J Saltzer

R. Freiburghouse
October 21, 1970

The Internal Representation of PL/1 Programs

Introduction

This document describes the internal format of a PL/1 program during compilation. Although some fields and values are not developed until the completion of declaration processing or semantic translation, the description is generally valid for the entire compilation.

This description is a complete definition of the input to the code generator and may be used by other Multics compiler writers who wish to use the code generator or by projects who wish to build code generators for other machines or environments.

This document is designed for use as reference material by compiler writers and maintenence personnel. The reader is assumed to completely understand the PL/1 language and should have read "The Multics PL/1 Compiler" by R. Freiburghouse published in the proceedings of the 1969 FJCC.

Contents

1. An Overview

The internal representation of the program being compiled serves as the interface between phases of the compiler. The internal representation is organized into a modified tree structure (the program tree) consisting of nodes which represent the component parts of the program, such as blocks, statements, operators, operands, and declarations. Each node may be logically connected to any number of other nodes by the use of pointers.

Each source program block is represented in the program tree by a block node which has two lists connected to it: a statement list and a declaration list. The elements of the declaration list are symbol table nodes representing declarations of identifiers within that block. The elements of the statement list are nodes representing the source statements of that block. Each statement node contains the root of a computation tree which represents the operations to be performed by that statement. This computation tree consists of operator nodes and reference nodes.

The operators of the internal representation are n-operand operators whose meaning closely parallels that of the PL/1 source operators. The form of a reference is changed by certain phases, but references generally refer to a declaration of some variable or constant. Each reference also serves as the root of a computation tree which describes the computations necessary to locate the item at run time.

This internal representation is machine independent in that it does not reflect the instruction set, the addressing properties, or the register arrangement of the GE645. The first four phases of the compiler are also machine independent since they deal only with this machine independent internal representation. Figure 1 shows the internal representation of a simple program.

```
FACT:   PROC;
        DCL I FIXED,PRINT ENTRY, F ENTRY RETURNS(FIXED) INT;
        DO I = 1 TO 10;
        CALL PRINT("Factorial is", F(I));
        END;
F:      PROC(N) FIXED;
        DCL N FIXED;
        IF N = 0 THEN RETURN(1);
        RETURN(N*F(N-1));
        END F;
        END FACT;
```

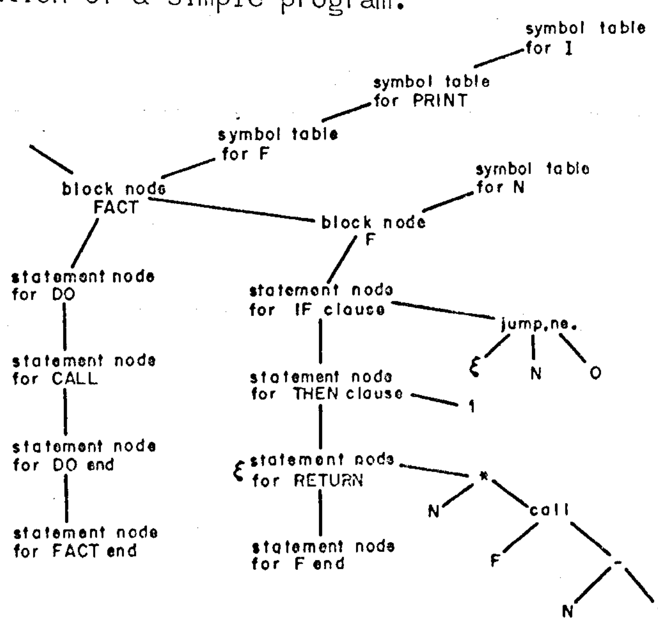


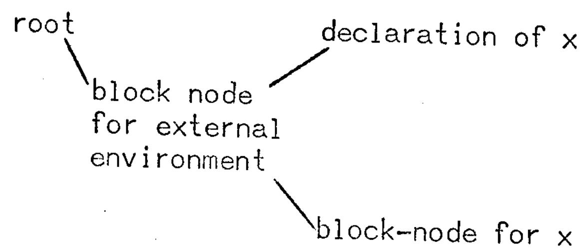
Figure 1—The internal representation of a program. The example is greatly simplified. Only the statements of procedure *F* are shown in detail.

2. The Block Structure

Each begin block, procedure, or on-unit is represented by a block node. The entire tree is found via the external static pointer "root". The outside or external environment of the outermost procedure is represented by a block node whose type is "root_block" and which contains the block which represents the external procedure.

Example:

```
x:  proc;
      :
    end;
```



Format:

```
dcl     1 block               based,
        2 node_type           bit(9),
        2 level               bit(9),
        2 max_arg_no          bit(9),
        2 max_param_no        bit(9),
        2 first_temp          bit(18),
        2 last_temp           bit(18),
        2 father              ptr,
        2 brother             ptr,
        2 son                 ptr,
        2 declaration         ptr,
        2 end_declaration     ptr,
        2 default             ptr,
        2 end_default         ptr,
        2 context             ptr,
        2 prologue            ptr,
        2 end_prologue        ptr,
        2 main                ptr,
        2 end_main            ptr,
        2 last_auto_loc       bit(18),
        2 prefix              bit(12),
        2 block_type          bit(9),
        2 descriptors_used    bit(1),
        2 no_stack            bit(1);
```

<u>node_type</u> - has a value of "000000001"b which identifies this as a block node.

<u>level</u> - is the nesting level of this block.

<u>max_arg_no</u> - the maximum number of arguments in any call made by this block.

max_param_no - the maximum number of parameters in all entries to this block.

first_temp, last_temp - are used by the code generator to remember how it has allocated fixed size temporaries.

father - points to the immediately containing block. This pointer is null for the root block.

brother - points to the next block at this nesting level which has the same father.

son - points to the first contained block.

declaration - points to the first symbol or label node declared in this block.

last_declaration - points to the last symbol or label node declared in this block.

context - used by the parse and declaration processor and is ignored by the code generator.

prologue - points to the first statement node of the prologue.

end_prologue - points to the last statement node of the prologue.

main - points to the first statement node of the main statement sequence.

end_main - points to the last statement node of the main statement sequence.

last_auto_loc - used by the storage allocator as a location counter for allocating constant size automatic variables and temporaries.

prefix - the condition prefix of this block. See section 4.1 for a definition of each bit.

block_type - defines the kind of block this represents. The valid codes are given in the "block-types" include file listed in the appendix.

descriptors_used - this block has a parameter whose extents are given as an asterisk.

no_stack - this block shares its stack frame with its parent block.

3. The Representation of Declarations

Two data bases are used to represent declarations: the token table and the symbol table. The token table contains an entry for each unique token (operator, delimiter, identifier, constant) in the source program. It does not reflect the block structure of the program and can be considered a vector. The symbol table consists of lists of symbol and label nodes attached to block nodes. Each block node contains a uni-directional list of symbol and label nodes which represent the declarations made in that block.

3.1 The Token Table

Each token table entry represents a unique token found in the source program or generated by the compiler.

