SECTION I

INTRODUCTION

AN OVERVIEW OF THE COMPILER

The PL/1 compiler translates a source program written in the PL/1 language into an equivalent Multics standard object segment. This compiler represents an implementation of the PL/1 language as defined in the PL/1 Language Manual (Order No. AG94). The entire compiler is written in the same language, and therefore, is self reproduceable.

The compiler is organized into five phases: Syntactic Translation, Declaration Processing, Semantic Translation, Optimization, and Code Generation. Each phase is a set of procedures grouped together to perform a major logical function.

The internal representation of the program being compiled serves as the interface between phases of the compiler. To have a thorough understanding of how the compiler works requires an in depth knowledge of the internal representation scheme adopted by this implementation.

This pl1 command program is the interface between the user and the compiler. It is also the interface between the compiler and the Multics operating system. All calls to Multics system subroutines are made in this command program.

NAME: pl1

Function:

1. It initializes the various static variables of the compiler.

2. It processes all the options to the command:
   check
   list
   time
   source
   brief
   symbols
   assembly
   severity
   cpdcls
   debug
   optimize
   table
   brief_table
   parse
   profile
   link

3. It gets the pointer to the source segment.

4. It makes the object segment and the listing segment if required.

5. It calls the multi-segment-file manager if the listing requires more than one segment.

6. It sets up a default handler.

7. It sets up a cleanup handler, in case a compiler should abort in the middle of a compilation.

8. It invokes the various phases of the compiler:
   parse
   semantic_translator
   optimizer
   code_generator
   prepare_symbol_map_

Entry:

Usage:

pl1 pathname -control_arg1 ... -control_argn

1. pathname — is the path name of a PL/1 source segment to be translated by the PL/1 compiler. If the source segment does not have a suffix of .pl1, then one is assumed.

2. control_argi — can be chosen from a list of options. Refer to the Multics Programmers' Manual 'pl1' command for details.

Entry:

pl1$times, v2pl1$times

This entry, when called after a compilation, will print out a table giving the time, the number of page faults, and the amount of storage used by each phase of the compiler. The phases include setup, parse, semantics, optimizer, code generator, and the lister.

Usage:

pl1$times

Entry:

pl1$epilogue, v2pl1$epilogue
pl1$clean_up, v2pl1$clean_up

These entries are called after an aborted compilation, so that cleanup jobs will be done.

Usage:

```
pl1$epilogue
pl1$clean_up
```

Entry:

```
pl1$blast,  v2pl1$blast
```

This entry is called to turn on the blast message, to turn off the blast message, or to rewrite the blast message.

If the blast message is on, the blast message will be given at the start of the first compilation in the process.

Usage:

```
pl1$blast -on
pl1$blast -off
pl1$blast -set blast_message
```

Internal Procedures:

```
none
```

External Variables:

```
cg_static_$debug
cg_static_$stop_id
cg_static_$support
error_table_$badopt
error_table_$entlong
error_table_$noarg
error_table_$translation_failed
error_table_$zero_length_seg
pl1_blast_$blast_message
pl1_blast_$blast_on
pl1_blast_$blast_time
```

pl1_stat_$abort_label
pl1_stat_$brief_error_mode
pl1_stat_$char_pos
pl1_stat_$compiler_name
pl1_stat_$constant_list
pl1_stat_$debug_semant
pl1_stat_$dummy_block
pl1_stat_$error_messages
pl1_stat_$error_width
pl1_stat_$generate_symtab
pl1_stat_$greatest_severity
pl1_stat_$index
pl1_stat_$line_count
pl1_stat_$list_ptr
pl1_stat_$listing_on
pl1_stat_$max_list_size
pl1_stat_$max_node_type
pl1_stat_$node_name
pl1_stat_$node_size
pl1_stat_$node_uses
pl1_stat_$ok_list
pl1_stat_$optimize
pl1_stat_$options
pl1_stat_$pathname
pl1_stat_$phase
pl1_stat_$print_cp_dcl
pl1_stat_$profile_length
pl1_stat_$root
pl1_stat_$seg_name
pl1_stat_$severity_plateau
pl1_stat_$source_index
pl1_stat_$source_ptr
pl1_stat_$stop_id
pl1_stat_$table
pl1_stat_$temporary_list
pl1_stat_$tree_vec_index
pl1_stat_$user_id
pl1_stat_$validate_proc
tree_$
v2pl1$
xeq_tree_$

Internal Static Variables:

none

Programs Called:

```
bindec
clock_
code_gen_
code_gen_$return_bit_count
com_err_
cu_$arg_ptr
cv_dec_
date_time_
db
default_handler_$set
error_$finish
establish_cleanup_proc_
expand_path_
get_group_id_
get_wdir_
hcs_$get_max_length_seg
hcs_$get_usage_values
hcs_$initiate_count
hcs_$make_ptr
hcs_$truncate_seg
hmu
ioa_
ioa_$nnl
ios_$changemode
lex$scan_token_table
lex$terminate_source
msf_manager_$get_ptr
optimizer
parse
pl1_print$non_varying
pl1_print$non_varying_nl
pl1_print$varying_nl
pl1_signal_catcher
pl1_symbol_print
prepare_symbol_map_
prepare_symbol_table
record_command_usage_$enter
record_command_usage_$exit
revert_cleanup_proc_
semantic_translator
tree_manager$init
tree_manager$truncate
tssi_$clean_up_file
tssi_$clean_up_segment
tssi_$finish_file
tssi_$finish_segment
tssi_$get_file
tssi_$get_segment
v2pl1$epilogue
```

