software, and includes examples of execution of the programs.

1.1 Data Flow Programs and Graphs

A data flow program expresses a computation by explicitly indicating the data dependencies among the operators involved in the computation. It is generally represented as a *data flow graph*, or *DFG*[7]. Figure 1 shows an example of a DFG that computes $2*(x^2+y^2)$ (where $*$ is multiplication). The circles represent operators and the arrows show the direction of data flow. The DFG of Fig. 1 shows that the multiplication operators (1) and (2) are independent of each other. They are thus *concurrent* in the sense that they *may* be executed in any order (including simultaneously) without affecting the result of the whole computation. A set of output values (tokens) will be produced on each output arc of the graph (in this case there is only one output arc) for each set of input values (tokens) sent to the input arcs of the graph. However, since there is no ordering between the executions of operator 1 and operator 2, it is possible that operator 1 may be ready to fire a second time before operator 2 has fired once. In order for the data flow graph to be *safe* (that is, to prevent the possibility of generating two tokens on the arc from operator 1 to the plus

---

Fig. 1. Data flow graph to compute $2*(x^2+y^2)$.

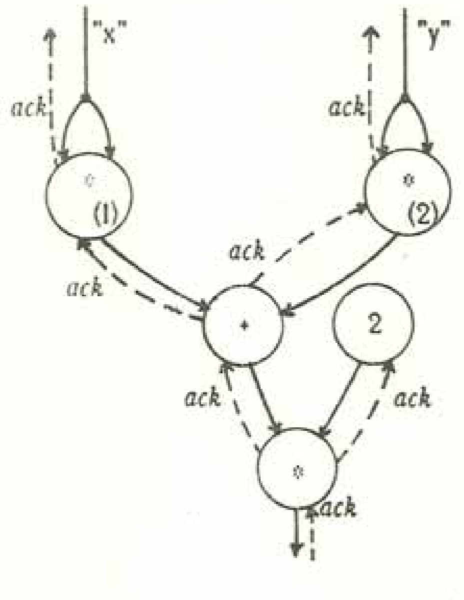


operator), we need to add the concept of *acknowledge* arcs to data flow graphs. (Of course, if arcs of the graph were considered to have unbounded buffering capability, then this would not be needed, as the presence of two tokens on the same output arc would not then interfere with the deterministic behavior of the graph.)

An acknowledge arc insures that an operator cannot fire until its output arcs are empty (that is, all operators attached to those arcs have fired, consuming the last output tokens produced). To add acknowledges to the graph of Fig. 1, we replace each data arc with a pair of data and acknowledge arcs, as shown in Fig. 2.

The firing rules for operators need not be changed -- tokens must be present on *all* input arcs of an operator (including acknowledge arcs) before it can fire; when an operator fires it consumes a token from each input arc and places an output token on each output arc (including acknowledge arcs). With every data path of a data flow graph replaced by a data-acknowledge pair, the safety of the graph is guaranteed, and no new difficulties are introduced[5].

Fig. 2. Revised graph of Fig. 1.



As will be discussed in section 4 of this thesis, acknowledge arcs are not required on all data paths, and an optimization phase may eliminate some of them; for now, however, we can assume that all data paths will have an acknowledge arc corresponding to the data arc.

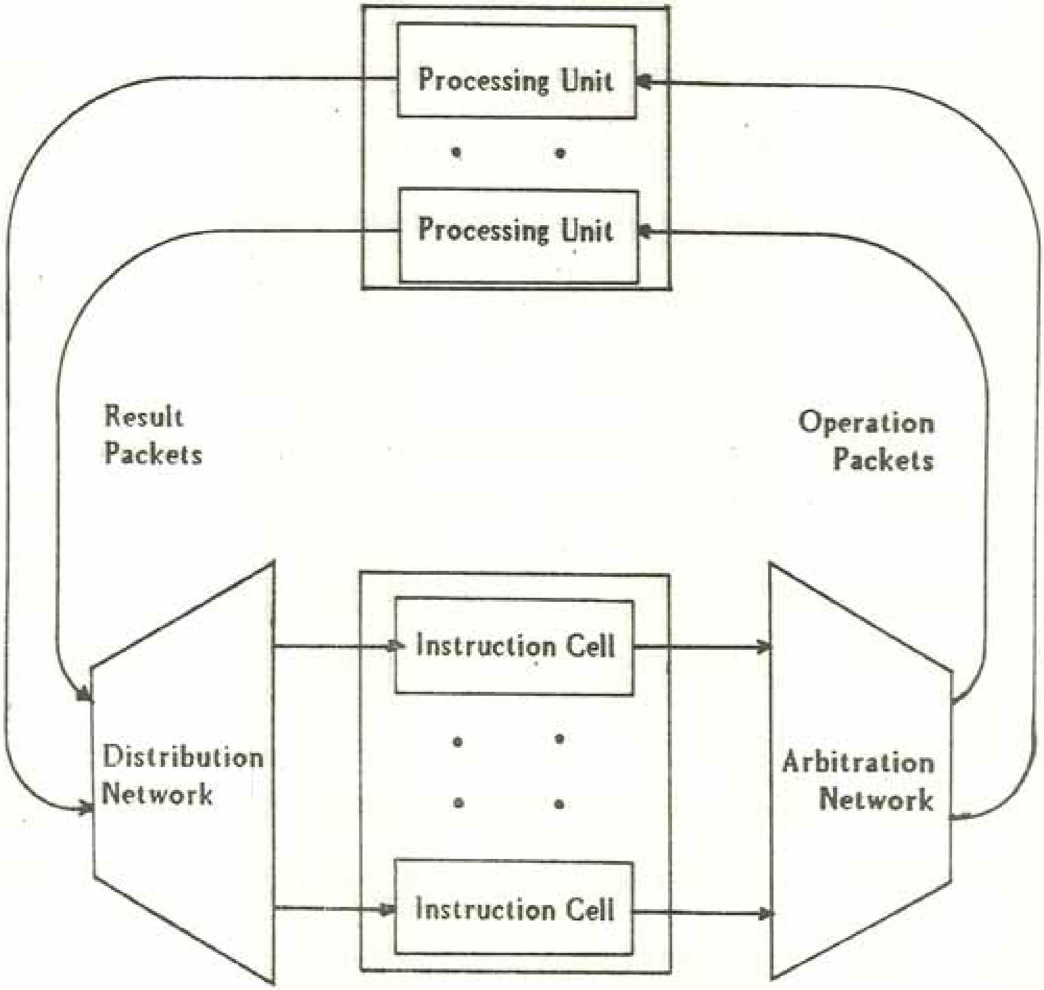
1.2 A Data Flow Computer

The Computation Structures Group at MIT has been developing a *data flow computer*[6,8] which can take advantage of the concurrency of data flow programs by executing independent operations such as (1) and (2) of Fig. 1 in parallel. Figure 3 illustrates the basic form of data flow computer. It is a *packet communication system* in which data (operation requests and operands) flow in packets in the directions indicated by the arrows.

Data flow program instructions reside in the *instruction memory* awaiting the arrival of their operands. These operands are delivered to the appropriate instruction cell by the *distribution network*. When all the operands needed by a particular instruction are available the instruction cell *fires*, delivering an *operation packet* to the *arbitration network*. These operation packets consist of an operation code, operand data, and destination addresses.

The arbitration network delivers operation packets to appropriate *processing units*, which perform the required operations and emit *result packets*. The distribution network then delivers the result packets to their destinations. All parts of the machine operate in parallel, asynchronously. At any given time many instruction cells can be enabled for firing, so that very high throughput can be achieved. More details of the data flow computer architecture can be found in [6] and [8].

Fig. 3. Form 1 data flow computer.



1.3 VAL -- A Data Flow Source Language

A high-level programming language named **VAL** is being developed for source language program specification[2]. **VAL** is an applicative (side-effect-free) language. Consequently, **VAL** programs are *functions* (whose bodies are *expressions* composed of *subexpressions*), rather than *statements*. **VAL** is a strongly-typed language with a rich set of primitive types;[1] however, as it is not my intention to define a **VAL** compiler in this thesis I will generally omit type specifications, structured data types, and type checking from my examples. The language constructs I am most interested in are **if** ... **then** ... **else** ... **end**, **for** ... **do** ... **iter** ... **end**, and **let** ... **in** ... **end** (previously written as **begin** ... **result** ... **end**). Examples of these constructs will be shown in Section 3. Their semantics can be briefly described as follows.

The expression "**if** <exp_1> **then** <exp_2> else <exp_3> **end**" represents the conventional (applicative) conditional expression. The value of the expression is either the value of <exp_2> or that of <exp_3>, depending on the (boolean) value of <exp_1>. Only one of the **then** or **else** clauses is evaluated when the **if** expression is evaluated.

The expression "**for** <*binding-expression*> **do** <*body-exp*> **end**" represents an iterative expression. The notation <*binding-expression*> represents an expression of the form <*identifier-list*> := <*exp-list*>, where <*identifier-list*> is a set of variable names separated by commas, and <*exp-list*> is a set (of the same arity) of expressions, also separated by commas. The value of the **for** construct is the value of <*body-exp*> evaluated in an environment in

---

[1] In order to allow structured data types to be tokens on the data flow computer it is necessary to add a *structure memory* and *structure controller* to the architecture of Fig. 3. This does not affect the basic form of data flow programs, so will not be explained in detail in this thesis. Details can be found in [1].

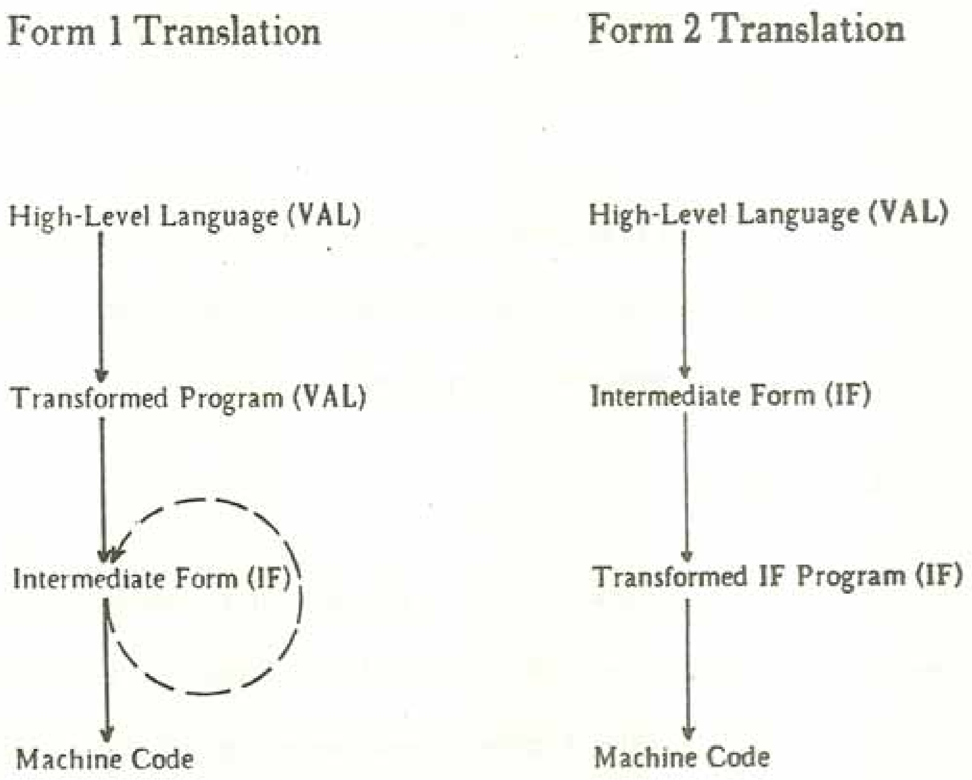
which each identifier (iteration variable) in the binding expression is bound to the value of the corresponding expression. If *<body-exp>* contains a subexpression of the form "**iter** *<binding-expression>*", then the value of that subexpression is the value of the **for** expression evaluated in an environment with the iteration variables given the *new* bindings.

Finally, the expression "**let** *<binding-expression>* **in** *<body-exp>* **end**" is exactly equivalent to the **for** expression except that no **iter** subexpression is permitted in the body, and thus no iteration occurs; the bindings are performed only for one evaluation of the body. (Of course, the body of a **let** expression may contain a **for** loop as a subexpression, but then the **iter** subexpression is part of the **for** construct containing it.)

Expressions in VAL can also be *multi-expressions*, which are tuples of basic expressions, normally written as several expressions separated by commas. For the most part, I will show only expressions of arity *one* in my examples, simply for clarity. The programs that implement the *operator* cluster and the code demonstrating the transformations of VAL expressions into their *operator* representations are, of course, able to handle the general case of higher arity.

The translation process from VAL to data flow machine code may take one of two basic forms, as illustrated in Fig. 4. In the first form, the programmer would initially construct his program in the high level language (VAL). From this program a *transformed program* (in VAL) would be produced with the aid of the computer checking the validity of the transformations (probably these transformations could not be *completely* automated). In constructing this transformed program the programmer would make decisions about the degree to which operations are to be performed in parallel as opposed to iteratively, and other types of space-time tradeoffs. This would involve choosing among alternate control

Fig. 4. Possible translation sequences.



---

structures and alternate data structures. For example, performing the same computation over several data objects could be done sequentially by treating the data as a *stream* of objects fed one at a time to the operator involved, or by treating the data as an *array* of objects, and operating on each element in parallel, which would require many more instruction cells to be used. From the transformed program the compiler would generate an intermediate form of the program, to be used in the final phase of code generation.

In the second form of translation process the transformed program would be generated from the intermediate form rather than the high level form of the initial program. This arrangement would not appear to work well unless the transformations could be completely automated, as the programmer would want to work with his program in the high level form. Because of this we prefer the first form of translation process, and will assume that this is the method to be used. However, I assume that further transformations will be done on the intermediate form to perform optimizations specific to data flow graphs as well as more traditional optimizations such as movement of loop invariants. This will be dicussed more in section 4.

Currently work is being performed by other members of the Computation Structures Group on a VAL translater[3]. This thesis will influence the further development of this work, and has in turn been influenced by its goals.

2. The Operator Cluster

The intermediate form presented in this thesis is an abstract data type (**cluster**) named *operator*, implemented in the language CLU[10]. It represents a data flow program as an *operator*, which is either a primitive operation or a graph of other (interconnected) *operators*. The translation of source constructs into *operators* is performed in a bottom-up fashion, building the graph for an expression out of the subgraphs derived from its subexpressions. By linking the *operator* representation to the nodes of the program's semantic tree, each graph can be built up as the tree is built. The translation defined by Brock[4] is used in building the network of operators.

The language CLU was chosen primarily because of its well-structured form and its concept of *data abstractions*[9]. A **cluster** (such as *operator*) is a user-defined abstract data type with a very restricted interface between the defining module and the using modules. CLU provides that the only interface between a cluster and the programs that use it is in the operations (procedures and iterators) defined for the cluster. In particular, the actual representation (**rep** data type), and any utility procedures defined within the cluster but not listed in the **cluster is . . .** header cannot be accessed outside of the cluster. Thus the behavioral specifications of the cluster operations completely define the cluster for any using programs.

An *operator* is either a *primitive* operator, or a *graph* operator. The set of primitive operators is fixed, and information about each primitive operator is kept in a table accessed by the operations of the cluster. Graph operators are built out of other operators (both primitives and graphs). The operators making up a graph are its *components*; a graph is its components' *owner*. Each component of a graph has a unique integer *id* assigned by order of inclusion into the graph (starting with 1). Components of a graph can be selected by this id in the same way that elements of an array are selected. Components can also be selected by following a graph's inputs or outputs into the corresponding component operator inputs or outputs (see below).

An operator has an *opname*, that is the name for the type of operation performed by the operator (for example, the opnames of some primitive operators are "+", "-", "and", "constant"). Its connections to other operators occur at its *inputs* and *outputs*. Primitive operators have a fixed number of distinct inputs and outputs (almost all primitive operators have only one distinct output). The inputs (outputs) of a graph correspond to *subinputs* (*suboutputs*), which are certain of the inputs (outputs) of its components. This correspondence is made when the graph is *sealed*, as described below.

The components of a graph can have *attachments* to other components of the same graph according to the following rules:

> 1) every attachment is a connection between some operator's input and some operator's output,
>
> 2) an operator whose output is attached to operator O's *i*th input is called the *source* of that input.

3) the operators with inputs attached to operator O's *i*th output are called the *destinations* of that output.

4) no input of any operator has more than one source.

Operators also have *acknowledge* inputs and outputs. All primitive operators have only one acknowledge input, and all but the merge operator have only one acknowledge output; graphs may have several acknowledge inputs and outputs, each corresponding to an acknowledge input or output of a component of the graph. The correspondence between graph acknowledge inputs and outputs and those of the components is made explicitly by calls to the operations *operator$make_ack_input* and *operator$make_ack_output*. Acknowledge arcs are attached in a way similar to data arcs, but there is one important difference: an acknowledge input of an operator can have any number of sources. This is because there is no *value* to an acknowledge token as there is to a data token; there is no need to know which acknowledge token arrived on which input, since they are not *operands* of the operator, only signals. For this reason primitive operators need only one acknowledge input, and the concept of numbered acknowledge inputs exists only to provide a consistent treatment of acknowledges for graphs (in which, clearly, it is necessary to separate acknowledges destined for different components).

Each primitive operator *expects* a certain number of acknowledge signals to be enabled for firing; in addition, it is initialized to have already "received" a certain number of acknowledges. This is necessary to ensure that the graph is *live*, that is, until execution is complete there must always be some operators enabled for firing. Normally the number of acknowledges *expected* (to enable the operator to fire) is equal to the number of acknowledge arcs pointing to it. However, this is not always the case, as it may be desired, in making optimizations on the graph, to acknowledge an operator from *either* of two (mutually exclusive) alternatives dependent on the output of the operator. Then the number of acknowledges expected would be *one* although *two* acknowledge arcs would point to the operator. Because of these considerations, the following rule was adopted in the design of the *operator* representation: when acknowledge arcs are connected (via the *operator$acknowledge* operation), the number of acknowledges expected by the receiving operator is incremented, so that unless otherwise changed the number expected is equal to the number of arcs pointing to the operator; however, the number expected can be explicitly changed (via the *operator$set_acks_expected* operation). In all cases the number of acknowledges initially received is explicitly set by the operation *operator$set_acks_received*.

A graph can be either *sealed* or *unsealed*; it is unsealed until the operation *operator$seal* is performed on it, and it remains sealed from that point on. Attachments can only be made within a graph before it is *sealed*. A graph can be included as a component operator within another graph only after it has been *sealed*. Primitive operators are always sealed.

The act of *sealing* a graph causes any unconnected inputs to components of the graph to become inputs to the graph operator itself. Thus, attachments to the inputs of a (sealed) graph operator are equivalent (in terms of the final data flow graph) to attachments to the corresponding inputs to components of the graph. The correspondence between particular graph inputs and component inputs is made according to the order of inclusion of the component operators in the graph (that is, by increasing order of *id*). Suppose operator x is included in an empty graph g, and then y is included in g. When g is sealed, any inputs to x that are still unconnected to any other components of g will correspond to inputs to g. The number one input to g will correspond to the first unconnected input of x (in increasing order of input number), the second input to g will correspond to the second unconnected input of *x*, and so on until there are no more unconnected inputs to *x*. The remaining inputs to *g* will be the unconnected inputs of *y*, in order.

Particular inputs and outputs of an operator are identified by their input/output *numbers*. However, a mechanism is provided to give individual inputs and outputs *names* (i.e. character string identifiers). The principal use of this feature is to follow the identifier binding mechanisms in the source language. In the first example in the next section, the association of the identifiers "i", "j", and "k" with their respective inputs is effected by giving those inputs the *names* "i", "j", and "k" via the *name_input* operation. Thereafter, the input numbers that these names refer to can be looked up via the *input_no* operation, and the entire set of names associated with inputs to the graph can be yielded, one at a time, by the *input_names* iterator. In terms of the binding of identifiers in the source language, these names identify the *free variables* referred to in the source text that corresponds to the operator.

