A Re-Usable Code Generator for Prime 50-Series Computers

User's Guide

T. Allen Akin

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

March, 1983

TABLE OF CONTENTS

Foreword

Although the School of Information and Computer Science has operated Prime 400 and 550 computers for over four years, as yet there has been no successful local attempt to produce a compiler for them. The main reasons for this failure are the irregularity of the architecture and existing system software, the complexity of Prime's standard object code format, and the lack of documentation on matters of importance to compiler writers.

This paper discusses the design, implementation, and usage of a re-usable code generator. This program can serve as a common "back-end" for a number of language translators, producing 64V-mode assembly language code suitable for execution on the P400 and higher numbered processors in Prime's "50" series. Furthermore, it could be tailored to match specific front-ends, when needs for special optimizations arise.

A preliminary version of the code generator is available for general use.

How to Use This Guide

The first chapter of this **Guide** is the Overview. The Overview is a brief summary of the design and construction of the code generator. This chapter may be of general interest, but it is not necessary to read it in order to learn to use the code generator.

The Code Generator Usage chapter describes the location of the code generator and its associated run-time support libraries, as well as the Software Tools Subsystem commands necessary to access them. Recommended procedure is to study this section, then generate command language programs to do the low-level file access operations.

Input Data Stream Formats gives a bird's-eye view of the formats of the three code generator input streams. This chapter merits some study, although it is supplemented by the Extended Examples.

The three operator definitions chapters (Operators Useful in the Static Data Stream, Operators Useful in the Procedure Definition Stream, Operators Useful in Procedure Definitions) provide a detailed reference for the intermediate form operators interpreted by the code generator. One or two readings through this chapter are desirable; thereafter, it can be used as a reference with the Operator/Function Index and the Table of Contents used as entry points.

The Extended Examples are comprised of several short (but complete) programs written in the language C. These examples include the original C code, annotated versions of the three code generator input streams, and an annotated listing of the code generator's assembly language output. The chapter should be useful in learning how the various intermediate form operators work together, and may be used as a reference when building a new front end.

'Drift' is a very small expression-based language whose structure closely mimics the code generator's internal world-model. The 'Drift' Compiler is a complete, working compiler using the code generator as a back-end. It serves as an example of one way to construct a front-end for the VCG.

For ease of reference, all the intermediate form operators have been organized by subject in the Intermediate Form Operator/Function Index. Typically, one would look up some function (e.g., "subscripting") in the Index, find the name of the appropriate intermediate form operator (e.g., INDEX_OP), then look up that operator in the table of contents to find its complete description.

Overview

Philosophy

Design Considerations

The design of the code generator (hereinafter referred to as VCG, for "V-mode Code Generator") was driven by a number of considerations:

- For experimental language translators, code generation should be fast and straightforward. This is necessary both for fast turnaround and ease of debugging in the development stage, and for fast turnaround in typical educational applications.

- The VCG should insulate front-ends from details of storage allocation and data format selection, as well as instruction generation. This encourages inter-language compatibility at the object code level, as well as providing a framework for easily retargetable front-ends.

- The intermediate form (IMF) that is processed by the VCG should be simple to generate and display (for debugging purposes). Furthermore, it should not unduly restrict extension for additional functionality or optimization.

- The output object code should conform to Prime's current standards, and should include at least minimal provisions for separate compilation and run-time debugging.

Implementation Approaches

After some time, consideration of the goals above led to the following approaches to the implementation of the VCG:

- The basic IMF handled by both the front end and the VCG should be a tree structure. A tree is easily generated from the information available on the semantic stack during a bottom-up parse, and can be generated directly without an explicit stack during a top-down parse. A number of operations like constant folding, reordering of operands of commutative operators, and global context propagation are readily performed on a tree structure. Furthermore, use of a tree can eliminate the need for generation and tracking of temporary variables in the front end.

- The IMF operators should be close to the constructs used in an algorithmic language of the level of, say, Pascal. This permits straightforward translation of most algorithmic

languages, and provides enough additional context to simplify many optimization tasks. For example, the IMF resembles the program's flow graph closely enough that simple global register allocation can be performed without graph reduction.

. One of the basic functions of the VCG is the mapping of data descriptions supplied by the front end into physical storage layouts. The goal of complete machine data structure independence in the front end cannot be met without compromising the code generator's utility for languages that allow storage layout specification (C and Ada are notable examples). Therefore, the IMF should contain descriptions of data structures in terms of a small set of primitive data modes that can easily be parameterized in front-end code. Simple variables, structures, and arrays defined in the front end must be converted to single or multiple instances of the following basic machine data modes: 16-bit signed integer, 16-bit unsigned integer, 32-bit signed integer, 32-bit unsigned integer, 32-bit floating point, and 64-bit floating point.

. The IMF tree should be linearized and passed to the VCG as a stream of data in prefix Polish notation. The linearized form partly reflects the usual Software Tools methodology of expressing even complex data transformations as "filters." However, there are other advantages, particularly in storing and interpreting the IMF for debugging. Prefix Polish was chosen because it can be generated easily from the internal representation of the tree, and because it minimizes the amount of state information that must be explicitly maintained by both the front end and the VCG in order to output or input the IMF.

. The final output of the VCG should be a stream of Prime Macro Assembly Language source code. Although the time required to assemble this source imposes a significant penalty on code generator performance, it appears to be unavoidable if the compiler writer is to be insulated from Prime's object code format. (In addition, Prime has scheduled object code format changes, and it would not be wise to invest heavily in the present format.)

Structure

The VCG "main loop" simply reads each module present on its input, rebuilds the tree represented by the input, transforms the tree to a linked list of machine instructions, performs register tracking optimizations on that list, and finally converts the list to assembly language and outputs it.

The input and output routines are straightforward and relatively uninteresting.

The optimization routines amount to about 13 pages of Ratfor code, and work by simulating the effect of each machine instruction on the contents of the six registers that are tracked. At the moment, three types of optimization are performed: redundant loads are eliminated, some memory references are eliminated in favor of register-to-register transfers, and general instruction sequences are replaced with special-case code.

The heart of the code generator is the set of transformation routines that convert the tree representation to the doubly-linked list of machine instructions. The transformation routines exhibit a great deal of knowledge about the machine architecture, but actually employ only very simple algorithms for code generation.

IMF operators may appear in one of several "contexts," identified internally by the following terms:

Reach. An operator evaluated in reach context yields the address of a word in memory containing the result of the operation, if possible. At present, only the object, constant, pointer dereferencing, array indexing, and structure member selection operators yield addresses. All other operators behave as if they were evaluated in "load" context.

Load. An operator evaluated in load context yields a value in a machine register. The particular register used depends only on the basic machine data mode of the operation. Most IMF operators are evaluated only in this context.

Void. An operator evaluated in void context yields side effects only. In a very few cases, this results in an opportunity to exploit special-case machine instructions that perform some calculation without making the result available in a register (incrementing a memory location, for example).

Flow. An operator evaluated in flow context yields a change in flow-of-control rather than a value. For example, a "test for equality" operator would return 1 or 0 in a load context, but in flow context would cause a jump to a given label depending on the outcome of the test.

AP. An operator evaluated in AP context yields an "argument pointer" rather than a value. Argument pointers are used to pass parameters to procedures.

Context information is propagated top-down by the code generator as it scans the IMF tree. Additional information in the form of register requirements is propagated from the bottom up during the same scan. Together, context and register usage determine with fair accuracy the optimal code sequence to be generated for a given operator.

Input/Output Semantics

Input Structure

The IMF passed to the VCG consists of a sequence of modules. A module is a sequence of procedure definitions, static data definitions, and entry point declarations. The static data definitions build a data area that is shared by all procedures in the module, while the procedure definitions build code and data areas that are strictly local to each procedure, and the entry point declarations make the static data area or procedures visible to Prime's link editor.

Prime's Fortran compilers currently generate code that is equivalent to one procedure per module under this scheme; Prime's PL/I and Pascal compilers generate code that is equivalent to a single IMF module. The VCG module structure permits compatibility with either of these alternatives, as well as compromise forms that are more suitable for other languages.

> Note: Separate compilation capability directly affects module structure. At present, there is no way for separately compiled procedures to share a static data area. Furthermore, separately-compiled static objects must be referenced by a unique 8 or fewer character name made visible to the loader. A Fortran COMMON block definition can be used to reduce the number of such external symbols, but COMMON definitions must match exactly in all separately-maintained modules. In addition, note that Prime's current loader software requires that external objects be referenced through an indirect address, which can cause a significant reduction in performance.

Each static data definition allocates space for an object and may specify an initial value for the object. A static data declaration names an object that is defined outside the current module, but provides no other information about the object.

Each procedure definition consists of information associated with a closed routine defined by the front end. In particular, the procedure's argument types and code tree are included.

The bulk of the IMF will be in subtrees defining the code associated with procedures. Most storage allocators, arithmetic operators, and flow controllers are straightforwardly expressed in tree form; a description of these IMF components is available elsewhere.

Output Structure

Each VCG input module generates a single PMA input module, terminated by an END pseudo-op. The PMA input module may be

assembled, link-edited, and subsequently executed. The concatenation of all static data definitions and declarations forms a link frame that is shared by all procedures in the module. Each procedure definition yields an entry control block (ECB) and a chunk of machine code that implements the function of the procedure, including the allocation of space in the procedure's stack frame for local variables.

Code Generator Usage

The code generator currently resides in the file =bin=/vcg. The three input streams can be read from the three standard inputs, or from three files (if a standard naming convention is used). The PMA output stream is produced on standard output 1, and should be redirected to a file for assembly.

Assume temporary files will be used for communications between the front end and the code generator. The temporary files must have names of the form "xxx.ct1" (for IMF stream 1), "xxx.ct2" (for IMF stream 2), and "xxx.ct3" (for IMF stream 3), where "xxx" is completely arbitrary but must be the same for all of the three temporary files in a given run. When the code generator is invoked, the string "xxx" must be passed to it as a command line argument.

To use the code generator, first run the front end to produce the temporary files:

```
front_end
```

Say, for example, this produces files "temp.ct1", "temp.ct2", and "temp.ct3". Next, run the code generator and produce the assembly language output:

```
vcg temp >temp.s
```

Run the assembler to convert the PMA source to relocatable binary code:

```
pmac temp.s
```

Finally, run the link editor to load the VCG main program, the binary code for your program, and all required library routines:

```
ld =lib=/vcg_main temp.b =lib=/vcglib -o program
```

This produces an object program (in the file "program") which may be executed simply by typing its name:

```
program
```

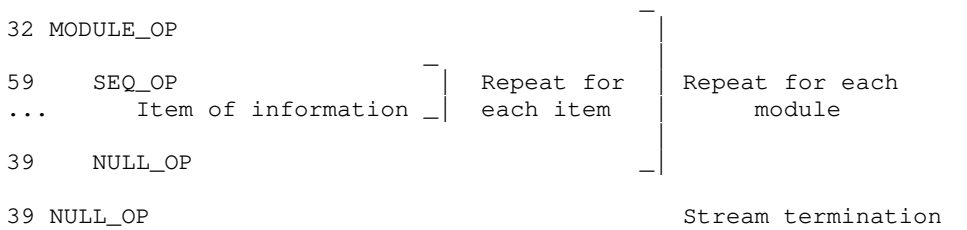
All run-time support routines called by the output of the code generator are available in the library =lib=/vcglib. The stub main program in =lib=/vcg_main calls a procedure named MAIN; therefore, the user's main program must be named MAIN. (This is the usual case in C environments.)

One miscellaneous note: if the front end is being written in Ratfor, the complete set of macro definitions for the intermediate form operators can be obtained by simply including

the file "=incl=/vcg_defs.r.i". If the front end is being writ-
ten in Pascal, the complete set of constant definitions for the
intermediate form operators can be obtained by including the file
"=incl=/vcg_defs.p.i".

Input Data Stream Formats

This section describes the formats of the three code generator input streams. Note that all three have the same basic format:



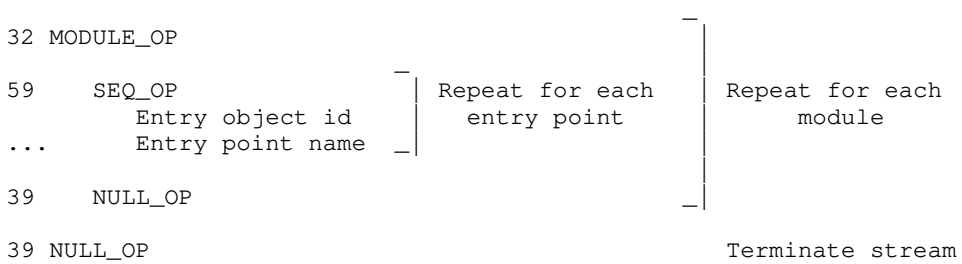
Detailed examples of the code generator input can be found in the "Extended Examples" section of this guide.

Stream 1 --- Entry Point Declarations

The first intermediate form stream consists of one or more modules. Each module consists of a MODULE_OP, a list of entry point declarations separated by SEQ_OPs, and a NULL_OP terminating the list of entry point declarations. The list of modules is terminated by a final NULL_OP.

Each entry point declaration is an object identification number followed by a character string, expressed as the length of the string followed by the ASCII character codes for the characters in the string. Each such string is assumed to be the name of a location defined in the current input module, and is made available to the link editor for resolving references made by other modules.

A template for stream 1 would look something like this:

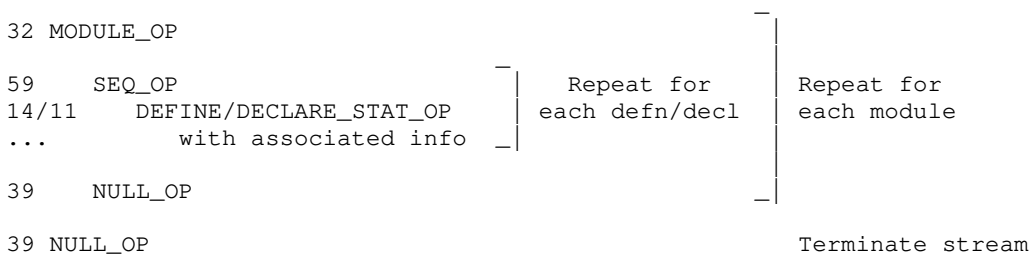


Stream 2 --- Static Data Declarations/Definitions

In C terminology, a data "definition" reserves storage space for an object and possibly initializes that space, whereas a data "declaration" simply indicates that the storage space for an object resides outside the current module. The second intermediate form input stream defines or declares static data (objects that are not automatically allocated on the stack when a procedure is entered).

The input stream consists of a series of modules, terminated by a NULL_OP. Each module contains a sequence of DEFINE STAT OPs and DECLARE STAT OPs, terminated by a NULL_OP.

A template for the static data stream would look something like this:

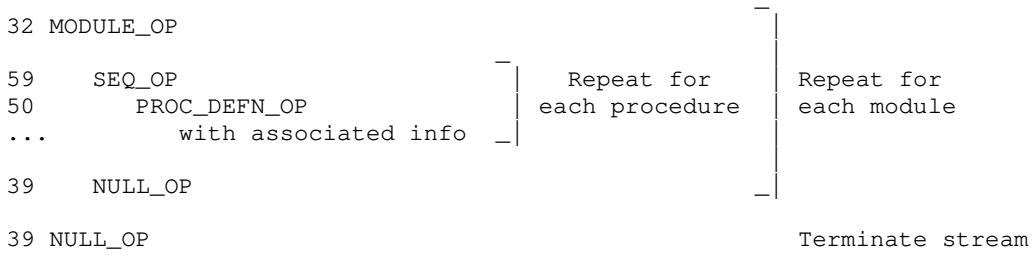


Stream 3 --- Procedure Definitions

The third intermediate form input stream consists of one or more modules, terminated by a NULL_OP. Each module contains a list of PROC DEFN OPs, separated by SEQ_OPs and terminated with a NULL_OP.

Each PROC_DEFN_OP causes a procedure to be defined and code for it to be generated.

A template for stream 3 would look something like this:



Primitive Data Modes

The following primitive data modes are presently handled by the code generator:

INT_MODE 1

Integer objects are one 16-bit word in size. They have integral values in the range (-2**15) to (2**15 - 1), inclusive.

LONG_INT_MODE 2

Long integer objects are two 16-bit words in size. They have integral values in the range (-2**31) to (2**31 - 1), inclusive.

UNS_MODE 3

Unsigned objects are nominally one 16-bit word in size. They have integral values in the range 0 to (2**16 - 1). Bit fields (see FIELD_OP) can be of mode UNSIGNED, and may range from 1 bit to 16 bits in length (with consequent change in the range of values they can represent).

LONG_UNS_MODE 4

Long unsigned objects are nominally two 16-bit words in size. They have integral values in the range 0 to (2**32 - 1). Machine addresses (pointers) are represented as long unsigned quantities. Bit fields (see FIELD_OP) can be of mode LONG UNSIGNED, and may range from 1 bit to 32 bits in length (with consequent change in the range of values they can represent).

FLOAT_MODE 5

Floating point objects are two 16-bit words in size.

LONG FLOAT_MODE 6

Long floating point objects are four 16-bit words in size.

STOWED_MODE 7

STOWED mode is the mode assigned to structured objects like arrays and structs (Pascal "records"). STOWED objects may be any size from 1 to 65536 16-bit words; IMF operators that need to know the size of a STOWED object invariably have a "length" or "size" parameter to carry that information.

Operators Useful in the Static Data Stream

DECLARE_STAT_OP 11

```
int 11
int object_id
string external_name
```

DECLARE_STAT informs the code generator that an object defined outside the current module will be referenced by a given integer object id. The parameter 'external_name' is a character string, represented in the IMF by a length followed by a stream of ASCII characters (one per word, right justified, zero filled). The external name is used by the link editor and the loader to resolve actual references to the object.

Example: extern int abc
where 'abc' is assigned the object id 6

```
11       DECLARE_STAT_OP
6           Object id of 'abc'
3           Length of name 'abc'
225            character 'a'
226            character 'b'
227            character 'c'
```

DEFINE_STAT_OP 14

```
int 14
int object_id
tree init_list
int size
```

This operator causes storage for the object identified by 'object_id' to be allocated in the current link frame (static data area). 'Object_id' must be used in all subsequent references to the object, and the object's definition with DEFINE_STAT must precede all such references. The init_list is a list of initializers whose values will be assigned to successive portions of the newly-declared object. The size parameter specifies the amount of storage to be reserved for the object, in words. (Slightly fewer than 65,535 words are available for static storage in each module.)

Example: static int abc[100]
where abc is assigned the object id 6

```
14       DEFINE_STAT_OP
6           Object id for 'abc'
39          NULL_OP (no initializers present)
```

```
100         Object is 100 words long
```

Operators Useful in the Procedure Definition Stream

PROC_DEFN_OP 50

```
int 50
int object_id
int number_of_args
string proc_name
tree argument_list
tree code
```

Each procedure to be generated by the code generator is defined by a PROC_DEFN_OP. The 'object_id' is an integer identifier that must be used on calls to the procedure and other references to its entry control block (for example, pointers to functions as used in C). 'Number_of_args' should be self-explanatory. 'Proc_name' is a string (in the IMF, a length followed by ASCII character values) giving the internal name of the procedure. (This information is used to print trace information during debugging.) Each formal parameter (argument) is described by a PROC_DEFN_ARG_OP; 'argument_list' is simply a linked list of those descriptions. 'Code' is a subtree containing the body of the procedure: local variable definitions and expressions to be evaluated.