Include Files used:

none

Errors Diagnosed:

Errors diagnosed by this program are not errors in the source program, but rather errors found in the use of the command pl1.

r

SECTION II

INTERNAL REPRESENTATION

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 action 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/I source operators. References are represented by reference nodes which point 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.

Except for some fields of the reference node used only by the code generator, this internal representation is machine independent in that it does not reflect the instruction set, the addressing properties, or the register arrangement of the target machine (645 or 6180). All phases of the compiler, except the code generator, are also machine independent since they deal only with this machine independent internal representation. Figure 2-1 shows the internal representation of a simple program.

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. See Figure 2-2.

Format:

```
     dcl  1 block                      based aligned,
          2 node_type                  bit(9) unaligned,
          2 source_id                  structure unaligned,
            3 file_number              bit(8),
            3 line_number              bit(14),
            3 statement_number         bit(5),
          2 father                     ptr unaligned,
```

```
          2 brother                    ptr unaligned,
          2 son                        ptr unaligned,
          2 declaration                ptr unaligned,
          2 end_declaration            ptr unaligned,
          2 default                    ptr unaligned,
          2 end_default                ptr unaligned,
          2 context                    ptr unaligned,
          2 prologue                   ptr unaligned,
          2 end_prologue               ptr unaligned,
          2 main                       ptr unaligned,
          2 end_main                   ptr unaligned,
          2 return_values              ptr unaligned,
          2 return_count               ptr unaligned,
          2 plio_ps                    ptr unaligned,
          2 plio_fa                    ptr unaligned,
          2 plio_ffsb                  ptr unaligned,
          2 plio_ssl                   ptr unaligned,
          2 plio_fab2                  ptr unaligned,
          2 block_type                 bit(9) unaligned,
          2 prefix                     bit(12) unaligned,
          2 like_attribute             bit(1) unaligned,
          2 no_stack                   bit(1) unaligned,
          2 get_data                   bit(1) unaligned,
          2 flush_at_call              bit(1) unaligned,
          2 processed                  bit(1) unaligned,
          2 skip                       bit(1) unaligned,
          2 number                     fixed bin(8) unaligned,
          2 free_temps                 dimension(3) ptr,
          2 temp_list                  ptr,
          2 entry_list                 ptr,
          2 o_and_s                    ptr,
          2 max_display_steps          fixed(17),
          2 display_vector             fixed(17),
          2 number_of_entries          fixed(17),
          2 level                      fixed(17),
          2 last_auto_loc              fixed(17),
          2 symbol_block               fixed(17),
          2 entry_info                 fixed(18),
          2 enter                      structure unaligned,
            3 start                    fixed(17),
            3 end                      fixed(17),
          2 leave                      structure unaligned,
            3 start                    fixed(17),
            3 end                      fixed(17);
```

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

source_id - (treated as a triple of numbers)

file_number - 0 for main source file, and indexes include files in sequential order of inclusion. Any include file may be included more than once; each occurrence will have a distinct file_number.

line_number - line number within source file (see file_number) of line on which statement begins.

statement_number - 1 + number of statements that finish ahead of the current statement on the line on which the current statement begins.

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 that has the same father.

son - points to the first contained block.

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

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

default - points to a uni-directional chain of default nodes each representing a default statement in this block. The default nodes are used only during declaration processing and are of no interest to the code generator.

end_default - points to the last default node 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 statement sequence.

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

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.

return_values - points to a chain of list nodes each of which points to a symbol node representing a unique kind of value returned by the return statements of this procedure.

return_count - if this procedure returns more than one kind of value, this points to a declaration of an integer declared in the block which is used to determine what kind of value is to be returned. This information as well as the list of return values is not used by the code generator; it is created and used by the semantic translator.

plio_ps - if non-null, points to the symbol-node for PS, the storage block used in I/O statements. If non-null, the code generator will compile code in the block prologue to set PS.stack_frame_p, stack.psp, and, if there is to be a runtime symbol table, PS.ST_top_p and PS.ST_block_p.

plio_fa - if non-null, points to the symbol-node for the format-area, used by edit-directed get- and put-statements. If non-null, the code generator will

compile code in the block prologue to set PS.format_area_p to its address.

plio_ffsb - if non-null, points to the symbol-node for "fake FSB", a pseudo-file-control-block used for get- and put-statements with string option.

plio_ssl - if non-null, points to the symbol-node for ss_list, storage used for the put data statement. If non-null, the code generator will cause its address to be stored in PS.ss_list_p during block prologue.