Format:

```
dcl        1 token                          based,
             2 node_type                    bit(9),
             2 type                         bit(9),
             2 size                         fixed bin(15),
             2 declaration                  ptr,
             2 next                         ptr,
             2 string                       char(n refer(token.size)) aligned;
```

node_type - has a value of "000000101"b which identifies this node as a token table entry.

type - has one of the values listed in the appendix. This value describes the kind of token represented by this node.

size - is the length of the token.

declaration - points to a uni-directional chain of symbol and label nodes which describe the declarations of this token. This pointer is null for tokens other than identifiers.

next - points to the next entry in the token table.

string - is the character string representation of the token.

3.2 The Symbol Table

The symbol table consists of lists of symbol and label nodes attached to block nodes. Each block node contains a pointer to a uni-directional chain of symbol and label nodes, each of which represents a declaration in the block.

3.2.1 Label Nodes

A label node represents the declaration of a statement label or format label constant. It may be a scalar or array. Entry labels are represented by symbol nodes, not label nodes.

Format:

```
dcl     1 label                  based,
        2 node_type              bit(9),
        2 source_id,
        3 line_number            bit(18),
        3 statement_number       bit(9),
        2 unused                 bit(13),
        2 dcl_type               bit(3),
        2 unused2                bit(2),
        2 array                  bit(1),
        2 allocated              bit(1),
        2 location               bit(16),
        2 block_node             ptr,
        2 token                  ptr,
        2 next                   ptr,
        2 multi_use              ptr,
        2 cross_reference        ptr,
        2 statement              ptr,
        2 low_bound              fixed bin(31),
        2 high_bound             fixed bin(31);
```

<u>node_type</u> - has a value of "000001111"b which identifies this node as a label node.

<u>source_id</u> - describes the statement on which this label appeared. For label arrays it identifies the first statement on which one of the array elements appeared.

<u>dcl_type</u> - describes the manner in which the label was declared: "010"b means that the label appeared in the program as was declared by explicit context. "101"b means that the compiler created this label.

array - identifies this as a constant label array.

allocated - indicates that the storage allocator has assigned an actual location in the object program for this label.

location - the address assigned to this label by the storage allocator.

block_node - points to the block node which owns this declaration.

token - points to the token table entry for this identifier.

next - points to the next symbol or label node in this block.

multi_use - points to the next declaration of this identifier (in any block).

cross_reference - points to a uni-directional chain of cross reference nodes, each of which contains a statement-id if a statement which references this label or label array.

statement - points to the statement node representing the statement on which this label appeared. For label arrays this points to the first statement on which one of the array elements appeared.

low_bound - the lower bound of the array.

high_bound - the high bound of the array.

3.2.2 Symbol Nodes

A symbol node represents the declaration of a variable or constant (other than label constants). All scalar and aggregate values are represented in a uniform manner. Variables, constants, entry names, file names, condition names, and temporaries are represented by symbol nodes with the proper storage class and type attributes.

Format:

```
dcl     1 symbol               based,
        2 node_type            bit(9),
        2 source_id,
        3 line_number          bit(18),
        3 statement_number     bit(9),
        2 level                bit(6),
        2 scale                bit(7),
        2 dcl_type             bit(3),
        2 boundary             bit(3),
        2 allocated            bit(1),
        2 location             bit(16),
        2 block_node           ptr,
        2 token                ptr,
        2 next                 ptr,
        2 multi_use            ptr,
        2 cross_references     ptr,
        2 initial              ptr,
        2 array                ptr,
```

```
2 descriptor          ptr,
2 equivalence         ptr,
2 reference           ptr,
2 general             ptr,
2 father              ptr,
2 brother             ptr,
2 son                 ptr,
2 word_size           ptr,
2 bit_size            ptr,
2 dcl_size            ptr,
2 c_word_size         fixed bin(31),
2 c_bit_size          fixed bin(31),
2 c_dcl_size          fixed bin(31),

2 structure           bit(1) aligned,
2 fixed               bit(1),
2 float               bit(1),
2 bit                 bit(1),
2 char                bit(1),
2 ptr                 bit(1),
2 offset              bit(1),
2 area                bit(1),
2 label               bit(1),
2 entry               bit(1),
2 file                bit(1),
2 arg_descriptor      bit(1),
2 storage_block       bit(1),
2 unused              bit(1),
2 condition           bit(1),
2 format              bit(1),
2 builtin             bit(1),
2 generic             bit(1),
2 picture             bit(1),

2 dimensioned         bit(1),
2 initialed           bit(1),
2 aligned             bit(1),
2 unaligned           bit(1),
2 connected           bit(1),
2 unconnected         bit(1),
2 varying             bit(1),
2 local               bit(1),
2 decimal             bit(1),
2 binary              bit(1),
2 real                bit(1),
2 complex             bit(1),
2 variable            bit(1),
2 reducible           bit(1),
2 irreducible         bit(1),
2 returns             bit(1),
2 position            bit(1),
2 internal            bit(1),
2 external            bit(1),
2 like                bit(1),
2 member              bit(1),
```

```
2 auto              bit(1),
2 based             bit(1),
2 static            bit(1),
2 controlled        bit(1),
2 defined           bit(1),
2 parameter         bit(1),
2 param_desc        bit(1),
2 constant          bit(1),
2 temporary         bit(1),
2 return_value      bit(1),

2 print             bit(1),
2 input             bit(1),
2 output            bit(1),
2 update            bit(1),
2 stream            bit(1),
2 bitstream         bit(1),
2 record            bit(1),
2 sequential        bit(1),
2 direct            bit(1),
2 transient         bit(1),
2 buffered          bit(1),
2 unbuffered        bit(1),
2 backwards         bit(1),
2 keyed             bit(1),
2 exclusive         bit(1),
2 environment       bit(1),

2 abnormal          bit(1),
2 packed            bit(1),
2 passed_as_arg     bit(1),
2 allocate          bit(1),
2 set               bit(1),
2 exp_extents       bit(1),
2 refer_extents     bit(1),
2 star_extents      bit(1),
2 no_arguments      bit(1),
2 no_return_value   bit(1);
```

node_type - has a value of "000000110"b which identifies this as a symbol node.

source_id - identifies the statement which declared this value. An all-zero source_id indicates a compiler created declaration.

level - is the structure level (0 and 1 are both "level one" declarations).

scale - is the arithmetic scale factor and is a signed quantity.

dcl_type - indicates how the declaration was established. The values of this field are defined in the "declare_types" include file listed in the appendix.

boundary - describes the storage boundary on which this item is to be allocated. The values of this field are defined in the boundary include file listed in the appendix.

allocated - indicates that the storage allocator has allocated this variable.

location - the address given to this item by the storage allocator.

block_node - points to the block node which owns this declaration;

token - points to the token table entry for this identifier. (Constants and temporaries will have a null value for this pointer).

next - points to the next symbol or label node in this block.

multi_use - if this declaration is a literal constant, this points to the next literal constant in the program. If this declaration is a temporary this points to the next temporary in the program. If this is a variable or named constant this points to another declaration of the same name.

cross_reference - points to a uni-directional chain of cross reference nodes each of which contains the source-id of a statement which references this declaration. (Items without names have a null value for this pointer.)

initial - if this item is an internal entry constant this points to the entry statement on which the entry name appeared. If this item is a named constant or initialized variable this points to a list node or tree of list nodes which represents the initial or constant attribute. If this item is a literal constant this points to the binary representation of the constant's value. If this is a "defined" value this points to the symbol node of the base value.

array - points to an array node which describes the number of dimensions, the bounds, and the multipliers of this array. Refer to section 3.2.3.

descriptor - points to a symbol node whose type is arg_descriptor and whose storage class is automatic, constant, or param_desc. If it is a constant it will appear in the constant list, otherwise it will be in the same block as the declaration which it describes. The semantic translator creates declarations of descriptors when it processes function references and calls. It generates assignment statements to assign the proper values to the descriptor - either in the prologue or immediately before the statement containing the call. If this is an array, the element descriptor is found via the array node.

equivalence - points to the reference given in the defined attribute or to the base constant of a group of equivalenced constants. (See section 3.2.5.) If the type is arg_descriptor for an array element, this may point to the basic descriptor of the entire array.

reference - points to a reference node which describes how to access this value at run-time. For arrays this reference node describes how to access the first element of the array.

general - A general purpose pointer whose meaning depends on other attributes.