When a graph operator is built out of component operators, any input/output names associated with the component operators are *inherited* by the graph operator in the following sense: when the graph is *seal*ed (i.e. its construction is complete), the unconnected inputs and outputs of each of the component operators become inputs and outputs of the graph; if any of these component operator inputs and outputs are *named*, then the corresponding graph input or output inherits the same name.

When two operators are included in a graph, each of which has an input with the same name (e.g. in the first example in the next section, both exp_1 and exp_2 have an input named "i" and will both be included in the graph for the entire *if* construct), then the two subinputs merge to form the *same* graph input. Conceptually this is the same as if an *identity* operator were included between the graph input and the two subinputs, but no such operator is explicitly added to the graph. Note that outputs from components of a graph cannot be merged into a single graph output in this way, since an operator cannot have two sources of the same input. It is, therefore, an error to attempt to name two distinct suboutputs of a graph with the same name.

It is important to realize that a graph does not inherit the names of its subinputs and suboutputs until the graph is sealed, since the association of graph inputs and outputs with subinputs and suboutputs cannot be made until that time. If attachments within a graph are made such that a named input or output of a component operator is connected, then that input or output will not be in the set of graph inputs or outputs when the graph is sealed, so the name will not become known at the outside of the graph (that is, it will not be a name for an input or output of the graph in question). Note, however, that making such an attachment within a graph amounts to *binding* the identifier corresponding to the name, and therefore this graph must correspond to the source construct in which the identifier is bound (i.e. a **for** . . . **do** . . . **end**, or **let** . . . **in** . . . **end** construct). In any case the input and output being connected cannot have *different* names, or else the signal *name_conflict* is raised.

It can be seen that this nicely parallels the scoping rules of the source language, in that a name is known within a graph only when the corresponding identifier is known within the source text corresponding to the graph.

The *description* field of an operator is meant to provide necessary semantic information (e.g. a *constant* operator is defined for all constant values, and its "initialization" value is defined in its *description*). The exact specification of what information goes here and how it is structured must wait for a more complete specification of the front end of the translater. However, as suggested in the examples, this information could include what role a subgraph plays in the containing graph, or any information deemed useful when the operator is displayed by the function *operator$write*.

3. Translations of Basic VAL Constructs

In this section a translation scheme for most of the basic VAL constructs will be presented, using the operations of the *operator* cluster. Entire VAL programs can be translated into their *operator* form by applying these procedures in a bottom-up fashion, translating each expression by first translating its subexpressions; this can be done in parallel with the construction of the semantic tree. Each node of the tree will have, as one of its components, the graph operator built to correspond to it. Nodes higher in the tree will point to graphs containing the graphs for their descendants as subgraphs. A representation of the semantic tree is not proposed here, but it appears that an obvious form of CLU *record* structure could be used, with the graph pointer simply a field of type *operator*.

3.1 Conditional Expression

Consider the VAL construct shown in Fig. 5. In terms of the operator cluster, each exp_i can be represented by a graph operator whose inputs correspond to the identifiers (that is, the free variables) used in the expression. The definitions of the exp_i operators (in terms of calls to operations of the *operator* cluster) are shown in Fig. 6. The operations of the *operator* cluster are defined in the appendix. The derivation of the code of Fig. 6 from the corresponding VAL expressions is straightforward for these lowest-level expressions, and therefore will be assumed as given.

Fig. 5. *If* construct and its semantic tree.

if i + j + k = 0 then i + 1 else j + 1 end

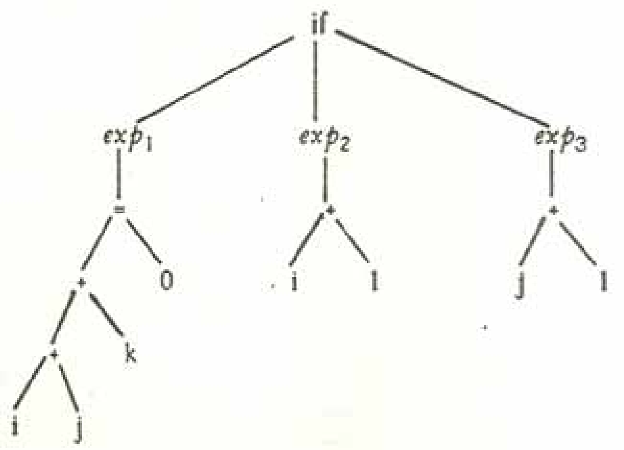
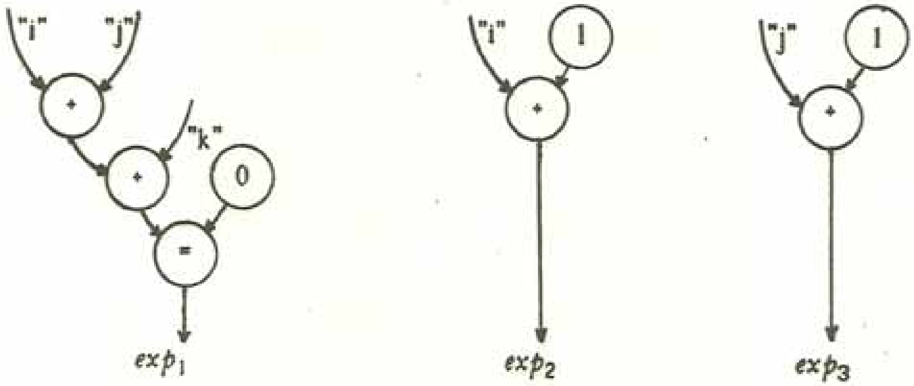


Figure 7 shows a method whereby the graph for the entire *if* construct of Fig. 5 could be constructed from the graph operators for the component graphs of Fig. 6 and additional primitive operators. Note that this method correctly handles the case where the *then* and *else* branches use *several* free variables, not only *one* as in the example. Compiler operations such as type- and (expression) arity- checking, while necessary, are not shown here. In order to perform these checks it will probably be necessary for each node of the semantic tree to have access to a list of free variables used at that level.

Note that acknowledge arcs have not been added to these graphs. Since we are not now considering optimizations on the graphs, the rule of one acknowledge arc for one data arc will be followed. Therefore, in all the examples of this section it can be assumed that whenever the operation *operator$attach(g, op1, outp, op2, inp)* is performed, the corresponding operation *operator$acknowledge(g, op2, 1, op1, 1)* is then performed (except when one of the operators is a graph, then the number of the acknowledge input or output is equal to the number of the graph input or output involved in the data attachment, after

Fig. 6. Graph operator definitions for *if* subexpressions.



```
OPTR = operator                    % OPTR is abbreviation for operator
DESC = array[string]               % Description type
ND: DESC := DESC$new()             % Null Description
```

exp_1:

```
exp1: OPTR := OPTR$create_graph("exp", ND)
exp11: OPTR := OPTR$create_primitive("+", ND)
OPTR$name_input(exp11, 1, "i")
OPTR$name_input(exp11, 2, "j")
exp12: OPTR := OPTR$create_primitive("+", ND)
OPTR$name_input(exp12, 2, "k")
OPTR$attach(exp1, exp11, 1, exp12, 1)
exp13: OPTR := OPTR$create("constant", DESC$["0"])
exp14: OPTR := OPTR$create("=", ND)
OPTR$attach(exp1, exp12, 1, exp14, 1)
OPTR$attach(exp1, exp13, 1, exp14, 2)
OPTR$seal(exp1, DESC$["if-exp", "if1"])
```

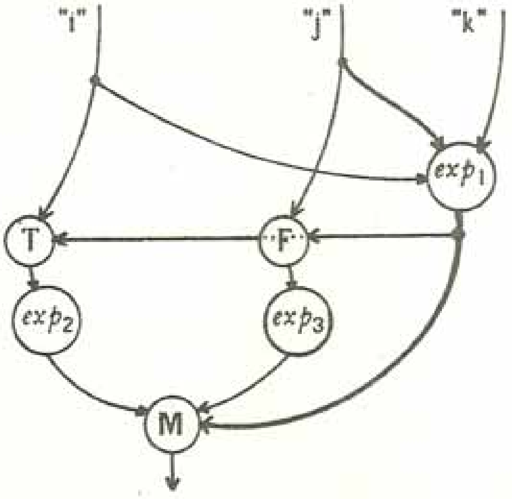
exp_2:

```
exp2: OPTR := OPTR$create_graph("exp", ND)
exp21: OPTR := OPTR$create_primitive("+", ND)
OPTR$name_input(exp21, 1, "i")
exp22: OPTR := OPTR$create_primitive("constant", DESC$["1"])
OPTR$attach(exp2, exp22, 1, exp21, 2)
OPTR$seal(exp2, DESC$["if-then-exp", "if1"])
```

exp_3:

```
exp3: OPTR := OPTR$create_graph("exp", ND)
exp31: OPTR := OPTR$create_primitive("+", ND)
OPTR$name_input(exp31, 1, "j")
exp32: OPTR := OPTR$create_primitive("constant", DESC$["1"])
OPTR$attach(exp3, exp32, 1, exp31, 2)
OPTR$seal(exp3, DESC$["if-else-exp", "if1"])
```

Fig. 7. Graph operator definition for *if* construct.



```
if1: OPTR := OPTR$create_graph("if", DESC$["if1"])
OPTR$include(if1, exp1)
OPTR$include(if1, exp2)
OPTR$include(if1, exp3)

% Construct T gates for free variables of then clause

for var: string in OPTR$input_names(exp2)
do     t: OPTR := OPTR$create_primitive("T-Gate", DESC$[var])
       OPTR$attach(if1, exp1, 1, t, 1)              % t defines var for exp2
       OPTR$attach(if1, t, 1, exp2, OPTR$input_no(exp2, var))
       OPTR$name_input(t, 2, var)        % Now pass name (var) up to input of t.
end

% Construct F gates for free variables of else clause in exactly the same way.
            *       *       *       *
% Merge the results.  Note -- the following code will work only when the "if"
% is not within an iteration body.  The general case will be examined later,
% when the iteration construct is discussed.

next_output: int := 1
for i: int in OPTR$outdegree(exp2)
do     m: OPTR := OPTR$create_primitive("M-Gate", DESC$[var])
       OPTR$attach(if1, exp1, 1, m, 1)
       OPTR$attach(if1, exp2, i, m, 2)
       OPTR$attach(if1, exp3, i, m, 3)
end

OPTR$seal(if1, ND)
```

having been specified as a graph acknowledge input or output by the operation *operator$make_ack_input(output)*. Because this is always the same, it is omitted in these examples to avoid unnecessarily cluttering the code.

3.2 Identifier Binding Expression

In this section a similar translation will be performed for another VAL construct, the **let . . . in . . . end** expression. An attempt will be made to avoid repeating details already developed in the first example.

This construct, shown in Fig. 8, is relatively easy to translate into our *operator* representation. As we are assuming that type checking is done elsewhere, we have omitted the handling of type specification of variables.

Using the ideas developed in the previous example, we can assume we have already translated the <exp_i> expressions into their respective graphs. Now, to generate the graph for the binding expressions <var_i> := <exp_i>, it is only necessary to label the output of each <exp_i> graph with its name <var_i>. The graph for the entire **let** construct is then constructed by feeding the outputs from the binding expressions into the graph for the **in** expression (<*in-exp*>). The graph for Fig. 8 is generated with the code of Fig. 9.

Fig. 8. *Let . . . in . . . end* expression with bound variables.

let <var_1>, <var_2>, ... , <var_k> := <exp_1>, <exp_2>, ... , <exp_k> **in** <*in-exp*> **end**

Fig. 9. Construction of graph for *let* construct.

```
let_exp: OPTR := OPTR$create_graph("let", DESC$["let1"])
for i: int in <set of var-exp pairs of the binding expressions>
do      % Construct binding expression by labelling <$exp_i$>'s output
        OPTR$name_output(<$exp_i$>, 1, <$var_i$>)

        % Include it in the let expression only if it is actually used
        inp: int := OPTR$input_no(<in-exp>, <$var_i$>)

        if inp > 0
        then    OPTR$attach(let_exp, <$exp_i$>, 1, <in-exp>, inp)
        end

end

OPTR$seal(let_exp, ND)
```

3.3 Iteration Expression

In this section the final example VAL construct will be analyzed; this is the **for . . . do . . . iter** iteration construct. Its data flow graph must, of course, be cyclic, and up to now we have constructed only acyclic graphs. We might expect this to cause problems, but in fact we'll find that the translation scheme works rather well even in this situation.

One problem that *is* introduced by this construct is that under Brock's scheme[4] separate translation functions are used for iteration bodies and for code not within an iteration body. This would, of course, be a problem for our bottom-up translation scheme. The reason Brock found it necessary to use two distinct translation functions is that iteration bodies yield two types of values, *iteration* or *I* values (which are recycled through the beginning of the graph) and *return* or *R* values (which are eventually returned as the value of the expression). The iteration-body translation function must, therefore, generate these two sets of outputs, and an additional output named *iter?*, which is a truth value indicating whether the *R* or *I* output set contains valid results. Code that is not contained within the body of an iteration construct yields only *R* values, so it was considered most reasonable to use a separate translater for such code, that only returns *R* values and does not have to generate an *iter?* output for every graph. However, this is not necessary if we make certain assumptions. One translation function will be used for both types of expressions, but when a graph of a subexpression is to be incorporated into that of an expression that generates both *I* and *R* values, the following must be done: if the graph of the subexpression does not have an *iter?* output (as can be tested by *operator$output_no(subexp, "iter?") = 0*), then the translation function must supply a constant *false* operator with name *iter?* for this subexpression.

Whether a graph output is an *I* or *R* output can be determined by noting that only *I* outputs will have names (except for *iter?*, which is not a legal variable identifier). This is because outputs are only named when the operator with the named output is a binding construct for the variable corresponding to the name. This only occurs in VAL in a **let . . . in** or **for . . . iter** construct, and it is not possible for nested binding constructs to "overlap" their binding definitions; that is, if an expression has an *iter?* output it is then an iteration body of a **for . . . iter** loop and cannot also be a subexpression of the right-hand side of a **let** *definition*, unless the entire **for** loop is a subexpression.

Figure 10 shows a **for** loop that returns the sum of the integers from 1 to *n*. We will next translate this into our *operator* representation. As always, we proceed in a bottom-up fashion, noting how the translation function deals with expressions of the form **iter** x,y := . . . before dealing with the enclosing **for** loop. We will not deal with syntax and error checking, in that we will not make any attempt to verify that the **iter** expression is properly contained within a **do . . . end**, or that the **iter** variables are all included in the original list of variables in the **for** expression. It should be clear that this could be imposed on top of the basic structure of the translater described here.

The graphs generated for the lowest level subexpressions of this construct are straightforward applications of the algorithms already demonstrated. We assume, therefore, that these translations have already been made, and that the various subgraphs we need are as shown in Fig. 11. Note the use of the identity operator ① for expressions of the form *<variable>*. This identity operator is just a "placeholder" in that it will not become an actual data flow machine instruction. (Identity operators that *do* become machine instructions, called *buffers*, do have their uses, as will be discussed in a later section.)

To create the graph for an expression of the form "iter <exp>" we need merely add a constant **true** output labelled *iter?* to the graph constructed for <exp>. Thus, Fig. 12 shows code to construct the graph *iterexp* for the expression "iter i,s := i+1, s+i".

Fig. 10. Simple *for* loop.

```
for  i,s := 1,0
do          if i > n
            then    s
            else    iter i,s := i+1, s+i
            end
end
```

Fig. 11. Graphs for subexpressions of Fig. 10.

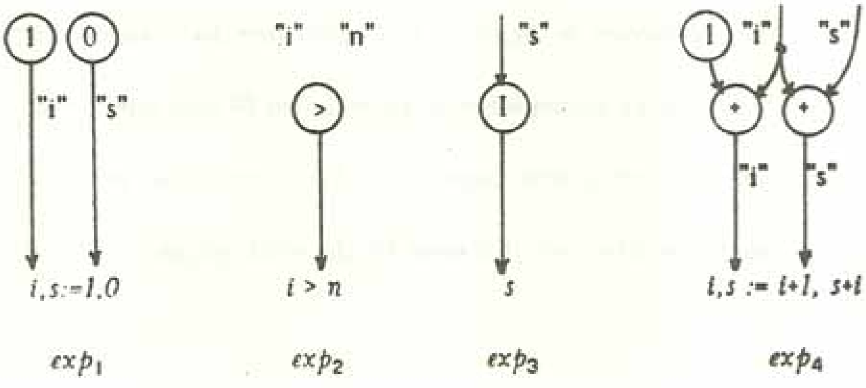


Fig. 12. Construction of graph for *iter* expression.

```
iterexp: OPTR := OPTR$create_graph("iterI", ND)
OPTR$absorb(iterexp, exp4)
OPTR$include(iterexp, OPTR$name_output(OPTR$create_primitive
                                    ("constant", DESC$[true]), 1, "iter?"))
```

Now we must use this graph to build the graph for the conditional expression containing it. This requires a few additions to the construction process of the first example. First, we check whether either of the *then* clause or the *else* clause has an *iter?* output. If both clauses did not have such an output, then the process of the first example could be used. In this example, however, the *else* clause has an *iter?* output, but the *then* clause does not. We must therefore use a constant **false** operator to produce an *iter?* output for the *then* clause.

In the most general case of **if** construct the *then* clause and the *else* clause may each have an *iter?* output, I (named) outputs, and R (unnamed) outputs. The *iter?* output of the entire **if** graph will then be selected from the *then* clause *iter?* or the *else* clause *iter?*, depending on the value of the conditional expression, and two separate **M** (merge) gates must be used to independently merge the I and R outputs. Further, the two clauses must generate the same number of R outputs *if both generate such outputs*, whereas the number (and names) of I outputs can differ. Any I outputs missing from one clause but supplied by the other must be represented in the merge gate input from that clause by the "old value" of that variable. This can be accomplished by naming that **M** gate input; this input will then be merged with other component inputs with the same name and become the input corresponding to the variable with that name for the whole graph.

There are several necessary preconditions, as mentioned above, for the subgraphs of an if expression to be valid. These are summarized as follows.

1. Either both clauses generate the same number of *R* outputs, or at least one of them has 0 *R* outputs.

2. If either clause has an *iter?* output it must have more than 0 *I* outputs.

3. If either clause has more than 0 *I* outputs it must have an *iter?* output.

4. If either clause does *not* have an *iter?* output it must have more than 0 *R* outputs (otherwise it would have no outputs at all).

All of these preconditions must be verified before the code presented in this section is invoked; I have omitted the details of this error checking to refrain from obscuring the code.

In this case the *then* clause has an *R* output, but no *I* outputs, and the *else* clause has *I* outputs, but no *R* outputs, so the merge gates disappear, as will be seen. To detect this case and still be able to handle the general case, the code of Fig. 13 is used.

Note the use of the IC gate to generate the *iter?* output for the whole graph. An IC gate selects the *iter?* output from either the *then* or *else* clause *iter?* output, depending on the control input, and also has two other outputs: a control for the *I* merge gates (output number two) and a control output for the *R* merge gates (output number three). Since neither merge gates are present in this case the corresponding IC control outputs are connected to sinks so that they do not become graph outputs. The final result of all this is the graph for the entire if expression, shown in Fig. 14.