Example: the following C function

```
main (argc, argv)
int argc;
char **argv;
   {
   int i;
   i = 4;
   }
```

```
50       PROC_DEFN_OP
1           Procedure is object number 1
2           Procedure has 2 arguments
4           Procedure name is 4 characters long
237            m
225            a
233            i
238            n
49          PROC_DEFN_ARG_OP
2              Argument is object number 2
1              INT_MODE
0              VAL_DISP; pass argument by value
1              Argument is 1 word long
49          PROC_DEFN_ARG_OP
3              Argument is object number 3
4              LONG_UNS_MODE (a pointer)
1              REF_DISP; pass argument by reference
2              Argument is 2 words long
```

```
39          NULL_OP; end of argument description list
59          SEQ_OP; beginning of procedure code
13             DEFINE_DYNM_OP
4                 Object id is 4
39                NULL_OP; no initializers
1                 Object is 1 word long
59          SEQ_OP; procedure code continues
5              ASSIGN_OP
1                 INT_MODE
40                OBJECT_OP
1                    INT_MODE
4                    Object id is 4
9                 CONST_OP
1                    INT_MODE
1                    Constant is 1 word long
4                    Constant has value 4
1                 Assignment transfers 1 word
39          NULL_OP; end of procedure code
```

PROC_DEFN_ARG_OP 49

```
int 49
int object_id
int mode
int disposition
int length
tree next_argument
```

Formal parameters to procedures are described by this operator. The 'object_id' is an integer identifier that must be supplied on subsequent references to the parameter (see OBJECT_OP). The 'mode' is the machine data type of the parameter. 'Disposition' indicates how the argument is to be treated on the call; the two alternatives at the moment are 0 (VALUE_DISP) for pass-by-value (aka copy in) and 1 (REF_DISP) for pass-by-reference. 'Length' gives the size of the argument in 16-bit words; it is primarily necessary for handling of STOWED arguments that are passed by value. 'Next_argument' is simply a link to the next PROC_DEFN_ARG_OP in a procedure's argument descriptor list, or a NULL_OP.

See PROC_DEFN_OP for examples of PROC_DEFN_ARG_OP.

Operators Useful in Procedure Definitions

ADDAA_OP 1

```
int 1
int mode
tree left
tree right
```

The result of this operator is an rvalue, the sum of the values of the left and right operands. As a side effect, the sum is stored back into the left operand. The left operand must be an lvalue or a bit field (see FIELD_OP). Both operands must have the same mode as the ADDAA operation. The operation mode may not be STOWED.

ADDAA stands for "add and assign." This operator is normally used to implement the addition assignment operator ("+=" in C, "+:=" in Algol 68).

Example: i += 1 (where i is an integer object with object id 12)

```
1               ADDAA_OP
1                  INT_MODE
40                 OBJECT_OP
1                     INT_MODE
12                    Object id 12
9                  CONST_OP
1                     INT_MODE
1                     length is 1 word
1                     value of first word is 1
```

ADD_OP 2

```
int 2
int mode
tree left
tree right
```

The result of this operator is an rvalue, the sum of the values of the left and right operands. Both operands must have the same mode as the ADD, and STOWED mode is not allowed.

ADD is used to implement simple addition of fixed or floating point values.

Example: i + 1 (where i is an integer object with object id 12)

```
2               ADD_OP
1                  INT_MODE
40                 OBJECT_OP
1                     INT_MODE
```

```
12                   Object id 12
9                 CONST_OP
1                    INT_MODE
1                    length is 1 word
1                    value of first word is 1
```

ANDAA_OP 3

```
int 3
int mode
tree left
tree right
```

The result of this operator is an rvalue, the bitwise logical "and" of the values of the left and right operands. As a side effect, the conjunction is stored back into the left operand. The left operand must be an lvalue or a bit field (see FIELD_OP). Both operands must have the same mode as the ANDAA operation; the only allowable modes are INT, UNSIGNED, LONG INT, and LONG UNSIGNED.

ANDAA stands for "'and' and assign." ANDAA_OP is used to implement the logical-and assignment operator ("&=" in C).

Example: i &= 1 (where i is an integer object with object id 12)

```
3              ANDAA_OP
1                 INT_MODE
40                OBJECT_OP
1                    INT_MODE
12                   Object id 12
9                 CONST_OP
1                    INT_MODE
1                    length is 1 word
1                    value of first word is 1
```

AND_OP 4

```
int 4
int mode
tree left
tree right
```

The result of this operator is an rvalue, the bitwise logical-"and" of the values of the left and right operands. Both operands must have the same mode as the AND operation; the only allowable modes are INT, LONG INT, UNSIGNED, and LONG UNSIGNED.

AND_OP is normally used to implement the bitwise logical conjunction of integers ("&" in C). Although AND_OP can be used to implement conjunction in Boolean expressions, the short-circuit

conjunction operator (SAND_OP) is probably a better choice, since it guarantees evaluation order and prevents undesirable side effects.

```
Example:  i & 1 (where i is an integer object with object id 12)
         4              AND_OP
         1                 INT_MODE
         40                OBJECT_OP
         1                    INT_MODE
         12                   Object id 12
         9                 CONST_OP
         1                    INT_MODE
         1                    length is 1 word
         1                    value of first word is 1
```

ASSIGN_OP 5

```
int 5
int mode
tree left
tree right
int length
```

The result of this operator is an rvalue, namely the value of the right operand. As a side effect, the result is stored into the left operand. The left operand must be an lvalue or a bit field (see FIELD_OP). Both operands must have the same mode as the ASSIGN operation. Any mode is allowable, but the parameter 'length' must be set to the operand length, in 16-bit words.

ASSIGN implements the semantics of assignment statements in most algorithmic languages. Note that STOWED mode values are allowed, so things like Pascal record assignment can be handled.

```
Example:  i = 1 (where i is an integer object with object id 12)
         5              ASSIGN_OP
         1                 INT_MODE
         40                OBJECT_OP
         1                    INT_MODE
         12                   Object id 12
         9                 CONST_OP
         1                    INT_MODE
         1                    length is 1 word
         1                    value of first word is 1
         1                 length of assigned quantity is 1 word
```

BREAK_OP 6

```
int 6
int levels
```

BREAK_OP yields no result value, but causes an exit from one or more enclosing loops or multiway-branch ("switch," in C terminology; "case" in Pascal) statements. The operand 'levels' is an integer giving the number of nested loops and multiway branches to terminate. Obviously, 'levels' must be between 1 and the number of nested loops and multiway branch statements currently active, inclusive.

BREAK is mainly intended to implement premature loop exits. Because of (inadequate) historical reasons, a BREAK is also required to force control out of a multiway-branch alternative to the end of the statement. Thus, in implementing a Pascal-style case statement with the SWITCH_OP described below, each alternative would end with a BREAK_OP with 'levels' equal to 1. If the BREAK_OP was missing, control would fall through from case to case, as it does in C.

```
Example:  break 2  (terminate 2 enclosing loops)
          6              BREAK_OP
          2                 Levels to break
```

CASE_OP 7

```
int 7
tree value
tree actions
tree next_case
```

CASE is used to label an alternative in a multiway branch statement (like 'switch' in C or 'case' in Pascal). The 'value' parameter is the case label value for the alternative; it must be a CONST_OP node of the same mode as the switch expression (see SWITCH_OP). The mode may not be STOWED. The 'actions' parameter is the code to be executed for the given case label. The 'next_case' operand is a DEFAULT_OP or another CASE_OP or a NULL_OP (for the last alternative in the multiway-branch).

CASE_OP is simply a structural device; it organizes the alternatives in a multiway-branch so that variable-sized SWITCH operators are not necessary.

```
Example:  case 10: i += 1   (i is an integer with object id 12)
          7              CASE_OP
          9                 CONST_OP
          1                    INT_MODE
          1                    length is 1 word
          10                   value of first word is 10
          1                 ADDAA_OP
          1                    INT_MODE
          40                   OBJECT_OP
          1                       INT_MODE
          12                      Object id 12
          9                    CONST_OP
```

```
1                       INT_MODE
1                       length is 1 word
1                       value of first word is 1
7 or 12 or 39     CASE_OP or DEFAULT_OP or NULL_OP,
                  depending on next element of SWITCH
```

CHECK_LOWER_OP 72

```
int 72
int mode
tree expression
tree lower_bound
int source_line_number
```

The result of this operator is an rvalue, the value of the parameter 'expression'. The expression must have the mode given by the parameter 'mode', and may not be FLOAT, LONG_FLOAT, or STOWED. If at run time the value of the expression is less than the value of the expression given by the parameter 'lower_bound', an error message is printed and a RANGE_ERROR exception raised. The parameter 'source_line_number' is printed as part of the error message, and is identified as the number of the source code line that caused the range check to be generated.

This operator would normally be used in a situation that permitted optimized range checking, like assignment of one integer subrange variable to another.

Example: var i: 0..100; j: 1..100; begin ...; j := i; ... end
(where i has object id 12 and j has object id 13,
and the code above appears on line 14)

```
5        ASSIGN_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
13             Object id for j
72          CHECK_LOWER_OP
1              INT_MODE
40             OBJECT_OP
1                 INT_MODE
12                Object id for i
9              CONST_OP
1                 INT_MODE
1                 Length is 1 word
1                 Value is 1
14             Line number in source code
1           Length of assigned quantity is 1 word
```

CHECK_RANGE_OP 70

```
int 70
int mode
tree expression
tree lower_bound
tree upper_bound
int source_line_number
```

The result of this operator is an rvalue, the value of the parameter 'expression'. The expression must have the mode given by the parameter 'mode', and may not be FLOAT, LONG_FLOAT, or STOWED. If at run time the value of the expression is less than the value of the expression given by the parameter 'lower_bound' or greater than the value of the expression given by the parameter 'upper_bound' an error message is printed and a RANGE_ERROR exception raised. The parameter 'source_line_number' is printed as part of the error message, and is identified as the number of the source code line that caused the range check to be generated.

This operator would normally be used where a complete range check was necessary (an array subscripted by an unconstrained integer variable, for example).

Example: var a: array 1..10 of integer; i: integer; ...a[i]...
where 'a' has object id 4, 'i' has id 12,
and the subscripting operation appears on line 97
of the source code:

```
25       INDEX_OP
1           INT_MODE (element type of a)
40          OBJECT_OP; this is the base address of 'a'
7              STOWED_MODE
4              Object id of 'a'
70          CHECK_RANGE_OP; this is the index expression
1              INT_MODE
40             OBJECT_OP
1                 INT_MODE
12                Object id of 'i'
9              CONST_OP; this is the lower bound
1                 INT_MODE
1                 Length of constant is 1 word
1                 Value of constant is 1
9              CONST_OP; this is the upper bound
1                 INT_MODE
1                 Length of constant is 1 word
10                Value of constant is 10
97             Range check is on line 97
1           Array element size is 1 word
```

CHECK_UPPER_OP 71

```
int 71
int mode
tree expression
tree upper_bound
int source_line_number
```

The result of this operator is an rvalue, the value of the parameter 'expression'. The expression must have the mode given by the parameter 'mode', and may not be FLOAT, LONG_FLOAT, or STOWED. If at run time the value of the expression is greater than the value of the expression given by the parameter 'upper_bound', an error message is printed and a RANGE_ERROR exception raised. The parameter 'source_line_number' is printed as part of the error message, and is identified as the number of the source code line that caused the range check to be generated.

Like CHECK_LOWER, this operator is normally used in situations that permit optimized range checks.

Example: var i: 1..100; j: 1..10; begin ...; j := i; ... end
(where i has object id 12 and j has object id 13,
and the code above appears on line 14)

```
5        ASSIGN_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
13             Object id for j
71          CHECK_UPPER_OP
1              INT_MODE
40             OBJECT_OP
1                 INT_MODE
12                Object id for i
9              CONST_OP
1                 INT_MODE
1                 Length is 1 word
10                Value is 10
14             Line number in source code
1           Length of assigned quantity is 1 word
```

COMPL_OP 8

```
int 8
int mode
tree operand
```

The result of this operator is an rvalue, the bitwise complement of the operand. The operand must have the same mode as the COMPL operation; the only allowable modes are INT, LONG INT, UNSIGNED, and LONG UNSIGNED.

This operator implements bitwise complementation in languages that support bit operations (e.g. the "~" operator in C). In most cases, it should not be used for logical negation; the NOT_OP is more appropriate.

```
Example:  ~i (i is an integer object with id 12)
         8                 COMPL_OP
         1                    INT_MODE
         40                   OBJECT_OP
         1                       INT_MODE
         12                      Object id is 12
```

CONST_OP 9

```
int 9
int mode
int length
int word[1]
int word[2]
...
int word[length]
```

The result of this operator is an rvalue, equivalent to the value of the constant it defines. 'Length' is the length of the constant in 16-bit machine words. 'Mode' may take on any of the operand mode values, although STOWED constants are not of much use outside initializers.

CONST_OP is the only operator whose IMF representation varies in length depending on its contents. Most literals in a source language program eventually are expressed as CONST_OPs in the IMF.

```
Example:  14 (an integer constant)
         9                 CONST_OP
         1                    INT_MODE
         1                    length is 1 word
         14                   first word has value 14
```

CONVERT_OP 10

```
int 10
int source_mode
int destination_mode
tree operand
```

The result of this operator is an rvalue, namely the value of the operand converted to the data mode specified by 'destination_mode'. The operand mode must be the same as 'source_mode'. STOWED mode is not permissible in either mode

parameter. Note that in most cases, no range checking is performed; it is possible, for example, to convert an UNSIGNED quantity into an negative INT quantity. Floating point to integer conversions are performed by truncation.

CONVERT is the only means of converting data from one mode to another; the code generator never coerces data from one mode to another, unless the coercion is called for by a CONVERT operator.

```
Example:  x = i (x is a FLOAT object, with id 6;
                 i is an INT object, with id 12)
         5                 ASSIGN_OP
         5                    FLOAT_MODE
         40                   OBJECT_OP
         5                       FLOAT_MODE
         6                       Object id is 6
         10                   CONVERT_OP
         1                       from INT_MODE
         5                       to FLOAT_MODE
         40                      OBJECT_OP
         1                          INT_MODE
         12                         Object id is 12
```

DECLARE_STAT_OP 11

```
int 11
int object_id
string external_name
```

See "Operators useful in the Static Data Stream".

DEFAULT_OP 12

```
int 12
tree actions
tree next_case
```

This operator is used to label the default action in a multiway-branch statement. (In C, the default action is labeled "default"; in Pascal, it is labeled "otherwise".) The DEFAULT_OP need not be the last alternative in the list of alternatives following a SWITCH. A DEFAULT_OP behaves much like a CASE_OP, in that control will fall through to the next alternative unless the actions conclude with a BREAK_OP.

```
Example:  default: i += 1 (where i is an integer object with id 12)
         12             DEFAULT_OP
         1                 ADDAA_OP
         1                    INT_MODE
```

```
40                   OBJECT_OP
1                       INT_MODE
12                      Object id 12
9                    CONST_OP
1                       INT_MODE
1                       length is 1 word
1                       value of first word is 1
7 or 39           CASE_OP or NULL_OP, depending on structure
                  of SWITCH
```

DEFINE_DYNM_OP 13

```
int 13
int object_id
tree init_list
int size
```

This operator causes storage for the object identified by 'object_id' to be allocated in the current stack frame. It is generated for local variable declarations and for temporary variables allocated by the front end. 'Object_id' must be used in all subsequent references to the object, and the object's definition with DEFINE_DYNM must precede all such references. The init_list is a list of expressions whose values will be assigned to successive words of the newly-declared object (see INITIALIZER_OP and ZERO_INITIALIZER_OP). The size parameter specifies the amount of storage to be reserved for the object, in 16-bit words. (Slightly fewer than 65,535 words are available for local storage in each procedure.)

When processing a declaration, the front-end should assign each declared variable an integer "object id." To be safe, the object id should be unique within an IMF module. This object id must be used whenever the variable being declared is referenced.

```
Example:  int blank = 160;    (a local declaration; assume 'blank'
                              is assigned the object id 4)
         13             DEFINE_DYNM_OP
         4                 Object id is 4
         26                INITIALIZER_OP
         1                    INT_MODE
         9                    CONST_OP
         1                       INT_MODE
         1                       Length is 1 word
         160                     Value of first word is 160
         39                   NULL_OP (end of init list)
         1                 Size is 1 word
```

DEFINE_STAT_OP 14

```
int 14
int object_id
tree init_list
int size
```

This operator causes storage for the object identified by 'object_id' to be allocated in the current link frame (static data area). It is normally generated by the front end for global variable declarations. 'Object_id' must be used in all subsequent references to the object, and the object's definition with DEFINE_STAT must precede all such references. The init_list is a list of constants whose values will be assigned to successive words of the newly-declared object (see INITIALIZER_OP and ZERO_INITIALIZER_OP). The size parameter specifies the amount of storage to be reserved for the object, in 16-bit words. (Slightly fewer than 65,535 words are available for static storage in each module.)

Any storage reserved by a DEFINE_STAT_OP that is not filled by an initializer will be set to zero.

When processing a declaration, the front-end should assign each declared variable an integer "object id." To be safe, the object id should be unique within an IMF module. This object id must be used whenever the variable being declared is referenced.

Example: int blank = 160; (a global declaration; assume 'blank' is assigned the object id 4)

```
14              DEFINE_STAT_OP
4                  Object id is 4
26                 INITIALIZER_OP
1                     INT_MODE
9                     CONST_OP
1                        INT_MODE
1                        Length is 1 word
160                      Value of first word is 160
39                    NULL_OP (end of init list)
1                  Size is 1 word
```

DEREF_OP 15

```
int 15
int mode
tree operand
```

The result of this operator is an lvalue, the object whose address is given by the value of the operand. The operand must yield a 32-bit LONG INT or LONG UNSIGNED value. The operation mode is not restricted.

DEREF is one of the few operators that yield an lvalue, and are

therefore allowed as left-operands of assignments. DEREF is normally used for indirection through pointers in languages that support them explicitly (eg "^" operator in Pascal, or unary "*" in C), although it is also useful in obtaining the value of a variable that is passed to a procedure by reference.

```
Example:  i = *p  (or i = p^ in Pascal)
                  (i is an integer object with id 12;
                  p is a long unsigned object with id 32)
         5              ASSIGN_OP
         1                 INT_MODE
         40                OBJECT_OP
         1                    INT_MODE
         12                   Object id is 12
         15                DEREF_OP
         1                    INT_MODE
         40                   OBJECT_OP
         4                       LONG_UNS_MODE
         32                      Object id is 32
```

DIVAA_OP 16

```
int 16
int mode
tree left
tree right
```

The result of this operator is an rvalue, the quotient of the value of the left operand divided by the value of the right. As a side effect, the quotient is stored back into the left operand. The left operand must be an lvalue or a bit field (see FIELD_OP). Both operands must have the same mode as the DIVAA operation; any mode other than STOWED is acceptable.

DIVAA stands for "divide and assign." The operator is usually used to implement the division assignment operator ("/=" in C, "/:=" or "divab" in Algol 68).

If the operation mode is UNSIGNED or LONG UNSIGNED and the right operand is a power of 2, the division will be performed by a right logical shift.

```
Example:  i /= 10 (where i is an integer object with object id 12)
         16             DIVAA_OP
         1                 INT_MODE
         40                OBJECT_OP
         1                    INT_MODE
         12                   Object id 12
         9                 CONST_OP
         1                    INT_MODE
         1                    length is 1 word
         10                   value of first word is 1
```

DIV_OP 17

```
int 17
int mode
tree left
tree right
```

The result of this operator is an rvalue, the quotient of the value of the left operand divided by the value of the right. Both operands must have the same mode as the DIV operation, and the mode STOWED is not allowed.

DIV is used to implement simple division.

If the operation mode is UNSIGNED or LONG UNSIGNED and the right operand is a power of 2, the division will be performed by a right logical shift.

Example: i / 10 (where i is an integer object with object id 12)

```
17              DIV_OP
1                  INT_MODE
40                 OBJECT_OP
1                     INT_MODE
12                    Object id 12
9                  CONST_OP
1                     INT_MODE
1                     length is 1 word
10                    value of first word is 1
```

DO_LOOP_OP 18

```
int 18
tree body
tree condition
```

This operator implements a test-at-the-bottom loop. 'Body' specifies the operations to be performed in the loop. The loop is performed until the value of the expression specified by 'condition' is non-zero. A BREAK_OP may be used to terminate execution of the loop from within the body, and a NEXT_OP may be used to cause an immediate transfer to the condition test from within the body.

It is not kosher to use a DO_LOOP as a value-returning construct.

Example: do i *= 2 until (i > j)
(where i and j are integer objects, with ids 12 and 60)

```
18              DO_LOOP_OP
33                 MULAA_OP
1                     INT_MODE
40                    OBJECT_OP
1                        INT_MODE
12                       Object id is 12
```

```
9                    CONST_OP
1                       INT_MODE
1                       Length is 1 word
2                       Value is 2
23                GT_OP
1                    INT_MODE
40                   OBJECT_OP
1                       INT_MODE
12                      Object id is 12
40                   OBJECT_OP
1                       INT_MODE
60                      Object id is 60
```

EQ_OP 19

```
int 19
int mode
tree left
tree right
```

The result of this operator is an rvalue: 1 if the value of the left operand equals the value of the right, and 0 otherwise. Both operands must have the mode specified by the parameter 'mode', but note that the result mode of EQ is always INTEGER. The operation mode may not be STOWED.