1. offset data - points to the area reference given in the offset attribute.

2. pictured data - points to the token table entry representing the picture.

3. entry - points to a uni-directional chain of list nodes each of which points to a symbol node describing a parameter of the entry.

4. generic - points to a uni-directional chain of list nodes each of which points to a symbol node describing an entry descriptor, and to an entry reference.

5. structure - points to the reference given with the like attribute.

6. file constant - points to the declaration of the file block used at run-time.

father - points to the symbol node of the immediately containing structure.

brother - points to the symbol node of the next structure member at this level.

son - points to the first member of this structure. (null for non-structures).

word_size - points to an expression giving the size of this item in words (rounded if necessary). If the size is constant this field is null.

bit_size - points to an expression giving the size of this item in bits. If the size is constant this field is null. (Both bit and word size of dimensioned data is the total array size, not the element size).

dcl_size - points to the declared size of areas or the declared length of strings.

c_word_size - constant size in words (rounded if necessary).

c_bit_size - constant size in bits.

c_dcl_size - constant area size, string length, or arithmetic precision.

The bits of the symbol node are generally self explanatory and are derived from the declare statement and default rules of the language. The compiler-created attributes are described below:

abnormal - computations involving this value cannot be optimized because the value may change without any explicit indication in this program. A value is abnormal if:

1. it is based, defined, parameter, or external
2. it is passed by reference
3. it is used in an addr built-in function
4. it is a member of an abnormal structure or is a structure containing abnormal values

packed - this value is:

1. a packed aggregate (a packed aggregate contains only packed data)
2. unaligned arithmetic data
3. unaligned non-varying string data
4. unaligned pointer data

set - this item appears on the left side of an assignment, in a get list, a read into() statement, or as an argument passed by reference.

star_extents - this item has asterisk extents.

exp_extents - this item has non-constant extents

refer_extents - this item has refer extents or belongs to a structure which has refer extents.

no_return_value - this entry is not a function.

no_arguments - this entry has no arguments.

3.2.3 Array Nodes

The array node and its associated chains of bound pairs serve to describe the elements of an array and provide pre-computed multipliers for use by the subscript processor module of the semantic translator.

Format:

```
dcl        1 array                          based,
           2 node_type                      bit(9),
           2 number_of_dimensions           bit(7),
           2 offset_units                   bit(3),
           2 c_element_size                 fixed bin(31),
           2 c_element_size_bits            fixed bin(31),
           2 c_virtual_origin               fixed bin(31),
           2 element_size                   ptr,
           2 element_size_bits              ptr,
           2 virtual_origin                 ptr,
           2 bounds                         ptr,
           2 desc_bounds                    ptr,
           2 element_descriptor             ptr;

dcl        1 bound                          based,
           2 node_type                      bit(9),
           2 c_lower                        fixed bin(31),
           2 c_upper                        fixed bin(31),
           2 c_multiplier                   fixed bin(31),
           2 next                           ptr,
           2 lower                          ptr,
           2 upper                          ptr,
           2 multiplier                     ptr;
```

node_type - has a value of "000001000"b which identifies this node as an array node.

number_of_dimensions - the number of declared dimensions, plus all dimensions inherited from containing structures.

offset_units - indicates the units of the multipliers. The permitted values are defined by the boundary include file listed in the appendix. Note: descriptor multipliers are always in bits if the item is packed, words if it is not.

c_element_size - constant element size in words (rounded if necessary).

c_element_size_bits - constant element size in bits.

c_virtual_origin - constant offset of the 0th element (when subtracted from the offset of the first element).

element_size - points to an expression giving the element size in words.

element_size_bits - points to an expression giving the element size in bits.

virtual_origin - points to an expression which when subtracted from the offset of the first element gives the offset of the 0th element.

bounds - points to a uni-directional chain of bounds nodes each of which gives a lower bound, upper bound and multiplier. These multipliers are measured in the units indicated by offset_units.

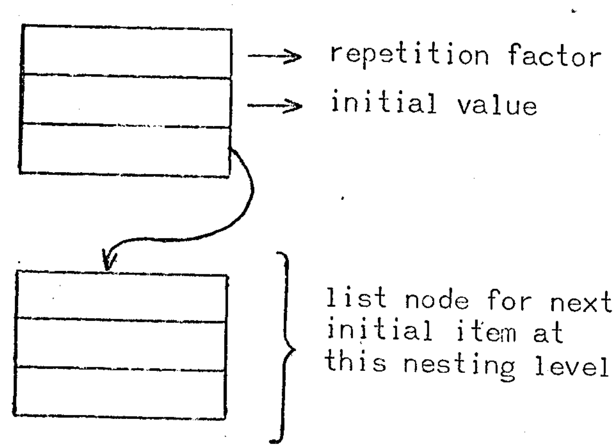
desc_bounds - points to a uni-directional chain of bounds nodes each of which gives a lower bound, upper bound and multiplier. These multipliers are used in constructing argument descriptors. The multipliers are in bits if the item is packed, otherwise they are in words.

element_descriptor - points to a symbol node whose type is arg_descriptor. That descriptor describes the elements of this array and is used when one of those elements is passed as an argument to any entry which requires descriptors. If the equivalence pointer of the element descriptor is not null, it points to the descriptor for the entire array.

3.2.4 The Initial Attribute

The initial attribute of PL/1 is a list of initial items each with a repetition factor or implied repetition factor of one. Each initial item is either an expression, an asterisk, or another initial list.

The parse of an initial attribute is a uni-directional chain of list nodes each representing a single initial item. The nesting of the initial attribute is reflected in the parse as shown below:



The repetition factor is an expression. The initial value is either an expression, a token table entry for an asterisk, or another chain of list nodes representing the parse of the nested initial list.

3.2.5 Storage Classes

The storage mechanism used to contain a value at run-time is defined by the storage class bits of the symbol node.

3.2.5.1 Automatic

If the size (extents) of the value are variable the prologue will contain a statement explicitly allocating the value using an "allot-auto" operator. This operator returns a pointer value which is used to qualify all references to the variable. The code generator does not allocate such variables and it assumes that all necessary pointer qualification has been done by the semantic translator.

Constant size automatic values are allocated by the storage allocator module of the code generator. It only allocates this value if the "allocate" bit is on. Having allocated the value, it sets the "allocated" bit and fills in the "location" field of the symbol node. The location field contains the stack offset of the value. The code generator will add this stack offset to any address it prepares for the value.

