Published: 07/25/69

Identification

Format of PL/I Programs' Internal Representation
R. Freiburghouse, J. Mills

1. Introduction

This document describes the format of the internal representation of PL/I programs during their compilation. It does not discuss the respresentation of declarations (the symbol table) nor does it describe data bases which are unique to one or two phases.

The internal representation discussed in this document is produced by the parse and is later modified by the semantic translator. However, the basic relationships between the elements of this representation remain essentially unchanged until it is converted into machine code by the code generator. The detailed representation of each statement is given in MSPM BZ.8.12 THE OUTPUT OF THE SYNTACTIC TRANSLATOR.

The representation is a structure consisting of various kinds of components (nodes) which are linked to each other by pointers. It may be considered to be basically a tree structure containing back pointers and cross pointers. The major types of nodes in this structure are briefly described below:

symbol table nodes - represent the declarations of PL/I data.

block nodes - represent the block structure of the program, created for procedures, begin blocks, and ON units.

group nodes - represent DO groups with specifications, these nodes facilitate DO group optimization.

statement nodes - represent source statements and statements created by the compiler.

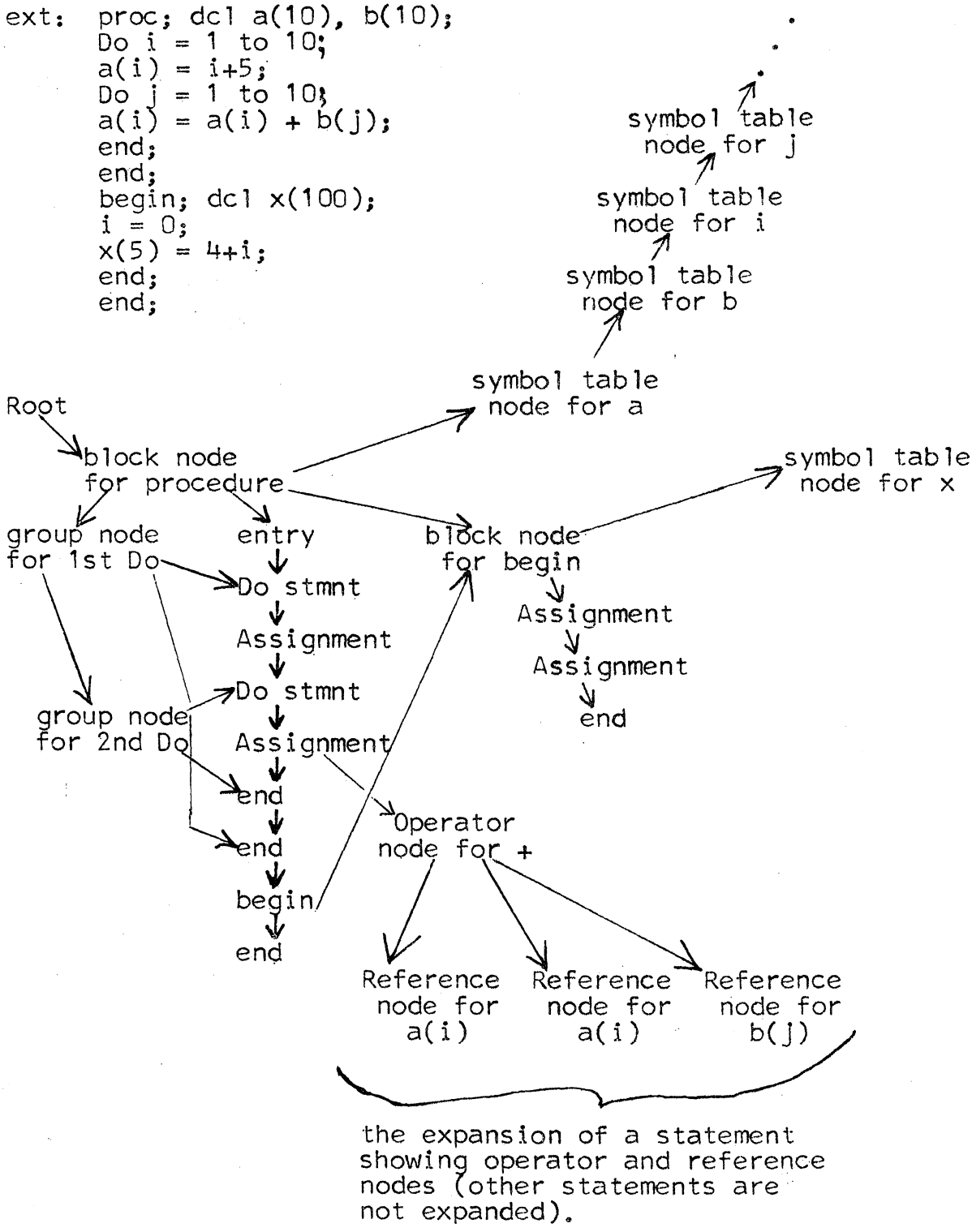
operator nodes - represent the operations to be performed.