EQ is used to implement test-for-equality for both expressions yielding Boolean values and for control flow tests. The restriction against STOWED operands will hopefully be lifted in the near future.

Example: i == 1 (where i is an integer object with object id 12)

```
19             EQ_OP
1                 INT_MODE
40                OBJECT_OP
1                    INT_MODE
12                   Object id 12
9                 CONST_OP
1                    INT_MODE
1                    length is 1 word
1                    value of first word is 1
```

FIELD_OP 69

```
int 69
int mode
int offset_from_msb
int length_in_bits
tree base_address
```

FIELD is used to select a partial field of a word or double word. It may be used on the left hand side of assignments, to cause the right hand side value to be placed in the field, or as an rvalue, to yield the value stored in the field. The operation mode must be INT, UNSIGNED, LONG INT, or LONG UNSIGNED. The parameter 'base_address' is an lvalue which specifies the first 16-bit word containing any portion of the bit field. The parameter 'offset_from_msb' gives the offset, in bits, of the beginning of the field from the left-hand (most significant) bit of the first word. The parameter 'length_in_bits' gives the length of the bit field. Bit fields may be 1 to 32 bits in length, and must be aligned so as not to cross more than one word boundary.

FIELDs behave like lvalues in most circumstances; for instance, they can be used in left-hand-sides of assignments. However, bit fields cannot be addressed, so they may not be passed by reference on procedure calls or used as an operand of the REFTO operator. FIELDs can always be used as rvalues.

Bit fields may not cross more than one word boundary, since this would require 48 bit shifts for field extraction. Formally, this means that 'offset_from_msb' + 'length_in_bits' must be less than or equal to 32.

Example: Fetching the right-hand byte of a 16-bit word in the integer object i, with id 12:

```
69       FIELD_OP
1           INT_MODE
8           Bit field begins 8 bits from the most
            significant bit
8           Bit field is 8 bits long
40          OBJECT_OP; the base address of the field
1              INT_MODE
12             Object id for 'i'
```

FOR_LOOP_OP 20

int 20
tree init
tree cond
tree reinit
tree body

The FOR_LOOP_OP implements the general-purpose C 'for' loop. The parameters 'init', 'reinit', and 'body' correspond to statements; 'cond' corresponds to a Boolean expression. The for-loop

```
for (init; cond; reinit) statement
```

is equivalent to

```
          init; while cond do begin statement; reinit end
```

A typical application in languages other than C might be the construction of an arithmetic loop like the Pascal 'for' or the Fortran 'do'.

Within the body of the loop, a BREAK_OP may be used to cause early loop termination, and a NEXT_OP may be used to cause an immediate jump to the 'reinit' code in preparation for another iteration.

It is not reasonable to use a FOR_LOOP as a value-returning construct.

Example:
```
for (i = 1; i <= n; i += 1)
    j *= i;
```
(where i, j, n are integers with object ids 12, 60, 44)

```
20              FOR_LOOP_OP
5                  ASSIGN
1                     INT_MODE
40                    OBJECT_OP
1                        INT_MODE
12                       Object id 12
9                     CONST_OP
1                        INT_MODE
1                        Length is 1 word
1                        Value is 1
1                     Assign 1 word
28                 LE_OP
1                     INT_MODE
40                    OBJECT_OP
1                        INT_MODE
12                       Object id 12
40                    OBJECT_OP
1                        INT_MODE
44                       Object id 44
1                  ADDAA_OP
1                     INT_MODE
40                    OBJECT_OP
1                        INT_MODE
12                       Object id 12
9                     CONST_OP
1                        INT_MODE
1                        Length is 1 word
1                        Value is 1
33                 MULAA_OP
1                     INT_MODE
40                    OBJECT_OP
1                        INT_MODE
60                       Object id 60
40                    OBJECT_OP
1                        INT_MODE
12                       Object id 12
```

GE_OP 21

```
int 21
int mode
tree left
tree right
```

The result of this operator is an rvalue, 1 if the value of the left operand is greater-than-or-equal-to the value of the right, 0 otherwise. Both operands must have the mode given in the parameter 'mode'; note, however, that the result of GE is always of mode INTEGER. The operation mode may not be STOWED. Note that if the operands are unsigned, a "magnitude" comparison is performed to insure correct results.

GE_OP implements the test for greater-or-equal in both Boolean expressions and flow-of-control tests. The restriction against STOWED operands may be lifted someday.

Example: i >= 1 (where i is an integer object with object id 12)

```
         21             GE_OP
         1                 INT_MODE
         40                OBJECT_OP
         1                    INT_MODE
         12                   Object id 12
         9                 CONST_OP
         1                    INT_MODE
         1                    length is 1 word
         1                    value of first word is 1
```

GOTO_OP 22

```
int 22
int object_id
```

GOTO_OP is used to implement unrestricted 'goto' statements in languages that support such nonsense. The parameter 'object_id' is the integer object identifier of the label which is the target of the goto. (See LABEL_OP).

The stack is not adjusted if the target label is outside the current procedure.

Example: goto label (where 'label' has object id 99)

```
         22             GOTO_OP
         99                Object ID of target label
```

GT_OP 23

```
int mode
tree left
tree right
```

The result of this operator is an rvalue, 1 if the value of the left operand is greater than the value of the right, 0 otherwise. Both operands must have the mode given by the parameter 'mode'; but note that GT always returns a value of mode INTEGER. The operation mode may not be STOWED. Note that if the operands are of mode unsigned, a "magnitude" comparison will be performed to insure correct results.

GT implements the test for greater-than for Boolean expressions and expressions in flow-of-control context. The restriction against STOWED operands might be lifted if the public demands it.

Example: i > 1 (where i is an integer object with object id 12)

```
         23             GT_OP
         1                 INT_MODE
         40                OBJECT_OP
         1                    INT_MODE
         12                   Object id 12
         9                 CONST_OP
         1                    INT_MODE
         1                    length is 1 word
         1                    value of first word is 1
```

IF_OP 24

```
int 24
int mode
tree condition
tree then_part
tree else_part
```

IF can be used to implement conditional expressions or conditional evaluation of statements; it always returns an rvalue. If the value of the condition is non-zero, the 'then_part' will be evaluated; otherwise, the 'else_part' will be evaluated. Either 'then_part' or 'else_part' may be omitted (ie, replaced by a NULL_OP). The operation mode may not be STOWED; if the operator is used to return a value (as in a conditional expression) then the modes of both the 'then_part' and the 'else_part' must be the same as the operation mode.

IF is most often used to implement conditional statements (eg the 'if' statement of most algorithmic languages). Since the code generator tends to view operators as value-returning, IF may also be used to implement conditional expressions ('if'-'then'-'else' in the Algol family, or '?:' in C).

```
Example:  if a < b then m = a else m = b
         (where a, b, m are floating point objects with id's 1, 2, 13)
         24             IF_OP
         5                 FLOAT_MODE
         31                LT_OP
         5                    FLOAT_MODE
         40                   OBJECT_OP
         5                       FLOAT_MODE
         1                       Object id 1
         40                   OBJECT_OP
         5                       FLOAT_MODE
         2                       Object id 2
         5                 ASSIGN_OP
         5                    FLOAT_MODE
         40                   OBJECT_OP
         5                       FLOAT_MODE
         13                      Object id 13
         40                   OBJECT_OP
         5                       FLOAT_MODE
         1                       Object id 1
         2                    Length is 2 words
         5                 ASSIGN_OP
         5                    FLOAT_MODE
         40                   OBJECT_OP
         5                       FLOAT_MODE
         13                      Object id 13
         40                   OBJECT_OP
         5                       FLOAT_MODE
         2                       Object id 2
         2                    Length is 2 words
```

INDEX_OP 25

```
int 25
int mode
tree array_base
tree index_expression
int element_size
```

The result of this operator is an lvalue, one member of a vector of identical objects. The parameter 'array_base' is the base of the vector; it is typically a simple OBJECT_OP, although it may be an expression yielding the base address of the vector (a dereferenced pointer, for example). It must be an lvalue. The 'index_expression' selects the particular vector element desired; it should have a value greater than or equal to zero and less than the number of elements in the vector. (Note that this implies zero-origin addressing.) (Note furthermore that there is no subscript checking.) The 'index_expression' must be of mode INTEGER or UNSIGNED (indexing across 64K-word segment boundaries produces incorrect results in V mode). 'Element_size' is the size of one element of the vector, in 16-bit words. The operation mode must be the same as the mode of the vector elements,

but is otherwise unrestricted; in particular, STOWED mode is allowed.

INDEX is used to implement array subscripting. The operator has deliberately been made rather primitive, to allow the front-end greater freedom in selecting storage layouts. For example, multidimensional arrays may be implemented by treating arrays as vector elements, and subsuming the additional addressing calculations in the 'index_expression'. This allows a compiler to select row- or column-major addressing. Note that subscripting is vastly more efficient if vector elements are a power of 2 words in length, and furthermore that lengths 1, 2, and 4 are most efficient.

```
Example:  a[i + 1]  (where a is a floating point object with id 1,
                    and i is an integer object with id 12)
          25            INDEX_OP
          5                FLOAT_MODE
          40               OBJECT_OP
          7                   STOWED_MODE
          1                   Object id 1
          2                ADD_OP
          1                   INT_MODE
          40                  OBJECT_OP
          1                      INT_MODE
          12                     Object id 12
          9                   CONST_OP
          1                      INT_MODE
          1                      Length is 1 word
          1                      Value is 1
          2                Array element is 2 words long
```

INITIALIZER_OP 26

```
int 26
int mode
tree expression
tree next_initializer
```

Initializers are the initial-value expressions that appear in definitions of variables in C (see DEFINE_DYNM_OP and DEFINE_STAT_OP). In the case of local variables, which are reinitialized whenever they are allocated, these expressions are arbitrary. In the case of static variables, these expressions must be constants or REFTO operators whose operands are constants or OBJECT_OPs.

Initializers are formed by linking a number of INITIALIZER_OPs and ZERO_INITIALIZER_OPs together through their 'next_initializer' fields. ZERO_INITIALIZER_OP is a compact representation of an initializer consisting of all zeros.

Any mode is allowable in an INITIALIZER. INT and UNSIGNED

initializers cause one word to be filled; LONG INT, LONG UNSIGNED, and FLOAT cause two words to be filled; LONG FLOAT causes four words to be filled; STOWED expressions fill as many words as the size of the expression allows (STOWED mode CONST_OPs are particularly useful here).

```
Example:  int ai[3] = {1, 2, 3}
          (a local declaration, where ai is assigned object id 8)
         13          DEFINE_DYNM_OP
         8              Object has id 8
         26             INITIALIZER_OP
         1                 INT_MODE
         9                 CONST_OP (the init. expression)
         1                    INT_MODE
         1                    Constant has length 1
         1                    Constant has value 1
         26                INITIALIZER_OP
         1                    INT_MODE
         9                    CONST_OP
         1                       INT_MODE
         1                       Constant has length 1
         2                       Constant has value 2
         26                   INITIALIZER_OP
         1                       INT_MODE
         9                       CONST_OP
         1                          INT_MODE
         1                          Constant has length 1
         3                          Constant has value 3
         39                      NULL_OP (end of initializers)
         3              Object has size 3 words
```

As an alternative,

```
         13          DEFINE_DYNM_OP
         8              Object has id 8
         26             INITIALIZER_OP
         7                 STOWED_MODE
         9                 CONST_OP
         7                    STOWED_MODE
         3                    Constant is 3 words long
         1                    First word is 1
         2                    Second word is 2
         3                    Third word is 3
         39                NULL_OP (end of initializers)
         3              Object is 3 words long
```

LABEL_OP 27

```
int 27
int object_id
```

LABEL_OP is used to place the target label for 'goto' statements. The parameter 'object_id' is the integer identifier used by

GOTO_OPs to identify their target labels.

```
Example:  label lab;.... lab:
          (assume the label declaration causes 'lab' to be assigned
          the object id 6)
         27          LABEL_OP
         6              The object ID of the label
```

LE_OP 28

```
int 28
int mode
tree left
tree right
```

The result of this operator is an rvalue, 1 if the value of the left operand is less than or equal to the value of the right, 0 otherwise. Both operands must have the mode specified by the parameter 'mode'; STOWED mode is not allowable. Note that LE always returns a value of mode INTEGER. Magnitude comparisons are generated for unsigned operands, to insure correct results.

Use LE_OP to implement all tests for less-than-or-equal-to, whether they appear in boolean expressions or flow-of-control tests. The restriction against STOWED operands may be lifted if the author feels sufficiently threatened.

Example: i <= 1 (where i is an integer object with id 12)

```
         28       LE_OP
         1           INT_MODE
         40          OBJECT_OP
         1              INT_MODE
         12             Object id 12
         9           CONST_OP
         1              INT_MODE
         1              Constant has length 1
         1              Constant has value 1
```

LSHIFTAA_OP 29

```
int 29
int mode
tree left
tree right
```

The result of this operator is an rvalue, the value of the left operand shifted logically (zero-fill) left the number of bit places specified by the value of the right operand. As a side effect, the result is stored back into the left operand. The

left operand must be an lvalue or a bit field (see FIELD_OP). The operation mode may be INT, LONG INT, UNSIGNED, or LONG UNSIGNED, and the left operand must have the same mode. The right operand must be of mode INT or UNSIGNED, and really should have a value between 0 and the length of the left operand, inclusive. (Reasonable results outside this range are not guaranteed.)

LSHIFTAA stands for "left-shift and assign." The operator is used to implement "<<=" in C.

Example: i <<= 1 (where i is an integer object with id 12)

```
29       LSHIFTAA_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object id 12
9           CONST_OP
1              INT_MODE
1              Constant has length 1
1              Constant has value 1
```

LSHIFT_OP 30

```
int 30
int mode
tree left
tree right
```

The result of this operator is an rvalue, the value of the left operand shifted left logically (zero-fill) the number of bit places specified by the value of the right operand. The operation mode may be INT, LONG INT, UNSIGNED, or LONG UNSIGNED, and the left operand must have the same mode. The right operand must be of mode INT or UNSIGNED, and really should have a value between 0 and the length of the left operand, inclusive. (Reasonable results outside this range are not guaranteed.)

LSHIFT is used to implement the "<<" operator in C.

Example: i << 1 (where i is an integer object with id 12)

```
30       LSHIFT_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object id 12
9           CONST_OP
1              INT_MODE
1              Constant has length 1
1              Constant has value 1
```

LT_OP 31

```
int 31
int mode
tree left
tree right
```

The result of this operator is an rvalue, 1 if the value of the left operand is less than the value of the right, 0 otherwise. Both operands must have the mode given in the parameter 'mode'. Note that LT always returns a value of mode INTEGER, no matter what the operation mode was. The operation mode may not be STOWED. Magnitude comparisons are used if the operands are unsigned, to insure correct results.

LT is used to implement the test for less-than, in both Boolean expressions and flow-of-control expressions. The restriction against STOWED operands may be removed if an angry armed mob storms the author's office.

Example: i < 1 (where i is an integer object with id 12)

```
31       LT_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object id 12
9           CONST_OP
1              INT_MODE
1              Constant has length 1
1              Constant has value 1
```

MODULE_OP 32

```
int 32
```

This operator is not used in procedure definitions; it is used strictly to separate modules in input streams.

MULAA_OP 33

```
int 33
int mode
tree left
tree right
```

The result of this operation is an rvalue, the product of the value of the left operand and the value of the right. As a side effect, the product is stored into the left operand. The left

operand must be an lvalue or a bit field. Both operands must have the same mode as the operation, and that mode may not be STOWED.

MULAA stands for "multiply and assign." It is used to implement the multiplication assignment operators ("*=" in C, "*:=" or "mulab" in Algol 68). When either operand is known to be a power of 2, the multiplication will be replaced by a left logical shift.

Example: i *= 10 (where i is an integer object with id 12)

```
33       MULAA_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object id 12
9           CONST_OP
1              INT_MODE
1              Constant has length 1
10             Constant has value 10
```

MUL_OP 34

```
int 34
int mode
tree left
tree right
```

The result of this operation is an rvalue, the product of the value of the left operand and the value of the right. Both operands must have the same mode as the operation, and that mode may not be STOWED.

MUL_OP is used to implement simple multiplication. When either operand is known to be a power of 2, the multiplication will be replaced by a left logical shift.

Example: i * 2 (where i is an integer object with id 12)

```
34       MUL_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object id 12
9           CONST_OP
1              INT_MODE
1              Constant has length 1
2              Constant has value 2
```

NEG_OP 35

```
int 35
int mode
tree operand
```

The result of this operator is an rvalue, the additive inverse of the value of the operand. Unsigned operands are subtracted from 2**n, where n is the number of bits used to represent them (16 or 32, in this implementation). The operation mode must be the same as the mode of the operand, and may not be STOWED.

NEG_OP implements the unary minus (negation) operator for all the primitive arithmetic data modes.

Example: -i (where i is an integer object with id 12)

```
35       NEG_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object has id 12
```

NEXT_OP 36

```
int 36
int levels
```

NEXT_OP yields no result value, but causes an immediate restart of a particular enclosing loop. 'Levels' - 1 enclosing loops are terminated (see BREAK_OP) and then a branch is taken to the proper restart point in the next enclosing loop. For the FOR_LOOP, the restart point is the re-initialization statement at the end of the body. For DO_LOOPs and WHILE_LOOPs, the restart point is the evaluation of the iteration condition.

Example: next 2 (break 1 loop, continue the next outermost)

```
36       NEXT_OP
2           Levels
```

NE_OP 37

```
int 37
int mode
tree left
tree right
```

The result of this operator is an rvalue, 1 if the value of the left operand does not equal the value of the right, 0 otherwise.

The modes of both operands must match the mode of the operation, and STOWED mode is not allowed. Note that NE_OP always returns a value of mode INTEGER, no matter what operation mode is specified.

NE implements the test for inequality in all contexts. Use of nuclear weapons might be enough to convince the author to lift the restriction against STOWED operands.

Example: i <> 1 (where i is an integer object with id 12)

```
37       NE_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object id 12
9           CONST_OP
1              INT_MODE
1              Constant has length 1
1              Constant has value 1
```

NOT_OP 38

```
int 38
int mode
tree operand
```

The result of this operator is an rvalue, the logical negation of the operand value. (Ie, if the operand has value zero, the result of the NOT_OP will be 1; if the operand is non-zero, the result of the NOT_OP will be zero.) The mode of the operand must be the same as the mode of the operation, and STOWED mode is not allowed. The result of a NOT_OP is <u>always</u> of mode INTEGER, no matter what the operation mode.

NOT_OP is normally used to implement Boolean negation. For bitwise complementation, use COMPL_OP.

Example: !i (where i is an integer object with id 12)

```
38       NOT_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object has id 12
```

NULL_OP 39

```
int 39
```

The null operator is usually used to terminate lists constructed with the sequence operator SEQ_OP, or to indicate that a subtree has been omitted. For example, if a conditional has no else_part, the missing subtree must be represented by a NULL_OP. SEQ also acts as a delimiter at several places in the input stream.

Example:

```
         39       NULL_OP
```

OBJECT_OP 40

```
int 40
int mode
int object_id
```

The result of this operator is an lvalue, corresponding to a variable defined by the front end. 'Mode' is unrestricted; objects may have any primitive data mode, including STOWED (for arrays and records). The 'object_id' parameter gives the identification number that was supplied in the definition or declaration of the object.

Normally, each occurrence of a variable in the source program produces an OBJECT_OP in the intermediate form. OBJECTs are the primitive lvalues from which all other lvalue-producing constructs are derived.

Each object that is referenced in the intermediate form must be identified by a simple integer known as the "object id." Typically these ids are assigned at declaration time (for variables) or at time of first reference (for locations, like procedure names or statement labels). Object ids should be unique within each IMF module.