The last thing to do is to construct the graph for the entire for loop. This is in fact quite straightforward. The graph for the *iteration* subgraph is the *if2* graph just generated. We need merely construct a graph that feeds *if2* the proper values for its named inputs,

Fig. 13. Construction of graph for *if* expression -- general case.

```
%       Then_I is number of I outputs from then clause, then_R number of R outputs

then_I : int := OPTR$named_outdegree(exp3) - 1             % Dont count iter? output
then_R : int := OPTR$outdegree(exp3) - then_I - 1
if then_I < 0
then    then_I := 0
end

%       Define else_R and else_I the same way

else_I : int := OPTR$named_outdegree(iterexp) - 1
else_R : int := OPTR$outdegree(iterexp) - else_I - 1
if else_I < 0
then    else_I := 0
end

%       Find iter? outputs, if any
then_iter : int := OPTR$output_no(exp3, "iter?")
else_iter : int := OPTR$output_no(iterexp, "iter?")

%       --At this point the preconditions should be checked and any errors signalled,
%         then create if2 graph and construct T and F gates as in the first example--

%       Now construct the iter? output for the whole graph if one is needed

ic: OPTR        % Ic will generate graph iter? and M gate control outputs (if needed)
if else_iter > 0 | then_iter > 0
then    ic := OPTR$create_primitive("IC-Gate", DESC$["if2"])
        OPTR$attach(if2, exp2, 1, ic, 1)    % Conditional exp controls ic gate
        if then_iter > 0                    % If exp2 true, take then iter?, or constant
        then    OPTR$attach(if2, exp3, then_iter, ic, 2)    % false if no then iter?
        else    OPTR$attach(if2, OPTR$create_primitive("constant", DESC$["false"]),
                                                                        1, ic, 2)
        end

        if else_iter > 0                    % Same as above for else clause
        then    OPTR$attach(if2, iterexp, else_iter, ic, 3)
        else    OPTR$attach(if2, OPTR$create_primitive("constant", DESC$["false"]),
                                                                        1, ic, 3)
        end

        OPTR$name_output(ic, 1, "iter?")
```

Fig. 13 (continued)

```
        % If both clauses have an iter? output . . .
        if then_iter > 0  &  else_iter > 0
        then    % . . . merge I results from both clauses
                % then clause
                for var: string in OPTR$output_names(exp3)
                do      % ignore iter? outputs
                        if var = "iter?" then continue end

                        m: OPTR := OPTR$create_primitive("M-Gate", DESC$[var])
                        OPTR$attach(if2, ic, 2, m, 1)
                        OPTR$attach(if2, exp3, OPTR$output_no(exp3, var), m, 2)

                        k: int := OPTR$output_no(iterexp, var)
                        if k > 0 % Merge with else output or old value
                        then    OPTR$attach(if2, iterexp, k, m, 3)
                        else    OPTR$name_input(m, 3, var)
                        end
                        OPTR$name_output(m, 1, var)
                end

                % else clause
                for var: string in OPTR$output_names(iterexp)
                do      k: int := OPTR$output_no(iterexp, var)
                        if  array[inconn]$size(OPTR$dests(iterexp, k)) = 0
                        then    % Output k has not been connected to an M gate,
                                % so then clause has no var output.
                                m: OPTR := OPTR$create_primitive("M-Gate",
                                                                 DESC$[var])
                                OPTR$attach(if2, ic, 2, m, 1)
                                OPTR$attach(if2, iterexp, k, m, 3)
                                % use old value for then part of merge
                                OPTR$name_input(m, 2, var)
                                OPTR$name_output(m, 1, var)
                        end
                end

        else    % If only the then clause or the else clause has any I
                % outputs, they will become the named outputs of the graph when
                % it is sealed, so sink the ic merge control output
                OPTR$attach(if2, ic, 2, OPTR$create_primitive("sink", ND), 1)
        end
else    % No iter? outputs at all, so no IC gate
end
```

Fig. 13 (concluded)

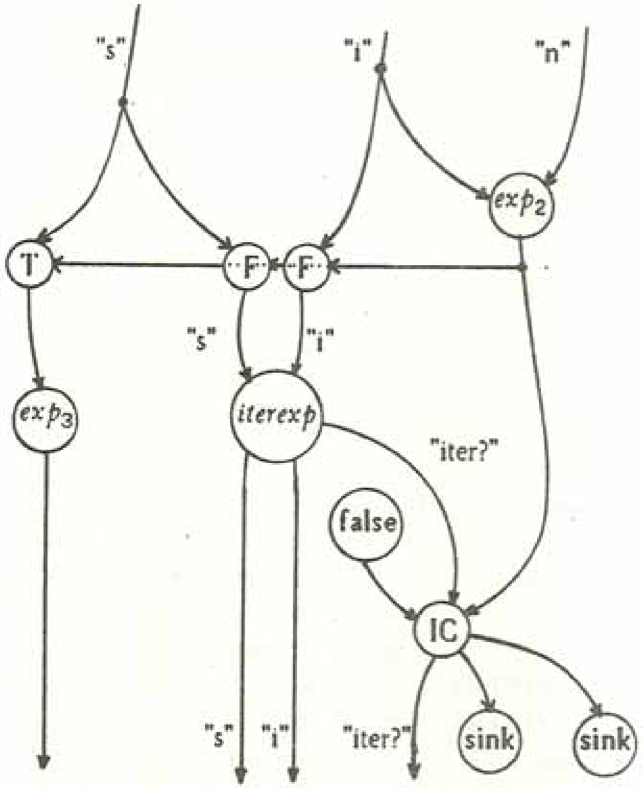
```
%       Lastly, merge the R outputs, if any

if else_R > 0 & then_R > 0
then    % Preconditions demand that then_R = else_R, so iterate over each
        % clause's unnamed outputs in order, merging them.
        next_t: int := 1
        next_e: int := 1
        for i: int in int$from_to(1, then_R)
        do      % Find next unnamed then and else outputs . . .
                while OPTR$output_name(exp3, next_t) ~= ""
                do      next_t := next_t + 1
                end

                while OPTR$output_name(iterexp, next_e) ~= ""
                do      next_e := next_e + 1
                end

                % . . . and merge them
                m: OPTR := OPTR$create_primitive("M-Gate", DESC$["R" ||
                                                    int$unparse(i), "if2"])
                OPTR$attach(if2, ic, 3, m, 1)          % ic gate controls m
                OPTR$attach(if2, exp3, next_t, m, 2)          % then clause
                OPTR$attach(if2, iterexp, next_e, m, 3)       % else clause
        end
elseif then_iter > 0 | else_iter > 0          % i.e. if there is an IC gate
then    % Any unnamed outputs from either clause alone will become the unnamed
        % outputs from the graph when sealed, so sink the ic merge control output
        OPTR$attach(if2, ic, 3, OPTR$create_primitive("sink", ND), 1)
end
OPTR$seal(if2, ND)
```

Fig. 14. Graph for *if* expression with *iter* subexpression.



that is the initial bindings of the iteration variables defined by the graph *exp*$_1$ on the first iteration, followed by the *I* results from *if2* on subsequent iterations, until *if2*'s *iter?* output is **false**.

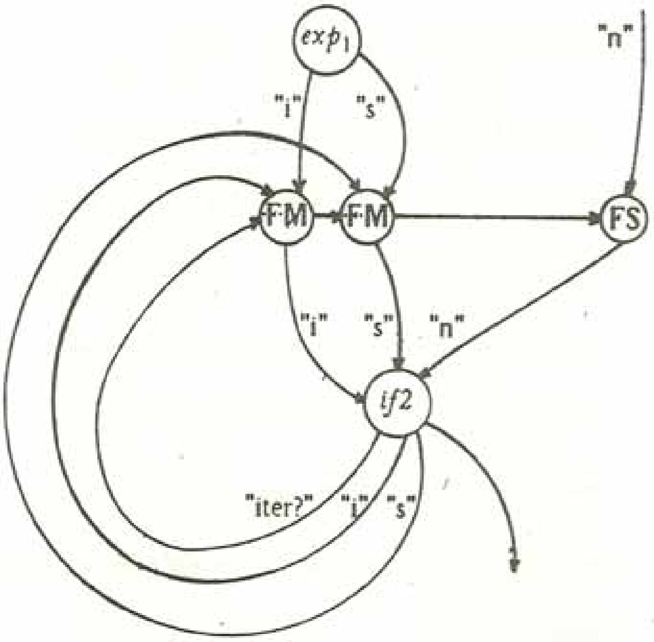
To do this we need two new types of special gates, **FM** and **FS** gates. The **FM** gate is like the **M** gate except that it has an initial **false** token built into it. The **FS** gate gives us the ability to store a data token and continually output it until its control gate goes **false**. This is used for the inputs to *if2* that are *not* iteration variables, because the *same* values must be used for those inputs each iteration. The code to construct the graph is shown in Fig. 15. Note that, as usual, I am omitting most error checking. The resulting graph is shown in Fig. 16.

Fig. 15. Construction of graph for *for* loop.

```
forl: OPTR := OPTR$create_graph("for", DESC$["forl"])
OPTR$include(forl, expl)
OPTR$include(forl, if2)
iter_out: int := OPTR$output_no(if2, "iter?")
%       Merge I results from iteration subgraph with initial bindings
if iter_out > 0
then    for var: string in OPTR$output_names(expl)
        do      % For each iteration variable used by if2, check to see
                % if it is ever reset by an iter expression in if2;
                % if so, then merge it with its initial defn from expl;
                % if not, then it enters if2 via an FS gate
                inp: int := OPTR$input_no(if2, var)
                outp: int := OPTR$output_no(if2, var)
                if inp > 0  &  outp > 0    % i.e. var is used and reset
                then    fm: OPTR := OPTR$create_primitive("FM-Gate", DESC$[var])
                        OPTR$attach(forl, if2, iter_out, fm, 1)
                        OPTR$attach(forl, if2, outp, fm, 2)
                        OPTR$attach(forl, expl, OPTR$output_no(expl, var), fm, 3)
                        OPTR$attach(forl, fm, 1, if2, inp)
                elseif inp > 0               % used but not reset
                then    fs: OPTR := OPTR$create_primitive("FS-Gate", DESC$[var])
                        OPTR$attach(forl, if2, iter_out, fs, 1)
                        OPTR$attach(forl, expl, OPTR$output_no(expl, var), fs, 2)
                        OPTR$attach(forl, fs, 1, if2, inp)
                else    % if never used, sink it; should probably report an error
                        OPTR$attach(forl, expl, OPTR$output_no(expl, var),
                                                   OPTR$create_primitive("sink", ND), 1)
                end
        end

        % Now pass any other unreset inputs to if2 through FS gates
        for var: string in OPTR$input_names(if2)
        do      inp: int := OPTR$input_no(if2, var)
                if OPTR$null_source(if2, inp)
                then    % still unconnected, so needs an FS gate
                        fs: OPTR := OPTR$create_primitive("FS-Gate", DESC$[var])
                        OPTR$attach(forl, if2, iter_out, fs, 1)
                        OPTR$attach(forl, fs, 1, if2, inp)
                        OPTR$name_input(fs, 2, var)  % FS input will be graph input
                end
        end
else    % the iteration subgraph does not contain an iter expression
        % and should be treated simply as a let expression.
end
OPTR$seal(forl, ND)
```

Fig. 16. Graph for *for* loop.



3.4 Other VAL Constructs

There are some important VAL constructs that have not been discussed in the previous sections because a definite form of data flow graph has not been chosen for their representation. The two most important such constructs, the **forall** expression and procedure invocation, are discussed briefly in this section. Other VAL constructs, such as the **tagcase** expression, can clearly be implemented as modifications of VAL constructs already discussed. (For example, **tagcase** is a multi-branch conditional expression, which can be translated into the same type of graph as produced for a set of nested **if** expressions.

3.4.1 The Forall Expression

The **forall** expression, shown in its basic form in Fig. 17, provides explicit high-level parallelism in **VAL**. There are two basic forms of *body*: "**construct** *<expression>*", and "**eval** *<operation> <expression>*". In the first case the result of the **forall** expression is an array. Each component of the array is set to the value of the expression following the keyword **construct**, evaluated in an environment in which the **forall** index (*<identifier>*) is bound to an integer between *<integer-exp*$_1$*>* and *<integer-exp*$_2$*>* (inclusive), and the temporary names are bound to their definitions. The low and high bounds of the array are *<integer-exp*$_1$*>* and *<integer-exp*$_2$*>*, and the elements are ordered according to the value of the index used in evaluating them.

In the second form of **forall** body a single value is returned which is constructed from the values of the expression following the keyword **eval** with the index and temporary names bound as in the first case. The result value is obtained by applying *<operation>* to each of these evaluations of the expression. There is a limited set of valid *<operation>*s (*plus*, *times*, *min*, *max*, *and*, and *or*). Figure 18 shows an example of each type of **forall** body. The first evaluates to an array of integers whose indexes are 1 to 5 and whose elements are 1, 4, 9, 16, and 25. The second evaluates to an integer which is the sum of the first *n* squares.

Fig. 17. Basic *forall* expression.

```
forall <identifier> in [<integer-exp1> , <integer-exp2>]
  {<declarations and definitions of temporary names>}
  <body>
end
```

Fig. 18. *Forall* **expression -- two types.**

```
forall i in [1, 5]
   construct i*i
end
```

```
forall j in [1, n]
   eval plus j*j
 end
```

In the most general case there can exist several **eval** and **construct** clauses in the *same* **forall** expression, yielding a multi-expression. Each expression to be **constructed** or **evaled** should be independent of the others, and each evaluation of the same **eval** or **construct** expression should also be independent, so that each "iteration" of the loop can in fact be evaluated simultaneously rather than iteratively.

This expression clearly presents a number of problems for our translation algorithm. First, since the range of the index is not necessarily known at translation time, a data flow graph allowing the maximum amount of parallelism would have to be dynamic; when new values for the index range arrive, the number of branches of the graph would have to change, which clearly is impractical.

One method of dealing with this problem is to require that the index high and low bounds be known at translation time, that is, they must be constant expressions. The translater could then decide whether to generate a graph in which each index value is computed and used in parallel, or to transform the **forall** into a **for** expression containing a smaller **forall**. In this way the total range of the index would be divided into subranges such that the subranges are invoked iteratively but within each subrange each branch is evaluated in parallel.

An alternative approach is more general but much more difficult to implement. With this method the index range need not be known at translation time. What the translater could then do is to divide the range into subranges as above, such that the size of each subrange is fixed, and the number of iterations is variable. Each branch of the graph to evaluate a given subrange would then have to check that the index is within the range of the **forall** before evaluation, which leads to a very complicated graph in which it is unclear whether the **forall** expression has really gained much efficiency by introducing parallelism (since each parallel branch must now do careful checking before deciding whether to evaluate or not, and since joining the results of the evaluations of each branch is made much more complex).

Another alternative is described in [12]. This method implements **foralls** as a form of recursive procedure. This and other possibilities are under consideration, but no approach has been clearly decided upon, and each seems to have its problems. Because of these difficulties, a translation of **forall** constructs into an *operator* representation is not proposed in this thesis, and further analysis of the expression is required.

3.4.2 Procedure Invocation

Like almost all modern programming languages, **VAL** allows a block of code to be written as a procedure that can then be invoked from several different points within other procedures; however, an implementation of procedure invocation for data flow machines has not been decided upon. This is a complex issue, well beyond the scope of this thesis, but it is necessary to discuss it at least briefly in terms of the data flow program representation scheme proposed here.

If procedures are restricted to being nonrecursive, then they can be implemented easily by simply copying the graph for the procedure at each point of invocation. That is, a procedure is then a graph operator with named inputs corresponding to the names of its formal parameters. The point of invocation can be considered to be equivalent to a "**let . . .**

in" expression with the formal parameters being the identifiers to be bound, the actual parameters being their values, and the graph for the procedure being the "in . . ." expression subgraph. This implementation introduces no new difficulties in the use of the representation scheme of this thesis, but is not completely satisfactory for a number of reasons. First, of course, is the lack of recursion. There is also the issue of the space (i.e. number of instruction cells) taken up by multiple copies of the procedure graph.

If procedures are allowed to be recursive (as is clearly desirable), then such a straightforward approach cannot be used. One approach to implementing recursive procedures is detailed in [12]. This method involves the use of procedure *activation records* (similar to those used in the conventional stack-based recursive procedure implementation) implemented as data structures, and an *execution controller* which creates the desired instruction packet upon the arrival of all operands to a particular instruction of any activation record of a procedure.

It is hard to evaluate the impact of the method described above on the *operator* representation scheme, since it is a fairly abstract proposal; however, it seems reasonable to believe that the *operator* representation scheme proposed in this thesis is as capable of being extended sufficiently to handle this approach as any other reasonable implementation of data flow programs. A detailed analysis of this is clearly beyond the present paper.

4. Transformations and Optimizations

Two basic types of optimizing transformations specific to data flow graphs are described in [5] and [11]. The first transformation is aimed at decreasing the number of tokens sent around the system by eliminating unnecessary acknowledge arcs. The second transformation increases throughput by allowing pipelined execution of sections of the graph.

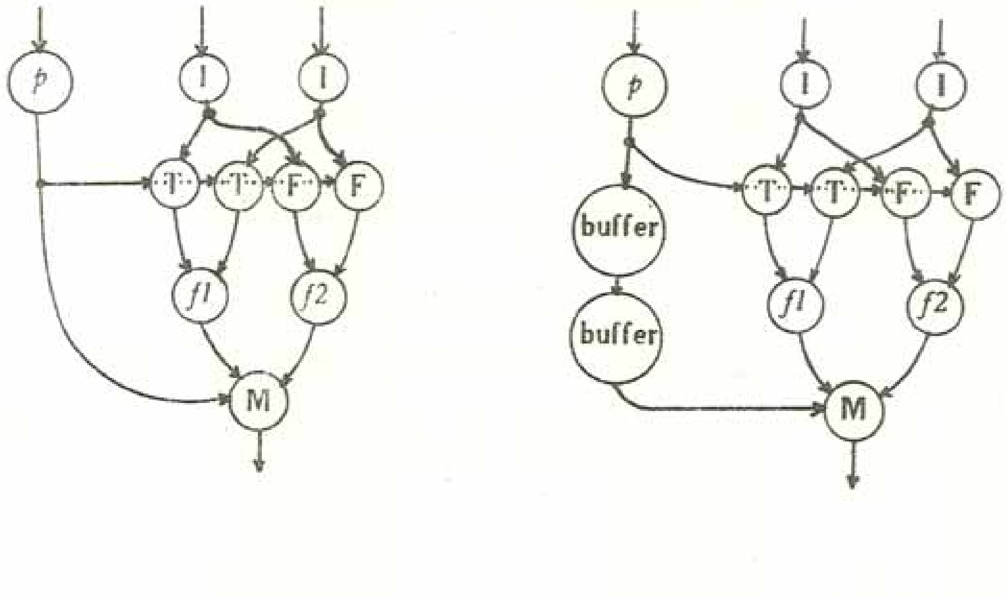
An example of the first type of transformation, taken from [5], arises in the graph for a **for** loop, such as that of Fig. 16. The *iter?* output of the subgraph for the iteration body (*if2*) is, of course, dependent on the inputs to the body ("i", "s", and "n"); however, these values are in turn passed to the iteration body only on the receipt of the old *iter?* value at their controlling **FS** and **FM** gates. Thus the old *iter?* value is guaranteed to be consumed before a new one can be generated, and the acknowledge arc along this path (from *if2* to the **FS** and **FM** gates) is unnecessary.

There are other similar cases in which acknowledge arcs can be removed on determination that they are unneeded to ensure safe execution of the program. The *operator* cluster provides the operation *detach_ack* to remove acknowledge arcs that have already been attached. Making such a transformation on the *operator* representation is therefore quite simple.