operand nodes - represent the operands of the program. After the parse they represent only the syntax of the original source program operands. The semantic translator replaces these nodes with reference nodes which do convey the semantics of the operands.

reference nodes - represent the operands of the program after the semantic translation. These nodes reference the symbol table declaration of the source operand. They also contain the computations necessary to locate the item at run time.

The relationship between these nodes is shown in the example which follows. Note that the arrows represent pointers and also that the example is somewhat simplified to retain some measure of clarity.

```
ext:  proc; dcl a(10), b(10);
      Do i = 1 to 10;
      a(i) = i+5;
      Do j = 1 to 10;
      a(i) = a(i) + b(j);
      end;
      end;
      begin; dcl x(100);
      i = 0;
      x(5) = 4+i;
      end;
      end;
```



the expansion of a statement showing operator and reference nodes (other statements are not expanded).

Code Expansion

During the parse and the semantic translation, a good deal of code expansion occurs as operations which were implied in the source program become explicit in the internal representation of the program. This expansion is done in the following ways:

1. By the addition of new statement nodes - e.g., the generation of prologue code, the expansion of ON statements, etc.

2. By the addition of more operations in some computation tree already rooted in some statement node.

Code Ordering

Compilation also results in code reordering. This is done in the following ways:

1. Internal procedures and ON units will be removed and placed at the end of the procedure segment. The example shows how the internal representation facilitates this.

2. Code insertions will be done into one of two code sequences within some block. For this purpose each block node contains a pointer to the current end of the prologue and main code sequences.

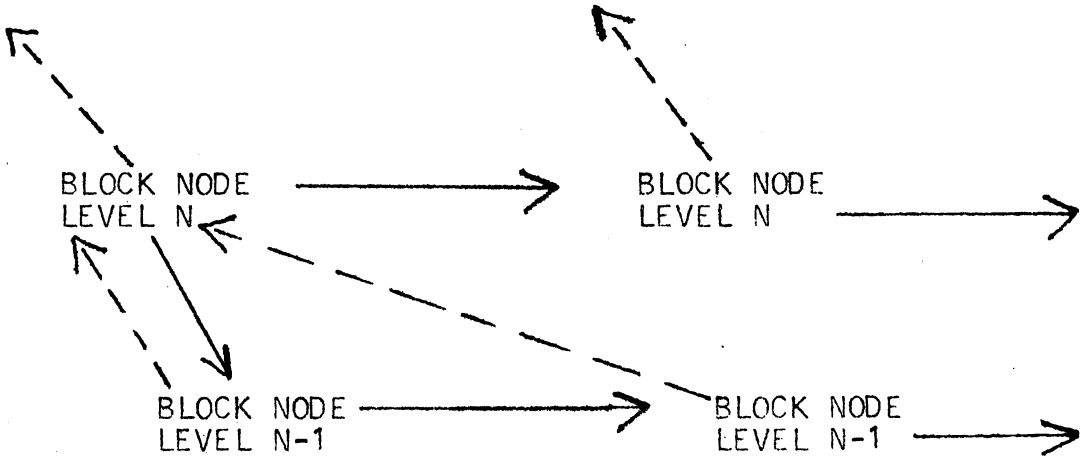
3. The reordering of operations within some statement. This can be accomplished by the appropriate linking and unlinking of branches in the computation trees.