The code generator always creates accessing code with the proper block qualification (or display) pointers. The block qualification is not explicitly described in the internal representation.

3.2.5.2 Based

The code generator does not allocate based values. It computes their addresses by evaluating the offset and qualifier expressions found in the reference node used to access the value.

3.2.5.3 Static

Internal static values are allocated by the storage allocator module of the code generator. If the set bit is on, the value is placed in internal static storage (the linkage section) and the "allocated" bit is turned on. The location field is set to contain the offset of the value within the linkage section. This offset is added to any address developed by the code generator.

If the value is not set but is referenced (the "allocate" bit is on) and does not have an initial attribute the storage allocator issues a diagnostic warning the user that the value is used but not set. If the value is used, not set, and is initialized the value has its storage class changed to constant and is allocated within the text of the object program.

Internal static values are initialized by the storage allocator and do not result in the creation of initialization code in the object program. (Areas are an exception and are done the first time the prologue is executed). Note that areas must be initialized to the empty state by explicit code in the prologue.

External static values result in the generation of a link (symbolic reference) in the linkage section of the object program. The storage allocator creates the link and sets the "allocated" bit on. The "location" field is set to contain the offset of this link. All addresses developed by the code generator are effectively indirect references through the link.

If the name of the variable has no $, the link contains information used by the linker (via datmk) which allocates and initializes the variable in stat_ the first time it is referenced in the process. The initial value is compiled into the text of the object program. Areas are initialized by datmk. If the name contains a $, the link does not include initialization or dynamic allocation information.

3.2.5.4 Controlled

Controlled is not supported by this version of the compiler.

3.2.5.5 Defined

No storage is allocated for the value. The code generator develops addresses by combining the address developed from the reference node, and the "location" field of the symbol node found via the "initial" pointer. The initial pointer points to the symbol node of the base value on which this value is defined. If the base value is external static the final reference created by the code generator is indirect through a link.

3.2.5.6 Parameter

Two methods are used to access a parameter and its descriptor:

If a parameter appears in the same position within all entries the "allocated" bit is set on and the "location" field gives the parameter's position. All references to the parameter are qualified by a locator expression consisting of a "param_ptr" operator. The parameter's descriptor is similarly qualified by a "param_desc_ptr" operator. Both of these operators select the kth argument or descriptor pointer.

If a parameter appears in more than one position within different entries, the "allocated" bit is off and the "location" field is zero. The reference node used to access the parameter will be qualified by a unique automatic pointer declared by the compiler. A similar pointer will be used to qualify the parameter's descriptor. Both of these pointers will be set by assignment statements generated at each entry. They are set by param_ptr or "param_desc_ptr" operators. Refer to section 4.5.6.

3.2.5.7 Param-Desc

This storage class is used for parameter descriptors and functions exactly like the parameter storage class. The compiler may create additional declarations of this storage class for entry(), returns(), and generic() attributes. Such declarations have no meaning after semantic translation and have no effect on the code generator since it never finds any references to them.

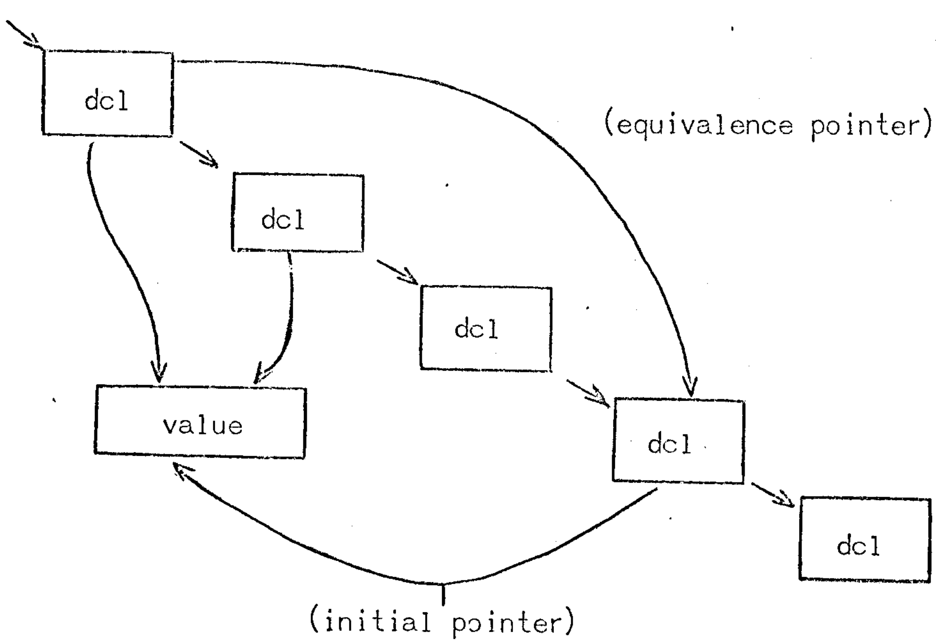
3.2.5.8 Constants

Named constants such as entry and file constants are represented by symbol nodes whose storage class is constant and whose type bits are file or entry. They are not part of the pooling mechanism used for literal constants.

Literal constants may result from source program constants or may be compiler-created. They have no name and therefore do not refer to a token table entry. Each declaration of a constant consists of a symbol node and associated reference node. All such declarations are threaded on a uni-directional chain beginning with the external static pointer "constant_list". Each symbol node contains attributes which describe a value. The binary internal representation of the value is referenced by the "initial" field of the symbol node.

The chain of literal constant declarations is maintained in order of increasing size of the constant's value. More than one declaration may refer to the same value. Such groups of constants are said to be equivalenced. All declarations which have been equivalenced to another have their equivalence pointer set to refer to the symbol node of the constant to which they are equivalenced. A constant which is the base of other equivalenced constants is itself never equivalenced.

Example of equivalenced constants:



3.2.5.9 Temporary Values

The result of each operator is represented by a declaration of a temporary value. Each declaration consists of a symbol node and associated reference node. The symbol node contains all the attributes of the value and has a storage class of "temporary" or "return-value".

All such temporaries are threaded on a uni-directional chain beginning with the external static pointer "temporary_list". The procedure "declare_temporary" does its best to pool temporary declarations to minimize the amount of compiler storage needed to represent these declarations.

Values which are never referenced elsewhere in the program have a storage class of "temporary", and a zero "allocate" bit. They are allocated and freed by the code generator at its discretion.

Values which must be maintained for an extended period of time because they are referenced elsewhere within the same region of the program have a storage class of "temporary" and a "1"b allocate bit. When the object program first compiles their value it retains it until the next statement having a free_temp attribute. (See statement node in section 4.1).

Values which must be maintained for the duration of the block are represented by automatic variables declared in the symbol table. Such variables are not shared and appear as normal variables except that they have no name.

Values returned by functions whose return attribute contains asterisks (returns(char(*))) are represented by declarations whose storage class is "return_value". These termporaries are allocated by the called program but exist in the caller's stack. They continue to exist until a statement having a free-temp attribute is executed.

4. The Representation of Executable Statements

The executable statements of a block are represented by two bi-directional chains of statement nodes attached to the block node. One chain represents the prologue statements generated by the compiler, the other represents the statements written by the programmer or generated from statements written by the programmer.