The other basic type of data flow graph optimization is illustrated in Fig. 19 (also taken from [5]). Adding the buffers to the first graph enables more overlapped execution to take place. This is because the control operation p cannot fire until all tokens on its output

---

Fig. 19. Adding pipelining to a data flow graph.



arcs have been consumed. In the first graph this requires the merge gate (**M**) to fire, generating the graph output for one set of input values, before a new set of input values can be gated into *f1* or *f2*. In the second graph *p* can fire again before the **M** gate consumes the token generated by the last firing of *p*, since *p*'s output arc (to the first buffer) is empty. This type of transformation can be accomplished within the *operator* representation by using the operation *detach* to first disconnect the attachment between *p* and **M**, and then using *include* and *attach* to make the new attachments. (See the appendix for complete descriptons of these operations, and examples of their use.)

Other, more conventional types of transformations, such as code movement, can be done in the same way, removing operators and inserting new ones, or detach connections and reattaching them in new ways. The operations of the *operator* cluster appear to be sufficient for any such manipulation.

All such transformations could be made by directly modifying the original graph, or in a more applicative way by first *copying* the original graph and then making the changes in the copy. To obtain an unsealed copy of a sealed graph, the graph can first be *copied* (which returns a sealed copy), and then *absorbed* into an empty graph. In order to make the attachments in the copy it is necessary to be able to obtain a reference to the components of the copy that correspond to specific components of the original. This can be done by use of the operations *get_id* and *fetch* (that is, if *g2* is a copy of *g1*, and *c1* is a component of *g1*, then the component of *g2* corresponding to *c1* is that component with the same *id* as *c1*'s, *g2*[*c1*.id]). This method is illustrated in the last section of the appendix.

5. Summary of Results and Conclusions

This thesis has presented a CLU implementation of a scheme for the representation of data flow programs. It involves the definition of an abstract data type called *operator*. An *operator* represents either a (primitive) node of a data flow graph or an interconnected set of *operators* which together form a *graph*.

An *operator* which is a graph can be considered to be similar to a primitive operator of a data flow graph, in that it can be connected to other operators at a fixed set of inputs and outputs (for both data and acknowledge arcs). The inputs and outputs of a graph corresond to inputs and outputs of its components in a natural way, so that a graph *operator* connected to other *operators* acts as an "abbreviation" for the larger graph that would result from expanding the graph *operator* into its interconnected components, and attaching those components to the *operators* attached at the corresponding graph inputs and outputs.

The operations of the *operator* cluster presented here are sufficient and convenient for the construction of such graphs, and the resulting graphs provide a convenient representation of data flow programs. Transformations of a type known to improve the execution performance of data flow programs can also be conveniently made using the operations of this cluster.

The translations of most data flow source language expressions written in the language VAL into their *operator* representations can be done by following the generalized scheme presented in this thesis, which involves a bottom-up approach, translating the subexpressions of a VAL expression into graphs which will then become subgraphs of the translation of the entire expression. This approach is consistent with, and can parallel, a bottom-up parse of VAL programs, and is therefore an acceptable method for use in a VAL compiler.

Translation schemes have not been presented for certain VAL constructs for which an adequate data flow graph representation has not yet been chosen, but it is reasonable to believe that this representation is at least as powerful as any other reasonable representation in its ability to handle such constructs. A more complete analysis of these remaining constructs should be undertaken to determine the truth of this conjecture.

No attempt has been made to implement these programs in a high-performance "production sytem", and thus their speed of operation could undoubtedly be greatly improved with some reprogramming.

It would also be interesting to use the structure of this representation as a basis for a data flow simulator; this would involve the addition of data and acknowledge *tokens* to the graphs, an indication of operator *enablement*, and a step-by-step updating of the state of each operator.