Example: i (where i is an integer object, with object id 12)

```
         40       OBJECT_OP
         1           INT_MODE
         12          Object id 12
```

ORAA_OP 41

```
int 41
int mode
tree left
tree right
```

The result of this operator is an rvalue, the bitwise inclusive-or of the values of the left and right operands. As a side

effect, the result is stored back into the left operand. The left operand must be an lvalue or a bit field (see FIELD_OP). The operation mode must be INT, LONG INT, UNSIGNED, or LONG UNSIGNED, and the modes of both operands must match the operation mode.

ORAA stands for "logical or and assign." It is used to implement the C assignment operator "|=".

Example: i |= 1 (where i is an integer object with id 12)

```
41       ORAA_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object id 12
9           CONST_OP
1              INT_MODE
1              Constant has length 1
1              Constant has value 1
```

OR_OP 42

```
int 42
int mode
tree left
tree right
```

The result of this operator is an rvalue, the bitwise inclusive-or of the values of the left and right operands. The operation mode must be INT, LONG INT, UNSIGNED, or LONG UNSIGNED, and the modes of both operands must match the operation mode.

OR is used to implement bit-oriented logical operations, like the "|" operator of C. Although OR can be used in Boolean expressions, the sequential-OR operator SOR_OP is usually more appropriate.

Example: i | 1 (where i is an integer object with id 12)

```
42       OR_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object id 12
9           CONST_OP
1              INT_MODE
1              Constant has length 1
1              Constant has value 1
```

POSTDEC_OP 43

```
int 43
int mode
tree left
tree right
```

The result of this operator is an rvalue, the value of the left operand before the operator is executed. As a side effect, the left operand is decremented by the value of the right operand. The left operand must be an lvalue or a bit field (see FIELD_OP), and must have the same mode as the operation. The right operand must be a CONST_OP, with the same mode as the operation.

The POSTDEC operator corresponds to the C postfix autodecrement construct.

Example: p-- (where p is a long unsigned (pointer) object with object id 15, and p is intended to point to integers)

```
43       POSTDEC_OP
4           LONG_UNS_MODE
40          OBJECT_OP
4              LONG_UNS_MODE
15             Object id of p
9           CONST_OP
4              LONG_UNS_MODE
2              Constant has length 2
0              Constant has value...
1              ...1, expressed as a long integer
```

POSTINC_OP 44

```
int 44
int mode
tree left
tree right
```

The result of this operator is an rvalue, the value of the left operand before the operator is executed. As a side effect, the left operand is incremented by the value of the right operand. The left operand must be an lvalue or a bit field (see FIELD_OP), and must have the same mode as the operation. The right operand must be a CONST_OP, with the same mode as the operation.

The POSTINC operator corresponds to the C postfix autoincrement construct.

Example: p++ (where p is a long unsigned (pointer) object with object id 15, and p is intended to point to integers)

```
44       POSTINC_OP
4           LONG_UNS_MODE
```

```
40          OBJECT_OP
4              LONG_UNS_MODE
15             Object id of p
9           CONST_OP
4              LONG_UNS_MODE
2              Constant has length 2
0              Constant has value...
1              ...1, expressed as a long integer
```

PREDEC_OP 45

```
int 45
int mode
tree left
tree right
```

The result of this operator is an rvalue, the value of the left operand decremented by the value of the right operand. As a side effect, the result is stored back into the left operand. The left operand must be an lvalue or a bit field (see FIELD_OP), and must have the same mode as the operation. The right operand must be a CONST_OP, with the same mode as the operation.

The PREDEC operator corresponds to the C prefix autodecrement construct.

Example: --p (where p is a long unsigned (pointer) object with object id 15, and p is intended to point to integers)

```
45       PREDEC_OP
4           LONG_UNS_MODE
40          OBJECT_OP
4              LONG_UNS_MODE
15             Object id of p
9           CONST_OP
4              LONG_UNS_MODE
2              Constant has length 2
0              Constant has value...
1              ...1, expressed as a long integer
```

PREINC_OP 46

```
int 46
int mode
tree left
tree right
```

The result of this operator is an rvalue, the value of the left operand incremented by the value of the right operand. As a side effect, the result is stored back into the left operand. The

left operand must be an lvalue or a bit field (see FIELD_OP), and must have the same mode as the operation. The right operand must be a CONST_OP, with the same mode as the operation.

The PREINC operator corresponds to the C prefix autoincrement construct.

Example: ++p (where p is a long unsigned (pointer) object with object id 15, and p is intended to point to integers)

```
46       PREINC_OP
4           LONG_UNS_MODE
40          OBJECT_OP
4              LONG_UNS_MODE
15             Object id of p
9           CONST_OP
4              LONG_UNS_MODE
2              Constant has length 2
0              Constant has value...
1              ...1, expressed as a long integer
```

PROC_CALL_ARG_OP 47

```
int 47
int mode
tree expression
tree next_argument
```

Procedure call arguments are specified in a linked list of PROC_CALL_ARG_OPs attached to a PROC_CALL_OP. An argument expression is specified by the parameter 'expression'; its mode must be given by the parameter 'mode'. The parameter 'next_argument' is simply the next procedure argument in the list. Any mode expression is allowable as an argument, since the Prime procedure call convention passes a fixed-size pointer to the actual argument, rather than the argument itself.

Note that arguments (with the exception of bit fields) are always passed by reference. If arguments are to be copied on procedure entry or exit, the called procedure must do the copying. (See PROC_DEFN_ARG_OP; an argument will be copied automatically if it is given the disposition VALUE_DISP.) Bit fields are an exception; they are not addressable objects, and so are always passed by value.

See PROC_CALL_OP for examples of PROC_CALL_ARG_OP.

PROC_CALL_OP 48

```
int 48
int mode
```

```
tree procedure
tree argument_list
```

The PROC_CALL_OP is used to generate a call to a procedure. The parameter 'mode' is the mode of the return value of the procedure, if any. The parameter 'procedure' is an lvalue representing the address of the procedure to be called; the most common case is simply an OBJECT_OP with an object id equal to the id of a declared procedure (see PROC_DEFN_OP). The parameter 'argument_list' is a singly-linked list of expressions to be passed as arguments to the procedure; each expression in the argument list is contained in a PROC_CALL_ARG_OP subtree, and the entire list is terminated with a NULL_OP.

PROC_CALL implements invocation of both "procedures" and "functions" (or "value-returning procedures").

Example: l = strlen (s)
where l is an integer object with id 13,
s is a STOWED object (an array of integers) with id 14,
and strlen is a procedure with id 50.

```
5        ASSIGN_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
13             object id for l
48          PROC_CALL_OP
1              INT_MODE
40             OBJECT_OP; this gives the procedure address
7                 STOWED_MODE
50                Object id for strlen
47             PROC_CALL_ARG_OP; description of first arg
7                 STOWED_MODE
40                OBJECT_OP
7                    STOWED_MODE
14                   Object id for s
39                NULL_OP; ends list of arguments
1           Number of words transferred by ASSIGN
```

PROC_DEFN_ARG_OP 49

```
int 49
int object_id
int mode
int disposition
int length
tree next_argument
```

This operator cannot be used as part of the code of a procedure. See "Operators Useful in the Procedure Definition Stream".

PROC_DEFN_OP 50

```
int 50
int object_id
int number_of_args
string proc_name
tree argument_list
tree code
```

This operator cannot be used as part of the code of a procedure. See "Operators Useful in the Procedure Definition Stream".

REFTO_OP 51

```
int 51
int mode
tree operand
```

The result of this operator is an rvalue, the virtual memory address of the operand. The operand must be an lvalue, but it can have any mode. In particular, the operand may not be a bit field. The operation mode must be LONG INT or LONG UNSIGNED.

REFTO implements the unary "&" operator in C.

Example: &i (where i is an integer object with id 12)

```
51       REFTO_OP
4           LONG_UNS_MODE; pointers are generally of this mode
40          OBJECT_OP
1              INT_MODE
12             Object id for i
```

REMAA_OP 52

```
int 52
int mode
tree left
tree right
```

The result of this operation is an rvalue, the remainder resulting from division of the value of the left operand by the value of the right. As a side effect, the result is stored back into the left operand. The left operand must be an lvalue or a bit field. Both operands must have the same mode as the operation, and the operation mode may not be STOWED, FLOAT, or LONG FLOAT. (The restriction against floating point operands may be lifted in the near future.)

Note that this operator produces the remainder resulting from the division; the remainder may be negative. If a true modulus is desired, the absolute value of the left operand should be remaindered by the right operand, instead.

Example: i %= 2 (where i is an integer object with id 12)

```
52       REMAA_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object id for i
9           CONST_OP
1              INT_MODE
1              Length of constant is 1 word
2              Value of constant is 2
```

REM_OP 53

```
int 53
int mode
tree left
tree right
```

The result of this operation is an rvalue, the remainder resulting from division of the value of the left operand by the value of the right. Both operands must have the same mode as the operation, and the operation mode may not be STOWED, FLOAT, or LONG FLOAT. (The restriction against floating point operands may be lifted in the near future.)

Note that this operator produces the remainder resulting from the division; the remainder may be negative. If a true modulus is desired, the absolute value of the left operand should be remaindered by the right operand, instead.

Example: i % 2 (where i is an integer object with id 12)

```
53       REM_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object id for i
9           CONST_OP
1              INT_MODE
1              Length of constant is 1 word
2              Value of constant is 2
```

RETURN_OP 54

```
int 54
int mode
tree operand
```

The operand is evaluated and returned as the result of the current procedure. If the operand is absent (represented by a NULL_OP) a procedure return takes place, but no effort is made to return a particular value. The operation mode may not be STOWED.

This operator is used to implement the "return" statement in many algorithmic languages. All procedures should end with a RETURN_OP.

Example: return (0)

```
54       RETURN_OP
9           CONST_OP
1              INT_MODE
1              Constant has length 1
0              Constant has value 0
```

RSHIFTAA_OP 55

```
int 55
int mode
tree left
tree right
```

The result of this operator is an rvalue, the value of the left operand shifted right the number of bit places specified by the value of the right operand. As a side effect, the result is stored back into the left operand. The left operand must be an lvalue or a bit field (see FIELD_OP). The operation mode may be INT, LONG INT, UNSIGNED, or LONG UNSIGNED, and the left operand must have the same mode. The right operand must be of mode INT or UNSIGNED, and really should have a value between 0 and the length of the left operand, inclusive. (Reasonable results outside this range are not guaranteed.)

RSHIFTAA stands for "right-shift and assign." The operator is used to implement ">>=" in C. If the operation mode is UNSIGNED or LONG UNSIGNED, the vacated bits on the left are zero-filled (logical shift); if the operation mode is INT or LONG INT, the vacated bits on the left are sign-filled (arithmetic shift).

Example: i >>= 1 (where i is an integer object with id 12)

```
55       RSHIFTAA_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
```

```
12             Object id for i
9           CONST_OP
1              INT_MODE
1              Length of constant is 1 word
1              Value of constant is 1
```

RSHIFT_OP 56

```
int 56
int mode
tree left
tree right
```

The result of this operator is an rvalue, the value of the left operand shifted right the number of bit places specified by the value of the right operand. The operation mode may be INT, LONG INT, UNSIGNED, or LONG UNSIGNED, and the left operand must have the same mode. The right operand must be of mode INT or UNSIGNED, and really should have a value between 0 and the length of the left operand, inclusive. (Reasonable results outside this range are not guaranteed.)

This operator is used to implement ">>" in C. If the operation mode is UNSIGNED or LONG UNSIGNED, the vacated bits on the left are zero-filled (logical shift); if the operation mode is INT or LONG INT, the vacated bits on the left are sign-filled (arithmetic shift).

Example: i >> 1 (where i is an integer object with id 12)

```
56       RSHIFT_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object id for i
9           CONST_OP
1              INT_MODE
1              Length of constant is 1 word
1              Value of constant is 1
```

SAND_OP 57

```
int 57
int mode
tree left
tree right
```

The result of this operation is an rvalue. The left operand is evaluated first. If it is zero, the result of the operation is zero and evaluation is terminated. If it is non-zero, then the

value of the right operand is returned as the result of the operator. The modes of both operands must be the same as the mode of the result.

SAND is used to implement sequential ("short-circuit") logical conjunctions.

Example: i && j (where i, j are integer objects with ids 12 and 13)

```
57       SAND_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object id for i
40          OBJECT_OP
1              INT_MODE
13             Object id for j
```

SELECT_OP 58

```
int 58
int mode
int offset
tree structure
```

The result of this operator is an lvalue, one member of a heterogeneous data structure (ala the Pascal "record" or the C "struct"). The parameter 'mode' is the mode of the element selected; it is unrestricted. The parameter 'structure' is an lvalue expression yielding the base address of the structure. Typically it is an OBJECT_OP with an object_id field equal to the object id of a STOWED object defined by DEFINE_STAT or DEFINE_DYNM.

Example: rec.field
(rec is a record with object id 4;
field is an integer field offset 3 words from the beginning of the record)

```
58       SELECT_OP
1           INT_MODE
3           Offset from beginning of struct
40          OBJECT_OP
7              STOWED mode
4              Object id of 'rec'
```

SEQ_OP 59

```
int 59
tree left
```

tree right

SEQ causes the left operand to be evaluated, then the right operand. The result is the result of the right operand.

SEQ_OP corresponds roughly to the "," operator in C and the semicolon statement separator in Pascal.

Example: i = 1; j = 2
(where i, j are integer objects with ids 12, 13)

```
59       SEQ_OP
5           ASSIGN_OP
1              INT_MODE
40             OBJECT_OP
1                 INT_MODE
12                Object id of 'i'
9              CONST_OP
1                 INT_MODE
1                 Constant length is 1 word
1                 Constant value is 1
1              Assignment transfers 1 word
5           ASSIGN_OP
1              INT_MODE
40             OBJECT_OP
1                 INT_MODE
13                Object id of 'j'
9              CONST_OP
1                 INT_MODE
1                 Constant length is 1 word
1                 Constant value is 1
1              Assignment transfers 1 word
```

A frequently-used alternative to the above is

```
59       SEQ_OP
5           ASSIGN_OP
1              INT_MODE
40             OBJECT_OP
1                 INT_MODE
12                Object id of 'i'
9              CONST_OP
1                 INT_MODE
1                 Constant length is 1 word
1                 Constant value is 1
1              Assignment transfers 1 word
59       SEQ_OP
5           ASSIGN_OP
1              INT_MODE
40             OBJECT_OP
1                 INT_MODE
13                Object id of 'j'
9              CONST_OP
1                 INT_MODE
1                 Constant length is 1 word
1                 Constant value is 1
```

```
1              Assignment transfers 1 word
39       NULL_OP; end of sequence
```

SOR_OP 60

```
int 60
int mode
tree left
tree right
```

The result of this operator is an rvalue. The left operand is evaluated first. If it is non-zero, it is returned as the result of the operation. If it is zero, the value of the right operand is returned as the result of the operation. The mode of the operation result is always INTEGER. The operands may be of any mode other than STOWED.

SOR is used to implement sequential ("short-circuit") logical disjunctions.

Example: i || j (where i, j are integer objects with ids 12 and 13)

```
60       SOR_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object id for i
40          OBJECT_OP
1              INT_MODE
13             Object id for j
```

SUBAA_OP 61

```
int 61
int mode
tree left
tree right
```

The result of this operator is an rvalue, the value of the left operand minus the value of the right operand. As a side effect, the difference is stored back into the left operand. The left operand must be an lvalue or a bit field (see FIELD_OP). Both operands must have the same mode as the operation, and the mode may not be STOWED.

SUBAA stands for "subtract and assign." It is used to implement the "-=" operator of C and the "-:=" or "minusab" operator of Algol 68.

Example: i -= 1 (where i is an integer object with id 12)

```
61       SUBAA_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object id for 'i'
9           CONST_OP
1              INT_MODE
1              Constant is of length 1
1              Constant has value 1
```

SUB_OP 62

```
int 62
int mode
tree left
tree right
```

The result of this operator is an rvalue, the value of the left operand minus the value of the right operand. Both operands must have the same mode as the operation, and that mode may not be STOWED.

Example: i - 1 (where i is an integer object with id 12)

```
62       SUB_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object id for 'i'
9           CONST_OP
1              INT_MODE
1              Constant is of length 1
1              Constant has value 1
```

SWITCH_OP 63

```
int 63
int mode
tree selector
tree alternative_list
```

SWITCH_OP is used to generate a multiway-branch statement, like the 'switch' of C or the 'case' of Pascal. When the SWITCH is used as a value-returning construct, the modes of all the CASESs must match the operation mode, and must not be STOWED. The parameter 'selector' is an expression to be evaluated and compared with all alternative values in CASE_OPs. 'Alternative_list' is a singly-linked list of CASE_OPs and at most one DEFAULT_OP, terminated with a NULL_OP.

Note that there is no automatic jump from the end of an alternative to the end of the switch; if one is desired, a BREAK_OP should be used. This behavior allows construction of alternatives with multiple case labels, as illustrated in the example below.

Example: The following Pascal 'case' statement, assuming i and j are integer variables with object ids 12 and 13, respectively

```
case i of
   1: j := 6;
   2, 4: j := 10;
   otherwise j := 9;
   end;
```

```
63       SWITCH_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object id for 'i'
7           CASE_OP; the first alternative
9              CONST_OP
1                 INT_MODE
1                 Length of constant is 1
1                 Value of constant is 1
59             SEQ_OP; actions for first CASE
5                 ASSIGN_OP
1                    INT_MODE
40                   OBJECT_OP
1                       INT_MODE
13                      Object id for 'j'
9                    CONST_OP
1                       INT_MODE
1                       Length of constant is 1 word
6                       Value of constant is 6
1                    Assignment transfers 1 word
59             SEQ_OP; continuing CASE actions
6                 BREAK_OP
1                    1 Level (the SWITCH)
39             NULL_OP; end of CASE actions
7           CASE_OP; second alternative
9              CONST_OP
1                 INT_MODE
1                 Constant has length 1
2                 Constant has value 2
39             NULL_OP; no actions, control falls through
7           CASE_OP; second case of second alternative
9              CONST_OP
1                 INT_MODE
1                 Constant has length 1
4                 Constant has value 4
59             SEQ_OP; beginning of actions
5                 ASSIGN_OP
1                    INT_MODE
40                   OBJECT_OP
1                       INT_MODE
```

```
13                      Object id for 'j'
9                    CONST_OP
1                       INT_MODE
1                       Constant has length 1
10                      Constant has value 10
1                    Assignment transfers 1 word
59                SEQ_OP; actions continue
6                    BREAK_OP
1                       1 Level
39                NULL_OP; end of actions
12             DEFAULT_OP; default actions for SWITCH
59                SEQ_OP; beginning of actions
5                    ASSIGN_OP
1                       INT_MODE
40                      OBJECT_OP
1                          INT_MODE
13                         Object id for 'j'
9                       CONST_OP
1                          INT_MODE
1                          Length 1
9                          Value 9
1                       Assignment transfers 1 word
59                SEQ_OP; default actions continue
6                    BREAK_OP
1                       1 Level
39                NULL_OP; end of default actions
39             NULL_OP; end of alternatives for SWITCH
```

UNDEFINE_DYNM_OP 64

int 64
int object_id

UNDEFINE_DYNM is used to release space assigned to an object allocated in the current local storage area. The parameter 'object_id' is the object identifier used in the DEFINE_DYNM_OP that assigned space to the object.

This operator is rarely used; it is normally unnecessary unless the language supported by the front-end allows nested blocks or the front-end generates and deallocates temporary variables explicitly.

Example: If object number 44 has been allocated by the front end as a temporary, it can be deallocated with

```
64          UNDEFINE_DYNM_OP
44             ID of object to be deallocated
```

WHILE_LOOP_OP 65

```
int 65
tree condition
tree body
```

WHILE_LOOP_OP generates a test-at-the-top loop. The parameter 'condition' must be an expression yielding a result of zero (for loop termination) or non-zero (for loop continuation). The parameter 'body' is the body of the loop (which may contain BREAK ops for early termination or NEXT ops for explicit continuation).