4.1 Statement Nodes

Each statement is represented by a statement node.

Format:

```
dcl     1 statement                     based,
        2 node_type                     bit(9),
        2 statement_type                bit(9),
        2 source_id,
        3 line_number                   bit(18),
        3 statement_number              bit(9),
        2 prefix                        bit(12),
        2 optimized                     bit(1),
        2 generated                     bit(1),
        2 free_temps                    bit(1),
        2 reference_count               bit(12),
        2 next                          ptr,
        2 back                          ptr,
        2 root                          ptr,
        2 labels                        ptr,
        2 reference_list                ptr;
```

node_type - has a value of "000000001"b which identifies this as a statement node.

statement_type - identifies the kind of statement. Its value is one of the values defined by the "statement-types" include file listed in the appendix.

source_id - identifies the original statement in the source text. Compiler-generated statements will carry the source_id of the original statement from which they were generated or will be zero if no original exists.

prefix - describes the condition prefix found on this source statement or inherited from the block. A value of "1"b means the condition is enabled.

Bit	Meaning
1	underflow
2	overflow
3	zerodivide
4	fixedoverflow
5	*conversion
6	*size
7	subscriptrange
8	stringrange
9	*stringsize
10-12	unused

* not supported by remainder of compiler

optimized - this bit is set on by the optimizer when it first attaches a list of available values to the reference list.

generated - this bit is set on if the statement is compiler-generated.

free_temps - when the code generator encounters a statement node with this attribute it releases all allocated temporaries and return values.

reference_count - indicates the total number of references made by the program to all labels of this statement. It is used by the optimizer for flow analysis.

next - points to the next statement node in this block.

back - points to the previous statement node in this block.

root - points to the computation tree which represents the operators and operands of this statement.

labels - points to a uni-directional chain of list nodes, each of which points to a label node representing the declaration of a label that appeared on this statement. Subscripted labels are represented by a reference node which points to a label node. The offset field of the reference node indicates which element of the label array appeared as a label on this statement.

reference_list - used by the optimizer to collect a list of values which are known to be available when control reaches this statement.

4.2 Reference Nodes

All values (except scalar label constants) are accessed via a reference node. This node contains the offset, length, and other attributes which may be unique for each reference.

Each symbol node has an associated reference node constructed by the declaration processor. This node contains the offset of the item from its level one containing structure. For arrays this node references the first element.

References which are not subscripted, or which do not otherwise have unique offsets (via substr or based structure element references) all share the reference node associated with the symbol node. References with unique offsets, lengths, etc., do not share the symbol tables reference node but use their own unique node. This sharing has no logical significance but does reduce the size of the internal representation.

Format:

```
dcl     1 reference                    based,
        2 node_type                    bit(9),
        2 array_ref                    bit(1),
        2 varying_ref                  bit(1),
        2 padded_ref                   bit(1),
        2 bit                          bit(1),
        2 byte                         bit(1),
        2 half_word                    bit(1),
        2 word                         bit(1),
        2 c_offset                     fixed bin(31),
        2 offset                       ptr,
        2 symbol                       ptr,
        2 qualifier                    ptr,
        2 length                       ptr,
        2 c_length                     fixed bin(31);
```

node_type -- has a value of "000000100"b which identifies this as a reference node.

array_ref - indicates that this is an array reference, not an array element reference.

varying_ref - indicates that this is a reference to a varying string. (This is unique because substr(x,i,j) = y results in a non-varying reference to x even when x is varying).

padded_ref - indicates that the last word of the value is not shared with another value.

bit, byte, half_word, word - indicate the units of the offset expression or constant offset.

c_offset - the constant offset. This field will be zero if the offset is variable.

offset - points to the offset expression. If the offset is entirely constant this field is null.

symbol - points to the symbol or label node which represents the declaration of this value.

qualifier - points to the locator expression used to qualify this reference.

length - points to the length expression giving the current length of the string value.

c_length - the constant current length of a string value.

4.3 List Nodes

The list node is a general purpose node used to chain together other types of nodes. It is used to:

1. chain together the label nodes or label reference nodes which represent the label prefix.
2. chain together parameter descriptors of an entry()or returns() attribute.
3. chain together the members of a generic() attribute.
4. to represent the initial attribute.
5. to represent argument lists and descriptor lists of arg_list operators.

Format:

```
dcl      1 list                    based,
         2 node_type               bit(9),
         2 number                  fixed bin(15),
         2 element(n refer(list.number)) ptr;
```

node_type - has a value of "000001011"b which identifies the node as a list node.

4.4 Operator Nodes

Each operation to be performed by the object program is represented by an operator node. All source language operators and all compiler generated operators have the same form and are subjected to the same optimizations.

Format:

```
dcl      1 operator                based,
         2 node_type               bit(9),
         2 op_code                 bit(9),
         2 shared                  bit(1),
         2 optimized               bit(1),
         2 number                  fixed bin(15),
         2 operand(n refer(operator.number)) ptr;
```

node_type - has a value of "000000011"b which identifies this as an operator node.

op_code - is one of the op codes of the internal representation.

shared - indicates that this operator appears as a subexpression of another computation elsewhere in this program. The optimizer uses this bit to keep itself from getting into trouble.

optimized - this computation has been previously performed and it does not need to be re-evaluated. Operand one contains the correct value. This bit is the means by which the optimizer tells the code generator to suppress redundant computations.

number - the number of operands

operand - pointers to the operands

4.5 The Operators

4.5.1 Arithmetic Operators

Arithmetic operands are:

1. binary fixed {real|complex}
2. binary float {real|complex}
3. decimal {fixed|float}{real|complex}

The code generator performs all necessary conversions between mode for cases 1 and 2. It performs conversions of mode and type for case 3. These conversions are done by the code generator because it can exploit particular hardware features.

Operands may be any precision and scale. The desired output is defined by the attributes of operand one.

Op Code	Value	Definition
add	"000100001"b	opnd(1)<-opnd(2)+opnd(3)
sub	"000100010"b	opnd(1)<-opnd(2)-opnd(3)
mult	"000100011"b	opnd(1)<-opnd(2)*opnd(3)
div	"000100100"b	opnd(1)<-opnd(2)/opnd(3)
negate	"000100101"b	opnd(1)<- -opnd(2)

4.5.2 String Operators

The operands of string operators are scalar string values. They are either a all bit-strings or all character-strings. The boolean operators only allow bit-string operands while the concatenation operator allows either. The reference given as operand one describes the desired result.

```
Op Code         Value             Definition

and_bits        "001000001"b      opnd(1)<-opnd(2) & opnd(3)
or_bits         "001000010"b      opnd(1)<-opnd(2)|opnd(3)
xor_bits        "001000011"b      opnd(1)<-opnd(2) xor opnd(3)
not_bits        "001000100"b      opnd(1)<- ^opnd(2)
cat_string      "001000101"b      opnd(1)<-opnd(2)||opnd(3)
```

4.5.3 Assignment Operators

Assignment operators are used to assign values to variables or to perform conversions of values. They represent special cases of assignment which can either result in very efficient code sequences or which allow the target of the assignment to be accessed without regard to its declared attributes (these are pseudo-variables).