Appendix I - Implementation

1.1 The Operator Cluster

The following is the CLU code which implements the *operator* cluster. Each operation includes a header which describes its behavior in terms of its interface to the "outside world" (i.e. any programs using the *operator* cluster). These headers constitute the behavioral specifications of the cluster.

```
% NEEDS table.specs, acat.specs  to compile

operator = cluster is  absorb, ack_dests, ack_indegree, ack_outdegree, acknowledge,
                       attach, components, copy, create_graph, create_primitive, dests,
                       detach, detach_ack, equal, fetch, free, get_acks_expected,
                       get_acks_received, get_description, get_id, get_opname,
                       get_owner, in_suback, include, indegree, input_name,
                       input_names, input_no, is_graph, is_primitive, is_sealed,
                       make_ack_input, make_ack_output, name_input, name_output,
                       named_indegree, named_outdegree, null_source, out_suback,
                       outdegree, output_name, output_names, output_no, remove,
                       seal, set_acks_expected, set_acks_received, source, subinput,
                       suboutput, write

%       Operators are the nodes of a data flow graph.  A data flow graph is itself
%       an operator, called a GRAPH OPERATOR.  Within a graph, an operator can
%       be ATTACHed to other operators at a specific input or output.  An operator
%       can have only one source operator for each of its distinct inputs, but
%       can have many destination operators for each of its distinct outputs.
%       Acknowledge attachments can also be made between acknowledge inputs and
%       acknowledge outputs; acknowledge inputs can have more than one source.

% Abbreviations:
        OPTR = operator
        DESC = array[string]            % Description data type
        al = array[link]                % Links are defined below
        aic = array[inconn]             % Inconns and outconns are explained
        aoc = array[outconn]            %       below
        tbl = table[string, int]        % Used to remember input/output names
        lkp = string$equal              % TABLE lookup operation
        row = record[opname: string, inputs, outputs, ack_inputs, ack_outputs: int]
                % Row is used in reading from the primitive operator table

  rep = record[kind_of_op: op_kind, opname: string, inputs, outputs: al,
                      owned: owner, description: DESC, id: int,
                      in_names, out_names: tbl, ack_to: array[aic],
                      ack_inputs: int]

  op_kind = oneof[primitive: prim_op,  graph: graph_op]

  prim_op = record[acks_expected: int, init_acks_received: int]

  graph_op = record[components: array[OPTR], next_id: int,
                      subinputs: al, suboutputs: aoc, in_subacks: aic,
                      out_subacks: aoc, sealed: bool]

  owner = oneof[free: null, owned_by: OPTR]
```

```
%       Links are the data arcs of a data flow graph.  Each link has at most
%       one SOURCE operator, and an arbitrary number of DEST operators.  Data
%       flow is from source to dests.  An unconnected (null) link has empty
%       source and dests; an unnamed link has null string as name.

 link = record[source: aoc, dests: aic, name: string]


%       An inconn (input connection) "ic" is a pair representing an operator
%       (ic.op) and a specific input number of that operator (ic.inp).
%       Similarly an outconn is an output connection.  Inconns and outconns
%       are used to identify attachments between operators.

 inconn = record[op: OPTR, inp: int]
 outconn = record[op: OPTR, outp: int]


% Cluster Operations:


 absorb = proc(g: OPTR, op: OPTR) returns(OPTR) signals(already_owned,
                              cant_include_self, not_graph, sealed)

        % If op is a primitive operator then absorb acts the same as include;
        % if op is a free graph operator (sealed or unsealed) then each
        % component of op is included in g (with the respective attachments),
        % rather than the graph operator op; the components of op will then
        % become components of g instead, and op will become an empty (and
        % unsealed) graph.  THIS IS A SIDE EFFECT, and care should be taken
        % with this operation; in particular, if op is a graph then there
        % should be no other pointer to that graph when this program is called.
        % Note that any graph acknowledge inputs or outputs of op will become
        % new graph acknowledge inputs or outputs of g when op's components are
        % absorbed into g.  The signals possible here are identical to those
        % for include, except that there is no signal for op being unsealed.
        % The argument g is returned.

        tagcase down(op).kind_of_op
        tag primitive:
                return(OPTR$include(g, op))       % If op not graph, just include
```

```
tag graph(gop: graph_op):                % otherwise include its components
        if ~OPTR$free(op)
        then    signal already_owned

        elseif op = g
        then    signal cant_include_self

        elseif ~OPTR$is_graph(g)
        then    signal not_graph

        elseif OPTR$is_sealed(g)
        then    signal sealed
        end

        g2: graph_op := op_kind$value_graph(down(g).kind_of_op)

        for c: OPTR in OPTR$components(op)
        do      % Assign c's owner and id
                down(c).owned := owner$make_owned_by(g)
                down(c).id := g2.next_id

                % Add c to g's components
                array[OPTR]$addh(g2.components, c)
                g2.next_id := g2.next_id + 1
        end

        % Add op's subacks to g's subacks

        g2.in_subacks := array_cat[inconn](g2.in_subacks, gop.in_subacks)

        for oc: outconn in aoc$elements(gop.out_subacks)
        do      aoc$addh(g2.out_subacks, oc)
                array[aic]$addh(down(g).ack_to, aic$new())
        end

        % Remove op's components  --NOTE THIS SIDE EFFECT--

        dop: rep := down(op)
        dop.inputs := al$new()
        dop.outputs := al$new()
        dop.in_names := tbl$create()
        dop.out_names := tbl$create()

        gop.components := array[OPTR]$new()
        gop.next_id := 1
        gop.subinputs := al$new()
        gop.suboutputs := aoc$new()
        gop.in_subacks := aic$new()
```

```
                gop.out_subacks := aoc$new()
                gop.sealed := false
                return(g)
         end
 end absorb


ack_dests = proc(op: OPTR, ack_no: int)  returns(aic)
                                          signals(acks_range, free_operator)

        % Returns an array whose elements are inconns whose op components
        % are the operators that receive op's number ack_no acknowledge
        % output, and whose inp components are the acknowledge inputs of
        % those operators that receive the acknowledge from op.  If op has
        % no number ack_no acknowledge output, then "acks_range" is signalled;
        % if op is free, then "free_operator" is signalled.

        if OPTR$free(op)
        then    signal free_operator
        end

        ai: aic := down(op).ack_to[ack_no]
           except
                when bounds:    signal acks_range
           end

        result: aic := aic$new()
        for ic: inconn in aic$elements(ai)
        do      aic$addh(result, inconn$copy1(ic))
        end

        return(result)
end ack_dests


ack_indegree = proc(op: OPTR) returns(int) signals(unsealed)

        % Returns the number of acknowledge inputs defined for op.  This information
        % is in the operator table for primitive operators, and for graphs
        % depends on the number of MAKE_ACK_INPUT operations on op.  Signals
        % if op is unsealed.

        if ~OPTR$is_sealed(op)
        then    signal unsealed
        end

        return(down(op).ack_inputs)
end ack_indegree
```

```
ack_outdegree = proc(op: OPTR) returns(int) signals(unsealed)

        % Returns the number of acknowlede outputs defined for op.  If op is
        % primitive this number is fixed at create time (the information is in
        % the primitive operator table); if op is a graph, this number is equal
        % to the number of MAKE_ACK_OUTPUT operations performed on the graph.

        if ~OPTR$is_sealed(op)
        then    signal unsealed
        end

        return(array[aic]$size(down(op).ack_to))
end ack_outdegree


acknowledge = proc(g: OPTR, ops: OPTR, ack_no: int, opr: OPTR, ack_inp: int)
                                returns(OPTR)
                                signals(already_owned, cant_include_self,
                                        cant_include_unsealed_graph,
                                        in_range, out_range, not_graph,
                                        sealed)

        % Attaches the number ack_no acknowledge arc from ops (the sending op)
        % to the number ack_inp acknowledge input of opr (the receiving op)
        % within graph g.  If either of ops and opr is free it is first included
        % in g (with ops included before opr), so all signals of include can
        % occur.  Also signals "in_range" if opr has no number ack_inp acknowledge
        % input, and "out_range" if ops has no number ack_no output.  The
        % argument g is returned.

        if (~OPTR$free(ops)  cand  ops.owner ~= g)
          | (~OPTR$free(opr)  cand  opr.owner ~= g)
        then    signal already_owned

        elseif ~OPTR$is_graph(g)
        then    signal not_graph

        elseif OPTR$is_sealed(g)
        then    signal sealed

        elseif OPTR$ack_indegree(opr) < ack_inp
        then    signal in_range

        elseif OPTR$ack_outdegree(ops) < ack_no
        then    signal out_range
        end
```

```
        if OPTR$free(ops)
        then    OPTR$include(g, ops)
        end

        if OPTR$free(opr)
        then    OPTR$include(g, opr)
        end

        aic$addh(down(ops).ack_to[ack_no], inconn${op: opr, inp: ack_inp})

        % Increment acks expected by primitive operator that receives
        % the new acknowledge arc.

        p: prim_op := find_receiver(opr, ack_inp)
        p.acks_expected := p.acks_expected + 1

        return(g)
end acknowledge


attach = proc(g: OPTR, op1: OPTR, outp: int, op2: OPTR, inp: int) returns(OPTR)
                                   signals(already_owned, cant_include_self,
                                           cant_include_unsealed_graph, not_graph,
                                           sealed, inputs_range, outputs_range,
                                           already_attached, name_conflict)

        % Attaches op1's outpth distinct output to op2's inpth distinct input,
        % within graph g.  The graph g must be free and unsealed.  If either of
        % op1 and op2 is free, it is first included in g (with op1 included
        % before op2), and therefore all signals of include can occur.  Other
        % signals: "already_attached" if op2's inpth input has a source;
        % "inputs_range" if inp is outside the range of valid inputs for op2;
        % "outputs_range" if outp is outside the range of valid outputs for op1;
        % "name_conflict" if the input and output arcs in the connection have
        % different names.  (After the attachment is made both the input and
        % output arc involved in the attachment will have the same name, even
        % if previously only one of them was named.  The argument g is returned.

        if (~OPTR$free(op1)  cand  op1.owner ~= g)
           | (~OPTR$free(op2)  cand  op2.owner ~= g)
        then    signal already_owned

        elseif ~OPTR$is_graph(g)
        then    signal not_graph

        elseif OPTR$is_sealed(g)
        then    signal sealed
```

```
        elseif OPTR$indegree(op2) < inp
        then    signal inputs_range

        elseif OPTR$outdegree(op1) < outp
        then    signal outputs_range

        elseif ~OPTR$null_source(op2, inp)
        then    % Already an attachment there
                signal already_attached
        end

        r1: rep := down(op1)
        r2: rep := down(op2)
        l1: link := r1.outputs[outp]
        l2: link := r2.inputs[inp]

        if l2.name ~= ""
        then    if l1.name ~= ""
                then    if l1.name ~= l2.name
                        then    signal name_conflict
                        end

                else    % Pass l2's name to l1
                        l1.name := l2.name
                        tbl$insert(l2.name, outp, r1.out_names)
                end

        elseif l1.name ~= ""
        then    tbl$insert(l1.name, inp, r2.in_names)
        end

        if OPTR$free(op1)
        then    OPTR$include(g, op1)
        end

        if OPTR$free(op2)
        then    OPTR$include(g, op2)
        end

        if null_link(l1)
        then    aoc$addh(l1.source, outconn${op: op1, outp: outp})
        end

        aic$addh(l1.dests, inconn${op: op2, inp: inp})
        r2.inputs[inp] := l1
        return(g)
    end attach
```

```
components = iter(g: OPTR)  yields(OPTR) signals(not_graph, unsealed)

        % An iterator over all the components of graph g.  If g is not a graph
        % then "not_graph" is signalled; "unsealed" is signalled if g is
        % unsealed.  The components are yielded in the order in which they
        % were included in g, that is by order of increasing id.

        if ~OPTR$is_sealed(g)
        then    signal unsealed
        end

        gop: graph_op := op_kind$value_graph(down(g).kind_of_op)
            except
                when wrong_tag:         signal not_graph
            end

        for c: OPTR in array[OPTR]$elements(gop.components)
        do      yield(c)
        end

        return
end components


copy = proc(op: cvt, descr: DESC) returns(OPTR) signals(unsealed)

        % Returns a free, sealed copy of the operator op, but with description as
        % given, and with 0 acknowledges expected and 0 acknowledges initially
        % received.  The set of input and output names of op is also copied in
        % the returned operator.  If op is a graph then each component of op is
        % (recursively) copied, and attached in the returned operator in the same
        % way as in op.  The description of each component operator is copied
        % unchanged, as are the initial acknowledges received and expected fields
        % of each component.  The graph must be sealed or else "unsealed" is
        % signalled.

        tagcase op.kind_of_op
        tag primitive:
                target: OPTR := OPTR$create_primitive(op.opname, descr)

                % Copy set of input names

                for s: string in OPTR$input_names(up(op))
                do      OPTR$name_input(target, OPTR$input_no(up(op), s), s)
                end

                % Copy set of output names
```

```
          for s: string in OPTR$output_names(up(op))
          do      OPTR$name_output(target, OPTR$output_no(up(op), s), s)
          end
          return(target)

     tag graph(g: graph_op):
          if ~g.sealed
          then    signal unsealed
          end

          target: OPTR := OPTR$create_graph(op.opname, descr)
          trep: rep := down(target)
          tg: graph_op := op_kind$value_graph(trep.kind_of_op)

          % First, copy each component

          for o: OPTR in array[OPTR]$elements(g.components)
          do      o2: OPTR := OPTR$copy(o, o.description)
                  OPTR$include(target, o2)

                  if OPTR$is_primitive(o2)
                  then    o2.acks_expected := o.acks_expected
                          o2.acks_received := o.acks_received
                  end
          end

          % Now attach them in target as attached in op.  To avoid
          % redundant attachments, copy the attachments at the source of
          % each input of each component operator; also copy acknowledge arcs.

          for i: int in array[OPTR]$indexes(g.components)
          do      for inp: int in int$from_to(1,
                                          OPTR$indegree(g.components[i]))
                  do      if ~OPTR$null_source(g.components[i], inp)
                          then    % Map attachment in g onto tg
                                  oc: outconn :=
                                          OPTR$source(g.components[i], inp)
                                  OPTR$attach(target,
                                          tg.components[oc.op.id],
                                          oc.outp, tg.components[i], inp)
                          end
                  end
```

```
        c: OPTR := g.components[i]

        for j: int in array[aic]$indexes(down(c).ack_to)
        do      for ic: inconn in aic$elements(down(c).ack_to[j])
                do      OPTR$acknowledge(target, tg.components[i],
                                j, tg.components[ic.op.id], ic.inp)

                        % Reset acks_expected field of target

                        p: prim_op := find_receiver(
                                tg.components[ic.op.id], ic.inp)
                        p.acks_expected := p.acks_expected - 1
                end
        end
end

OPTR$seal(target, DESC$new())

% Now copy any input or output names not inherited from components

for s: string in OPTR$input_names(up(op))
do      inp: int := OPTR$input_no(up(op), s)

        if OPTR$input_name(target, inp) ~= s
        then    OPTR$name_input(target, inp, s)
        end
end


for s: string in OPTR$output_names(up(op))
do      outp: int := OPTR$output_no(up(op), s)

        if OPTR$output_name(target, outp) ~= s
        then    OPTR$name_output(target, outp, s)
        end
end

% Copy graphs acknowledge inputs and outputs (inherited from
% components via MAKE_ACK_INPUT and MAKE_ACK_OUTPUT.

for ic: inconn in aic$elements(g.in_subacks)
do      aic$addh(tg.in_subacks, inconn${op: tg.components[ic.op.id],
                                         inp: ic.inp})
end

trep.ack_inputs := op.ack_inputs
```

```
            for oc: outconn in aoc$elements(g.out_subacks)
            do      aoc$addh(tg.out_subacks, outconn${op: tg.components[oc.op.id],
                                                      outp: oc.outp})
                    array[aic]$addh(trep.ack_to, aic$new())
            end

            return(target)
        end

    end copy


    create_graph = proc(name: string, description: DESC) returns(cvt)

        % Returns a new free unsealed graph operator with opname NAME, and
        % description as specified.

        return (rep${   kind_of_op: op_kind$make_graph(graph_op${
                                                components: array[OPTR]$new(),
                                                next_id: 1,
                                                subinputs: al$new(),
                                                suboutputs: aoc$new(),
                                                in_subacks: aic$new(),
                                                out_subacks: aoc$new(),
                                                sealed: false}),
                        opname: name,
                        inputs: al$new(),
                        outputs: al$new(),
                        owned: owner$make_free(nil),
                        description: description,
                        id: 0,
                        in_names: tbl$create(),
                        out_names: tbl$create(),
                        ack_to: array[aic]$new(),
                        ack_inputs: 0 })

    end create_graph
```

```
create_primitive = proc(name: string, description: DESC) returns(cvt)
                                              signals(not_primitive_opname,
                                                      no_operator_table)

     % Returns a new free primitive operator of type NAME, with
     % description as specified, or signals not_primitive_opname if
     % NAME is not a valid op name.

     r: row := lookup_opname(name)
        except
             when not_primitive_opname:      signal not_primitive_opname
             when no_operator_table:         signal no_operator_table
        end

     ins: al := al$new()
     outs: al := al$new()
     ack_outs: array[aic] := array[aic]$new()

     for i: int in int$from_to(1, r.inputs)
     do      % Set up inputs array with unconnected links
             al$addh(ins, new_link())
     end

     for i: int in int$from_to(1, r.outputs)
     do      % Set up outputs array with unconnected links
             al$addh(outs, new_link())
     end

     for i: int in int$from_to(1, r.ack_outputs)
     do      % Set up acknowledge destinations array with no dests.
             array[aic]$addh(ack_outs, aic$new())
     end

     return (rep${    kind_of_op: op_kind$make_primitive(prim_op${
                                    acks_expected: 0,
                                    init_acks_received: 0 }),
                      opname: name,
                      inputs: ins,
                      outputs: outs,
                      owned: owner$make_free(nil),
                      description: description,
                      id: 0,
                      in_names: tbl$create(),
                      out_names: tbl$create(),
                      ack_to: ack_outs,
                      ack_inputs: r.ack_inputs })

end create_primitive
```

```
dests = proc(op: OPTR, outp: int) returns(aic) signals(outputs_range, unsealed,
                                                                free_operator)

        % Returns an array (possibly empty) whose elements are the input
        % connections of op's outpth output.  If op has no outpth output, then
        % "outputs_range" is signalled; if op is free then "free_operator"
        % is signalled; if op is unsealed then "unsealed" is signalled.

        if ~OPTR$is_sealed(op)
        then    signal unsealed
        end

        if OPTR$free(op)
        then    signal free_operator
        end

        l: link := down(op).outputs[outp]
          except
                when bounds: signal outputs_range
          end

        % Generate a new array containing copies of the destinations of l.

        destlist: aic := aic$new()
        for ic: inconn  in aic$elements(l.dests)
        do      % Copy each inconn
                aic$addh(destlist, inconn$copy1(ic))
        end

        return(destlist)
end dests


detach = proc(g: OPTR, op1: OPTR, outp: int, op2: OPTR, inp: int) returns(OPTR)
                              signals(not_graph, sealed, not_included,
                                        not_attached, inputs_range, outputs_range)

        % Breaks the attachment made by the corresponding call to ATTACH.
        % If EITHER the input or the output arc involved in the attachment was
        % named before the attachment was made, then BOTH arcs will retain
        % this name even after detachment.  Signals are: "not_graph" if g is not a
        % graph, "sealed" if g has been sealed, "not_included" if op1 or op2 is
        % not in g, "not_attached" if the indicated attachment does not exist,
        % "inputs_range" if op2 has no number inp input, "outputs_range" if op1
        % has no number outp output.
```

```
        if ~OPTR$is_graph(g)
        then    signal not_graph

        elseif OPTR$is_sealed(g)
        then    signal sealed

        elseif (OPTR$free(op1) | OPTR$free(op2))
              cor (op1.owner ~= g | op2.owner ~= g)
        then    signal not_included

        elseif OPTR$indegree(op2) < inp
        then    signal inputs_range

        elseif OPTR$outdegree(op1) < outp
        then    signal outputs_range
        end

        r1: rep := down(op1)
        r2: rep := down(op2)
        l: link := r1.outputs[outp]

        if l ~= r2.inputs[inp]
        then    signal not_attached
        end

        % Remove op2 from destination list of l

        pos: int := 0
        for i: int in aic$indexes(l.dests)
        do      if l.dests[i].op = op2 & l.dests[i].inp = inp
                then    pos := i
                        break
                end
        end
        l.dests[pos] := aic$top(l.dests)
        aic$remh(l.dests)

        if aic$size(l.dests) = 0                % Make the link null if dests empty
        then
                l.source := aoc$new()
        end

        % Give op1 a new (null) <inp>th input link
        r2.inputs[inp] := new_link()
        r2.inputs[inp].name := l.name

        return(g)
    end detach
```

```
detach_ack = proc(g: OPTR, ops: OPTR, ack_no: int, opr: OPTR, ack_inp: int)
                              returns(OPTR)
                              signals(not_graph, sealed, not_included,
                                      not_attached, in_range, out_range)

        % Like DETACH, but for acknowledge arcs; breaks the attachment made by the
        % corresponding call to ACKNOWLEDGE.  Signals are: "not_graph" if g is not
        % a graph, "sealed" if g has been sealed, "not_included" if ops or opr is
        % not in g, "not_attached" if the indicated acknowledge attachment does
        % not exist, "in_range" if ops has no number ack_inp acknowledge input,
        % "out_range" if opr has no number ack_no acknowledge output.

        if ~OPTR$is_graph(g)
        then    signal not_graph

        elseif OPTR$is_sealed(g)
        then    signal sealed

        elseif (OPTR$free(ops) | OPTR$free(opr))
              cor (ops.owner ~= g | opr.owner ~= g)
        then    signal not_included

        elseif OPTR$ack_indegree(opr) < ack_inp
        then    signal in_range

        elseif OPTR$ack_outdegree(ops) < ack_no
        then    signal out_range
        end

        % Find opr's position (pos) among destinations of ops's output

        pos: int := 0
        ai: aic := down(ops).ack_to[ack_no]

        for i: int in aic$indexes(ai)
        do      if ai[i].op = opr  &  ai[i].inp = ack_inp
                then    pos := i
                        break
                end
        end

        if pos = 0
        then    signal not_attached
        end
```

```
        % Remove opr from destinations of ops's output

        ai[pos] := aic$top(ai)
        aic$remh(ai)

        % Decrement acks expected by primitive target operator

        p: prim_op := find_receiver(opr, ack_inp)
        p.acks_expected := p.acks_expected - 1

        return(g)
end detach_ack


equal = proc(ol, o2: cvt) returns(bool)
        return(ol = o2)
end equal


fetch = proc(g: cvt, i: int) returns(OPTR) signals(not_graph, bounds)

        % Returns the ith component of graph g (whether g is sealed or unsealed).
        % If g is not a graph, signals "not_graph"; if g has no ith component,
        % signals "bounds". Note that operator$fetch can be invoked by the
        % shorthand form for array subscript referencing, e.g. "op[i]" .

        gop: graph_op := op_kind$value_graph(g.kind_of_op)
            except
                when wrong_tag:         signal not_graph
            end

        return(gop.components[i])
            except
                when bounds:    signal bounds
            end
end fetch


free = proc(op: cvt) returns(bool)

        % Returns true if op belongs to no graph, else false.

        tagcase op.owned
        tag free:               return(true)
        tag owned_by:   return(false)
        end

end free
```

```
% The following "get_..." operations can be invoked by the shorthand for
% record component selection, e.g. "op.acks_expected" .

get_acks_expected = proc(op: cvt) returns(int) signals(not_primitive)

        % Returns the number of acknowledges expected by the (primitive)
        % operator op, that is, the number of acks that must be received
        % before the operator can fire.  This information is defined at
        % create time (it is in the primitive operator table).  If op is
        % not a primitive operator, then "not_primitive" is signalled.

        p: prim_op := op_kind$value_primitive(op.kind_of_op)
            except
                when wrong_tag:         signal not_primitive
            end
        return(p.acks_expected)
end get_acks_expected


get_acks_received = proc(op: cvt) returns(int) signals(not_primitive)

        % Returns the number of acknowledges considered to be initially
        % received by the (primitive) operator op.  This information is
        % defined at create time (it is in the primitive operator table).
        % If op is a graph, then "not_primitive" is signalled.

        p: prim_op := op_kind$value_primitive(op.kind_of_op)
            except
                when wrong_tag:         signal not_primitive
            end
        return(p.init_acks_received)
end get_acks_received


get_description = proc(o: cvt)  returns(DESC)

        % Returns the description of o

        return(o.description)
end get_description


get_id = proc(o: cvt) returns(int)

        % Returns id of o

        return(o.id)
end get_id
```

```
get_opname = proc(o: cvt) returns(string)

        % Returns the opname (i.e. operator type) of o

        return(o.opname)
end get_opname


get_owner = proc(o: cvt) returns(OPTR) signals(free_operator)

        % Returns the owner of o if o is not free, or signals "free_operator"

        tagcase o.owned
        tag free:
                signal free_operator
        tag owned_by(o2: OPTR):
                return(o2)
        end

end get_owner


in_suback = proc(g: OPTR, ack_inp: int) returns(inconn)
                                        signals(not_graph, unsealed, in_range)

        % Returns an inconn whose op component is the operator that receives
        % graph g's number ack_inp acknowledge input (created by the operation
        % MAKE_ACK_INPUT), and whose inp component is the corresponding
        % acknowledge input of that operator.

        if ~OPTR$is_graph(g)
        then    signal not_graph

        elseif ~OPTR$is_sealed(g)
        then    signal unsealed

        elseif OPTR$ack_indegree(g) < ack_inp
        then    signal in_range
        end

        gop: graph_op := op_kind$value_graph(down(g).kind_of_op)
        return(inconn$copy1(gop.in_subacks[ack_inp]))
end in_suback
```

```
include = proc(g: OPTR, op: OPTR) returns(OPTR) signals(already_owned,
                        cant_include_self, cant_include_unsealed_graph,
                        not_graph, sealed)

        % Includes op in g; assigns g as op's owner, and assigns the next id
        % number for g to op.  If op is already owned by a graph, signals
        % "already_owned";  if op = g then "cant_include_self" is signalled;
        % if op is an unsealed graph then "cant_include_unsealed_graph" is
        % signalled;  if g is not a graph operator, signals "not_graph"; if g
        % has already been sealed, signals "sealed".
        % The argument g is returned.

        if ~OPTR$free(op)
        then    signal already_owned

        elseif op = g
        then    signal cant_include_self

        elseif ~OPTR$is_sealed(op)
        then    signal cant_include_unsealed_graph

        elseif ~OPTR$is_graph(g)
        then    signal not_graph

        elseif OPTR$is_sealed(g)
        then    signal sealed
        end

        dg: rep := down(g)
        g2: graph_op := op_kind$value_graph(dg.kind_of_op)
        dop: rep := down(op)

        %       Assign op's owner and id
        dop.owned := owner$make_owned_by(g)
        dop.id := g2.next_id

        %       Add op to components of g
        array[OPTR]$addh(g2.components, op)
        g2.next_id := g2.next_id + 1

        return(g)
end include
```

```
indegree = proc(op: OPTR)  returns(int) signals(unsealed)

        % Returns the number of inputs defined for op (whether primitive
        % or graph), or signals if op is an unsealed graph.

        if ~OPTR$is_sealed(op)
        then     signal unsealed
        end

        return(al$size(down(op).inputs))
end indegree


input_name = proc(op: OPTR, inp: int)  returns(string) signals(unsealed,
                                                                inputs_range)

        % The inverse of input_no.  Signals "unsealed" if op is not
        % sealed; signals "inputs_range" if op has no inpth input.

        if ~OPTR$is_sealed(op)
        then     signal unsealed
        end

        return(down(op).inputs[inp].name)
            except
                 when bounds:    signal inputs_range
            end

end input_name


input_names = iter(op: OPTR) yields(string) signals(unsealed)

        % An iterator over all the input names defined for op.
        % Signals "unsealed" if op is not sealed.

        if ~OPTR$is_sealed(op)
        then     signal unsealed
        end

        for name: string, dummy: int in tbl$elements(down(op).in_names)
        do       % Yield each name in the order delivered by TABLE cluster
                 yield(name)
        end
        return

end input_names
```

```
input_no = proc(op: OPTR, name: string) returns(int) signals(unsealed)

        % Returns the input number of the input associated with name, or 0 if
        % no such name is assigned for op.  Signals "unsealed" if op is unsealed.

        if ~OPTR$is_sealed(op)
        then    signal unsealed
        end

        return(tbl$lookup(name, lkp, down(op).in_names))
            except
                when no_match: return(0)
            end
end input_no


is_graph = proc(op: cvt) returns(bool)

        % Returns TRUE if op is a graph operator, FALSE if
        % op is a primitive operator.  (equivalent to ~is_primitive(op) )

        return(op_kind$is_graph(op.kind_of_op))
end is_graph


is_primitive = proc(op: cvt) returns(bool)

        % Returns TRUE if op is a primitive operator, FALSE if
        % op is a graph operator.  (equivalent to ~is_graph(op) )

        return(op_kind$is_primitive(op.kind_of_op))
end is_primitive


is_sealed = proc(op: cvt) returns(bool)

        % Returns false iff op is an unsealed graph, else returns true.

        gop: graph_op := op_kind$value_graph(op.kind_of_op)
            except
                when wrong_tag:         return(true)
            end

        if gop.sealed
        then    return(true)
        else    return(false)
        end
end is_sealed
```

```
make_ack_input = proc(g: OPTR, op: OPTR, ack_inp: int) returns(OPTR)
                            signals(in_range, not_included, not_graph,
                                    sealed)

        % Causes graph g to "inherit" the number ack_inp acknowledge input
        % of its component operator op -- that is, this acknowledge input
        % will become the next acknowledge input to the whole graph g.
        % The argument g is returned.  Signals are: "in_range" if op has
        % no number ack_inp acknowledge input, "not_included" if op is not
        % a component of g, "not_graph" if g is not a graph, "sealed"
        % if g has been sealed.

        if ~OPTR$is_graph(g)
        then    signal not_graph

        elseif OPTR$is_sealed(g)
        then    signal sealed

        elseif OPTR$free(op)  cor  op.owner ~= g
        then    signal not_included

        elseif OPTR$ack_indegree(op) < ack_inp
        then    signal in_range
        end

        gop: graph_op := op_kind$value_graph(down(g).kind_of_op)

        aic$addh(gop.in_subacks, inconn${op: op, inp: ack_inp})
        down(g).ack_inputs := down(g).ack_inputs + 1

        return(g)
end make_ack_input
```

```
make_ack_output = proc(g: OPTR, op: OPTR, ack_no: int) returns(OPTR)
                                    signals(out_range, not_included, not_graph,
                                            sealed)

        % Causes graph g to "inherit" the number ack_no acknowledge output
        % of its component operator op -- that is, this acknowledge output
        % will become the next acknowledge output from the whole graph g.
        % The argument g is returned.  Signals are: "out_range" if op has
        % no number ack_no acknowledge output, "not_included" if op is not
        % a component of g, "not_graph" if g is not a graph, "sealed"
        % if g has been sealed.

        if ~OPTR$is_graph(g)
        then    signal not_graph

        elseif OPTR$is_sealed(g)
        then    signal sealed

        elseif OPTR$free(op)  cor  op.owner ~= g
        then    signal not_included

        elseif OPTR$ack_outdegree(op) < ack_no
        then    signal out_range
        end

        gop: graph_op := op_kind$value_graph(down(g).kind_of_op)

        aoc$addh(gop.out_subacks, outconn${op: op, outp: ack_no})
        array[aic]$addh(down(g).ack_to, aic$new())

        return(g)
 end make_ack_output
```

```
name_input = proc(op: OPTR, inp: int, name: string) returns(OPTR)
                                  signals(inputs_range, unsealed,
                                          name_already_defined, multiple_names)

        % Associates name as the name of input inp of operator op.
        % Op must be sealed, or else "unsealed" is signalled.
        % Signals "inputs_range" if inp is not a valid input of op;
        % signals "name_already_defined" if op already has an input with
        % this name;  signals "multiple_names" if this input has another name.
        % The argument op is returned.

        if ~OPTR$is_sealed(op)
        then    signal unsealed
        end

        r: rep := down(op)
        l: link := r.inputs[inp]
            except
                 when bounds:    signal inputs_range
            end

        if tbl$is_in(name, lkp, r.in_names)
        then    signal name_already_defined
        end

        if l.name ~= ""
        then    signal multiple_names
        end

        l.name := name
        tbl$insert(name, inp, r.in_names)

        return(op)
    end name_input
```

```
name_output = proc(op: OPTR, outp: int, name: string) returns(OPTR)
                                        signals(outputs_range, unsealed,
                                              name_already_defined, multiple_names)

        % Associates name as the name of output outp of operator op.  Signals
        % "outputs_range" if outp is not a valid output of op; other signals
        % are identical to those of name_input.  The argument op is returned.

        if ~OPTR$is_sealed(op)
        then    signal unsealed
        end

        r: rep := down(op)
        l: link := r.outputs[outp]
            except
                when bounds:    signal outputs_range
            end

        if tbl$is_in(name, lkp, r.out_names)
        then    signal name_already_defined
        end

        if l.name ~= ""
        then    signal multiple_names
        end

        l.name := name
        tbl$insert(name, outp, r.out_names)

        return(op)
end name_output


named_indegree = proc(op: OPTR)  returns(int) signals(unsealed)

        % Returns number of named inputs defined for op (whether primitive or graph),
        % or signals "unsealed" if op is not sealed.  In all cases named_indegree(op)
        % is less-than-or-equal-to indegree(op).

        if ~OPTR$is_sealed(op)
        then    signal unsealed
        end

        return(tbl$size(down(op).in_names))
end named_indegree
```

```
named_outdegree = proc(op: OPTR) returns(int) signals(unsealed)

        % Like named_indegree, but for named outputs.

        if ~OPTR$is_sealed(op)
        then    signal unsealed
        end

        return(tbl$size(down(op).out_names))
end named_outdegree


null_source = proc(op: OPTR, inp: int)  returns(bool) signals(inputs_range, unsealed)

        % Returns true if op's inpth input has no source operator, else false; if
        % free(op) then true is returned.  If inp is outside the range of valid
        % inputs for op, then "inputs_range" is signalled.  Signals "unsealed"
        % if op is unsealed.

        if ~OPTR$is_sealed(op)
        then    signal unsealed
        end

        r: rep := down(op)

        if null_link(r.inputs[inp])
        then    return(true)
        else    return(false)
        end  except
                when bounds: signal inputs_range
             end
end null_source


out_suback = proc(g: OPTR, ack_no: int) returns(outconn)
                                        signals(not_graph, unsealed, out_range)

        % Returns an outconn whose op component is the operator that generates
        % graph g's number ack_no acknowledge output (created by the operation
        % MAKE_ACK_OUTPUT), and whose outp component is the corresponding
        % acknowledge output of that operator.

        if ~OPTR$is_graph(g)
        then    signal not_graph

        elseif ~OPTR$is_sealed(g)
        then    signal unsealed
```

```
        elseif OPTR$ack_outdegree(g) < ack_no
        then    signal out_range
        end

        gop: graph_op := op_kind$value_graph(down(g).kind_of_op)
        return(outconn$copy1(gop.out_subacks[ack_no]))
end out_suback


outdegree = proc(op: OPTR)  returns(int) signals(unsealed)

        % Returns the number of outputs defined for op (whether primitive
        % or graph), or signals if op is an unsealed graph.

        if ~OPTR$is_sealed(op)
        then    signal unsealed
        end

        return(al$size(down(op).outputs))
end outdegree

output_name = proc(op: OPTR, outp: int)  returns(string) signals(unsealed,
                                                                  outputs_range)

        % The inverse of output_no.  Signals "unsealed" if op is not
        % sealed; signals "outputs_range" if op has no outpth output.

        if ~OPTR$is_sealed(op)
        then    signal unsealed
        end

        return(down(op).outputs[outp].name)
            except
                when bounds:    signal outputs_range
            end
end output_name
```

```
output_names = iter(op: OPTR) yields(string) signals(unsealed)

        % An iterator over all the output names defined for op.
        % Signals "unsealed" if op is not sealed.

        if ~OPTR$is_sealed(op)
        then    signal unsealed
        end

        for name: string, dummy: int in tbl$elements(down(op).out_names)
        do      % Yield each name in the order delivered by TABLE cluster
                yield(name)
        end
        return

end output_names


output_no = proc(op: OPTR, name: string) returns(int) signals(unsealed)

        % Returns the output number of the output associated with name, or 0
        % if no such name is assigned for op.  Signals "unsealed" if op
        % is unsealed.

        if ~OPTR$is_sealed(op)
        then    signal unsealed
        end

        return(tbl$lookup(name, lkp, down(op).out_names))
            except
                when no_match: return(0)
            end

end output_no


remove = proc(g: OPTR, op: OPTR) returns(OPTR) signals(not_graph, sealed,
                                                        not_included)

        % Removes the operator op from graph g, breaking (via DETACH and
        % DETACH_ACK) all the attachments to and from op, and making op free
        % again (NOTE THESE SIDE EFFECTS).  Signals are: "not_graph" if g is
        % not a graph, "sealed" if g has been sealed, "not_included" if op is
        % not a component of g.

        if ~OPTR$is_graph(g)
        then    signal not_graph
```

```
elseif OPTR$is_sealed(g)
then    signal sealed

elseif OPTR$free(op)  cor  op.owner ~= g
then    signal not_included
end

grep: rep := down(g)
gop: graph_op := op_kind$value_graph(grep.kind_of_op)
oprep: rep := down(op)

found: bool := false        % Set when op's position in g is found

for i: int in array[OPTR]$indexes(gop.components)
do      o: OPTR := gop.components[i]
        if found
        then    % Move all operators down 1 place, thus removing op
                gop.components[i-1] := o

        elseif o = op
        then    found := true
                continue        % Dont process op
        end

        % At this point we know o is NOT equal to op;  remove any
        % acknowledges sent to op from o.

        for ack_no: int in array[aic]$indexes(down(o).ack_to)
        do      for ic: inconn in aic$elements(down(o).ack_to[ack_no])
                do      if ic.op = op
                        then    % Remove this acknowledge arc
                                OPTR$detach_ack(g, o, ack_no, op, ic.inp)
                                break
                        end
                end
        end
end

% Trim components array, having removed op from it in loop above
array[OPTR]$remh(gop.components)

% Now remove all acknowledge outputs from op

for i: int in array[aic]$indexes(oprep.ack_to)
do      for ic: inconn in aic$elements(aic$copy1(oprep.ack_to[i]))
        do      OPTR$detach_ack(g, op, i, ic.op, ic.inp)
        end
end
```

```
    % Remove from set of g's acknowledge inputs and outputs any subacks
    % inherited from op.

    new_in_subacks: aic := aic$new()
    for ic: inconn in aic$elements(gop.in_subacks)
    do      if ic.op ~= op
            then    aic$addh(new_in_subacks, ic)
            end
    end
    gop.in_subacks := new_in_subacks

    new_ack_to: array[aic] := array[aic]$new()
    new_out_subacks: aoc := aoc$new()

    for i: int in aoc$indexes(gop.out_subacks)
    do      if gop.out_subacks[i].op ~= op
            then    aoc$addh(new_out_subacks, gop.out_subacks[i])
                    array[aic]$addh(new_ack_to, grep.ack_to[i])
            end
    end

    gop.out_subacks := new_out_subacks
    grep.ack_to := new_ack_to

    % Now break all attachments to op's inputs

    for inp: int in int$from_to(1, OPTR$indegree(op))
    do      if ~OPTR$null_source(op, inp)
            then    oc: outconn := OPTR$source(op, inp)
                    OPTR$detach(g, oc.op, oc.outp, op, inp)
            end
    end

    % Now break all attachments to op's outputs

    for outp: int in int$from_to(1, OPTR$outdegree(op))
    do      for ic: inconn in aic$elements(OPTR$dests(op, outp))
            do      OPTR$detach(g, op, outp, ic.op, ic.inp)
            end
    end

    % Make op free

    oprep.owned := owner$make_free(nil)
    oprep.id := 0

    return(g)
end remove
```

```
seal = proc(op: cvt, descr: DESC) returns(cvt) signals(not_graph, already_sealed,
                                                   output_name_multiply_defined)

        % Seals op so that no more attachments can be made within it and so it
        % can be used as a component operator in other graphs.  Op must be a
        % graph, or "not_graph" is signalled.  If op is already sealed, signals
        % "already_sealed"; if there is more than one unconnected component
        % output with the same name, then "output_name_multiply_defined" is
        % signalled, since these outputs would become graph outputs with the
        % same name.  No such signal occurs for inputs, since all component inputs
        % with the same name will be merged into one graph input. Any description
        % provided in this call is appended to the description given the
        % operator at creation time.  The argument (op) is returned.


        g: graph_op := op_kind$value_graph(op.kind_of_op)
            except
                when wrong_tag:         signal not_graph
            end

        if g.sealed
        then    signal already_sealed
        end

        for c: OPTR in array[OPTR]$elements(g.components)
        do      % Make each unconnected component input a graph input
                for inp: int in int$from_to(1, OPTR$indegree(c))
                do      if OPTR$null_source(c, inp)
                        then    name: string := OPTR$input_name(c, inp)
                                if name ~= ""  cand
                                  tbl$is_in(name, lkp, op.in_names)
                                then    % Merge c's inpth input with
                                        % correspondingly named graph input
                                        k: int := tbl$lookup(name,lkp,op.in_names)
                                        aic$addh(g.subinputs[k].dests,
                                                        inconn${op: c, inp: inp})
                                else    % New graph input
                                        al$addh(op.inputs, new_link())
                                        al$top(op.inputs).name := name
                                        al$addh(g.subinputs, link${
                                                source: aoc$new(),
                                                dests: aic$[inconn${op: c, inp: inp}],
                                                name: name})
                                        if name ~= ""
                                        then    tbl$insert(name, al$high(op.inputs),
                                                                op.in_names)
                                        end
                                end
```

```
                        end
                end

                % Now do same for unconnected component outputs
                for outp: int in int$from_to(1, OPTR$outdegree(c))
                do      if aic$size(OPTR$dests(c, outp)) = 0
                        then    % Make c's outpth output a graph output
                                name: string := OPTR$output_name(c, outp)
                                if name ~= "" cand
                                   tbl$is_in(name, lkp, op.out_names)
                                then    signal output_name_multiply_defined
                                else    % New graph output
                                        al$addh(op.outputs, new_link())
                                        al$top(op.outputs).name := name
                                        aoc$addh(g.suboutputs,
                                                outconn${op: c, outp: outp})
                                        if name ~= ""
                                        then    tbl$insert(name, al$high(op.outputs),
                                                                        op.out_names)
                                        end
                                end
                        end
                end
        end

        g.sealed := true                % Seal the graph and add to description
        op.description := array_cat[string](op.description, descr)
        return(op)
end seal


% The following "set_..." operations can be invoked by the shorthand for
% record component update, e.g. "op.acks_expected := ...".


set_acks_expected = proc(op: cvt, i: int) signals(not_primitive)

        % Sets the number of acknowledges expected by the primitive operator
        % op to i, or signals if op is not primitive.  NOTE -- this operation
        % should be used ONLY to specially set the acknowledges expected to
        % a value other than the number of acknowledge arcs pointing to the
        % operator, as that is the default value (set by calls to the
        % operation ACKNOWLEDGE).

        p: prim_op := op_kind$value_primitive(op.kind_of_op)
            except
                when wrong_tag:         signal not_primitive
            end
```

```
        p.acks_expected := i
        return
 end set_acks_expected


 set_acks_received = proc(op: cvt, i: int) signals(not_primitive)

        % Sets the number of acknowledges received by the primitive operator
        % op to i, or signals if op is not primitive.

        p: prim_op := op_kind$value_primitive(op.kind_of_op)
           except
              when wrong_tag:          signal not_primitive
           end

        p.init_acks_received := i
        return
 end set_acks_received


 source = proc(op: OPTR, inp: int) returns(outconn) signals(no_source, inputs_range,
                                                            unsealed, free_operator)

        % Returns the output connection that is the source of op's inpth input,
        % or signals "no_source" if op is not free but has no source at that
        % input, "inputs_range" if op has no inpth input, "unsealed" if op is
        % not sealed, or "free_operator" if op is free.

        if ~OPTR$is_sealed(op)
        then    signal unsealed
        end

        if OPTR$free(op)
        then    signal free_operator
        end

        l: link := down(op).inputs[inp]
          except
              when bounds: signal inputs_range
          end

        if null_link(l)
        then    signal no_source
        else    return(outconn$copy1(l.source[1]))
        end
 end source
```

```
subinput = proc(g: OPTR, inp: int) returns(aic)
                                   signals(inputs_range, unsealed, not_graph)

        % Returns the input connections to the component operators of g
        % corresponding to graph input inp of g, or signals as above.

        if ~OPTR$is_sealed(g)
        then    signal unsealed
        end

        dg: graph_op := op_kind$value_graph(down(g).kind_of_op)
            except
                when wrong_tag:         signal not_graph
            end

        % copy inconn list for this subinput
        subs: aic := aic$new()
        for ic: inconn in aic$elements(dg.subinputs[inp].dests)
        do      aic$addh(subs, inconn$copyl(ic))
        end  except
                when bounds: signal inputs_range
            end

        return(subs)
end subinput


suboutput = proc(g: OPTR, outp: int) returns(outconn)
                                     signals(outputs_range, unsealed, not_graph)

        % Returns the output connection from the component operators of g
        % corresponding to graph output outp of g, or signals as above.

        if ~OPTR$is_sealed(g)
        then    signal unsealed
        end

        dg: graph_op := op_kind$value_graph(down(g).kind_of_op)
            except
                when wrong_tag:         signal not_graph
            end

        return(outconn$copyl(dg.suboutputs[outp]))
            except
                when bounds:    signal outputs_range
            end
end suboutput
```

```
write = proc(op: OPTR, s: stream)  returns(OPTR) signals(unsealed, not_possible)

        % Writes a description of op to stream s.  If g is not sealed, signals
        % "unsealed"; if s cannot be written to, signals "not_possible".
        % The argument op is returned.

        output_comma: bool
        indent: string := "  "

        if ~stream$can_write(s)
        then    signal not_possible
        end

        if ~OPTR$is_sealed(op)
        then    signal unsealed
        end


        if OPTR$is_primitive(op)
        then    % Just give "top level" description of op
                put_description(s, op)
                ins: int := OPTR$indegree(op)
                outs: int := OPTR$outdegree(op)

                stream$puts(s, indent || "inputs: " || int$unparse(ins))
                output_comma := false
                %       Write out set of input names
                for n: string in OPTR$input_names(op)
                do      if output_comma
                        then    stream$putc(s, ',')
                        else    stream$puts(s, "  names:")
                                output_comma := true
                        end

                        stream$puts(s, " \"" || n || "\"("
                                || int$unparse(OPTR$input_no(op, n)) || ")" )
                end
                stream$putc(s, '\n')

                stream$puts(s, indent || "outputs: " || int$unparse(outs))
                output_comma := false
```

```
        %       Write out set of output names
        for n: string in OPTR$output_names(op)
        do      if output_comma
                then    stream$putc(s, ',')
                else    stream$puts(s, "  names:")
                        output_comma := true
                end

                stream$puts(s, " \"" || n || "\"("
                        || int$unparse(OPTR$output_no(op, n)) || ")" )
        end
        stream$putc(s, '\n')

        stream$putl(s, indent || "acknowledge inputs: " ||
                                int$unparse(OPTR$ack_indegree(op)))
        stream$putl(s, indent || "acknowledge outputs: " ||
                                int$unparse(OPTR$ack_outdegree(op)))
        stream$putl(s, indent || "acknowledges expected: " ||
                                int$unparse(op.acks_expected))
        stream$putl(s, indent || "acknowledges initially received: " ||
                                int$unparse(op.acks_received))
        return(op)
end

% If op is a graph, first give top level description of g

stream$puts(s, "graph ")
put_description(s, op)


% Now describe graph inputs of op

gop: graph_op := op_kind$value_graph(down(op).kind_of_op)
k: int := OPTR$indegree(op)
stream$puts(s, indent || "inputs(" || int$unparse(k) || ")")
output_comma := false

for i: int in int$from_to(1, k)
do      if output_comma
        then    stream$putc(s, ',')
        else    stream$putc(s, ':')
                output_comma := true
        end

        l: link := gop.subinputs[i]
        stream$puts(s, " {")
        output_inner_comma: bool := false
```

```
          for ic: inconn in aic$elements(l.dests)
          do      if output_inner_comma
                  then    stream$puts(s, ", ")
                  else    output_inner_comma := true
                  end

                  stream$puts(s, "op" || int$unparse(ic.op.id) || "•"
                                                          || int$unparse(ic.inp))
          end

          stream$putc(s, '}')
          n: string := down(op).inputs[i].name
          if n ~= ""
          then    stream$puts(s, "(\"" || n || "\")")
          end
end

stream$putc(s, '\n')

% Describe graph outputs of op in exactly the same way

k := OPTR$outdegree(op)
stream$puts(s, indent || "outputs(" || int$unparse(k) || ")")
output_comma := false

for i: int in int$from_to(1, k)
do        if output_comma
          then    stream$putc(s, ',')
          else    stream$putc(s, ':')
                  output_comma := true
          end

          oc: outconn := gop.suboutputs[i]
          stream$puts(s, " op" || int$unparse(oc.op.id) || "•"
                                                          || int$unparse(oc.outp))
          n: string := down(op).outputs[i].name
          if n ~= ""
          then    stream$puts(s, "(\"" || n || "\")")
          end
end
stream$putc(s, '\n')

% Describe acknowledge inputs of op

k := OPTR$ack_indegree(op)
stream$puts(s, indent || "acknowledge inputs(" || int$unparse(k) || ")")
```

```
output_comma := false

for i: int in int$from_to(1, k)
do      if output_comma
        then    stream$putc(s, ',')
        else    stream$putc(s, ':')
                output_comma := true
        end

        ic: inconn := gop.in_subacks[i]
        stream$puts(s, " op" || int$unparse(ic.op.id) || "."
                                        || int$unparse(ic.inp))
end
stream$putc(s, '\n')

% Describe acknowledge outputs of op

k := OPTR$ack_outdegree(op)
stream$puts(s, indent || "acknowledge outputs(" || int$unparse(k)
                                                        || ")")
output_comma := false

for i: int in int$from_to(1, k)
do      if output_comma
        then    stream$putc(s, ',')
        else    stream$putc(s, ':')
                output_comma := true
        end

        oc: outconn := gop.out_subacks[i]
        stream$puts(s, " op" || int$unparse(oc.op.id) || "."
                                        || int$unparse(oc.outp))
end
stream$putc(s, '\n')

% Describe components of op
stream$putl(s, indent || "components:")

for c: OPTR in OPTR$components(op)
do      stream$puts(s, indent||indent || "op" || int$unparse(c.id) || ": ")

        if OPTR$is_graph(c)
        then    stream$puts(s, "graph ")
        end

        put_description(s, c)
```

```
% Describe input attachments of c

k := OPTR$indegree(c)
stream$puts(s, indent||indent||indent || "inputs(" || int$unparse(k)
                                                  || ")" )
output_comma := false

for i: int in int$from_to(1, k)
do      if output_comma
        then    stream$putc(s, ',')
        else    stream$puts(s, " attached:")
                output_comma := true
        end

        l: link := down(c).inputs[i]

        if null_link(l)
        then    stream$puts(s, " <graph input>")
        else    oc: outconn := l.source[1]
                stream$puts(s, " op" || int$unparse(oc.op.id)
                                || "•" || int$unparse(oc.outp))
        end

        if l.name ~= ""
        then    stream$puts(s, "(\"" || l.name || "\")")
        end
end

stream$putc(s, '\n')

% Describe output attachments of c in the same way

k := OPTR$outdegree(c)
stream$puts(s, indent||indent||indent || "outputs(" || int$unparse(k)
                                                  || ")" )
output_comma := false

for i: int in int$from_to(1, k)
do      if output_comma
        then    stream$putc(s, ',')
        else    stream$puts(s, " attached:")
                output_comma := true
        end

        l: link := down(c).outputs[i]
```

```
        if null_link(l)
        then    stream$puts(s, " <graph output>")
        else    stream$puts(s, " {")
                output_inner_comma: bool := false

                for ic2: inconn in aic$elements(l.dests)
                do      if output_inner_comma
                        then    stream$puts(s, ", ")
                        else    output_inner_comma := true
                        end

                        stream$puts(s, "op" || int$unparse(ic2.op.id)
                                        || "." || int$unparse(ic2.inp))
                end

                stream$putc(s, '}')
        end


        if l.name ~= ""
        then    stream$puts(s, "(\"" || l.name || "\")")
        end
end
stream$putc(s, '\n')

% Describe acknowledge inputs of c

stream$putl(s, indent||indent||indent || "acknowledge inputs: "
                || int$unparse(OPTR$ack_indegree(c)))

% Describe acknowledge outputs of c in same way as outputs

k := OPTR$ack_outdegree(c)
stream$puts(s, indent||indent||indent || "acknowledge outputs("
                                || int$unparse(k) || ")" )

output_comma := false
for i: int in int$from_to(1, k)
do      if output_comma
        then    stream$putc(s, ',')
        else    stream$putc(s, ':')
                output_comma := true
        end

        ai: aic := down(c).ack_to[i]
```

```
            if aic$size(ai) = 0
            then    stream$puts(s, " not sent")
            else    stream$puts(s, " sent to: {")
                    output_inner_comma: bool := false

                    for ic: inconn in aic$elements(ai)
                    do      if output_inner_comma
                            then    stream$puts(s, ", ")
                            else    output_inner_comma := true
                            end

                            stream$puts(s, "op"||int$unparse(ic.op.id)
                                            || "*" || int$unparse(ic.inp))
                    end
                    stream$putc(s, '}')
            end

            % See if c's ith acknowledge output is graph ack.

            for j: int in aoc$indexes(gop.out_subacks)
            do      if gop.out_subacks[j].op = c
                      & gop.out_subacks[j].outp = i
                    then    stream$puts(s, "<graph acknowledge " ||
                                            int$unparse(j) || ">" )
                            break
                    end
            end

        end
        stream$putc(s, '\n')

        % If c is primitive, describe its acks expected and received

        if OPTR$is_primitive(c)
        then    stream$putl(s, indent||indent||indent ||
                                "acknowledges expected: " ||
                                int$unparse(c.acks_expected))

                stream$putl(s, indent||indent||indent ||
                                "acknowledges initially received: " ||
                                int$unparse(c.acks_received))
        end

    end
    return(op)

end write
```

```
% ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ %
% Utility functions . . . %
% ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ ◊ %

 lookup_opname = proc(name: string)  returns(row)
                                signals(not_primitive_opname, no_operator_table)

        % Returns the number of inputs and outputs for an operator whose
        % operation name is NAME.  Not_primitive_opname is signalled when
        % name is not in the optable, and no_operator_table is signalled
        % if the optable cannot be found or accessed.

        optable: istream := istream$open(file_name$parse("optabl.dfg"), "read")
            except
                others: signal no_operator_table
            end
        while ~istream$empty(optable)
        do      % Read in each row of the table to find entry for "name"
                r: row := row$decode(optable)
                if r.opname = name
                then    istream$close(optable)
                        return(r)
                end
        end
        signal not_primitive_opname
 end lookup_opname


 new_link = proc() returns(link)

        % Returns a new link unconnected to any operators (a null link).

        return(link${source: aoc$new(), dests: aic$new(), name: ""})
 end new_link


 null_link = proc(l: link) returns(bool)

        % Returns TRUE if l is a newly created link,
        % i.e. unconnected to any operators at source
        % or dest.

        if aoc$size(l.source) = 0  &  aic$size(l.dests) = 0
        then    return(true)
        else    return(false)
        end

 end null_link
```

```
find_receiver = proc(op: OPTR, inp: int) returns(prim_op) signals(in_range)

        % Returns the primitive operator that receives op's number inp
        % acknowledge input, or signals in_range if op has no number inp
        % acknowledge input.

        while(OPTR$is_graph(op))
        do      ic: inconn := OPTR$in_suback(op, inp)
                    except
                        when in_range:  signal in_range
                    end

                op := ic.op
                inp := ic.inp
        end

        return(op_kind$value_primitive(down(op).kind_of_op))
end find_receiver


put_description = proc(s: stream, o: OPTR)

        % Prints top line of description of o (for operator$write)

        output_comma: bool := false
        stream$puts(s, "\"" || o.opname || "\" description: [")

        for d: string in array[string]$elements(o.description)
        do      if output_comma
                then    stream$puts(s, ", ")
                else    output_comma := true
                end

                stream$puts(s, "\"" || d || "\"")
        end

        stream$puts(s, "]\n")
        return

end put_description

end operator
```