Example: while (i < j) do i <<= 1;
where i, j are integer objects with ids 12 and 13

```
65       WHILE_LOOP_OP
31          LT_OP
1              INT_MODE
40             OBJECT_OP
1                 INT_MODE
12                Object ID for i
40             OBJECT_OP
1                 INT_MODE
13                Object ID for j
29          LSHIFTAA_OP
1              INT_MODE
40             OBJECT_OP
1                 INT_MODE
12                Object ID for i
9              CONST_OP
1                 INT_MODE
1                 Length 1
1                 Value 1
```

XORAA_OP 66

```
int 66
int mode
tree left
tree right
```

The result of this operator is an rvalue, the bitwise exclusive-or of the values of the left and right operands. As a side effect, the result is stored back into the left operand. The left operand must be an lvalue or a bit field (see FIELD_OP). Both operands must have the same mode as the operation, and only modes INT, LONG INT, UNSIGNED, and LONG UNSIGNED are allowable.

XORAA stands for "exclusive-or and assign." It is used to implement the "^=" operator of C.

Example: i ^= 1 (where i is an integer object with id 12)

```
66       XORAA_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object id for 'i'
9           CONST_OP
1              INT_MODE
1              Constant is of length 1
1              Constant has value 1
```

XOR_OP 67

```
int 67
int mode
tree left
tree right
```

The result of this operator is an rvalue, the bitwise exclusive-or of the values of the left and right operands. Both operands must have the same mode as the operation, and only modes INT, LONG INT, UNSIGNED, and LONG UNSIGNED are allowable.

Example: i ^ 1 (where i is an integer object with id 12)

```
67       XOR_OP
1           INT_MODE
40          OBJECT_OP
1              INT_MODE
12             Object id for 'i'
9           CONST_OP
1              INT_MODE
1              Constant is of length 1
1              Constant has value 1
```

ZERO_INITIALIZER_OP 68

```
int 68
int size
tree next_initializer
```

Initializers are the initial-value expressions that appear in definitions of variables in C (see DEFINE_DYNM_OP and DEFINE_STAT_OP). Local variables are reinitialized whenever the procedure containing them is entered; global (static) variables are initialized only when the program containing them is loaded.

ZERO_INITIALIZER provides a compact way of specifying an all-zeros initializer. The parameter 'size' is the number of 16-bit zero words to be generated; 'next_initializer' is simply a link to the next INITIALIZER or ZERO_INITIALIZER in a variable's

initial-value list.

Example: int a[3] = {0, 0, 0} (a global declaration where 'a' has object id 1)

```
14       DEFINE_STAT_OP
1           Object id for 'a'
68          ZERO_INITIALIZER_OP
3              Fill 3 words with zero
39             NULL_OP; no more initializers
3           Size of 'a', in 16-bit words
```

Extended Examples

These examples should illustrate some global aspects of using the code generator. They include a segment of C source code, the (annotated) intermediate form code produced by the C front end, and the (annotated) assembly language generated by the VCG.

Basic VCG Input

C Code

```
extern int e1, e2;      /* defined outside this module */

int v1, v2;             /* defined here, visible outside */

static int s1, s2;      /* defined here, not visible outside */

proc1 ()                /* procedure defined here, visible outside */
   {
   }

proc2 ()                /* more of the same */
   {
   }
```

IMF Stream 1

```
32    A MODULE_OP; begins the input module
59       SEQ_OP; initiates the sequence of entry points
7           Object number 7 is an entry point...
5              whose name is 5 characters long...
240               p
242               r
239               o
227               c
177               1
59       SEQ_OP; next member of the list of entry points
8           Object number 8 is an entry point...
5              whose name is 5 characters long...
240               p
242               r
239               o
227               c
178               2
59       SEQ_OP; next member of the list of entry points
3           Object number 3 is an entry point...
2              whose name is 2 characters long...
246               v
```

```
177               1
59       SEQ_OP; next member of the list of entry points
4           Object number 4 is an entry point...
2              whose name is 2 characters long...
246               v
178               2
39       NULL_OP; terminates the list of entries in this module
39    NULL_OP; terminates the list of modules in the input
```

IMF Stream 2

```
32    MODULE_OP; beginning of this input module
59       SEQ_OP; beginning of static data declarations list
14          DEFINE_STAT_OP; reserve space for an object
3              Object ID is 3
39             NULL_OP; there are no initializers for this object
1              Its size is 1 word
59       SEQ_OP; next element of declarations list
14          DEFINE_STAT_OP; reserve space for an object
4              Object ID is 4
39             NULL_OP; there are no initializers for this object
1              Its size is 1 word
59       SEQ_OP; next element of declarations list
14          DEFINE_STAT_OP; reserve space for an object
5              Object ID is 5
39             NULL_OP; there are no initializers for this object
1              Its size is 1 word
59       SEQ_OP; next element of declarations list
14          DEFINE_STAT_OP; reserve space for an object
6              Object ID is 6
39             NULL_OP; there are no initializers for this object
1              Its size is 1 word
59       SEQ_OP; next element of declarations list
11          DECLARE_STAT_OP; declare object defined outside this module
1              Object ID is 1
2              Name has 2 characters...
229               e
177               1
59       SEQ_OP; next element of declarations list
11          DECLARE_STAT_OP; declare object defined outside this module
2              Object ID is 2
2              Name has 2 characters...
229               e
178               2
39       NULL_OP; end of static data definition/declaration list
39    NULL_OP; end of modules in input stream
```

IMF Stream 3

```
32    MODULE_OP; beginning of next module in input stream
59       SEQ_OP; first element of procedure definitions list
50          PROC_DEFN_OP; procedure definition follows
7              Procedure is object number 7
0              Procedure has no arguments
```

```
5              Procedure name is 5 characters long...
240               p
242               r
239               o
227               c
177               1
39             NULL_OP; empty argument description list
39             NULL_OP; no code for this procedure
59       SEQ_OP; next element of procedure definitions list
50          PROC_DEFN_OP; procedure definition follows
8              Procedure is object number 8
0              Procedure has no arguments
5              Procedure name is 5 characters long...
240               p
242               r
239               o
227               c
178               2
39             NULL_OP; empty argument description list
39             NULL_OP; no code for this procedure
39       NULL_OP; end of procedure definitions list (and this module)
39    NULL_OP; end of modules in this input stream
```

PMA Code

```
 SEG              Assemble in 64V mode
 RLIT             Place literals in procedure frame
 SYML             Allow 8-character external names
 ENT PROC1,L7_    PROC1 is an entry point with address L7_
 ENT PROC2,L8_    Similarly for PROC2,
 ENT V1,L3_          V1,
 ENT V2,L4_          and V2
 LINK             Output data in link (static) frame
L3_ EQU *
 BSZ '1           Reserve one word for L3_, init to zero
 PROC             Output data in proc (procedure) frame
 LINK
L4_ EQU *
 BSZ '1           Reserve one word for L4_, init to zero
 PROC
 LINK
L5_ EQU *
 BSZ '1           Reserve one word for L5_, init to zero
 PROC
 LINK
L6_ EQU *
 BSZ '1           Reserve one word for L6_, init to zero
 PROC
 LINK
 EXT E1           Declare symbol E1 external to this module
L1_ EQU *
 IP E1            Generate a pointer for the loader to fill in
 PROC
 LINK
 EXT E2           Declare symbol E2 external to this module
```

```
L2_ EQU *
 IP E2            Generate a pointer for the loader
 PROC
 PROC
L65535_ EQU *     Beginning of a procedure
 EAL L7_          Set up stack frame owner pointer for debugging
 STL SB%+18
 LDA =’4000
 STA% SB%
 PRTN             "Procedure Return" at end of procedure
L7_ ECB L65535_,,SB%+’0,0,’24    Entry control block for procedure
 DATA ’5          PL/I character varying form procedure name
 DATA ’170362
 DATA ’167743
 DATA ’130405
 PROC
L65534_ EQU *     Beginning of second procedure
 EAL L8_          Set up stack frame owner pointer
 STL SB%+18
 LDA =’4000
 STA% SB%
 PRTN
L8_ ECB L65534_,,SB%+’0,0,’24    Entry control block
 DATA ’5          Procedure name
 DATA ’170362
 DATA ’167743
 DATA ’131370
 END              End of this module
```

C Code

```
int   i,             /* a static integer variable */
      ii [10];       /* a static integer array */

struct
   {
   int f1, f2;
   } s;              /* a static structure with two integer fields */

main (argc, argv)    /* a non-trivial procedure, with arguments */
int argc;               /* integer argument */
char **argv;            /* pointer-to-pointer-to-character argument */
   {
   int li,              /* a local integer variable */
       lii [10];        /* a local integer array */

   struct
      {
      int m1, m2;
      } ls;             /* a local structure with two integer fields */

   i;                   /* use of various things in expressions */
   ii [0];
   s.f1;
   li;
   lii [0];
   ls.m1;
   argv;
   argc;
   }
```

IMF Stream 1

```
32    MODULE_OP; beginning of next module in input stream
59       SEQ_OP; beginning of entry point declaration list
1           Object number 1 is an entry point...
1              whose name is 1 character long...
233               i
59       SEQ_OP; next member of entry point list
3           Object number 3 is an entry point...
1              whose name is 1 character long...
243               s
59       SEQ_OP; next member of entry point list
4           Object number 4 is an entry point...
4              whose name is 4 characters long...
237               m
225               a
233               i
238               n
59       SEQ_OP; next member of entry point list
2           Object number 2 is an entry point...
```

```
2              whose name is 2 characters long...
233               i
233               i
39       NULL_OP; end of entry point list (and this module)
39    NULL_OP; end of modules in this input stream
```

IMF Stream 2

```
32    MODULE_OP; beginning of next module
59       SEQ_OP; beginning of static data declarations/definitions
14          DEFINE_STAT_OP; reserve space for a static variable
1              Object ID is 1
39             NULL_OP; no initializers for this variable
1              Object size is 1 word
59       SEQ_OP; next member of static data list
14          DEFINE_STAT_OP; reserve space for a static variable
2              Object ID is 2
39             NULL_OP; no initializers for this variable
10             Object size is 10 words
59       SEQ_OP; next member of static data list
14          DEFINE_STAT_OP; reserve space for a static variable
3              Object ID is 3
39             NULL_OP; no initializers for this variable
2              Object size is 2 words
39       NULL_OP; end of static data list
39    NULL_OP; end of modules in this input stream
```

IMF Stream 3

```
32    MODULE_OP; beginning of next module in input stream
59       SEQ_OP; beginning of procedure definition list
50          PROC_DEFN_OP; procedure definition follows
4              Object ID of procedure is 4
2              Procedure has 2 arguments
4              Procedure name is 4 characters long...
237               m
225               a
233               i
238               n
49             PROC_DEFN_ARG_OP; description of first argument
5                 Argument has object ID 5
1                 Argument has mode 1 (INTEGER)
0                 Argument has disposition 0 (pass-by-value)
1                 Argument is 1 word long
49                PROC_DEFN_ARG_OP; description of second argument
6                    Argument has object ID 6
4                    Argument has mode 4 (LONG UNSIGNED, or pointer)
1                    Argument has disposition 1 (pass-by-reference)
2                    Argument is 2 words long
39                   NULL_OP; end of argument descriptor list
59             SEQ_OP; beginning of procedure code
13                DEFINE_DYNM_OP; reserve space for local variable
7                    Object ID 7
39                   NULL_OP; no initializers
```

```
1                  Size 1 word
59              SEQ_OP; next element of procedure code
13                 DEFINE_DYNM_OP; reserve space for local variable
8                     Object ID 8
39                    NULL_OP; no initializers
10                    Size 10 words
59              SEQ_OP; next element of procedure code
13                 DEFINE_DYNM_OP; reserve space for local variable
9                     Object ID 9
39                    NULL_OP; no initializers
2                     Size 2 words
59              SEQ_OP; next element of procedure code
40                 OBJECT_OP; (this is actually an expression subtree)
1                     Mode 1 (INTEGER)
1                     Object ID 1
59              SEQ_OP; next element of procedure code
25                 INDEX_OP; again, the top of an expression subtree
1                     Mode 1 (INTEGER)
40                    OBJECT_OP; this one is the base address of the array
7                        Mode 7 (STOWED)
2                        Object ID 2
9                     CONST_OP; this one is the index expression
1                        Mode 1 (INTEGER)
1                        Length is 1 word
0                        Value of word is 0
1                     Array element size is 1 word
59              SEQ_OP; next element of procedure code
58                 SELECT_OP; again, the top of an expression subtree
1                     Mode 1 (INTEGER)
0                     Field to be selected has word offset 0 from base
40                    OBJECT_OP; the base address of the structure
7                        Mode 7 (STOWED)
3                        Object ID 3
59              SEQ_OP; next element of procedure code
40                 OBJECT_OP; an expression, again
1                     Mode 1 (INTEGER)
7                     Object ID is 7
59              SEQ_OP; next element of procedure code
25                 INDEX_OP; using an array element as an expression
1                     Mode 1 (INTEGER)
40                    OBJECT_OP; this is the base of the array being indexed
7                        Mode 7 (STOWED)
8                        Object ID is 8
9                     CONST_OP; this is the subscript expression
1                        MODE 1 (INTEGER)
1                        Length of constant is 1 word
0                        Value of constant is 0
1                     Array element size is 1 word
59              SEQ_OP; next element of procedure code
58                 SELECT_OP; using struct field as an expression
1                     Mode 1 (INTEGER)
0                     Offset of selected field is 0 words from base
40                    OBJECT_OP; this is the base address of the structure
7                        Mode 7 (STOWED)
9                        Object ID is 9
59              SEQ_OP; next element of procedure code
```

```
40                OBJECT_OP; just the top of an expression tree
4                    Mode 4 (LONG_UNSIGNED, or pointer)
6                    Object ID is 6
59             SEQ_OP; next element of procedure code
40                OBJECT_OP; an expression, again
1                    Mode 1 (INTEGER)
5                    Object ID is 5
39             NULL_OP; end of procedure body code (and proc defn)
39       NULL_OP; end of procedure defn list (and this module)
39    NULL_OP; end of this input stream
```

PMA Code

```
 SEG              Assemble in 64V mode
 RLIT             Place literals in procedure frame
 SYML             Allow 8-character external symbols
 ENT I,L1_        I is an entry point, with address L1_
 ENT S,L3_        S is an entry point, with address L3_
 ENT MAIN,L4_     MAIN is an entry point, with address L4_
 ENT II,L2_       II is an entry point, with address L2_
 LINK             Emit data in link (static data) frame
L1_ EQU *
 BSZ '1           Reserve 1 word for L1_
 PROC
 LINK
L2_ EQU *
 BSZ '12          Reserve 10 words ('12 octal) for L2_
 PROC
 LINK
L3_ EQU *
 BSZ '2           Reserve 2 words for L3_
 PROC
 PROC
L65535_ EQU *     Beginning of a procedure
 ARGT             Transfer arguments from caller
 EAL L4_          Set up stack frame owner pointer for debugging
 STL SB%+18
 LDA ='4000
 STA% SB%
 LDA SB%+'24,*    Make copy of pass-by-value arguments
 STA SB%+'24
 LDA LB%+'400     Evaluate expression 1,
 LDA LB%+'401                         2,
 LDA LB%+'413                         3,
 LDA SB%+'25                          4,
 LDA SB%+'32                          5,
 LDA SB%+'44                          6,
 LDL SB%+'27                          7,
 LDA SB%+'24                          8
 PRTN             Return from the procedure
L4_ ECB L65535_,,SB%+'24,2,'46      Entry control block
 DATA '4          PL/I char varying procedure name
 DATA '166741
 DATA '164756
 END              End of this PMA module
```

String Copy

C Code

```
strcpy (s, t)     /* copy string s to string t */
char s[], t[];
   {
   int i;         /* a local integer variable, for indexing */

   i = 0;         /* start at first char */
   while ((t[i] = s[i]) != '\0')    /* copy until a zero char is seen */
      i += 1;     /* incrementing the index each time */
   }
```

IMF Stream 1

```
32    MODULE_OP; begins the input module
59       SEQ_OP; begins sequence of entry points
1           Object number 1 is an entry point
6              whose name is 6 characters long...
243               s
244               t
242               r
227               c
240               p
249               y
39       NULL_OP; terminates entry point list
39    NULL_OP; terminates list of modules in the input
```

IMF Stream 2

```
32    MODULE_OP; begins the input module
39       NULL_OP; terminates the sequence of static data definitions
39    NULL_OP; terminates list of modules in the input
```

IMF Stream 3

```
32    MODULE_OP; begins next module in the input stream
59       SEQ_OP; first procedure definition follows
50          PROC_DEFN_OP; procedure definition follows
1              Procedure is object number 1
2              There are 2 arguments, described below.
6              Procedure name is 6 characters long...
243               s
244               t
242               r
227               c
240               p
249               y
49             PROC_DEFN_ARG_OP; description of argument number 1
2                 Argument is object number 2
4                 LONG_UNS_MODE; argument is a pointer
```

```
1                  REF_DISP; argument is passed-by-reference
2                  Argument is 2 words long
49                 PROC_DEFN_ARG_OP; description of argument number 2
3                     Argument is object number 3
4                     LONG_UNS_MODE; argument is a pointer
1                     REF_DISP; argument is passed-by-reference
2                     Argument is 2 words long
39                    NULL_OP; end of argument descriptions
59              SEQ_OP; beginning of procedure code list
13                 DEFINE_DYNM_OP; declare a local variable
4                     Variable has object id 4
39                    No initializers
1                     Variable is 1 word in length
59              SEQ_OP; next element of code list
5                  ASSIGN_OP
1                     INT_MODE
40                    OBJECT_OP
1                        INT_MODE
4                        Object id is 4
9                     CONST_OP
1                        INT_MODE
1                        Constant has length 1
0                        Constant has value 0
1                     Assignment transfers 1 word
59              SEQ_OP; next element of code list
65                 WHILE_OP
37                    NE_OP
1                        INT_MODE
5                        ASSIGN_OP
1                           INT_MODE
25                          INDEX_OP; the LHS of the assignment
1                              INT_MODE
15                             DEREF_OP; this is the base address
7                                 STOWED_MODE
40                                OBJECT_OP
4                                    LONG_UNS_MODE
3                                    Object id is 3
40                             OBJECT_OP; this is the subscript
1                                 INT_MODE
4                                 Object id is 4
1                              Array element size is 1 word
25                          INDEX_OP; the RHS of the assignment
1                              INT_MODE
15                             DEREF_OP; the base address expression
7                                 STOWED_MODE
40                                OBJECT_OP
4                                    LONG_UNS_MODE
2                                    Object id is 2
40                             OBJECT_OP; the subscript, again
1                                 INT_MODE
4                                 Object id is 4
1                              Array element size is 1 word
1                           Assignment transfers 1 word
9                        CONST_OP; the right operand of the NE_OP
1                           INT_MODE
1                           Constant is 1 word long
```

```
0                          Constant has value 0
59                     SEQ_OP; beginning of the body of the WHILE loop
1                         ADDAA_OP
1                            INT_MODE
40                           OBJECT_OP
1                               INT_MODE
4                               Object id is 4
9                            CONST_OP
1                               INT_MODE
1                               Length is 1 word
1                               Value is 1
39                     NULL_OP; end of the body of the WHILE loop
39               NULL_OP; end of the statements for the current procedure
39         NULL_OP; end of the procedure definitions in this module
39      NULL_OP; end of this input stream
```

PMA Code

```
 SEG
 RLIT
 SYML
 ENT STRCPY,L1_
 PROC
L65535_ EQU *
 ARGT              Transfer arguments from caller
 EAL L1_           Set up stack frame owner pointer for debugging
 STL SB%+18
 LDA =’4000
 STA% SB%
 CRA               Load A with zero
 STA SB%+’32       Store in i
 JMP L65533_       Enter the WHILE loop at the test
 FIN               (dump literals here)
L65532_ EQU *      Top of the WHILE loop body
 IRS SB%+’32       Increment i
 RCB                  (takes two instructions on this turkey machine)
L65533_ EQU *      WHILE loop test starts here
 LDX SB%+’32       Load index register with i
 LDA SB%+’24,*X    Load next character in string s
 STA SB%+’27,*X    Store it in next position in string t
 BNE L65532_       If it’s non-zero, go back for more characters
L65534_ EQU *      Exit label for the WHILE loop
 PRTN              Return from string copy procedure
L1_ ECB L65535_,,SB%+’24,2,’33
 DATA ’6
 DATA ’171764
 DATA ’171343
 DATA ’170371
 END
```