The general assignment operator allows operands of any data type. Conversions are permitted between any combination of arithmetic and string data, between offset and pointer, between pointer and offset, between packed and unpacked data, and it allows assignment of pointer to file, and integer to arg_descriptor, arg_descriptor to integer.

```
assign          "001100001"b      opnd(1)<-opnd(2)
```

Assign_size_ck allows assignments between any combination of arithmetic and string data. If the receiving value has insufficient precision or string length to hold the value the size or stringsize condition is signaled.

```
assign_size_ck  "001100010"b      opnd(1)<-opnd(2)
```

The special case integer assignment operators allow efficient code sequences to be produced for some integer arithmetic operations. Their operands are always fixed binary, single word integers.

```
assign_zero     "001100011"b      opnd(1)<-0
add_1_assign    "001100100"b      opnd(1)<-opnd(1)+1
incr_assign     "001100101"b      opnd(1)<-opnd(1)+opnd(2)
decr_assign     "001100110"b      opnd(1)<-opnd(1)-opnd(2)
diff_assign     "001100111"b      opnd(1)<-opnd(2)-opnd(1)
```

Note: It is somewhat inconsistent with the "machine independent" design philosophy for these special cases to be detected by the semantic translator. Later versions of the code generator may do these special cases thus eliminating these operators.

The copy words operator is created by the semantic translator for aggregate assignment when the two aggregates are of equal size, connected, and have identical composition. Its operands are two aggregate or scalar references and an integer expression giving the number of words to be copied.

```
copy_words "001101000"b   move opnd(2) to opnd(1) by opnd(3) words
```

The set_size operator is an accessing operator which allows assignment to the current size field of a varying string value. Operand one is a varying string and operand two is an integer expression.

```
set_size   "001101001"b   cur_size(opnd(1))<-opnd(2)
```

The set_desc_size and set_desc_type operators are accessing operators which allow assignment to the basic descriptor word of an argument descriptor. Operand one is an argument descriptor variable. Operand two of the set_desc_size operator is an integer expression, operand two of the set_desc_type operator is a bit-string expression.

```
set_desc_size  "001101110"b bits 30-35 of opnd(1) <- opnd(2)
set_desc_type  "001101111"b bits 0-29 of opnd(1) <- opnd(2)
```

The unspec_assign and string_assign operators are accessing operators which allow operand one to receive a value as if it were a bit or character string. Operand one may be a variable of any type. The length field of the reference will be correctly set to reflect the current length by the semantic translator. Operand two is any arithmetic or string expression.

```
unspec_assign  "001101010"b  unspec(opnd(1))<-opnd(2)
string_assign  "001101011"b  char_string(opnd(1))<-opnd(2)
```

The imag_assign and real_assign operators are used to assign to the component parts of complex variables. Operand one is a complex variable {fixed|float} and operand two is any arithmetic or string expression.

```
imag_assign   "001101100"b   imag(opnd(1))<-opnd(2)
real_assign   "001101101"b   real(opnd(1))<-opnd(2)
```

4.5.4 Relational Operators

Operand one of the relational operators is always a bit-string value of length one. The other two operands are either: both arithmetic (see 4.5.1), character-string, bit-string, pointer, offset, label variables, entry variables, or file variables.

Relational operators other than = and ≠ are illegal with complex, pointer, offset, label, entry or file operands. All operands are scalar. There may also be combinations of packed and unpacked values.

Op Code	Value	Definition
less_than	"010000001"b	opnd(1)<-opnd(2) < opnd(3)
greater_than	"010000010"b	opnd(1)<-opnd(2) > opnd(3)
equal	"010000011"b	opnd(1)<-opnd(2) = opnd(3)
not_equal	"010000100"b	opnd(1) <- opnd(2) ^= opnd(3)
less_or_equal	"010000101"b	opnd(1)<-opnd(2) <= opnd(3)
greater_or_equal	"010000110"b	opnd(1)<-opnd(2) >= opnd(3)

4.5.5 Transfer Operators

Operand one of a transfer operator is either a label node, a reference node refering to a label node, or a reference node refering to a symbol node which represents a declaration of a label variable.

The second operand of the jump_true and jump_false operators is a bit-string value. The second and third operands of other conditional transfer operators obey the rules specified for the operands of relational operators.

Op Code	Value	Definition
jump	"010100001"b	go to opnd(1) unconditionally
jump_true	"010100010"b	go to opnd(1) if opnd(2) is not 0
jump_false	"010100011"b	go to opnd(1) if opnd(2) is all 0
jump_if_lt	"010100100"b	go to opnd(1) if opnd(2) < opnd(3)
jump_if_gt	"010100101"b	go to opnd(1) if opnd(2) > opnd(3)
jump_if_eq	"010100110"b	go to opnd(1) if opnd(2) = opnd(3)
jump_if_ne	"010100111"b	go to opnd(1) if opnd(2) ^= opnd(3)
jump_if_le	"010101000"b	go to opnd(1) if opnd(2) <= opnd(3)
jump_if_ge	"010101001"b	go to opnd(1) if opnd(2) >= opnd(3)

4.5.6 Call, Save and Return Operators

The std_arg_list operator results in the creation of a Multics Standard Argument List in automatic storage. Operand one represents the argument list. During argument list creation all argument expressions are evaluated.

The quick_arg_list operator results in the creation of a quick argument List in automatic storage. Operand one represents the argument list.

Operand two is a list node containing a vector of pointers to the argument expressions. The last argument of function references is the return value and is a "return_value", allocated "temporary" or "automatic" value. "Return value" storage class means that the called procedure will allocate space for the return value. (See 3.2.5.9.)

Operand three is a list node containing a vector of pointers to the argument descriptors. If no descriptors are needed operand three is null.

```
std_arg_list   "011000001"b opnd(1)<-arglist(opnd(2) desclist(opnd(3)))
quick_arg_list "011000010"b opnd(1)<-arglist(opnd(2) desclist(opnd(3)))
```

The std_call operator results in a Multics Standard Call. The quick_call operator results in a quick call. Operand one is null if the call is not a function reference; otherwise it points to the reference node used to access the return value. Operand two is an entry expression giving the entry to be invoked. Operand three is null if there are no arguments or return value; otherwise it is an argument list operator which prepared the argument list.

```
std_call     "011000011"b    opnd(1)<-call opnd(2) with opnd(3)
quick_call   "011000100"b    opnd(1)<-call opnd(2) with opnd(3)
```

The std_entry operator results in the creation of entry descriptive information and a Multics Standard entry sequence in the object program. The entry descriptive information includes the number of parameters and type codes for each parameter.

```
std_entry    "011000101"b    entry(opnd(1)... opnd(n))
quick_entry  "011000110"b    entry(opnd(1)... opnd(n))
```

The quick_entry operator is used to define the entry to a quick subroutine. It causes the creation of a quick entry sequence in the object program.

```
ex_prologue "011000111"b   execute the prologue -no operands-
```

The ex_prologue operator causes the prologue to be evaluated.

```
allot_auto   "011001000"b    opnd(1)<- addrel(stack,opnd(2))
```

The allot_auto operator makes permanent allocations in the stack. It is a pointer valued operator whose second operand is an integer expression. The storage is released by the return or non-local go to operator.

```
param_ptr       "011001001"b opnd(1)<- ptr to kth argument, k=opnd(2)
param_desc_ptr  "011001010"b opnd(1)<- ptr to kth descriptor, k=opnd(2)
```