1.2 The Table Cluster

```
% From file "table.clu"

table = cluster[keyt, itemt: type] is create, insert, delete, is_in, lookup, elements, size

% Supports a table of items of type itemt, keyed by objects of type keyt.
% Note that this is a simple-minded implementation, most suitable for small
% tables.  For larger tables where search time becomes important a more
% sophisticated representation, such as a height-balanced tree, should be used.

  rep = array[row]
  row = record[key: keyt, item: itemt]

  %       Matcht procedures are used to compare keys when searching a table.
  matcht = proctype(keyt, keyt) returns(bool)

  create = proc() returns(cvt)
          % Creates a table of the given type
          return(rep$new())
  end create


  insert = proc(k: keyt, i: itemt, t: cvt)
          % Inserts item i with key k into table t
          rep$addh(t, row${key: k, item: i})
  end insert


  delete = proc(k: keyt, match: matcht, t: cvt) signals(no_match)

          % Deletes an item with key matching k (according to match)
          % from table t.  If no key in the table matches k then
          % no_match is signalled.

          for i: int in rep$indexes(t)
          do      % Find matching key, if any
                  if match(k, t[i].key)
                  then    % Delete row i from table
                          t[i] := rep$top(t)
                          rep$remh(t)
                          return
                  end
          end

          signal no_match
  end delete
```

```
is_in = proc(k: keyt, match: matcht, t: cvt) returns(bool)

        % Returns TRUE if some item in the table has a key matching
        % k, else returns FALSE.

        for r: row in rep$elements(t)
        do      if match(k, r.key)
                then    return(true)
                end
        end

        return(false)
end is_in


lookup = proc(k: keyt, match: matcht, t: cvt)  returns(itemt) signals(no_match)

        % Returns the item in t whose key is matched by k,
        % or signals no_match if no such item.

        for r: row in rep$elements(t)
        do      if match(k, r.key)
                then    return(r.item)
                end
        end
        signal no_match
end lookup


elements = iter(t: cvt) yields(keyt, itemt)

        % Yields the key and item of each element in the table.
        % The order of retrieval is not necessarily the order of insertion.

        for r: row in rep$elements(t)
        do      yield(r.key, r.item)
        end

        return
end elements


size = proc(t: cvt) returns(int)
        % Returns the number of items in the table
        return(array[row]$size(t))
end size

end table
```