Implementation

The Block Node

Each source language procedure, begin block, or ON unit; and each compiler created procedure is represented

by a block node in the internal representation. Block nodes are structured as indicated in the following example:



Definition of a Block Node

```
dcl  1  block                 based(p)
     2  node_type             fixed bin(15),
     2  block_type            fixed bin(15),
     2  last_auto_loc         fixed bin(31),
     2  bits,
     3  prefix                bit(12),
     3  recursive             bit(1),
     3  main                  bit(1),
     3  descriptors_used      bit(1),
     2  father                ptr,
     2  brother               ptr,
     2  son                   ptr,
     2  group                 ptr,
     2  declaration           ptr,
     2  end_declaration       ptr,
     2  context               ptr,
     2  allocate_stmnt        ptr,
     2  open_stmnt            ptr,
     2  entry_list            ptr,
     2  checked_list          ptr,
     2  prologue              ptr,
     2  end_prologue          ptr,
     2  main                  ptr,
```

```
    2  end_main          ptr,
    2  auto_adj_loc      ptr,
    2  first_temp        fixed bin(31),
    2  last_temp         fixed bin(31),
    2  max_arg_no        fixed bin(15),
    2  level             fixed bin(15),
    2  spare             fixed bin(15),
    2  spare_ptr         ptr;
```

node type - is a constant which identifies this node as a block node. See Appendix 1.

block type - has any of the values:

1. root block
2. external procedure
3. internal procedure
4. begin block
5. on unit

last auto loc - stack frame allocation counter used by the code generator only.

prefix - is a bit string whose bits represent the condition prefixes. A value of "1"b means the condition is set.

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

recursive - equal to "1"b if the recursive option was specified on an external procedure statement.

main - unused.

descriptors used - equal to "1"b if this block uses parameter descriptors.

father - is a pointer to the block node to which this node is immediately internal.

brother - is a pointer to a block node at this same nesting level.

son - is a pointer to the first immediately contained block.

group - is a pointer to the first DO group node in this block.

declaration - is a pointer to the first symbol table node which resulted from a declaration in this block. All such declarations are chained and each contains a pointer to this block node.

end declaration - is a pointer to the end of the declaration list which is associated with this block. It is used to facilitate the adding of declarations to the block.

context - ptr to a list of nodes which represent contextual information about identifiers in this block. This information is recorded by the parse and processed by the declaration processor. After declaration processing the context field is used to point to a list of temporary nodes for the block.

allocate stmnt - ptr to the first of a list of all the allocate statement nodes in the block.

open stmnt - ptr to the first of a list of all the open statement nodes in the block.

entry list - ptr to the first of a list of all the entry and procedure statement nodes in the block.

checked list - is a pointer to a list which represents the identifiers which appeared in this block's check condition prefix. Each node of the list is declared to be:

```
dcl 1 check_node        based(p),
    2 check_nocheck     bit(1),   /* = "1"b means check */
    2 next              ptr,      /* ptr to next check_node */
    2 identifier        ptr,      /* to token-table node for
                                     checked identifier */
```

prologue
end prologue,
main
end main

} are pointers to the beginnings and ends of lists of statements.

auto adj loc - stack frame allocation counter for prologue. Used by code generator.

first temp - beginning of temp storage area.