The "param_ptr" and "param_desc_ptr" are used to access the argument pointer and argument descriptor pointer which references the kth argument of the entry used to invoke the procedure. They are used to assign these pointers to the automatic pointers used to reference the parameter or parameter descriptor. See section 3.2.5.6.

```
std_return    "011001011"b   return -no arguments-
```

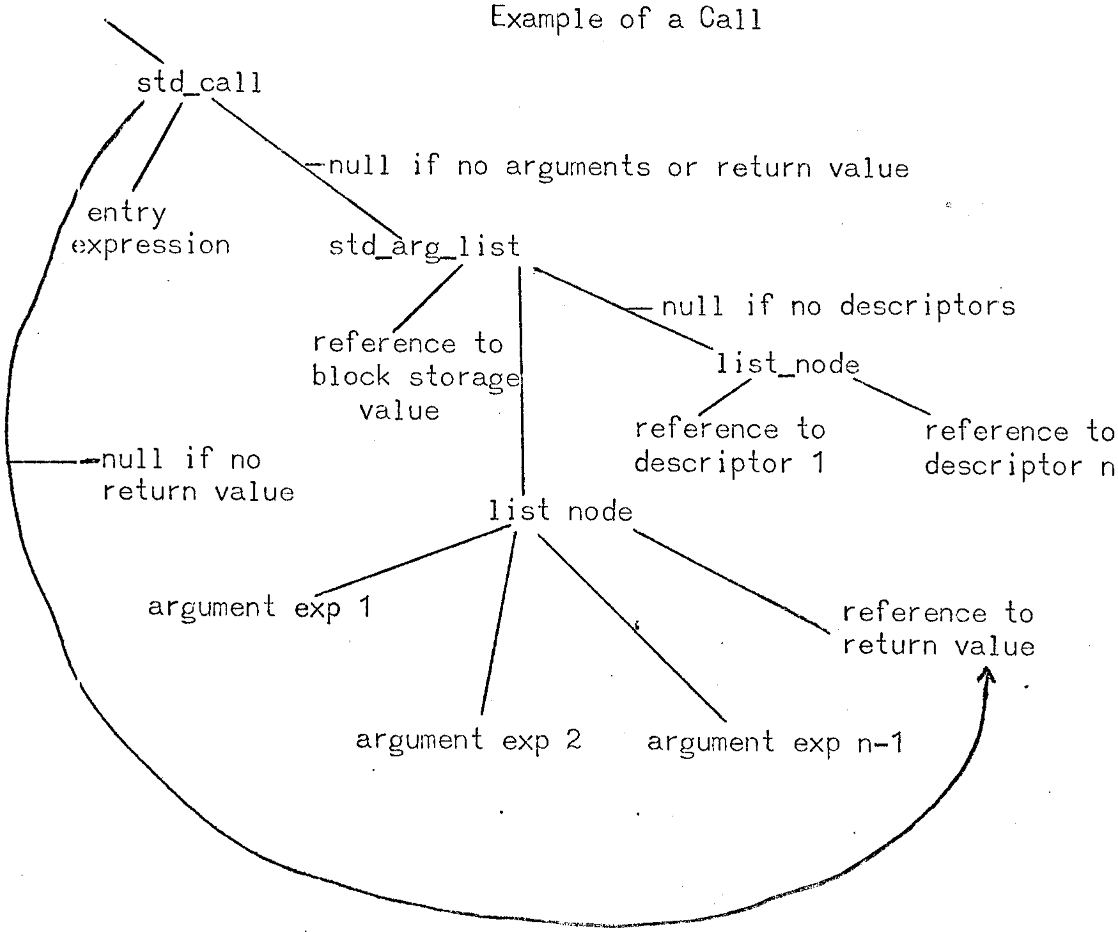
The std_return operator returns via the Multics Standard Return. It has no arguments - an assignment statement has already assigned the return value to the last parameter.

```
return_value     "011001100"b     return(opnd(1))
quick_return     "011001101"b     return -no arguments-
```

The return_value operator returns via the Multics standard return, but requires the evaluation, allocation, and assignment of the return value to the last parameter. The descriptor of the return value has already been set.

The quick_return operator performs a quick return. Any return value has already been assigned to the last parameter.

Example of a Call



4.5.7 Offset Operators

Offset operators are used to compute the addresses of values at run-time. Their output operands are binary integers and their input operands are usually binary integer expressions. The "desc_size" operator has an arg_descriptor as operand two, and the "bit_pointer" operator has a pointer value as operand two.

Op Code	Value	Definition
bit_to_char	"011100001"b	opnd(1) <- (opnd(2)+8)/9
bit_to_word	"011100010"b	opnd(1) <- (opnd(2)+35)/36
char_to_word	"011100011"b	opnd(1) <- (opnd(2)+3)/4
half_to_word	"011100100"b	opnd(1) <- (opnd(2)+1)/2
word_to_mod2	"011100101"b	opnd(1) <- (opnd(2)+1)/2*2
word_to_mod4	"011100110"b	opnd(1) <- (opnd(2)+3)/4*4
word_to_mod8	"011100111"b	opnd(1) <- (opnd(2)+7)/8*8
bit_pointer	"011101000"b	opnd(1) <- bit offset of opnd(2)
read_size	"011101001"b	opnd(1) <- current length of opnd(2)
bound_ck	"011101010"b	opnd(1)<-opnd(2) if <=opnd(4) & >=opnd(3)
desc_size	"011101110"b	opnd(1) <- bits 20-35 of desc

4.5.8 Built-in Function Operators

The built-in function operators are a miscellaneous group of operators which support PL/1 built-in functions. The types of their arguments are defined by the language. All argument conversions required by the language have been done and are not implied by the operator.

Op Code	Value	Definition
addr_fun	"100000001"b	opnd(1)<- addr(opnd(2))
addr_fun_bits	"100000010"b	opnd(1)<- addr(opnd(2))
ptr_fun	"100000011"b	opnd(1)<- ptr(opnd(2),opnd(3))
index_fun	"100000100"b	opnd(1)<- index(opnd(2),opnd(3))
off_fun	"100000101"b	opnd(1)<- offset(opnd(2),opnd(3))
sign_fun	"100000110"b	opnd(1)<-sign(opnd(2))
abs_fun	"100000111"b	opnd(1)<-abs(opnd(2))
imag_fun	"100001000"b	opnd(1)<-imag(opnd(2))
real_fun	"100001001"b	opnd(1)<-real(opnd(2))
complex_fun	"100001010"b	opnd(1)<-complex(opnd(2),opnd(3))
conjg_fun	"100001011"b	opnd(1)<-conjg(opnd(2),opnd(3))
min_fun	"100001100"b	opnd(1)<-min(opnd(2) ... opnd(n))
max_fun	"100001101"b	opnd(1)<-max(opnd(2) ... opnd(n))
mod_fun	"100001110"b	opnd(1)<-mod(opnd(2),opnd(3))
repeat_fun	"100001111"b	opnd(1)<-repeat(opnd(2),opnd(3))
translate	"100010000"b	opnd(1)<-translate(opnd(2),opnd(3))
verify	"100010001"b	opnd(1)<-verify(opnd(2),opnd(3))
rel_fun	"100010010"b	opnd(1)<-rel(opnd(2))
baseno_fun	"100010011"b	opnd(1)<-baseno(opnd(2))
baseptr_fun	"100010100"b	opnd(1)<-baseptr(opnd(2))
addrel_fun	"100010101"b	opnd(1)<-addrel(opnd(2),opnd(3))
lock_fun	"100010110"b	opnd(1)<-stac(opnd(2),opnd(3))
clock_fun	"100010111"b	opnd(1)<- clock reading
unspec_fun	"100011000"b	opnd(1)<-unspec(opnd(2))
string_fun	"100011001"b	opnd(1)<-char string(opnd(2))