1.3 Support Procedures

```
% From file "optabl.clu"

% Handles the file optabl.dfg -- table of primitive operators
% for the operator cluster (oper.clu)

  row = record[opname: string, inputs, outputs, ack_inputs, ack_outputs: int]

create_optabl = proc()

        % Creates the file "optabl.dfg", for use in the OPERATOR cluster.
        % The initial set of primitive operators is defined by this operation.
        % Additional operators can be defined with the add_row operation.

        outs: istream := istream$open(file_name$parse("optabl.dfg"), "write")

        append_row(outs, "+", 2, 1, 1, 1)
        append_row(outs, "-", 2, 1, 1, 1)
        append_row(outs, "*", 2, 1, 1, 1)
        append_row(outs, "/", 2, 1, 1, 1)
        append_row(outs, "=", 2, 1, 1, 1)
        append_row(outs, ">", 2, 1, 1, 1)
        append_row(outs, "<", 2, 1, 1, 1)
        append_row(outs, "and", 2, 1, 1, 1)
        append_row(outs, "or", 2, 1, 1, 1)
        append_row(outs, "not", 1, 1, 1, 1)
        append_row(outs, "I", 1, 1, 1, 1)
        append_row(outs, "sink", 1, 0, 1, 1)
        append_row(outs, "constant", 0, 1, 1, 1)
        append_row(outs, "negate", 1, 1, 1, 1)
        append_row(outs, "T-Gate", 2, 1, 1, 1)
        append_row(outs, "F-Gate", 2, 1, 1, 1)
        append_row(outs, "M-Gate", 3, 1, 1, 2)
        append_row(outs, "FS-Gate", 2, 1, 1, 1)
        append_row(outs, "FM-Gate", 3, 1, 1, 2)
        append_row(outs, "IC-Gate", 3, 3, 1, 1)

        istream$close(outs)

        return
end create_optabl
```

```
add_row = proc(name: string, inputs, outputs, ack_ins, ack_outs: int)

        % Appends rows to the file "optabl.dfg", for use in the
        % OPERATOR cluster.  To add a row to the table, type:
        %         add_row(name, inputs, outputs, ack_inputs, ack_outputs)

        outs: istream := istream$open(file_name$parse("optabl.dfg"), "append")

        append_row(outs, name, inputs, outputs, ack_ins, ack_outs)
        istream$close(outs)

        return
end add_row


append_row = proc(s: istream, name: string,
                                        inputs, outputs, ack_ins, ack_outs: int)

        % Appends a row to the stream s

        row$encode(row${opname: name, inputs: inputs, outputs: outputs,
                        ack_inputs: ack_ins, ack_outputs: ack_outs}, s)
        return
end append_row


list_optabl = proc()

        % Lists each row of the table

        ins: istream := istream$open(file_name$parse("optabl.dfg"), "read")

        while ~istream$empty(ins)
        do      r: row := row$decode(ins)
                stream$putl(stream$primary_output(),
                        "\"" || r.opname || "\"\tinputs: " ||
                        int$unparse(r.inputs) || "\toutputs: " ||
                        int$unparse(r.outputs) || "\tack inputs: " ||
                        int$unparse(r.ack_inputs) || "\tack outputs: " ||
                        int$unparse(r.ack_outputs) )
        end

        istream$close(ins)

        return
end list_optabl
```

```
% From file "acat.clu"
array_cat = proc[t: type](a, b: array[t]) returns(array[t])

        % Returns an array whose elements are the concatenation of the
        % elements of the arrays a and b.

        c: array[t] := array[t]$copy1(a)
        for elem: t in array[t]$elements(b)
        do      array[t]$addh(c, elem)
        end

        return(c)
end array_cat
```

1.4 Procedural Forms of Fig. 9, Fig. 13, and Fig. 15

The following shows the code of Fig. 9, Fig. 13, and Fig. 15, implemented as procedures. These procedures take as arguments the *operator* representation of the subexpressions of the let, if, or for graph being constructed, and return the completed graph. As before, the construction of the acknowledge arcs is not shown but is assumed to follow the construction of each data arc.

```
% From file "tester.clu"

% NEEDS oper.specs to compile


% Abbreviations:
        OPTR = operator
        DESC = array[string]              % Description data type
        aic = array[inconn]
        aoc = array[outconn]
        ast = array[string]
        aop = array[OPTR]

        inconn = record[op: OPTR, inp: int]
        outconn = record[op: OPTR, outp: int]


make_if = proc(if_exp, then_exp, else_exp: OPTR, descr: DESC) returns(OPTR)
                                                          signals(bad_args)

        % Returns a graph operator that is a (general) IF expression with
        % the OPTR arguments as subexpressions, opname "if", and the
        % given description.

        % Then_I is number of I outputs from then clause, (don't count iter?
        % output among I outputs), and  then_R is number of R outputs.

        then_I : int := OPTR$named_outdegree(then_exp) - 1
        then_R : int := OPTR$outdegree(then_exp) - then_I - 1
        if then_I < 0
        then    then_I := 0
        end

        % Define else_R and else_I the same way

        else_I : int := OPTR$named_outdegree(else_exp) - 1
        else_R : int := OPTR$outdegree(else_exp) - else_I - 1
        if else_I < 0
        then    else_I := 0
        end
```

```
% Find iter? outputs, if any

then_iter : int := OPTR$output_no(then_exp, "iter?")
else_iter : int := OPTR$output_no(else_exp, "iter?")

% Check preconditions

if then_R > 0  &  else_R > 0  &  then_R ~= else_R
 | then_iter > 0  &  then_I = 0
 | else_iter > 0  &  else_I = 0
 | then_I > 0  &  then_iter = 0
 | else_I > 0  &  else_iter = 0
 | then_I = 0  &  then_R = 0
 | else_I = 0  &  else_R = 0
then    signal bad_args
end

% Create if_graph and construct T and F gates that feed then
% and else clauses

if_graph: OPTR := OPTR$create_graph("if", descr)
OPTR$include(if_graph, if_exp)
OPTR$include(if_graph, then_exp)
OPTR$include(if_graph, else_exp)

for var: string in OPTR$input_names(then_exp)
do      t: OPTR := OPTR$create_primitive("T-Gate", DESC$[var])
        OPTR$attach(if_graph, if_exp, 1, t, 1)
        OPTR$name_input(t, 2, var)      % t defines var for then_exp
        OPTR$name_output(t, 1, var)
        OPTR$attach(if_graph, t, 1, then_exp,
                                        OPTR$input_no(then_exp, var))
end

for var: string in OPTR$input_names(else_exp)
do      f: OPTR := OPTR$create_primitive("F-Gate", DESC$[var])
        OPTR$attach(if_graph, if_exp, 1, f, 1)
        OPTR$name_input(f, 2, var)      % f defines var for else_exp
        OPTR$name_output(f, 1, var)
        OPTR$attach(if_graph, f, 1, else_exp,
                                        OPTR$input_no(else_exp, var))
end
```

```
% Now construct the iter? output for the whole graph if one is needed

ic: OPTR        % Ic generates graph iter? & M control outputs (if needed)
if else_iter > 0 | then_iter > 0
then    ic := OPTR$create_primitive("IC-Gate", DESC$["if_graph"])
        OPTR$attach(if_graph, if_exp, 1, ic, 1)     % if_exp controls ic

        % If if_exp true, take THEN iter?, or constant FALSE
        % if there is no THEN iter? output

        if then_iter > 0
        then    OPTR$attach(if_graph, then_exp, then_iter, ic, 2)
        else    OPTR$attach(if_graph, OPTR$create_primitive(
                                "constant", DESC$["false"]), 1, ic, 2)
        end

        if else_iter > 0    % Same as above for else clause
        then    OPTR$attach(if_graph, else_exp, else_iter, ic, 3)
        else    OPTR$attach(if_graph, OPTR$create_primitive(
                                "constant", DESC$["false"]), 1, ic, 3)
        end

        OPTR$name_output(ic, 1, "iter?")

        % If BOTH clauses have an iter? output . . .
        if then_iter > 0  &  else_iter > 0
        then    % . . . merge I results from both clauses

                % then clause
                for var: string in OPTR$output_names(then_exp)
                do      % ignore iter? output
                        if var = "iter?" then continue end

                        m: OPTR := OPTR$create_primitive("M-Gate",
                                                        DESC$[var])
                        OPTR$attach(if_graph, ic, 2, m, 1)
                        OPTR$attach(if_graph, then_exp,
                                OPTR$output_no(then_exp, var), m, 2)

                        k: int := OPTR$output_no(else_exp, var)

                        if k > 0
                        then    OPTR$attach(if_graph, else_exp, k, m, 3)
                        else    OPTR$name_input(m, 3, var)
                        end

                        OPTR$name_output(m, 1, var)
                end
```

```
                % else clause
                for var: string in OPTR$output_names(else_exp)
                do      k: int := OPTR$output_no(else_exp, var)

                        if  aic$size(OPTR$dests(else_exp, k)) = 0
                        then    % Output k is not connected to an M gate,
                                % so then clause has no <var> output.

                                m: OPTR := OPTR$create_primitive(
                                                "M-Gate", DESC$[var])
                                OPTR$attach(if_graph, ic, 2, m, 1)
                                OPTR$attach(if_graph, else_exp, k, m, 3)
                                OPTR$name_input(m, 2, var)
                                OPTR$name_output(m, 1, var)
                        end
                end

        else    % If only the then clause or the else clause has any I
                % outputs, they will become the named outputs of the graph
                % when it is sealed, so sink the ic merge control output

                OPTR$attach(if_graph, ic, 2, OPTR$create_primitive("sink",
                                                        DESC$new()), 1)
        end

else    % No iter? outputs at all, so no IC gate
end

%       Lastly, merge the R outputs, if any

if else_R > 0 & then_R > 0
then    % Preconditions demand that then_R = else_R, so iterate over
        % each clause's unnamed outputs in order, merging them.

        next_t: int := 1
        next_e: int := 1
        for i: int in int$from_to(1, then_R)
        do      % Find next unnamed then and else outputs . . .
                while OPTR$output_name(then_exp, next_t) ~= ""
                do      next_t := next_t + 1
                end

                while OPTR$output_name(else_exp, next_e) ~= ""
                do      next_e := next_e + 1
                end

                % . . . and merge them
```

```
                m: OPTR := OPTR$create_primitive("M-Gate",
                             DESC$["R"||int$unparse(i), "if_graph"])
                if then_iter > 0 | else_iter > 0
                then    OPTR$attach(if_graph, ic, 3, m, 1)
                else    OPTR$attach(if_graph, if_exp, 1, m, 1)
                end

                OPTR$attach(if_graph, then_exp, next_t, m, 2)
                OPTR$attach(if_graph, else_exp, next_e, m, 3)
            end

        elseif then_iter > 0 | else_iter > 0
        then    % Any unnamed outputs from either clause alone will become
                % the unnamed outputs from the graph when sealed, so sink
                % the ic merge control output

                OPTR$attach(if_graph, ic, 3, OPTR$create_primitive("sink",
                                                           DESC$new()), 1)
        end

        OPTR$seal(if_graph, DESC$new())

        return(if_graph)

    end make_if
```

```
make_let = proc(vars: ast, exps: aop, in_exp: OPTR, descr: DESC)
                                        returns(OPTR) signals(bad_args)

        % Returns the OPTR (graph) representation of a VAL let...in
        % construct with var[i] being set to exp[i] in in_exp.

        if ast$size(vars) ~= aop$size(exps)
        then    signal bad_args
        end

        let_exp: OPTR := OPTR$create_graph("let", descr)

        for i: int in ast$indexes(vars)
        do      % Construct binding expression by labelling exps[i] output
                OPTR$name_output(exps[i], 1, vars[i])

                % Include it in the let expression only if it is actually used
                inp: int := OPTR$input_no(in_exp, vars[i])

                if inp > 0
                then    OPTR$attach(let_exp, exps[i], 1, in_exp, inp)
                end

        end

        OPTR$seal(let_exp, DESC$new())
        return(let_exp)
end make_let
```

```
make_for = proc(vars: ast, exps: aop, iter_exp: OPTR, descr: DESC)
                                        returns(OPTR) signals(bad_args)

        % Returns the operator representation of a FOR loop with given
        % iteration variables (vars), initial values (exps), iteration body
        % (iter_exp), and description (descr).

        if ast$size(vars) ~= aop$size(exps)
        then    signal bad_args
        end

        iter_out: int := OPTR$output_no(iter_exp, "iter?")

        if iter_out > 0
        then    for_graph: OPTR := OPTR$create_graph("for", descr)

                % Merge I results from iteration subgraph with initial bindings
                for i: int in ast$indexes(vars)
                do      % For each iteration variable used by iter_exp, check
                        % if it is ever reset by an iter expression in iter_exp;
                        % if so, then merge it with its initial defn;
                        % if not, then it enters iter_exp via an FS gate

                        inp: int := OPTR$input_no(iter_exp, vars[i])
                        outp: int := OPTR$output_no(iter_exp, vars[i])

                        if inp > 0  &  outp > 0     % vars[i] is used and reset
                        then    fm: OPTR := OPTR$create_primitive(
                                                "FM-Gate", DESC$[vars[i]])
                                OPTR$attach(for_graph, iter_exp, iter_out, fm, 1)
                                OPTR$attach(for_graph, iter_exp, outp, fm, 2)
                                OPTR$name_output(exps[i], 1, vars[i])
                                OPTR$attach(for_graph, exps[i], 1, fm, 3)
                                OPTR$attach(for_graph, fm, 1, iter_exp, inp)
                        elseif inp > 0              % used but not reset
                        then    fs: OPTR := OPTR$create_primitive(
                                                "FS-Gate", DESC$[vars[i]])
                                OPTR$attach(for_graph, iter_exp, iter_out, fs, 1)
                                OPTR$name_output(exps[i], 1, vars[i])
                                OPTR$attach(for_graph, exps[i], 1, fs, 2)
                                OPTR$attach(for_graph, fs, 1, iter_exp, inp)
                        else    % if never used, dont do binding
                        end
                end
```

```
                % Now pass any other unreset inputs to iter_exp through FS gates
                for var: string in OPTR$input_names(iter_exp)
                do      inp: int := OPTR$input_no(iter_exp, var)
                        if OPTR$null_source(iter_exp, inp)
                        then    % still unconnected, so needs an FS gate
                                fs: OPTR := OPTR$create_primitive("FS-Gate",
                                                                   DESC$[var])
                                OPTR$attach(for_graph, iter_exp, iter_out, fs, 1)
                                OPTR$attach(for_graph, fs, 1, iter_exp, inp)
                                %       FS input will be graph input
                                OPTR$name_input(fs, 2, var)
                        end
                end

                OPTR$seal(for_graph, DESC$new())
                return(for_graph)

        else    % the iteration subgraph does not contain an iter expression
                % and should be treated simply as a let expression.
                return(make_let(vars, exps, iter_exp, descr))
        end

end make_for
```

1.5 Executing the Programs

These programs were written for the DECSYSTEM-20™ computer (under the TOPS-20 operating system) of the Laboratory for Computer Science at MIT. In this implementation CLU programs can be executed from a CLU "listen-loop" called CLUSYS. The CLUSYS allows for the definition of equates, loading of compiled CLU programs, invocation of CLU procedures and iterators, and immediate display of the results. A CLUSYS named *<name>* can be automatically invoked via a "*<name>*.EXE" file, which is invoked as a command from the terminal and controlled by a "_XFILE.*<name>*" file. This latter file contains lines to be typed to the CLUSYS as if they came directly from the terminal. Figure 20 shows the file "_XFILE.GRAPHS", which controls the execution of "GRAPHS.EXE" in this manner. Thus, the command "graphs" from the terminal will invoke a CLUSYS with the operator cluster and related programs loaded, and useful equates (abbreviations) defined.

The programs were tested in a similar way, by creating a batch control file that invoked the "graphs" command and made various calls to the operations of the cluster, keeping a log of the results. This control file was then executed whenever any change was made in the programs.