Tree Print

C Code

```
/* recursive tree-printing routine */

#define NULL 0    /* a nil pointer */

struct TNODE      /* the data structure out of which the tree is built */
   {
   int value;
   struct TNODE *left, *right;
   };
typedef struct TNODE tnode;      /* create a new type, for convenience */


treeprint (t)
tnode *t;
   {
   if (t != NULL)
      {
      treeprint (t->left);
      printf ("%4d\n", t->value);      /* output the 'value' field */
      treeprint (t->right);
      }
   }
```

IMF Stream 1

```
32    MODULE_OP; beginning of next module in input stream
59       SEQ_OP; beginning of list of entry point declarations
1           Object number 1 is an entry point
9              whose name is 9 characters long...
244               t
242               r
229               e
229               e
240               p
242               r
233               i
238               n
244               t
39       NULL_OP; end of entry point list for this module
39    NULL_OP; end of modules in this input stream
```

IMF Stream 2

```
32    MODULE_OP; beginning of next module in this input stream
59       SEQ_OP; beginning of list of static data definitions
11          DECLARE_STAT_OP; declare an externally-defined object
3              Object has object id 3
6              Name of object is 6 characters long...
240               p
```

```
242               r
233               i
238               n
244               t
230               f
39       NULL_OP; end of static data for this module
39    NULL_OP; end of modules in this input stream
```

IMF Stream 3

```
32    MODULE_OP; beginning of next module in input stream
59       SEQ_OP; beginning of list of procedure definitions
50          PROC_DEFN_OP; procedure definition follows
1              Procedure has object id 1
1              Procedure has 1 argument
9              Procedure name is 9 characters long...
244               t
242               r
229               e
229               e
240               p
242               r
233               i
238               n
244               t
49             PROC_DEFN_ARG_OP; description of procedure argument
2                 Argument has object id 2
4                 Argument has mode LONG_UNS (it's a pointer)
1                 Argument has REF disposition (pass-by-reference)
2                 Argument is 2 words long
39                NULL_OP; no further argument descriptions
59             SEQ_OP; beginning of statement list
24                IF_OP
1                    INT_MODE
37                   NE_OP
4                       LONG_UNS_MODE
40                      OBJECT_OP
4                          LONG_UNS_MODE
2                          Object has id 2
9                       CONST_OP
4                          LONG_UNS_MODE
2                          Constant has length 2
0                          First word of constant is 0
0                          Second word of constant is 0
59                   SEQ_OP; then-part of IF statement follows
48                      PROC_CALL_OP (for treeprint)
1                          INT_MODE
40                         OBJECT_OP; this is the base address
7                             STOWED_MODE; arbitrary, for procs.
1                             Object id of procedure is 1
47                         PROC_CALL_ARG_OP
7                             STOWED_MODE
15                            DEREF_OP
7                                STOWED_MODE
58                               SELECT_OP
```

```
4                                  LONG_UNS_MODE
1                                  Field offset is 1 word
15                                 DEREF_OP
7                                     STOWED_MODE
40                                    OBJECT_OP
4                                        LONG_UNS_MODE
2                                        Object id is 2
39                            NULL_OP; end of argument list
59                         SEQ_OP; next element of IF body follows
48                            PROC_CALL_OP (for printf)
1                                INT_MODE
40                               OBJECT_OP; the base address
7                                   STOWED_MODE; ignored in this case
3                                   Object id of procedure is 3
47                               PROC_CALL_ARG_OP
1                                   INT_MODE
15                                  DEREF_OP
1                                      INT_MODE
51                                     REFTO_OP
4                                         LONG_UNS_MODE (pointer to char)
9                                         CONST_OP; this is the string
7                                            STOWED_MODE
5                                            Length is 5 words
165                                          Value is %
180                                                    4
228                                                    d
138                                                    newline
0                                                      0
47                                  PROC_CALL_ARG_OP
1                                      INT_MODE
58                                     SELECT_OP
1                                         INT_MODE
0                                         Field is at offset 0
15                                        DEREF_OP; base address of struct
7                                            STOWED_MODE
40                                           OBJECT_OP
4                                               LONG_UNS_MODE
2                                        Object id is 2
39                                    NULL_OP; end of argument list
59                         SEQ_OP; next element of body of IF follows
48                            PROC_CALL_OP
1                                INT_MODE (ignored)
40                               OBJECT_OP; the procedure address
7                                   STOWED_MODE
1                                   Object id is 1
47                               PROC_CALL_ARG_OP
7                                   STOWED_MODE
15                                  DEREF_OP
7                                      STOWED_MODE
58                                     SELECT_OP
4                                         LONG_UNS_MODE
3                                         Field has offset 3 words
15                                        DEREF_OP
7                                            STOWED_MODE
40                                           OBJECT_OP
4                                               LONG_UNS_MODE
```

```
2                                        Object id is 2
39                              NULL_OP; end of treeprint args
39                      NULL_OP; end of then-part of IF
39                      NULL_OP; omitted else-part of the IF
39                NULL_OP; end of statements in this procedure
39          NULL_OP; end of procedure definition list (and this module)
39       NULL_OP; end of modules in this input stream
```

PMA Code

```
 SEG
 RLIT
 SYML
 ENT TREEPRINT,L1_   Make 'treeprint' available outside this module
 LINK
 EXT PRINTF
L3_ EQU *
 IP PRINTF           Use 'printf', defined outside this module
 PROC
 PROC
L65535_ EQU *        Beginning of 'treeprint'
 ARGT                Transfer arguments from caller
 EAL L1_             Set up stack frame owner pointer for debugging
 STL SB%+18
 LDA ='4000
 STA% SB%
 LDL SB%+'24         If the argument...
 BLEQ L65534_        ...is nonzero...
 LDX ='1                we first get the pointer in the 'left' field
 EAXB SB%+'24,*X        by addressing the field with XB
 LDL XB%+'0             then loading the value of the pointer into L
 STL SB%+'27            then storing it in a temporary
 PCL L1_                call 'treeprint' recursively
 AP SB%+'27,*SL         using the temporary to pass the value
 LINK
 DATA '245              This is the format string for 'printf'...
 DATA '264              ...value is "%4d\n\0"
 DATA '344
 DATA '212
 DATA '0
 PROC
 PCL LB%+'400,*         Here's the call to 'printf'
 AP LB%+'402,S             passing the formatting string
 AP SB%+'24,*SL            and the value field of the current tnode
 LDX ='3                Now we get the pointer in the 'right' field
 EAXB SB%+'24,*X        pretty much as we did it before
 LDL XB%+'0
 STL SB%+'27
 PCL L1_                and call 'treeprint',
 AP SB%+'27,*SL         passing it the pointer to the right subtree
L65534_ EQU *
 PRTN                   return from 'treeprint'
L1_ ECB L65535_,,SB%+'24,1,'31
 DATA '11
 DATA '172362
```

```
DATA '162745
DATA '170362
DATA '164756
DATA '172041
END
```

The 'Drift' Language

Description

'Drift' is an extremely simplified programming language for computers with Von Neumann-style architectures. While too restrictive to be generally useful, it does have a few interesting features. It is an expression-oriented language rather than statement-oriented; non-declarative constructs generally yield a value of some sort. The syntax is intended to be conducive to simple error recovery schemes (particularly to panic-mode symbol skipping) while retaining a reasonable degree of cleanliness and human engineering (for example, statements are terminated by end-of-line, rather than some delimiter like a semicolon; continuations across lines are represented explicitly by an '&'). The semantics of the language closely reflect the expression-oriented semantics of the VCG itself.

'Drift' programs are composed of variable declarations, function declarations, and expressions. Variables may be global in scope or restricted to the function in which they are declared. Function declarations may not be nested. All variables represent floating point quantities; all functions return floating point quantities. The return value of a function is the return value of the last expression in the expression series that comprises its body. Functions may be recursive and need not be defined before use. The function named 'main' is assumed to be the main program, and will be invoked by whatever environment supports 'drift' programs.

Expressions are made of the four standard operators (+, -, *, /), assignment ('=', treated uniformly as an arithmetic operator yielding the value of its right-hand-side), two-way selection ('if'), and a loop ('while'). Variables in expressions yield their values (or take on new ones if used as the left operand of an assignment operator). They must be declared before they are used. Function calls in expressions cause parameters to be passed by value to the named function; the value returned by the function then takes the place of the call in the expression. The quad ('#') is a pseudovariable used for input/output. When used in the right-hand-side of an assignment, it causes input of a floating point value from standard input; when used in the left-hand-side, it causes output of the right-hand-side to standard output.

BNF

The syntax of 'drift' presented below employs the extended BNF used throughout the Software Tools Subsystem documentation.

Alternatives enclosed in curly braces {} may be repeated any number of times, including zero. Alternatives enclosed in square brackets [] may be used once or not at all.

```
program ->
   newlines {declaration newlines} eof

declaration ->
      global_variable_declaration
   |  function_declaration

global_variable_declaration ->
   'float' identifier {',' newlines identifier}

identifier ->
   letter {letter | digit | '_'}

newlines ->
   {NEWLINE}

function_declaration ->
   'function' identifier '(' formal_parameters ')' newlines
      {local_variable_declaration newlines}
      series newlines
   'end_function'

formal_parameters ->
   [identifier {',' newlines identifier}]

local_variable_declaration ->
   'float' identifier {',' newlines identifier}

series ->
   expression newlines {expression newlines}

expression ->
   sum {'=' sum}

sum ->
   term {('+' | '-') term}

term ->
   primary {('*' | '/') primary}

primary ->
      '#'
   |  'null'
   |  number
   |  identifier
   |  identifier '(' actual_parameters ')'
   |  loop
   |  conditional
   |  '(' series ')'

loop ->
```

```
   'while' newlines series newlines
      'do' newlines
      series newlines
      'od'

conditional ->
   'if' newlines series newlines
   'then' newlines series newlines
   ['else' newlines series newlines]
   'fi'

actual_parameters ->
   [series {',' newlines series}]
```

Examples

The following programs compute the value of a base raised to a positive integer exponent. The first is iterative, while the second is recursive.

```
-- A sample program in 'drift'

   float x, y

   function power (base, exponent)

      float result

      result = 1
      while exponent       -- that is, while exponent <> 0
         do
         result = result * base
         exponent = exponent - 1
         od

      result
   end_function

   function main ()
      x = #
      y = #
      # = power (x, y)
   end_function




-- The same sample, only done recursively

   float x, y

   function power (base, exponent)
      if exponent
```

```
      then base * power (base, exponent - 1)
      else 1
      fi
   end_function

   function main ()
      x = #
      y = #
      # = power (x, y)
   end_function
```

The Compiler

The 'drift' compiler was implemented in Ratfor under the Software Tools Subsystem in about two man-days. Conceptually, it generates intermediate form code for the VCG in two passes: the first (lexical and syntactic) generates an internal form used only by the front end, while the second (semantic) does semantic checking and converts the internal form to IMF.

The lexical analyzer used in the compiler is a fairly standard one employed in a number of Software Tools Subsystem programs because of its compactness and high speed. It resides almost entirely in the subroutine 'getsym'.

The parser code is input to 'stacc', a recursive-descent parser generator that is part of the Software Tools package. The production for 'program' is actually the main routine of the compiler. Note that very little attempt is made to recover from syntactic errors; the purpose of the compiler is the demonstration of code generation, not parsing. The parser drives the compilation process, making calls on the lexical analyzer and internal form code generation routines as necessary.

The IMF generation process is concentrated in the subroutine 'semantic_analysis' and its descendents. This routine invokes 'void_context', 'lvalue_context', and 'rvalue_context' to propagate contextual information during a traversal of the internal form tree. The bulk of the IMF generation takes place in 'rvalue_context', since most operators yield floating point values. Special cases are handled in the other two contexts: left-hand-sides of assignments by 'lvalue_context' and constructs that don't yield values by 'void_context'. Since the internal form is tree structured, the translation to IMF is straightforward.

Global Variable Definitions

```
# global variables for 'drift' compiler

# dynamic storage used by symbol table routines:
   DS_DECL (Mem, MEMSIZE)
```

```
# symbol tables:
   pointer Functions, Globals, Locals, Reserved_words
   common /stcom/ Functions, Globals, Locals, Reserved_words

# lexical stuff:
   character Inbuf (INBUFSIZE), Symtext (MAX_SYM_LEN)
   integer Symbol, Ibp, Current_line
   real Symval
   common /lexcom/ Inbuf, Symtext, Symbol, Ibp, Current_line, Symval

# files for I/O:
   filedes In_stream, Ent_stream, Data_stream, Code_stream
   common /filcom/ In_stream, Ent_stream, Data_stream, Code_stream

# internal form memory:
   integer Ifmem (INTERNAL_FORM_MEMSIZE), Next_ifmem
   common /if1com/ Ifmem
   common /if2com/ Next_ifmem

# semantic stack:
   ifpointer Stack (SEMANTIC_STACK_SIZE)
   integer Sp
   common /semcom/ Sp, Stack

# other junk:
   integer Next_obj_id, Ex$in_id, Ex$out_id, Error_count
   common /miscom/ Next_obj_id, Ex$in_id, Ex$out_id, Error_count
```

Parser Source Code

```
# 'stacc' parser for drift

.common "drift_com.r.i";      # file containing global variables

.symbol Symbol;               # "current symbol" variable

.scanner getsym;              # name of lexical analysis routine

.state state;                 # "parse state" variable

.terminal                     # terminal symbols
   256                           # start higher than largest character value
   FLOAT_SYM
   ID_SYM
   FUNCTION_SYM
   END_FUNCTION_SYM
   NULL_SYM
   NUMBER_SYM
   WHILE_SYM
   DO_SYM
   OD_SYM
   IF_SYM
   THEN_SYM
   ELSE_SYM
```

```
   FI_SYM
   ;


   NEWLINE
   EOF
   ;



program ->
                  !  call begin_program
   nls
   {
      declaration
      nls
      }
   EOF.
                  !  call end_program
                  ?  call pmr ("EOF expected*n"p, state)
   ;



declaration ->
      global_variable_declaration
   |
      function_declaration
   ;



global_variable_declaration ->
   FLOAT_SYM
   ID_SYM
                  !  call declare_global_variable (Symtext)
                  ?   call  pmr  ("missing identifier*n"p, state,
state)
   {
      ',,'
      nls
      ID_SYM
                  !  call declare_global_variable (Symtext)
                  ?  call pmr ("missing identifier*n"p, state)
      }
   ;



nls ->
   {
      NEWLINE
      }
   ;
```

```
function_declaration ->
   FUNCTION_SYM
   ID_SYM
                  !  call begin_function (Symtext)
                  ?  call pmr ("missing function name*n"p, state)
   '('
                  !  call make_null
                  ?  call pmr ("missing parameters*n"p, state)
   formal_parameters
                  !  call make_function_parameters
   ')'
                  ?  call pmr ("missing ')'*n"p, state)
   nls
                  !  call make_null
   {
      local_variable_declaration
      nls
      }
   series
                  !  call make_function_body
                  ?  call pmr ("missing function body*n"p, state)
   nls
   END_FUNCTION_SYM
                  !  call end_function
                  ?   call  pmr   ("missing   'end_function'*n"p,
state)
   ;



formal_parameters ->
   ID_SYM
                  !  call declare_formal_parameter (Symtext)
   {
      ','
      nls
      ID_SYM
                  !  call declare_formal_parameter (Symtext)
                  ?  call pmr ("missing identifier*n"p, state)
      }
   |
      epsilon
   ;



local_variable_declaration ->
   FLOAT_SYM
   ID_SYM
                  !  call declare_local_variable (Symtext)
                  ?  call pmr ("missing identifier*n"p, state)
   {
      ','
      nls
      ID_SYM
```

```
                  !  call declare_local_variable (Symtext)
                  ?  call pmr ("missing identifier*n"p, state)
      }
   ;



series ->
   expression
   nls
   {
      expression
                  !  call sequentialize
      nls
      }
   ;



expression ->
   sum
   {
      '='
      sum
                  !  call make_dyad (ASSIGN_NODE)
                  ?    call  pmr  ("missing  right-hand-side*n"p,
state)
      }
   ;



sum ->
                  !  integer node
   term
   {
      (
            '+'
                  !  node = ADD_NODE
         |
            '-'
                  !  node = SUBTRACT_NODE
         )
      term
                  !  call make_dyad (node)
                  ?  call pmr ("missing right operand*n"p, state)
      }
   ;



term ->
                  !  integer node
   primary
   {
      (
```

```
            '*'
                  !  node = MULTIPLY_NODE
         |
            '/'
                  !  node = DIVIDE_NODE
         )
      primary
                  !  call make_dyad (node)
                  ?  call pmr ("missing right operand*n"p, state)
      }
   ;


primary ->
                  !  character id (MAX_SYM_LEN)
      '#'
                  !  call make_quad
   |
      NULL_SYM
                  !  call make_null
   |
      NUMBER_SYM
                  !  call make_constant (Symval)
   |
      ID_SYM
                  !  call scopy (Symtext, 1, id, 1)
      (
            '('
            actual_parameters
            ')'
                  !  call make_call (id)
                  ?  call pmr ("missing ')'*n"p, state)
         |
            epsilon
                  !  call make_object (id)
         )
   |
      loop
   |
      conditional
   |
      '('
      series
      ')'
                  ?  call pmr ("missing ')'*n"p, state)
   ;


loop ->
   WHILE_SYM
   nls
   series
                  ?   call  pmr  ("missing  loop   condition*n"p,
state)
```

```
   nls
   DO_SYM
                  ?  call pmr ("missing 'do'*n"p, state)
   nls
   series
                  ?  call pmr ("missing loop body*n"p, state)
   nls
   OD_SYM
                  !  call make_loop
                  ?  call pmr ("missing 'od'*n"p, state)
   ;



conditional ->
   IF_SYM
   nls
   series
                  ?    call  pmr  ("missing  'if'  condition*n"p,
state)
   nls
   THEN_SYM
                  ?  call pmr ("missing 'then'*n"p, state)
   nls
   series
                  ?  call pmr ("missing then_part*n"p, state)
   (
         ELSE_SYM
         nls
         series
                  ?  call pmr ("missing else_part*n"p, state)
         nls
      |
         nls
                  !  call make_null
      )
   FI_SYM
                  !  call make_conditional
                  ?  call pmr ("missing 'fi'*n"p, state)
   ;



actual_parameters ->
                  !  call make_null
      series
                  !  call make_actual_parameter
      {
         ','
         nls
         series
                  !  call make_actual_parameter
                  ?  call pmr ("missing parameter after  ','*n"p,
state)
         }
   |
```

```
      epsilon
   ;
```