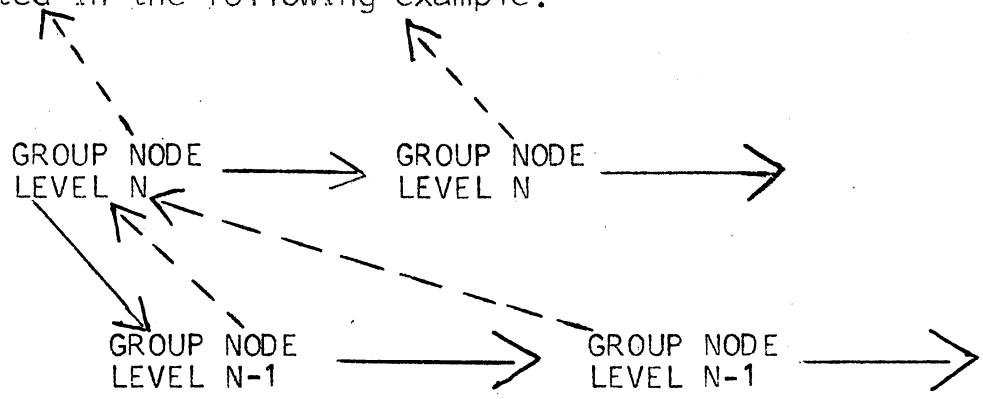
last temp - last used area in temp storage.

level - nesting level of block.

spare and spare ptr - unused.

The Group Node

Each source language DO statement with specifications is represented by a group node. Group nodes are constructed by the parse and are used by the DO processor to optimize the code within the DO group. Group nodes can be structured as indicated in the following example:



Definition of the Group Node

```
dcl  1  group          based(p),
     2  node_type      fixed,
     2  cv_list        ptr,
     2  father         ptr,
     2  brother        ptr,
     2  son            ptr,
     2  head           ptr,
     2  tail           ptr,
     2  list           ptr;
```

node_type - is a constant which indicates that this is a group node. See Appendix 1.

cv_list - is a pointer to a list of nodes which are used to transmit information about the control variable of the DO statement from the parse to the semantic translator.

father - is a pointer to the group or block node to which this group is immediately internal.

brother - is a pointer to a group node at this level.

son - is a pointer to the first group node which is immediately internal to this group.

head - is a pointer to the statement node representing the DO statement.

tail - is a pointer to the statement node of the end statement which marks the end of the group.

list - is a pointer to the data base built during the execution of the optimizer.

The Statement Node

Each source language statement or compiler generated statement is represented by a statement node. The statement node

contains information which is common to all types of statements. The specific operations to be performed by the execution of the statement are represented by a computation tree. A computation tree is a structure consisting of operator and reference nodes which has its root in the statement node. The statement nodes representing the source statements of some block are linked together into a chain. This chain can be considered to have two parts: the prologue part and the main part. The main part follows the prologue part. The block node contains pointers to each part of the chain so that each part can easily be updated by the addition or removal of nodes.

Definition of a Statement Node

```
dcl  1  statement          based(p),
     2  node_type          fixed bin(15),
     2  stmnt_type         fixed bin(15),
     2  reference_count    fixed bin(15),
     2  source_id          fixed bin(31),
     2  sub_stmnt_id       fixed bin(15),
     2  prefix             bit(12),
     2  back               ptr,
     2  next               ptr,
     2  next_of_kind       ptr,
     2  root               ptr,
     2  labels             ptr,
     2  reference_list     ptr;
```

node type - is a constant which indicates that this is a statement node. See Appendix 1.

stmnt type - indicates the kind of statement. See Appendix 2.

reference count - this is a count of the references to this statement.

source id - contains, in the left 16 bits, the number of the line on which the statement appeared. The right 15 bits contain the number of the statement on that line.

sub stmnt id - currently unused but reserved for use in identifying compiler produced statements.

prefix - is a bit string whose bits represent the condition prefixed. A value of "1"b means the condition is set.

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

back - is a pointer to the previous statement node in the statement list.

next - is a pointer to the next statement node in the statement list.

next of kind - for entry or procedure, open, and allocate statements this is a pointer to the next statement of that kind in this block.

root - is a pointer to the computation tree which represents the operations to be performed by the execution of this statement.

labels - is a pointer to a list of nodes representing the labels written on this statement.