The program "documt", mentioned in Fig. 20, allows the "graphs" command to be invoked as "graphs help", causing a brief display of documentation on the function of the command. The "add_script" procedure causes a record of the requests (and responses) typed to the CLUSYS to be kept in the file "GRAPHS.SAVE".

Fig. 20. Invoking the programs -- file _XFILE.GRAPHS.

```
load("oper")
load("acat")
load("table")
load("optabl")
load("tester")
load("clu:encdec")
load("clu:istream")
load("gjcl20")
load("documt")
jcl = get_jcl()
document(jcl, "graphs")
optr = operator
inconn = record[op:operator, inp:int]
outconn = record[op:operator, outp:int]
aoc = array[outconn]
aic = array[inconn]
desc = array[string]
ast = array[string]
aop = array[optr]
nd = desc$new()
savef = file_name$parse("graphs.save")
save = stream$open(savef, "write")
tty = stream$primary_output()
stream$add_script(tty, save)
```

1.6 Sample Execution

The following is a sample of the lines typed to the "graphs" CLUSYS (preceded by a colon) and the responses to those lines. Values returned from procedures are preceded by an arrow. Some of the returned values have been deleted because they are unreadable; for example, when an array or record structure (or cluster whose representation is such a structure) is displayed by the CLUSYS, it is not displayed in conveniently-readable form, as the response to the first line shows.

```
: plus = optr$create_primitive("+", nd)
=> [I [I: [I:]] [I:] 0...]
```

```
: optr$free(plus)
=> true

: optr$is_sealed(plus)
=> true

: optr$is_graph(plus)
=> false

: optr$is_primitive(plus)
=> true

: optr$indegree(plus)
=> 2

: optr$outdegree(plus)
=> 1

: optr$subinput(plus,1)
Signals: not_graph


: optr$write(plus, tty)
"+"  description: []
  inputs: 2
  outputs: 1
  acknowledge inputs: 1
  acknowledge outputs: 1
  acknowledges expected: 0
  acknowledges initially received: 0
=> ...

: optr$input_names(plus)


: optr$output_names(plus)


: optr$input_name(plus,1)
=> ""

: optr$output_name(plus,3)
Signals: outputs_range


: optr$input_name(plus, -1)
Signals: inputs_range
```

```
: optr$input_no(plus, "foo")
=> 0

: optr$get_owner(plus)
Signals: free_operator


: optr$get_opname(plus)
=> "+"

: optr$get_id(plus)
=> 0

: optr$get_acks_expected(plus)
=> 0

: optr$get_acks_received(plus)
=> 0

: optr$get_description(plus)
=> [1:]

: optr$name_input(plus, 2, "x")
=> ...

: optr$write(plus, tty)
"+"  description: []
  inputs: 2  names: "x"(2)
  outputs: 1
  acknowledge inputs: 1
  acknowledge outputs: 1
  acknowledges expected: 0
  acknowledges initially received: 0
=> ...

: g = optr$create_graph("test-graph", desc$["will contain PLUS"])
=> ...

: optr$is_graph(g)
=> true

: optr$is_primitive(g)
=> false

: optr$is_sealed(g)
=> false

: optr$free(g)
```

```
=> true

: optr$get_description(g)
=> ...

: optr$components(g)
Signals: unsealed


: optr$input_names(g)
Signals: unsealed


: optr$name_input(g,1,"foo")
Signals: unsealed


: optr$include(g, plus)
=> ...

: optr$free(plus)
=> false

: optr$free(g)
=> true

: optr$equal(g, optr$get_owner(plus))
=> true

: optr$attach(g, plus, 1, optr$create_primitive("*", nd), 1)
=> ...

: times := inconn$get_op(aic$fetch(optr$dests(plus, 1), 1))
=> ...

: optr$write(times, tty)
"*"  description: []
  inputs: 2
  outputs: 1
  acknowledge inputs: 1
  acknowledge outputs: 1
  acknowledges expected: 0
  acknowledges initially received: 0
=> ...

: optr$name_input(times, 2, "x")
=> ...
```

```
: optr$seal(g, desc$["and TIMES"])
=> ...

: optr$write(g, tty)
graph "test-graph"  description: ["will contain PLUS", "and TIMES"]
  inputs(2): {op1•1}, {op1•2, op2•2}("x")
  outputs(1): op2•1
  acknowledge inputs(0)
  acknowledge outputs(0)
  components:
    op1: "+"  description: []
      inputs(2) attached: <graph input>, <graph input>("x")
      outputs(1) attached: {op2•1}
      acknowledge inputs: 1
      acknowledge outputs(1): not sent
      acknowledges expected: 0
      acknowledges initially received: 0
    op2: "*"  description: []
      inputs(2) attached: op1•1, <graph input>("x")
      outputs(1) attached: <graph output>
      acknowledge inputs: 1
      acknowledge outputs(1): not sent
      acknowledges expected: 0
      acknowledges initially received: 0
=> ...
```

Adding acknowledge arcs to the above graph can be done as follows
(First get an unsealed copy of the graph)

```
: g2 = optr$absorb(optr$create_graph("test-graph", desc$["(copy)"]), optr$copy(g, nd))
=> ...
```

Original is unchanged since a copy was absorbed:

```
: optr$write(g, tty)
graph "test-graph"  description: ["will contain PLUS", "and TIMES"]
  inputs(2): {op1•1}, {op1•2, op2•2}("x")
  outputs(1): op2•1
  acknowledge inputs(0)
  acknowledge outputs(0)
  components:
    op1: "+"  description: []
      inputs(2) attached: <graph input>, <graph input>("x")
      outputs(1) attached: {op2•1}
      acknowledge inputs: 1
      acknowledge outputs(1): not sent
      acknowledges expected: 0
      acknowledges initially received: 0
```

```
    op2: "*"  description: []
      inputs(2) attached: op1•1, <graph input>("x")
      outputs(1) attached: <graph output>
      acknowledge inputs: 1
      acknowledge outputs(1): not sent
      acknowledges expected: 0
      acknowledges initially received: 0
=> ...

: optr$is_sealed(g2)
=> false

: plus2 = optr$fetch(g2, 1)
=> ...

: optr$write(plus2, tty)
"+"  description: []
  inputs: 2  names: "x"(2)
  outputs: 1
  acknowledge inputs: 1
  acknowledge outputs: 1
  acknowledges expected: 0
  acknowledges initially received: 0
=> ...

: times2 = optr$fetch(g2, 2)
=> ...

: optr$write(times2, tty)
"*"  description: []
  inputs: 2  names: "x"(2)
  outputs: 1
  acknowledge inputs: 1
  acknowledge outputs: 1
  acknowledges expected: 0
  acknowledges initially received: 0
=> ...

: optr$acknowledge(g2, times2, 1, plus2, 1)
=> ...

: optr$get_acks_expected(plus2)
=> 1

: optr$make_ack_output(g2, plus2, 1)
=> ...

: optr$make_ack_output(g2, times2, 1)
```

```
=> ...

: optr$make_ack_input(g2, times2, 1)
=> ...

: optr$seal(g2, nd)
=> ...

: optr$write(g2, tty)
graph "test-graph"  description: ["(copy)"]
  inputs(2): {op1•1}, {op1•2, op2•2}("x")
  outputs(1): op2•1
  acknowledge inputs(1): op2•1
  acknowledge outputs(2): op1•1, op2•1
  components:
   op1: "+"  description: []
     inputs(2) attached: <graph input>, <graph input>("x")
     outputs(1) attached: {op2•1}
     acknowledge inputs: 1
     acknowledge outputs(1): not sent<graph acknowledge 1>
     acknowledges expected: 1
     acknowledges initially received: 0
   op2: "*"  description: []
     inputs(2) attached: op1•1, <graph input>("x")
     outputs(1) attached: <graph output>
     acknowledge inputs: 1
     acknowledge outputs(1): sent to: {op1•1}<graph acknowledge 2>
     acknowledges expected: 0
     acknowledges initially received: 0
=> ...
```

To demonstrate the removal of an operator:

```
: g3 = optr$absorb(optr$create_graph("test-graph",desc$["(copy 2)"]), optr$copy(g2, nd))
=> ...

: times3 = optr$fetch(g3, 2)
=> ...

: optr$remove(g3, times3)
=> ...

: optr$seal(g3, nd)
=> ...

: optr$write(g3, tty)
graph "test-graph"  description: ["(copy 2)"]
  inputs(2): {op1•1}, {op1•2}("x")
```

```
  outputs(1): op1•1
  acknowledge inputs(0)
  acknowledge outputs(1): op1•1
  components:
    op1: "+"  description: []
      inputs(2) attached: <graph input>, <graph input>("x")
      outputs(1) attached: <graph output>
      acknowledge inputs: 1
      acknowledge outputs(1): not sent<graph acknowledge 1>
      acknowledges expected: 0
      acknowledges initially received: 0
=> ...

: optr$write(times3, tty)
"*"  description: []
  inputs: 2  names: "x"(2)
  outputs: 1
  acknowledge inputs: 1
  acknowledge outputs: 1
  acknowledges expected: 0
  acknowledges initially received: 0
=> ...

: optr$free(times3)
=> true
```

The following demonstrates the construction of the FOR loop of the examples (without acknowledge arcs defined)

```
: exp2 = optr$create_primitive(">",desc$["i>n"])
=> ...

: optr$name_input(exp2,1,"i")
=> ...

: optr$name_input(exp2,2,"n")
=> ...

: exp3 = optr$create_primitive("1",desc$["s"])
=> ...

: optr$name_input(exp3, 1, "s")
=> ...

: iterexp = optr$create_graph("iter-exp", desc$["i,s:=i+1,s+i"])
=> ...

: p1 = optr$create_primitive("+",nd)
```

```
=> ...

: p2 = optr$create_primitive("+",nd)
=> ...

: optr$attach(iterexp, optr$create_primitive("constant",desc$["1"]),1,p1,1)
=> ...

: optr$name_input(p1, 2, "i")
=> ...

: optr$name_input(p2, 1, "i")
=> ...

: optr$name_input(p2, 2, "s")
=> ...

: optr$include(iterexp, p2)
=> ...

: optr$name_output(p1,1,"i")
=> ...

: optr$name_output(p2,1,"s")
=> ...

: i = optr$create_primitive("constant", desc$["true"])
=> ...

: optr$include(iterexp, i)
=> ...

: optr$name_output(i, 1, "iter?")
=> ...

: optr$seal(iterexp,nd)
=> ...

: optr$write(iterexp, tty)
graph "iter-exp"  description: ["i,s:=i+1,s+i"]
  inputs(2): {op2•2, op3•1}("i"), {op3•2}("s")
  outputs(3): op2•1("i"), op3•1("s"), op4•1("iter?")
  acknowledge inputs(0)
  acknowledge outputs(0)
  components:
    op1: "constant"  description: ["1"]
      inputs(0)
      outputs(1) attached: {op2•1}
```

```
    acknowledge inputs: 1
    acknowledge outputs(1): not sent
    acknowledges expected: 0
    acknowledges initially received: 0
  op2: "+"  description: []
    inputs(2) attached: op1•1, <graph input>("i")
    outputs(1) attached: <graph output>("i")
    acknowledge inputs: 1
    acknowledge outputs(1): not sent
    acknowledges expected: 0
    acknowledges initially received: 0
  op3: "+"  description: []
    inputs(2) attached: <graph input>("i"), <graph input>("s")
    outputs(1) attached: <graph output>("s")
    acknowledge inputs: 1
    acknowledge outputs(1): not sent
    acknowledges expected: 0
    acknowledges initially received: 0
  op4: "constant"  description: ["true"]
    inputs(0)
    outputs(1) attached: <graph output>("iter?")
    acknowledge inputs: 1
    acknowledge outputs(1): not sent
    acknowledges expected: 0
    acknowledges initially received: 0
=> ...

: ifg = make_if(exp2, exp3, iterexp, desc$["if i>n then s else iter i,s:=i+1,s+i"])
=> ...

: optr$write(ifg, tty)
graph "if"  description: ["if i>n then s else iter i,s:=i+1,s+i"]
  inputs(3): {op1•1, op5•2}("i"), {op1•2}("n"), {op4•2, op6•2}("s")
  outputs(4): op2•1, op3•1("i"), op3•2("s"), op7•1("iter?")
  acknowledge inputs(0)
  acknowledge outputs(0)
  components:
   op1: ">"  description: ["i>n"]
    inputs(2) attached: <graph input>("i"), <graph input>("n")
    outputs(1) attached: {op4•1, op5•1, op6•1, op7•1}
    acknowledge inputs: 1
    acknowledge outputs(1): not sent
    acknowledges expected: 0
    acknowledges initially received: 0
   op2: "I"  description: ["s"]
    inputs(1) attached: op4•1("s")
    outputs(1) attached: <graph output>
    acknowledge inputs: 1
```

```
  acknowledge outputs(1): not sent
  acknowledges expected: 0
  acknowledges initially received: 0
op3: graph "iter-exp"  description: ["i,s:=i+1,s+i"]
  inputs(2) attached: op5•1("i"), op6•1("s")
  outputs(3) attached: <graph output>("i"), <graph output>("s"), {op7•3}("iter?")
  acknowledge inputs: 0
  acknowledge outputs(0)
op4: "T-Gate"  description: ["s"]
  inputs(2) attached: op1•1, <graph input>("s")
  outputs(1) attached: {op2•1}("s")
  acknowledge inputs: 1
  acknowledge outputs(1): not sent
  acknowledges expected: 0
  acknowledges initially received: 0
op5: "F-Gate"  description: ["i"]
  inputs(2) attached: op1•1, <graph input>("i")
  outputs(1) attached: {op3•1}("i")
  acknowledge inputs: 1
  acknowledge outputs(1): not sent
  acknowledges expected: 0
  acknowledges initially received: 0

op6: "F-Gate"  description: ["s"]
  inputs(2) attached: op1•1, <graph input>("s")
  outputs(1) attached: {op3•2}("s")
  acknowledge inputs: 1
  acknowledge outputs(1): not sent
  acknowledges expected: 0
  acknowledges initially received: 0
op7: "IC-Gate"  description: ["if_graph"]
  inputs(3) attached: op1•1, op8•1, op3•3("iter?")
  outputs(3) attached: <graph output>("iter?"), {op9•1}, {op10•1}
  acknowledge inputs: 1
  acknowledge outputs(1): not sent
  acknowledges expected: 0
  acknowledges initially received: 0
op8: "constant"  description: ["false"]
  inputs(0)
  outputs(1) attached: {op7•2}
  acknowledge inputs: 1
  acknowledge outputs(1): not sent
  acknowledges expected: 0
  acknowledges initially received: 0
op9: "sink"  description: []
  inputs(1) attached: op7•2
  outputs(0)
  acknowledge inputs: 1
```

```
      acknowledge outputs(1): not sent
      acknowledges expected: 0
      acknowledges initially received: 0
    op10: "sink"  description: []
      inputs(1) attached: op7•3
      outputs(0)
      acknowledge inputs: 1
      acknowledge outputs(1): not sent
      acknowledges expected: 0
      acknowledges initially received: 0
=> ...

: vars = ast$["i", "s"]
=> [1: "i" "s"]

: exps = aop$[optr$create_primitive("constant", desc$["1"]),
                  optr$create_primitive("constant", desc$["0"])]
=> ...

: make_for(vars, exps, ifg, desc$["entire for loop"])
=> ...

: forg = optr$get_owner(ifg)
=> ...

: optr$write(forg, tty)
graph "for"  description: ["entire for loop"]
  inputs(1): {op6•2}("n")
  outputs(1): op1•1
  acknowledge inputs(0)
  acknowledge outputs(0)
  components:
    op1: graph "if"  description:["if i>n then s else iter i,s:=i+1,s+i"]
      inputs(3) attached: op2•1("i"), op6•1("n"), op4•1("s")
      outputs(4) attached: <graph output>, {op2•2}("i"), {op4•2}("s"),
                                                    {op2•1, op4•1, op6•1}("iter?")
      acknowledge inputs: 0
      acknowledge outputs(0)
    op2: "I-M-Gate"  description: ["i"]
      inputs(3) attached: op1•4("iter?"), op1•2("i"), op3•1("i")
      outputs(1) attached: {op1•1}("i")
      acknowledge inputs: 1
      acknowledge outputs(2): not sent, not sent
      acknowledges expected: 0
      acknowledges initially received: 0
    op3: "constant"  description: ["1"]
      inputs(0)
      outputs(1) attached: {op2•3}("i")
```

```
      acknowledge inputs: 1
      acknowledge outputs(1): not sent
      acknowledges expected: 0
      acknowledges initially received: 0
    op4: "FM-Gate"  description: ["s"]
      inputs(3) attached: op1•4("iter?"), op1•3("s"), op5•1("s")
      outputs(1) attached: {op1•3}("s")
      acknowledge inputs: 1
      acknowledge outputs(2): not sent, not sent
      acknowledges expected: 0
      acknowledges initially received: 0
    op5: "constant"  description: ["0"]
      inputs(0)
      outputs(1) attached: {op4•3}("s")
      acknowledge inputs: 1
      acknowledge outputs(1): not sent
      acknowledges expected: 0
      acknowledges initially received: 0

    op6: "FS-Gate"  description: ["n"]
      inputs(2) attached: op1•4("iter?"), <graph input>("n")
      outputs(1) attached: {op1•2}("n")
      acknowledge inputs: 1
      acknowledge outputs(1): not sent
      acknowledges expected: 0
      acknowledges initially received: 0
=> ...

: bye()
```

References

[1] Ackerman, W. B. "A Structure Memory for Data Flow Computers", Laboratory for Computer Science (TR-186), M.I.T., Cambridge, Mass., August, 1977.

[2] Ackerman, W. B., Dennis, J. B. "VAL -- A Value-oriented Algorithmic Language: Preliminary Reference Manual", Computation Structures Group, Laboratory for Computer Science, M.I.T., Cambridge, Mass. In preparation.

[3] Ackerman, W. B., and Brock., J. D. Private communication.

[4] Brock, J. D. *Operational Semantics of a Data Flow Language*, Laboratory for Computer Science (TM-120), M.I.T., Cambridge, Mass., December 1978.

[5] Brock, J. D., and Montz, L. "Translation and Optimizations of Data Flow Programs", to appear in *Proceedings 1979 International Conference on Parallel Processing*, August, 1979.

[6] "Data Flow Computer Architecture", Computation Structures Group (Memo 160), Laboratory for Computer Science, M.I.T., Cambridge, Mass. May 1978.

[7] Dennis, J. B., Fosseen, J. B. "Introduction to Data Flow Schemas", Computation Structures Group (Memo 81-1), Laboratory for Computer Science, M.I.T., Cambridge, Mass. September 1973.

[8] Dennis, J. B., Misunas, D. P., Leung, C. K. "A Highly Parallel Processor Using a Data Flow Machine Language", Computation Structures Group (Memo 134), Laboratory for Computer Science, M.I.T., Cambridge, Mass. January 1977.

[9] Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C. "Abstraction Mechanisms in CLU", *Comm. of the ACM 20*, 8 (August, 1977), 564-576.

[10] Liskov, B., Moss, E., Schaffert, C., Scheifler, R., and Snyder, A. "CLU Reference Manual", Computation Structures Group (Memo 161), Laboratory for Computer Science, M.I.T., Cambridge, Mass. July 1978.

[11] Montz, L. *Safety and Optimization Transformations for Data Flow Programs*, S. M. Thesis, Dept. of Electrical Engineering and Computer Science, M.I.T., Cambridge, Mass. In preparation.

[12] Weng, K. S. *An Abstract Implementation For A Generalized Data Flow Language*, Ph.D. Thesis, Dept. of Electrical Engineering and Computer Science, M.I.T., Cambridge, Mass. May, 1979.