Remainder of Compiler Source Code

```
# drift --- sample compiler for VCG demonstration

define (GLOBAL_VARIABLES,"drift_com.r.i")

define (MAX_SYM_LEN, MAXLINE)
define (MEMSIZE, 4096)
define (SEMANTIC_STACK_SIZE, 100)
define (INTERNAL_FORM_MEMSIZE, 20000)
define (INBUFSIZE, 300)
define (PBLIMIT, 150)

define (UNDEFINED, 0)
define (DEFINED, 1)

define (ifpointer, integer)
define (unknown, integer)

# Types of internal form nodes:
define (ADD_NODE,1)
define (ARG_NODE,2)
define (ASSIGN_NODE,3)
define (CALL_NODE,4)
define (COND_NODE,5)
define (CONSTANT_NODE,6)
define (DECLARE_VAR_NODE,7)
define (DIVIDE_NODE,8)
define (FUNCTION_NODE,9)
define (IO_NODE,10)
define (LOOP_NODE,11)
define (MULTIPLY_NODE,12)
define (NULL_NODE,13)
define (PARAM_NODE,14)
define (SEQ_NODE,15)
define (SUBTRACT_NODE,16)
define (VAR_NODE,17)
define (LAST_NODE_TYPE,VAR_NODE)

# Elements of internal form records:
define (ARG_EXPR (n), Ifmem (n + 2))
define (ARG_LIST (n), Ifmem (n + 3))
define (COND (n), Ifmem (n + 2))
define (ELSE_PART (n), Ifmem (n + 4))
define (FUNC_BODY (n), Ifmem (n + 5))
define (LEFT (n), Ifmem (n + 2))
define (LINE_NUM (n), Ifmem (n + 1))
define (LOOP_BODY (n), Ifmem (n + 3))
define (NODE_TYPE (n), Ifmem (n))
define (NPARAMS (n), Ifmem (n + 4))
define (OBJ_ID (n), Ifmem (n + 2))
define (PARAM_LIST (n), Ifmem (n + 3))
```

```
define (RIGHT (n), Ifmem (n + 3))
define (THEN_PART (n), Ifmem (n + 3))
define (WORD1 (n), Ifmem (n + 2))
define (WORD2 (n), Ifmem (n + 3))

include "drift.stacc.defs"             # macro defns. produced by 'stacc'
include "/uc/allen/vcg/vcg_defs.r.i"   # macro defns. for IMF operators



   integer state

   call program (state)
   if (state ~= ACCEPT)
      call error ("syntactically incorrect program"p)
   stop
   end



include "drift.stacc.r"    # Ratfor source code produced by 'stacc'



# begin_function --- set up environment for compiling a function

   subroutine begin_function (name)
   character name (ARB)

   include GLOBAL_VARIABLES

   pointer mktabl

   integer info2 (2)
   integer lookup, gen_id

   ifpointer func_node
   ifpointer ialloc

   Next_ifmem = 1          # initialize internal form memory
   Locals = mktabl (1)     # initialize local variable symbol table
   Sp = 0                  # initialize semantic stack pointer

  # Place function name in 'Functions' table, if it's not already there
   if (lookup (name, info2, Functions) == YES)
      if (info2 (2) == DEFINED)
         call warning ("function *s multiply defined*n"p, name)
      else {
         info2 (2) = DEFINED
         call enter (name, info2, Functions)
         }
   else {
      info2 (1) = gen_id (1)
      info2 (2) = DEFINED
      call enter (name, info2, Functions)
      }
```

```
  # Output an entry point definition for the procedure:
   call emit (SEQ_OP, Ent_stream)
   call emit (info2 (1), Ent_stream)      # object id of function
   call emit_string (name, Ent_stream)    # function name

  # Put function node on semantic stack:
   func_node = ifalloc (FUNCTION_NODE)
   NPARAMS (func_node) = 0
   OBJ_ID (func_node) = info2 (1)
   call push (func_node)

   return
   end



# begin_program --- do pre-program initialization

   subroutine begin_program

   include GLOBAL_VARIABLES

   pointer mktabl

   filedes create, open

   character infile (MAXARG)

   integer getarg, gen_id

   call dsinit (MEMSIZE)      # init. dynamic storage
   Functions = mktabl (2)     # symbol table for function names
   Globals = mktabl (1)       # symbol table for global variables
   Reserved_words = mktabl (1)   # symbol table for reserved words
   Next_obj_id = 1            # for object id generator
   Error_count = 0
   Ibp = 1                    # buffer pointer...
   Inbuf (Ibp) = EOS          # ...and input buffer used by lexer
   Current_line = 0

  # open input file specified on command line:
   if (getarg (1, infile, MAXARG) == EOF)
      In_stream = STDIN
   else {
      In_stream = open (infile, READ)
      if (In_stream == ERR)
         call cant (infile)
      }

  # create temporary files for passing the IMF to the code generator:
   Ent_stream = create ("_drift_.ct1"s, READWRITE)
   Data_stream = create ("_drift_.ct2"s, READWRITE)
   Code_stream = create ("_drift_.ct3"s, READWRITE)
   if (Ent_stream == ERR || Data_stream == ERR || Code_stream == ERR)
      call error ("can't open temporary files _drift_.ct[1-3]"p)
```

```
   call emit (MODULE_OP, Ent_stream)
   call emit (MODULE_OP, Data_stream)
   call emit (MODULE_OP, Code_stream)

  # define object id's for the two run-time routines we'll need:
   Ex$in_id = gen_id (1)            # run-time routine for input
   call emit (SEQ_OP, Data_stream)
   call emit (DECLARE_STAT_OP, Data_stream)
   call emit (Ex$in_id, Data_stream)
   call emit_string ("EX$IN"s, Data_stream)

   Ex$out_id = gen_id (1)           # run-time routine for output
   call emit (SEQ_OP, Data_stream)
   call emit (DECLARE_STAT_OP, Data_stream)
   call emit (Ex$out_id, Data_stream)
   call emit_string ("EX$OUT"s, Data_stream)

  # build the reserved-word table used by the lexical analyzer:
   call enter ("do"s, DO_SYM, Reserved_words)
   call enter ("else"s, ELSE_SYM, Reserved_words)
   call enter ("end_function"s, END_FUNCTION_SYM, Reserved_words)
   call enter ("fi"s, FI_SYM, Reserved_words)
   call enter ("float"s, FLOAT_SYM, Reserved_words)
   call enter ("function"s, FUNCTION_SYM, Reserved_words)
   call enter ("if"s, IF_SYM, Reserved_words)
   call enter ("null"s, NULL_SYM, Reserved_words)
   call enter ("od"s, OD_SYM, Reserved_words)
   call enter ("then"s, THEN_SYM, Reserved_words)
   call enter ("while"s, WHILE_SYM, Reserved_words)

  # fire up lexical analysis:
   call getsym

   return
   end



# declare_formal_parameter --- put formal param name in table, assign obj id

   subroutine declare_formal_parameter (name)
   character name (ARB)

   include GLOBAL_VARIABLES

   integer obj_id
   integer lookup, gen_id

   ifpointer param_node
   ifpointer ifalloc

   if (lookup (name, obj_id, Locals) == YES) {
      call warning ("*s:  multiply declared*n"p, name)
      return
      }
```

```
   obj_id = gen_id (1)
   call enter (name, obj_id, Locals)

  # create new parameter node and combine it with previous sequence
  #   on the semantic stack:
   param_node = ifalloc (PARAM_NODE)
   OBJ_ID (param_node) = obj_id
   call push (param_node)
   call sequentialize
   NPARAMS (Stack (Sp - 1)) += 1

   return
   end



# declare_global_variable --- put name in global table, assign object id

   subroutine declare_global_variable (name)
   character name (ARB)

   include GLOBAL_VARIABLES

   integer obj_id
   integer lookup, gen_id

   if (lookup (name, obj_id, Globals) == YES) {
      call warning ("*s:  multiply declared*n"p, name)
      return
      }

   obj_id = gen_id (1)
   call enter (name, obj_id, Globals)

  # go ahead and reserve space in the static data storage area for
  #   the variable we just declared:
   call emit (SEQ_OP, Data_stream)
   call emit (DEFINE_STAT_OP, Data_stream)
   call emit (obj_id, Data_stream)
   call emit (NULL_OP, Data_stream)    # no initializers
   call emit (2, Data_stream)          # 2 words for a floating object

   return
   end



# declare_local_variable --- enter name in local table, assign object id

   subroutine declare_local_variable (name)
   character name (ARB)

   include GLOBAL_VARIABLES

   integer obj_id
```

```
   integer lookup, gen_id

   ifpointer decl_var_node
   ifpointer ifalloc

   if (lookup (name, obj_id, Locals) == YES) {
      call warning ("*s:  multiply declared*n"p, name)
      return
      }

   obj_id = gen_id (1)
   call enter (name, obj_id, Locals)

  # make new variable declaration node and put it into a sequence
  #   following all previously declared variables:
   decl_var_node = ifalloc (DECLARE_VAR_NODE)
   OBJ_ID (decl_var_node) = obj_id
   call push (decl_var_node)
   call sequentialize

   return
   end



# emit --- place value on an output stream

   subroutine emit (val, stream)
   integer val
   filedes stream

   call print (stream, "*i*n"s, val)

   return
   end



# emit_string --- place length of string and string on an output stream

   subroutine emit_string (str, stream)
   character str (ARB)
   filedes stream

   integer i
   integer length

   call emit (length (str), stream)
   for (i = 1; str (i) ~= EOS; i += 1)
      call emit (str (i), stream)

   return
   end
```

```
# end_function --- clean up after parse of a function is completed

   subroutine end_function

   include GLOBAL_VARIABLES

   call semantic_analysis (Stack (Sp))
   call rmtabl (Locals)    # get rid of all local variable information

   return
   end



# end_program --- clean up after the entire program is parsed

   subroutine end_program

   include GLOBAL_VARIABLES

   pointer position

   integer info2 (2)
   integer sctabl

   character sym (MAX_SYM_LEN)

   logical first

   call close (In_stream)

  # terminate IMF streams by ending sequence of definitions, then
  #   ending sequence of modules:
   call emit (NULL_OP, Ent_stream);  call emit (NULL_OP, Ent_stream)
   call emit (NULL_OP, Data_stream); call emit (NULL_OP, Data_stream)
   call emit (NULL_OP, Code_stream); call emit (NULL_OP, Code_stream)

   call close (Ent_stream)
   call close (Data_stream)
   call close (Code_stream)

  # check function table for names that were referenced but not
  #   declared; presumably these are externally compiled
   first = TRUE
   position = 0
   while (sctabl (Functions, sym, info2, position) ~= EOF)
      if (info2 (2) == UNDEFINED) {
         if (first) {
            call print (STDOUT, "External symbols:*n"p)
            first = FALSE
            }
         call print (STDOUT, "*s*n"p, sym)
         }

   return
   end
```

```
# gen_id --- generate new object identifiers

   integer function gen_id (num_ids)
   integer num_ids

   include GLOBAL_VARIABLES

   gen_id = Next_obj_id
   Next_obj_id += num_ids

   return
   end



# getsym --- get next symbol from input stream

   subroutine getsym

   include GLOBAL_VARIABLES

   procedure getchar forward
   procedure putback (c) forward
   procedure empty_buffer forward

   character c

   integer i
   integer getlin, lookup

   real ctor

   logical too_long, continuation

   continuation = FALSE    # true if we want to ignore a line boundary
   repeat { # until we find a legal symbol

      repeat
         getchar
         until (c ~= ' 'c)

      select (c)

         when (NEWLINE) {
            Current_line += 1
            Symbol = NEWLINE
            if (~continuation)
               break
            }

         when (';'c) {
            Symbol = NEWLINE     # but no line number advance
            if (~continuation)
```

```
            break
         }

      when ('-'c) {
         getchar
         if (c == '-'c) {     # -- begins comments
            empty_buffer
            Current_line += 1
            Symbol = NEWLINE
            if (~continuation)
               break
            }
         else {
            putback (c)
            Symbol = '-'c
            break
            }
         }

      when ('&'c)
         continuation = TRUE

      when ('+'c, '*'c, '/'c, '#'c, '('c, ')'c, ','c, '='c, EOF) {
         Symbol = c
         break
         }

      when (SET_OF_LETTERS) {    # a-z or A-Z; starting an identifier
         too_long = FALSE
         i = 1
         while (IS_LETTER (c) || IS_DIGIT (c) || c == '_'c) {
            Symtext (i) = c
            i += 1
            if (i > MAX_SYM_LEN) {
               i -= 1
               too_long = TRUE
               }
            getchar
            }
         putback (c)
         Symtext (i) = EOS
         if (too_long)
            call warning ("symbol beginning *s is too long*n"p, Symtext)
         if (lookup (Symtext, Symbol, Reserved_words) == NO)
            Symbol = ID_SYM
         break
         }

      when ('.'c, SET_OF_DIGITS) {
         putback (c)
         Symval = ctor (Inbuf, Ibp)    # advances Ibp
         Symbol = NUMBER_SYM
         break
         }

   else
```

```
      call warning ("'*c':  unrecognized character*n"p, c)

   }  # repeat until a valid symbol is found

return


# getchar --- get the next character from the input stream

   procedure getchar {

      if (Inbuf (Ibp) == EOS)       # time to read a new buffer?
         if (getlin (Inbuf (PBLIMIT), In_stream) == EOF)
            c = EOF
         else {
            c = Inbuf (PBLIMIT)     # pick up the first char read
            Ibp = PBLIMIT + 1
            }
      else {                        # text was already available
         c = Inbuf (Ibp)
         Ibp += 1
         }

      }


# putback --- push a character back onto the input stream

   procedure putback (c) {
      character c

      if (Ibp <= 1)
         call error ("too many characters pushed back"p)
      else {
         Ibp -= 1
         Inbuf (Ibp) = c
         }

      }


# empty_buffer --- kill remainder of line in input buffer

   procedure empty_buffer {

      Inbuf (Ibp) = EOS          # will force a read in 'getchar'

      }

end


# ifalloc --- allocate space for a particular type node in internal form memory

   ifpointer function ifalloc (node_type)
```

```
   integer node_type

   include GLOBAL_VARIABLES

  # These declarations assume that the internal form node types form
  #   a dense ascending sequence of integers from 1 to LAST_NODE_TYPE:
   integer sizeof (LAST_NODE_TYPE)
   data sizeof / _
      4,    # ADD_NODE
      3,    # ARG_NODE
      4,    # ASSIGN_NODE
      4,    # CALL_NODE
      5,    # COND_NODE
      4,    # CONSTANT_NODE
      3,    # DECLARE_VAR_NODE
      4,    # DIVIDE_NODE
      6,    # FUNCTION_NODE
      2,    # IO_NODE
      4,    # LOOP_NODE
      4,    # MULTIPLY_NODE
      2,    # NULL_NODE
      3,    # PARAM_NODE
      4,    # SEQ_NODE
      4,    # SUBTRACT_NODE
      3 _   # VAR_NODE
      /

   if (node_type < 1 || node_type > LAST_NODE_TYPE)
      call error ("ifalloc received bad node type"p)

   if (Next_ifmem + sizeof (node_type) > INTERNAL_FORM_MEMSIZE)
      call error ("insufficient internal form memory"p)

   ifalloc = Next_ifmem
   Next_ifmem += sizeof (node_type)

   NODE_TYPE (ifalloc) = node_type
   LINE_NUM (ifalloc) = Current_line

   return
   end



# lvalue_context --- generate VCG code for constructs used as lvalues
#     (assumes I/O quads have already been eliminated from LHS's)

   subroutine lvalue_context (node)
   ifpointer node

   include GLOBAL_VARIABLES

   select (NODE_TYPE (node))

      when (VAR_NODE) {
         call emit (OBJECT_OP, Code_stream)
```

```
         call emit (FLOAT_MODE, Code_stream)
         call emit (OBJ_ID (node), Code_stream)
         }

      when (SEQ_NODE) {
         if (NODE_TYPE (RIGHT (node)) == NULL_NODE)
            call lvalue_context (LEFT (node))
         else {
            call emit (SEQ_OP, Code_stream)
            call void_context (LEFT (node))
            call lvalue_context (RIGHT (node))
            }
         }

   else
      call warning ("assignment on line *i has an illegal left side*n"p,
         LINE_NUM (node))

   return
   end



# make_actual_parameter --- link actual parameter expression to list

   subroutine make_actual_parameter

   include GLOBAL_VARIABLES

   ifpointer act_param
   ifpointer ifalloc, pop

   act_param = ifalloc (ARG_NODE)
   ARG_EXPR (act_param) = pop (0)
   call push (act_param)
   call sequentialize

   return
   end



# make_call --- generate a call to a function

   subroutine make_call (name)
   character name (ARB)

   include GLOBAL_VARIABLES

   integer info2 (2)
   integer lookup, gen_id

   ifpointer call_node
   ifpointer ifalloc, pop

  # if function name is in Functions table, all is well; if not,
```

```
  #   we add it provisionally (it may be defined later).
   if (lookup (name, info2, Functions) == NO) {
      info2 (1) = gen_id (1)
      info2 (2) = UNDEFINED
      call enter (name, info2, Functions)
      }

   call_node = ifalloc (CALL_NODE)
   OBJ_ID (call_node) = info2 (1)
   ARG_LIST (call_node) = pop (0)
   call push (call_node)

   return
   end



# make_conditional --- make conditional (if-then-else-fi) node

   subroutine make_conditional

   include GLOBAL_VARIABLES

   ifpointer cond
   ifpointer if_alloc, pop

   cond = if_alloc (COND_NODE)
   ELSE_PART (cond) = pop (0)
   THEN_PART (cond) = pop (0)
   COND (cond) = pop (0)

   call push (cond)
   return
   end



# make_constant --- make constant node from given value

   subroutine make_constant (val)
   real val

   include GLOBAL_VARIABLES

   real rkluge
   integer ikluge (2)
   equivalence (rkluge, ikluge)  # used to unpack floating point constants

   ifpointer cnode
   ifpointer ifalloc

   cnode = ifalloc (CONSTANT_NODE)
   rkluge = val
   WORD1 (cnode) = ikluge (1)
   WORD2 (cnode) = ikluge (2)
```

```
   call push (cnode)
   return
   end


# make_dyad --- make node for a dyadic operator (=, +, -, *, /)

   subroutine make_dyad (node_type)
   integer node_type

   include GLOBAL_VARIABLES

   ifpointer node
   ifpointer ifalloc, pop

   node = ifalloc (node_type)
   RIGHT (node) = pop (0)
   LEFT (node) = pop (0)
   call push (node)

   return
   end


# make_function_body --- add function body to function definition node

   subroutine make_function_body

   include GLOBAL_VARIABLES

   ifpointer body
   ifpointer pop

   call sequentialize      # combine declarations and code
   body = pop (0)          # note deep-stack addressing makes sequencing
   FUNC_BODY (Stack (Sp)) = body       # necessary...

   return
   end


# make_function_parameters --- add params to function definition node

   subroutine make_function_parameters

   include GLOBAL_VARIABLES

   ifpointer params
   ifpointer pop

   params = pop (0)     # note:  deep-stack addressing makes use of
   PARAM_LIST (Stack (Sp)) = params    # a particular sequence necessary
```

```
   return
   end


# make_loop --- pop cond and body off stack, generate a loop node

   subroutine make_loop

   include GLOBAL_VARIABLES

   ifpointer loop
   ifpointer ifalloc, pop

   loop = ifalloc (LOOP_NODE)
   LOOP_BODY (loop) = pop (0)
   COND (loop) = pop (0)
   call push (loop)

   return
   end


# make_null --- push new "null operator" node on stack

   subroutine make_null

   include GLOBAL_VARIABLES

   ifpointer ifalloc

   call push (ifalloc (NULL_NODE))

   return
   end


# make_object --- push node referencing a variable on the stack

   subroutine make_object (name)
   character name (ARB)

   include GLOBAL_VARIABLES

   ifpointer node
   ifpointer ifalloc

   integer obj_id
   integer lookup

   node = ifalloc (VAR_NODE)

   if (lookup (name, obj_id, Locals) == NO
     && lookup (name, obj_id, Globals) == NO) {
```

```
      call warning ("*s:  undeclared identifier*n"p, name)
      obj_id = 0
      }

   OBJ_ID (node) = obj_id
   call push (node)

   return
   end


# make_quad --- generate an input/output operation node

   subroutine make_quad

   include GLOBAL_VARIABLES

   ifpointer ifalloc

   call push (ifalloc (IO_NODE))

   return
   end


# output_arguments --- output IMF for procedure call arguments

   subroutine output_arguments (arg_node)
   ifpointer arg_node

   include GLOBAL_VARIABLES

   select (NODE_TYPE (arg_node))

      when (ARG_NODE) {
         call emit (PROC_CALL_ARG_OP, Code_stream)
         call emit (FLOAT_MODE, Code_stream)
         call rvalue_context (ARG_EXPR (arg_node))
         }

      when (NULL_NODE)
         ;

      when (SEQ_NODE) {
         call output_arguments (LEFT (arg_node))
         call output_arguments (RIGHT (arg_node))
         }

   else
      call error ("in output_argument:  shouldn't happen"p)

   return
   end
```

```
# output_params --- output IMF for procedure formal parameter definitions

   subroutine output_params (param_node)
   ifpointer param_node

   include GLOBAL_VARIABLES

   select (NODE_TYPE (param_node))

      when (PARAM_NODE) {
         call emit (PROC_DEFN_ARG_OP, Code_stream)
         call emit (OBJ_ID (param_node), Code_stream)
         call emit (FLOAT_MODE, Code_stream)
         call emit (VALUE_DISP, Code_stream)
         call emit (2, Code_stream) # FLOATs are 2 words long
         }

      when (NULL_NODE)
         ;

      when (SEQ_NODE) {
         call output_params (LEFT (param_node))
         call output_params (RIGHT (param_node))
         }

   else
      call error ("in output_param:  shouldn't happen"p)

   return
   end



# pmr --- panic mode recovery for parser

   subroutine pmr (message, state)
   character message (ARB)
   integer state

   include GLOBAL_VARIABLES

   call warning (message)
   state = ACCEPT

   while (Symbol ~= EOF && Symbol ~= ')'c && Symbol ~= NEWLINE
     && Symbol ~= END_FUNCTION_SYM && Symbol ~= THEN_SYM
     && Symbol ~= ELSE_SYM && Symbol ~= FI_SYM && Symbol ~= DO_SYM
     && Symbol ~= OD_SYM && Symbol ~= ','c)
      call getsym

   return
   end
```

```
# pop --- pop a node pointer off the semantic stack

   ifpointer function pop (dummy)
   integer dummy     # needed to satisfy FORTRAN syntax requirements

   include GLOBAL_VARIABLES

   if (Sp < 1)
      call error ("semantic stack underflow"p)

   pop = Stack (Sp)
   Sp -= 1

   return
   end



# push --- push a node pointer onto the semantic stack

   subroutine push (node)
   ifpointer node

   include GLOBAL_VARIABLES

   if (Sp >= SEMANTIC_STACK_SIZE)
      call error ("semantic stack overflow"p)

   Sp += 1
   Stack (Sp) = node

   return
   end



# rvalue_context --- generate VCG code for constructs used as rvalues

   subroutine rvalue_context (node)
   ifpointer node

   include GLOBAL_VARIABLES

   select (NODE_TYPE (node))

      when (ADD_NODE, SUBTRACT_NODE, MULTIPLY_NODE, DIVIDE_NODE) {
         select (NODE_TYPE (node))
            when (ADD_NODE)
               call emit (ADD_OP, Code_stream)
            when (SUBTRACT_NODE)
               call emit (SUB_OP, Code_stream)
            when (MULTIPLY_NODE)
               call emit (MUL_OP, Code_stream)
            when (DIVIDE_NODE)
               call emit (DIV_OP, Code_stream)
```

```
   call emit (FLOAT_MODE, Code_stream)
   call rvalue_context (LEFT (node))
   call rvalue_context (RIGHT (node))
   }

when (ASSIGN_NODE) {
   if (NODE_TYPE (LEFT (node)) == IO_NODE) {
     # fake up output by calling 'ex$out' at run time:
      call emit (PROC_CALL_OP, Code_stream)
      call emit (FLOAT_MODE, Code_stream)
      call emit (OBJECT_OP, Code_stream)
      call emit (STOWED_MODE, Code_stream)
      call emit (Ex$out_id, Code_stream)
      call emit (PROC_CALL_ARG_OP, Code_stream)
      call emit (FLOAT_MODE, Code_stream)
      call rvalue_context (RIGHT (node))
      call emit (NULL_OP, Code_stream)
      }
   else {
      call emit (ASSIGN_OP, Code_stream)
      call emit (FLOAT_MODE, Code_stream)
      call lvalue_context (LEFT (node))
      call rvalue_context (RIGHT (node))
      call emit (2, Code_stream)    # assign 2 words
      }
   }

when (CALL_NODE) {
   call emit (PROC_CALL_OP, Code_stream)
   call emit (FLOAT_MODE, Code_stream)
   call emit (OBJECT_OP, Code_stream)
   call emit (STOWED_MODE, Code_stream)
   call emit (OBJ_ID (node), Code_stream)
   call output_arguments (ARG_LIST (node))
   call emit (NULL_OP, Code_stream)
   }

when (COND_NODE) {
   call emit (IF_OP, Code_stream)
   call emit (FLOAT_MODE, Code_stream)
   call rvalue_context (COND (node))
   call rvalue_context (THEN_PART (node))
   if (NODE_TYPE (ELSE_PART (node)) == NULL_NODE)
      call warning ("'if' on line *i needs an 'else' part*n"p,
         LINE_NUM (node))
   call rvalue_context (ELSE_PART (node))
   }

when (CONSTANT_NODE) {
   call emit (CONST_OP, Code_stream)
   call emit (FLOAT_MODE, Code_stream)
   call emit (2, Code_stream)    # 2-word floats
   call emit (WORD1 (node), Code_stream)
   call emit (WORD2 (node), Code_stream)
   }
```

```
      when (DECLARE_VAR_NODE) {
         call emit (DEFINE_DYNM_OP, Code_stream)
         call emit (OBJ_ID (node), Code_stream)
         call emit (NULL_OP, Code_stream)    # no initializers
         call emit (2, Code_stream)    # size is 2 words
         }

      when (IO_NODE) {
        # fake up input by calling 'ex$in' at run time:
         call emit (PROC_CALL_OP, Code_stream)
         call emit (FLOAT_MODE, Code_stream)
         call emit (OBJECT_OP, Code_stream)
         call emit (STOWED_MODE, Code_stream)
         call emit (Ex$in_id, Code_stream)
         call emit (NULL_OP, Code_stream)    # no arguments
         }

      when (LOOP_NODE)
         call warning ("while-loop at line *i is used as an rvalue*n"p,
            LINE_NUM (node))

      when (NULL_NODE)
         call emit (NULL_OP, Code_stream)

      when (SEQ_NODE) {
         if (NODE_TYPE (RIGHT (node)) == NULL_NODE)
            call rvalue_context (LEFT (node))
         else {
            call emit (SEQ_OP, Code_stream)
            call void_context (LEFT (node))  # can never yield a value
            call rvalue_context (RIGHT (node))
            }
         }

      when (VAR_NODE) {
         call emit (OBJECT_OP, Code_stream)
         call emit (FLOAT_MODE, Code_stream)
         call emit (OBJ_ID (node), Code_stream)
         }

   else
      call error ("in rvalue_context:  shouldn't happen"p)

   return
   end



# semantic_analysis --- check function and output VCG code for it

   subroutine semantic_analysis (func)
   ifpointer func

   include GLOBAL_VARIABLES

  # output the procedure definition node:
```

```
   call emit (SEQ_OP, Code_stream)
   call emit (PROC_DEFN_OP, Code_stream)
   call emit (OBJ_ID (func), Code_stream)
   call emit (NPARAMS (func), Code_stream)
   call emit_string (EOS, Code_stream) # we'll ignore this for now

  # take care of the formal parameter declarations:
   call output_params (ARG_LIST (func))
   call emit (NULL_OP, Code_stream)

  # finally, take care of local variables and the function code:
   call rvalue_context (FUNC_BODY (func))

   return
   end



# sequentialize --- combine two nodes with a "sequence" node

   subroutine sequentialize

   include GLOBAL_VARIABLES

   ifpointer seq_node
   ifpointer ifalloc, pop

   seq_node = ifalloc (SEQ_NODE)
   RIGHT (seq_node) = pop (0)
   LEFT (seq_node) = pop (0)
   call push (seq_node)

   return
   end



# void_context --- generate VCG code for constructs that yield no value

   subroutine void_context (node)
   ifpointer node

   include GLOBAL_VARIABLES

   select (NODE_TYPE (node))

      when (COND_NODE) {      # an 'if' used as a statement
         call emit (IF_OP, Code_stream)
         call emit (FLOAT_MODE, Code_stream)
         call rvalue_context (COND (node))
         call void_context (THEN_PART (node))
         call void_context (ELSE_PART (node))
         }

      when (LOOP_NODE) {
         call emit (WHILE_LOOP_OP, Code_stream)
```

```
         call rvalue_context (COND (node))
         call void_context (LOOP_BODY (node))
         }

      when (SEQ_NODE) {
         call emit (SEQ_OP, Code_stream)
         call void_context (LEFT (node))
         call void_context (RIGHT (node))
         }

   else
      call rvalue_context (node)

   return
   end



# warning --- print warning message

   subroutine warning (format, a1, a2, a3, a4, a5, a6, a7, a8, a9)
   character format (ARB)
   unknown a1, a2, a3, a4, a5, a6, a7, a8, a9

   include GLOBAL_VARIABLES

   call print (ERROUT, "*i:  "s, Current_line)
   call print (ERROUT, format, a1, a2, a3, a4, a5, a6, a7, a8, a9)
   Error_count += 1

   return
   end
```