The declaration of these nodes is:

```
dcl  1  label_node      based(p),
     2  next            ptr,
     2  reference       ptr;
```

where: next points to the next label and

reference points to a token-table node or tree representing the label.

reference list - is a pointer to a data base built and used by the optimization phase. The pointer is null if this statement is not referenced and none of its labels is passed as an argument or assigned to a label variable.

Computation Trees

The operations which are to be performed for the execution of a statement are expressed as a computation tree consisting of operator and various kinds of operand nodes.

The meaning of each of the fields in these nodes depends on whether or not the semantic translation has been done. In the discussion of the nodes we shall try to describe the fields both after the parse and after the semantic translation.

The syntaxes of the intermediate representation for each PL/I source statement, after the parse and after semantic translation, are given in Sections BZ.8.12, "OUTPUT OF THE SYNTACTIC TRANSLATOR" and BZ.8.13 "OUTPUT OF SEMANTIC TRANSLATION".

Operator Nodes

Operator nodes represent source operations or operations derived from or implied by the source. After the parse they largely represent source PL/I operators, but the semantic translator modifies these operators to reflect their implementation in terms of library calls or intermediate operators. The format of the operator node allows an operator to have any number of operands.

After the parse an operand is a token-table node if the source program operand was a simple identifier or a constant, an operand node if the source program was a construct of the form a(), or an operator may be an operand.

After semantic translation, computation trees consist of operators whose operands are:

reference nodes

string-reference nodes

temporary nodes

entry-attribute blocks

label-attribute blocks

parameter nodes

constant nodes

operator nodes

Of the nodes mentioned so far, the following are defined in MSPM BZ.8.10, "THE FORMAT OF DECLARATIONS".

token-table nodes

entry-attribute blocks

label-attribute blocks

constant nodes

Binary operators have three operands, the third being the description of the result. This facilitates the allocation of temporaries, the sharing of common subexpression values, and the representation of conversions.

Definition of an Operator Node

```
dcl  1  operator          based(p),
     2  node_type         fixed bin(15),
     2  number            fixed bin(15),
     2  back              ptr,
```

```
        2  op_code          fix bin(15),
        2  qualifier        fixed bin(15),
        2  operand(n)       ptr;
```

node_type - is a constant indicating that this is an operator node. See Appendix 1.

number - is the number of operands and is equivalent to n.

back - is a pointer to the node which references this node. It is set and used by the optimizer.

op_code - is a constant identifying the operation to be performed. The values are given in Appendix 3.

qualifier - currently unused.

operand(n) - is an array of pointers to the operands of this operator.

Operand Nodes

The parse produces an operand node, or list of operand nodes, for all source operands other than for a single unqualified identifier or constant. In other words - it produces an operand node for a subscripted reference, and for a procedure reference with arguments (which it can not distinguish from a subscripted reference).

Definition of the Operand Node

```
dcl  1  operand          based(p),
     2  node_type        fixed bin(15),
     2  number           fixed bin(15),
     2  identifier       ptr,
     2  list(n)          ptr;
```

node type - a constant which identifies this node as an operand node. See Appendix 1.

number - the number of expressions in the parenthesized list which follows the source identifier. If no such list exists its value is zero. It is equivalent to n.

identifier - a pointer to the token-table entry for the identifier which composes this operand.

list(n) - an array of pointers to computation trees which represent the expressions in the list. These expressions are either subscript expressions or arguments to a function but this is not known at the time of the parse.

Reference Nodes

Each source operand which consists of a reference to a variable will be transformed by the semantic translator into a reference node (or string - reference node). This reference node will be unique to the particular reference only if the reference is subscripted, derived from a substr builtin function, or pointer qualified. In all other cases the reference node will be connected to the symbol table and will be shared by all references. Reference nodes serve to carry the accessing computations which are required to locate the operand at run time. Offsets may be zero or constant or variable.