4.5.9 Input/Output Operators

Additional operators are used to drive the code generator into creating code for PL/1 input/output statements. These operators will be defined in a later version of this document.

r_parn	"100100001"b
l_parn	"100100010"b
r_format	"100100011"b
c_format	"100100100"b
f_format	"100100101"b
e_format	"100100110"b
b_format	"100100111"b
a_format	"100101000"b
x_format	"100101001"b
skip_format	"100101010"b
column_format	"100101011"b
page_format	"100101100"b
line_format	"100101101"b

Appendix - Codes Used In The Internal Representation

block_types.incl.pl1

```
dcl (      root_block                      initial("000000001"b),
           external_procedure              initial("000000010"b),
           internal_procedure              initial("000000011"b),
           begin_block                     initial("000000100"b),
           on_unit                         initial("000000101"b))
```

boundary.incl.pl1

```
dcl (      bit_boundary                    initial("001"b),
           character_boundary              initial("010"b),
           half_boundary                   initial("011"b),
           word_boundary                   initial("100"b),
           mod2_boundary                   initial("101"b),
           mod4_boundary                   initial("110"b),
           mod8_boundary                   initial("111"b))
```

declare_type.incl.pl1

```
dcl (      by_declare             initial("001"b),
           by_explicit_context    initial("010"b),
           by_context             initial("011"b),
           by_implication         initial("100"b),
           by_compiler            initial("101"b))
```

nodes.incl.pl1

```
dcl (      block_node                      initial("000000001"b),
           statement_node                  initial("000000010"b),
           operator_node                   initial("000000011"b),
           reference_node                  initial("000000100"b),
           token_node                      initial("000000101"b),
           symbol_node                     initial("000000110"b),
           context_node                    initial("000000111"b),
           array_node                      initial("000001000"b),
           bound_node                      initial("000001001"b),
           parameter_node                  initial("000001010"b),
           list_node                       initial("000001011"b),
           default_node                    initial("000001100"b),
           rand_node                       initial("000001101"b),
           address_node                    initial("000001110"b),
           label_node                      initial("000001111"b))

dcl        1 node     aligned based,
           2 type     bit(9);
```

statement_types.incl.pl1

```
        /* statement types */

dcl (   unknown_statement            initial("000000000"b),
        allocate_statement           initial("000000001"b),
        assignment_statement         initial("000000010"b),
        begin_statement              initial("000000011"b),
        call_statement               initial("000000100"b),
        close_statement              initial("000000101"b),
        declare_statement            initial("000000110"b),
        delay_statement              initial("000000111"b),
        delete_statement             initial("000001000"b),
        display_statement            initial("000001001"b),
        do_statement                 initial("000001010"b),
        else_clause                  initial("000001011"b),
        end_statement                initial("000001100"b),
        entry_statement              initial("000001101"b),
        exit_statement               initial("000001110"b),
        format_statement             initial("000001111"b),
        free_statement               initial("000010000"b),
        get_statement                initial("000010001"b),
        goto_statement               initial("000010010"b),
        if_statement                 initial("000010011"b),
        locate_statement             initial("000010100"b),
        null_statement               initial("000010101"b),
        on_statement                 initial("000010110"b),
        open_statement               initial("000010111"b),
        procedure_statement          initial("000011000"b),
        put_statement                initial("000011001"b),
        read_statement               initial("000011010"b),
        return_statement             initial("000011011"b),
        revert_statement             initial("000011100"b),
        rewrite_statement            initial("000011101"b),
        signal_statement             initial("000011110"b),
        stop_statement               initial("000011111"b),
        system_on_unit               initial("000100000"b),
        unlock_statement             initial("000100001"b),
        wait_statement               initial("000100010"b),
        write_statement              initial("000100011"b),
        default_statement            initial("000100100"b))
```

system.incl.pl1

```
dcl  (    max_p_flt_bin_1       initial(27),
          max_p_flt_bin_2       initial(63),
          max_p_fix_bin_1       initial(35),
          max_p_fix_bin_2       initial(71),

          max_p_flt_dec_1       initial(8),
          max_p_flt_dec_2       initial(18),
          max_p_fix_dec_1       initial(10),
          max_p_fix_dec_2       initial(21),

          max_scale_bin         initial(99999),
          max_scale_dec         initial(99999),
          max_bit_string        initial(2359296),
          max_char_string       initial(262144),
          max_area_size         initial(65536),

          bits_per_word         initial(36),
          characters_per_word   initial(4),
          bits_per_character    initial(9),

          default_area_size     initial(1024),
          default_flt_bin_p     initial(27),
          default_fix_bin_p     initial(17),
          default_flt_dec_p     initial(8),
          default_fix_dec_p     initial(5))
```

token_types.incl.pl1

```
dcl (    no_token                  initial("000000000"b),
         identifier                initial("100000000"b),
         isub                      initial("010000000"b),
         plus                      initial("001000001"b),
         minus                     initial("001000010"b),
         asterisk                  initial("001000011"b),
         slash                     initial("001000100"b),
         expon                     initial("001000101"b),
         not                       initial("001000110"b),
         and                       initial("001000111"b),
         or                        initial("001001000"b),
         cat                       initial("001001001"b),
         eq                        initial("001001010"b),
         ne                        initial("001001011"b),
         lt                        initial("001001100"b),
         gt                        initial("001001101"b),
         le                        initial("001001110"b),
         ge                        initial("001001111"b),
         ngt                       initial("001010000"b),
         nlt                       initial("001010001"b),
         assignment                initial("001010010"b),
         colon                     initial("001010011"b),
         semi_colon                initial("001010100"b),
         comma                     initial("001010101"b),
         period                    initial("001010110"b),
         arrow                     initial("001010111"b),
         left_parn                 initial("001011000"b),
         right_parn                initial("001011001"b),
         bit_string                initial("000100001"b),
         char_string               initial("000100010"b),
         bin_integer               initial("000110001"b),
         dec_integer               initial("000110011"b),
         fixed_bin                 initial("000110000"b),
         fixed_dec                 initial("000110010"b),
         float_bin                 initial("000110100"b),
         float_dec                 initial("000110110"b),
         i_bin_integer             initial("000111001"b),
         i_dec_integer             initial("000111011"b),
         i_fixed_bin               initial("000111000"b),
         i_fixed_dec               initial("000111010"b),
         i_float_bin               initial("000111100"b),
         i_float_dec               initial("000111110"b),

         The following four token_types are for fortran only

         label_argument                      initial("010000001"b),
         hollerith_constant_header           initial("010000010"b),
         x_format_f                          initial("010000011"b),
         logical_constant                    initial("000100001"b))
```