Run-Time Support Routines Source Code

```
* RUN-TIME SUPPORT FOR 'DRIFT'
*           UNFORTUNATELY, EX$IN MUST BE WRITTEN IN ASSEMBLER SINCE
*           FORTRAN DOESN'T ALLOW FUNCTIONS WITHOUT ARGUMENTS.
*           AN ALTERNATIVE WOULD BE TO HAVE THE COMPILER GENERATE
*           A DUMMY ARGUMENT ON THE CALL; THEN EX$IN AND EX$OUT
*           COULD BE WRITTEN IN RATFOR, PASCAL, OR WHAT HAVE YOU.


* EX$IN --- READ A LINE FROM STANDARD INPUT, CONVERT FROM CHARACTER
*           TO REAL, AND RETURN VALUE

            SEG
            RLIT
            SYML

            SUBR     EX$IN

EX$IN       ECB      EX$IN$

            DYNM     LINE(100),I
```

```
EX$IN$      EQU     *
            CALL    GETLIN          READ NEXT INPUT LINE
            AP      LINE,S
            AP      =-10,SL
            BGT     CVT_IN          IF WE HIT EOF,
            CALL    SWT                JUST QUIT
CVT_IN      EQU     *
            LT                      OTHERWISE,
            STA     I
            CALL    CTOR               CONVERT TO REAL
            AP      LINE,S
            AP      I,SL
            PRTN                       AND RETURN WITH VALUE IN F

            END


* EX$OUT --- WRITE REAL VALUE TO STANDARD OUTPUT, RETURN VALUE UNCHANGED

            SEG
            RLIT
            SYML

            SUBR    EX$OUT
EX$OUT      ECB     EX$OUT$,,VAL,1

            DYNM    VAL(3)

EX$OUT$     EQU     *
            ARGT
            CALL    PRINT           JUST USE SWT I/O TO OUTPUT VALUE
            AP      =-11,S             ON STDOUT
            AP      =C'*r*n.',S
            AP      VAL,*SL
            FLD     VAL,*           RETURN VALUE IN F SO THIS FUNCTION
            PRTN                       BEHAVES LIKE A PSEUDO-VARIABLE

            END
```

Intermediate Form Operator/Function Index

```
bit fields
   FIELD_OP

bitwise logical operators
   ANDAA_OP, AND_OP, COMPL_OP, LSHIFTAA_OP, LSHIFT_OP, ORAA_OP,
   OR_OP, RSHIFTAA_OP, RSHIFT_OP, XORAA_OP, XOR_OP

boolean operators
   NOT_OP, SAND_OP, SOR_OP

bounds checking
   CHECK_RANGE_OP, CHECK_LOWER_OP, CHECK_UPPER_OP

branch
   GOTO_OP, LABEL_OP

break (loop termination)
   BREAK_OP, NEXT_OP

byte access
   FIELD_OP

call
   procedures, functions, subroutines:  PROC_CALL_OP,
      PROC_CALL_ARG_OP

case statement
   SWITCH_OP

character operations
   FIELD_OP

checking
   CHECK_RANGE_OP, CHECK_UPPER_OP, CHECK_LOWER_OP

choice
   boolean:  IF_OP
   arithmetic:  SWITCH_OP

coercions
   CONVERT_OP

common blocks
   DECLARE_STAT_OP

comparison operators
   EQ_OP, GE_OP, GT_OP, LE_OP, LT_OP, NE_OP

complement
   COMPL_OP, NOT_OP

conditional expressions
   IF_OP

conjunction
   AND_OP, ANDAA_OP
```

```
constants
   CONST_OP

continuation of loops
   NEXT_OP

control flow
   BREAK_OP, DO_LOOP_OP, FOR_LOOP_OP, GOTO_OP, IF_OP, LABEL_OP,
   NEXT_OP, PROC_CALL_OP, RETURN_OP, SEQ_OP, SWITCH_OP,
   WHILE_LOOP_OP

conversions
   CONVERT_OP

copy
   ASSIGN_OP

data
   CONST_OP, INITIALIZER_OP, ZERO_INITIALIZER_OP

deallocation
   UNDEFINE_DYNM_OP

declarations
   DEFINE_DYNM_OP, DEFINE_STAT_OP, DECLARE_STAT_OP

decrement
   POSTDEC_OP, PREDEC_OP, SUBAA_OP

default case
   DEFAULT_OP

define
   procedures:  PROC_DEFN_OP
   storage:  DEFINE_DYNM_OP, DEFINE_STAT_OP

dereferencing
   DEREF_OP

descriptor
   address:  REFTO_OP

difference
   SUBAA_OP, SUB_OP

disjunction
   ORAA_OP, OR_OP

disposition of arguments
   PROC_DEFN_ARG_OP

division
   DIVAA_OP, DIV_OP, RSHIFTAA_OP, RSHIFT_OP

do loop
   C-style:  DO_LOOP_OP
```

```
   Fortran-style:  FOR_LOOP_OP

double precision
   LONG_FLOAT_MODE

dynamic variablesa
   DEFINE_DYNM_OP, UNDEFINE_DYNM_OP

element
   of an array:  INDEX_OP
   of a structure or record:  SELECT_OP

else
   IF_OP

entry points
   See descriptions of Intermediate Form stream 1

equality
   EQ_OP, NE_OP

exception
   No exception handling, yet

exclusive-or
   XORAA_OP, XOR_OP

exit
   from procedures:  RETURN_OP
   from loops:  BREAK_OP, NEXT_OP

external symbols
   DECLARE_STAT_OP

false
   zero

fields
   of words:  FIELD_OP
   of structures or records:  SELECT_OP

fixed-point modes
   INT_MODE, LONG_INT_MODE, UNS_MODE, LONG_UNS_MODE

floating-point modes
   FLOAT_MODE, LONG_FLOAT_MODE

flow of control
   BREAK_OP, DO_LOOP_OP, FOR_LOOP_OP, GOTO_OP, IF_OP, LABEL_OP,
   NEXT_OP, PROC_CALL_OP, RETURN_OP, SEQ_OP, SWITCH_OP,
   WHILE_LOOP_OP

formal parameters
   PROC_DEFN_ARG_OP
```

```
functions
   declaration:  PROC_DEFN_OP
   call:  PROC_CALL_OP

global variables
   declaration:  DECLARE_STAT_OP
   definition:  DEFINE_STAT_OP

goto
   GOTO_OP

greater-than
   GT_OP

guarantees
   None here.

immediate operands
   CONST_OP

inclusive-or
   ORAA_OP, OR_OP

incrementation
   ADDAA_OP, POSTINC_OP, PREINC_OP

indexing
   INDEX_OP

indirection
   DEREF_OP

inequality
   EQ_OP, NE_OP

initialization
   INITIALIZER_OP, ZERO_INITIALIZER_OP

integer
   modes:  INT_MODE, LONG_INT_MODE, UNS_MODE, LONG_UNS_MODE
   conversion:  CONVERT_OP

inverse
   additive:  NEG_OP
   bitwise:  COMPL_OP
   boolean:  NOT_OP

invocation
   of procedures:  PROC_CALL_OP

iteration
   DO_LOOP_OP, FOR_LOOP_OP, WHILE_LOOP_OP

jump
   GOTO_OP
```

```
multiway branch
   SWITCH_OP

negation
   NEG_OP

objects
   OBJECT_OP

or (logical)
   ORAA_OP, OR_OP, XORAA_OP, XOR_OP

otherwise
   in Pascal case statement:  DEFAULT_OP

packed data structures
   arrays:  no support
   structures:  FIELD_OP

parameters
   formal:  PROC_DEFN_ARG_OP
   actual:  PROC_CALL_ARG_OP
   pass-by-value:  see VALUE_DISP in PROC_DEFN_ARG_OP
   pass-by-reference:  see REF_DISP in PROC_DEFN_ARG_OP

partial fields
   FIELD_OP

passing parameters
   PROC_CALL_ARG_OP
   by value:  see VALUE_DISP in PROC_DEFN_ARG_OP
   by reference:  see REF_DISP in PROC_DEFN_ARG_OP

pointers
   obtaining them:  REFTO_OP
   indirection through them:  DEREF_OP

portions of a machine word
   FIELD_OP

postdecrement
   POSTDEC_OP

postincrement
   POSTINC_OP

predecrement
   PREDEC_OP

preincrement
   PREINC_OP

primitive data modes
   INT_MODE, LONG_INT_MODE, UNS_MODE, LONG_UNS_MODE, FLOAT_MODE,
   LONG_FLOAT_MODE, STOWED_MODE
```

```
procedure
   calling:  PROC_CALL_OP
   definition:  PROC_DEFN_OP

public symbols
   See description of IMF stream 1
   DECLARE_STAT_OP

quotient
   DIVAA_OP, DIV_OP

range checking
   CHECK_RANGE_OP, CHECK_LOWER_OP, CHECK_UPPER_OP

real
   FLOAT_MODE, LONG_FLOAT_MODE

records
   STOWED_MODE
   SELECT_OP

reference (pass-by)
   see REF_DISP in PROC_DEFN_ARG_OP

references
   REFTO_OP

remainder
   REMAA_OP, REM_OP

reserving storage
   DEFINE_DYNM_OP, DEFINE_STAT_OP

returning from procedures/function/subroutines
   RETURN_OP

semicolon
   SEQ_OP

sets
   bit vector implementations:  FIELD_OP

shift
   left:  LSHIFTAA_OP, LSHIFT_OP
   right:  RSHIFTAA_OP, RSHIFT_OP

short data modes
   INT_MODE, UNS_MODE, FLOAT_MODE

sign change
   NEG_OP

stack
   allocating storage on:  DEFINE_DYNM_OP
   deallocating storage on:  UNDEFINE_DYNM_OP
```

```
statements
   ASSIGN_OP, BREAK_OP, DO_LOOP_OP, FOR_LOOP_OP, GOTO_OP, IF_OP,
   NEXT_OP, PROC_CALL_OP, RETURN_OP, SWITCH_OP, WHILE_LOOP_OP

static variables
   DEFINE_STAT_OP, DECLARE_STAT_OP

storage
   allocation:  DEFINE_DYNM_OP, DEFINE_STAT_OP, DECLARE_STAT_OP
   deallocation:  UNDEFINE_DYNM_OP

structures
   STOWED_MODE
   SELECT_OP

subscripting
   INDEX_OP

subtraction
   SUBAA_OP, SUB_OP, PREDEC_OP, POSTDEC_OP

sum
   ADDAA_OP, ADD_OP, POSTINC_OP, PREINC_OP

switch
   SWITCH_OP, CASE_OP, DEFAULT_OP

target label
   LABEL_OP

temporary variables
   DEFINE_DYNM_OP, UNDEFINE_DYNM_OP

termination
   of procedures:  RETURN_OP

tests
   EQ_OP, GE_OP, GT_OP, LE_OP, LT_OP, NE_OP

transfers
   GOTO_OP

true
   non-zero

truncation
   CONVERT_OP

type
   primitive types:  INT_MODE, LONG_INT_MODE, UNS_MODE,
      LONG_UNS_MODE, FLOAT_MODE, LONG_FLOAT_MODE, STOWED_MODE

unary
   minus:  NEG_OP
   complementation:  COMPL_OP, NOT_OP
```

```
unsigned data modes
   UNS_MODE, LONG_UNS_MODE

upper bound checking
   CHECK_RANGE_OP, CHECK_UPPER_OP

use list
   DECLARE_STAT_OP

value (pass-by)
   see VALUE_DISP in PROC_DEFN_ARG_OP

variables
   OBJECT_OP

vector element selection
   INDEX_OP

zeros
   ZERO_INITIALIZER_OP
```

ADDENDUM

Arnold D. Robbins

August, 1984

Introduction

With the second release of the Georgia Tech C Compiler, 'vcg' has been changed in two ways. This addendum describes those changes.

Object Code Produced Directly

'Vcg' has been changed to directly generate 64V-mode relocatable object code, instead of symbolic assembly language. This enormously speeds up code generation time, since the Prime Macros Assembler, PMA, is no longer needed to turn the assembly code into binary.

As an option, 'vcg' will still produce PMA, which can be assembled normally. This is occasionally useful, since the object code routines still have some bugs buried deep within them. See the help on 'vcg' for details on producing assembly code.

Shift Instructions

Whenever a shift instruction is needed, 'vcg' used to generate code to build an instruction and then XEC it. Now, 'vcg' will generate a shortcall into a table of shift instructions. This table is included in the "vcglib" library, and in the "ciolib" library for C programs. This change saves code space for programs with a lot of shift instructions.