Definition of a Reference Node

```
dcl  1  reference              based(p),
     2  node_type              fixed bin(15),
     2  const_units_offset     fixed bin(31),
     2  units_offset           ptr,
     2  symbol                 ptr,
     2  bits,
     3  array_ref              bit(1);
```

node type - a constant which identifies this node as a reference node. See Appendix 1.

const units offset - this is the constant part of the offset expression which locates the variable. This offset is expressed in addressable units (words) and describes the distance from the origin of the level one containing aggregate.

units offset - is a pointer to a computation tree giving the variable part of the offset expression.

symbol - is a pointer to an attribute block of a symbol table entry.

array ref - if "1"b then the reference is to an array rather than an element of the array.

String-Reference Nodes

A string-reference node is the reference node created by the semantic translator for references to strings and structures.

Definition of a String-Reference Node

```
dcl  1  string_reference          based(p),
     2  node_type                 fixed bin(15),
     2  const_units_offset        fixed bin(31),
     2  units_offset              ptr,
     2  symbol                    ptr,
     2  fractional_offset         ptr,
     2  current_size              ptr,
     2  bits,
     3  padded                    bit(1),
     3  varying                   bit(1),
```

```
    3 array_ref                bit(1),
    2 const_current_size       fixed bin(31),
    2 const_fractional_offset  fixed bin(31);
```

node type - is a constant which identifies this node as a string-reference node. See Appendix 1.

const units offset - this is the constant part of the offset expression which locates the variable. This offset is expressed in addressable units (words) and describes the distance from the origin of the level one containing aggregate.

units offset - is a pointer to a computation tree giving the variable part of the offset expression.

symbol - is a pointer to an attribute block of a symbol table entry.

fractional offset - is a pointer to an expression which describes the bit offset. The bit offset is the number of bits between the origin of the variable and the previous boundary. If the bit offset is constant this pointer is null.

current size - is a pointer to an expression which describes the current size of a string variable. The value is measured in characters or bits depending on the type of the string.

padded - if equal to "1"b then the remainder of the last word of storage for this variable is not otherwise used.

varying - if equal to "1"b then this is an access to a varying string.

array ref - if equal to "1"b then the reference is to an array rather than to an element of the array.

const current size - this is the current length of the string measured in bits or characters. If the current length is variable this value is zero.

const fractional offset - this is the number of bits between the origin of the variable and the previous word boundary. If the bit offset is variable this value is zero.

Temporary Nodes

A temporary node represents the output of some operator. Unless their storage class is automatic they will be allocated storage at the discretion of the code generator. Their values are assumed to be destroyed upon completion of the execution of a statement. Automatic temporaries are identical to compiler created automatic variables and their values are assumed to be preserved throughout the execution of the block.

Temporary nodes, other than automatic, are shared by more than one operator. This sharing is done for efficiency reasons and does not affect the logical meaning of a temporary.

Definition of the Temporary Node

```
dcl  1  temporary_node      based(p),
     2  node_type           fixed bin(15),
     2  data_type           fixed bin(15),
     2  class               fixed bin(15),
     2  class_offset        fixed bin(31),
     2  size                fixed bin(31),
     2  scale               fixed bin(15),
     2  boundary            fixed bin(15),
     2  const_storage       fixed bin(15),
     2  variable_size       ptr,
     2  descriptor          ptr,
     2  next                ptr;
```

node type - is a constant which identifies this node as a temporary node. See Appendix 1.

data type - is a constant which specifies the data attributes of the temporary. The possible values and meanings are given in Appendix 4.

class - is a constant which specifies the storage class of the temporary. The values and meanings are:

a) 1 automatic - allocate by block prologue
b) 10 temp - allocated and re-used for each statement

class offset - the location of this temporary as determined by the storage allocator or code generator.

size - this is the precision of arithmetic temporaries and length of string temporaries.

scale - is the scale of fixed point temporaries.

boundary - indicates the starting address requirements of the temporary. The values and meanings are:

1	any bit
2	character boundary
3	word boundary
4	even word boundary
5	address 0 mod 4
6	address 0 mod 8
7	address 0 mod 16

const storage - the amount of storage, measured in words, required by this value. If the size of the temporary is variable this field is zero.

variable size - an expression which describes the length, measured in bits or characters, of this temporary value. If the size is constant this field is null.

descriptor - a pointer to the argument descriptor image created for this value by the storage allocator. This pointer is null if no descriptor is required.

next - points to the next temporary node in the pool of temporary nodes for a given block.

Parameter Node

The parameter node represents two distinct objects in the internal representation.

a) Parameter nodes whose number is zero represent a locator variable. During semantic translation the locator variable (p) which qualifies a based reference (p -> x) replaces all occurrences of parameter nodes within the size and offset expressions of the qualified reference (x). This mechanism implements the refer option and allows the fractional offset expression of based or parameter unaligned strings to reference the bit value of their qualifying locator variable.

b) Parameter nodes whose number is -1 are used to describe a "register" of the code generator. This register is assumed to contain the word offset derived from evaluation of a fractional offset expression. The fractional offset expression has a two part output - a bit and a word part. The word offset expression contains a parameter node which serves as a reference to the word part of the evaluated fractional offset.

Definition of a Parameter Node

```
dcl  1  parameter        based(p),
     2  node_type        fixed bin(15),
     2  number           fixed bin(15);
```

node type - is a constant indicating this is a parameter node. See Appendix 1.

number - an identification code whose use is described above.

APPENDIX 1

NODE TYPES

VALUE	MEANING
1	block node
2	statement node
3	operator node
4	operand node
5	temporary node
6	symbol-table node
7	temporary-attribute block
8	data-attribute block
9	condition-attribute block
10	file-attribute block
11	initial-link node
12	entry-attribute block
13	token-table node
14	reference node
15	string-reference node
16	constant node
17	structure-size node
18	local-offset node
19	array-attribute block
20	label-attribute block
21	parameter node
22	descriptor block
23	group node
24	rand node
25	address node
26	cv-node

APPENDIX 2

STATEMENT TYPES

VALUE	STATEMENT
0	unknown
1	allocate
2	assignment
3	begin
4	call
5	close
6	declare
7	delay
8	delete
9	display
10	do
12	end
13	entry
14	exit
15	format
16	free
17	get
18	goto
19	if
20	locate
21	null
22	on
23	open
24	procedure
25	put
26	read
27	return
28	revert
29	rewrite
30	signal
31	stop
33	unlock
34	wait
35	write

APPENDIX 3

VALUES FOR OP CODES

(SEE BZ.8.11 FOR FULL DOCUMENTATION)

VALUES	NAME
1	add
2	sub
3	mult
4	div
5	exp
6	negate
10	and-bits
11	or-bits
12	not-bits
13	single-cat
20	assign
21	assign-by-name
22	assign-with-size-ck
30	less-than
31	greater-than
32	equal
33	not-equal
34	less-or-equal
35	greater-or-equal
40	jump
41	jump-true
42	jump-false
43	jump-if-lt
44	jump-if-gt
45	jump-if-eq
46	jump-if-ne
47	jump-if-le
48	jump-if-ge
60	std-call
61	validate call
70	entry
71	std-return
72	func-return

APPENDIX 3 (cont.)

VALUES	NAME
81	dot
82	pointer
90	join
91	allot-based
92	free-based
93	ex-prologue
96	copy-words
97	onloc-name
98	enable-on
99	revert-on
102	signal-on
100	dope-copy
101	dope-fill

APPENDIX 4

DATA TYPES

VALUES	MEANING
1	real fixed binary single
2	real fixed binary double
3	real fixed decimal single
4	real fixed decimal double
5	real float binary single
6	real float binary double
7	real float decimal single
8	real float decimal double
21	complex fixed binary single
22	complex fixed binary double
23	complex fixed decimal single
24	complex fixed decimal double
25	complex float binary single
26	complex float binary double
27	complex float decimal single
28	complex float decimal double
31	non-varying character string
32	varying character string
41	non-varying bit string
42	varying bit string
51	label variable, local values only
52	label variable, any values permitted
60	pointer variable
61	offset variable
72	entry variable
80	structure
81	structure created for varying strings
82	cell
83	file name
84	area