plio_fab2 - points to the symbol-node for FAB2, storage used by the open statement to record file options, linesize, and pagesize.

block_type - defines the kind of block this node represents. The codes used in this field are given in the appendix.

prefix - the condition prefix of the block. See "Statement Nodes" on page 2- for a definition of each bit.

like_attribute - indicates that some declaration occurs in this block with a like attribute.

no_stack - this block shares its stack frame with its containing or brother block and can be called with a non-recursive call.

get_data - used by get-data to indicate that a full runtime symbol table is required for this block.

flush_at_call - indicates that some son of this block is assigned to an external static entry variable. Hence, any call may invoke it and change any automatic variable in this block.

processed - used and set by the code generator only.

skip - a filler.

number - this field is used to sequentially number all blocks. it is used by the part of the semantic translator which determines the set of blocks requiring stack frames.

free_temps - points to lists of free temporaries. (used and set only by the code generator).

temp_list - points to a list of allocated temporaries. (set and used only by the code generator).

entry_list - points to a list of all entry statements in this block.

o_and_s - used by the code generator to keep track of offset and size expressions.

max_display_steps - indicates the maximum number of environment pointers needed to reference automatic variables or label constants, etc., declared in outer blocks.

display_vector - used by the code generator to remember the location of the environment (display) pointers. (not used or set outside the code generator).

number_of_entries - the number of procedure and entry statements in this block.

level - set and used only by the code generator. "level" is the nesting level of this block in terms of stack-frame nesting depth. The "level" of a quick block is thus equal to the "level" of the block in which its automatic storage has been placed. The level of the root block is 0. The level of the external procedure block is 1.

last_auto_loc - used by the storage allocator as a location counter for allocating automatic storage.

symbol_block - holds the offset within the run-time symbol table of the runtime block node that corresponds to this block; used and set only by the code generator.

entry_info - used and set only by the code generator.

enter - used and set only by the code generator.

leave - used and set only by the code generator.

REPRESENTATION_OF_DECLARATIONS

Two data structures 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.

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 aligned,
          2 node_type         bit(9) unaligned,
          2 type              bit(9) unaligned,
          2 loc               bit(18) unaligned,
          2 declaration       ptr unaligned,
          2 next              ptr unaligned,
          2 size              fixed(9),
          2 string            char(n refer(token.size));
```

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.

loc - Position in runtime symbol table of this token. Used and set by the code generator only.

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.

size - is the length of the token, "token.string".

string - is the character string representation of the token. In the case of a character-string token, "string" is the string value. In the case of a bit-string token, "string" is the character-string obtained from the bit string by replacing "1"b with "1", "0"b with "0", and adding a final "b".

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.

LABEL NODES

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

have labels, but these are removed from the statement by io_statement_parse and changed into symbols with the initial pointer pointing at the format statement. The fields of the label node generally match the corresponding fields of the symbol node.

Format:

```
     dcl  1 label                       based aligned,
          2 node_type                   bit(9) unaligned,
          2 source_id                   structure unaligned,
            3 file_number               bit(8),
            3 line_number               bit(14),
            3 statement_number          bit(5),
          2 location                    fixed(17) unaligned,
          2 allocated                   bit(1) unaligned,
          2 dcl_type                    bit(3) unaligned,
          2 reserved                    bit(29) unaligned,
          2 array                       bit(1) unaligned,
          2 used_as_format              bit(1) unaligned,
          2 used_in_goto                bit(1) unaligned,
          2 symbol_table                bit(18) unaligned,
          2 low_bound                   fixed(17) unaligned,
          2 high_bound                  fixed(17) unaligned,
          2 block_node                  ptr unaligned,
          2 token                       ptr unaligned,
          2 next                        ptr unaligned,
          2 multi_use                   ptr unaligned,
          2 cross_reference             ptr unaligned,
          2 statement                   ptr unaligned;
```

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

source_id - describes the statement on which this label appeared. For label arrays it identifies the first statement on which one of the array elements appeared. (For further detail, see description in "Block Structure" on page 2-9.)

location - the address assigned to this label.

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

dcl_type - describes the manner in which the label was declared. The declare_types include file listed in the appendix defines the values used in this field.

array - identifies this as a constant label array.

used_as_format - used by FORTRAN to distinguish labels and format identifiers.

used_in_goto - used by FORTRAN to distinguish labels and format identifiers.

symbol_table - used and set by the code generator only. Records the location in the runtime symbol table of the runtime label node corresponding to this label.

low_bound - the observed lower bound of the array.

high_bound - the observed high bound of the array.

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 of 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 as a label prefix.

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 aligned,
          2 node_type                  bit(9) unaligned,
          2 source_id                  structure unaligned,
            3 file_number              bit(8),
            3 line_number              bit(14),
            3 statement_number         bit(5),
          2 location                   fixed(17) unaligned,
          2 allocated                  bit(1) unaligned,
          2 dcl_type                   bit(3) unaligned,
          2 reserved                   bit(6) unaligned,
          2 pix                        structure unaligned,
            3 pic_fixed                bit(1) unaligned,
            3 pic_float                bit(1) unaligned,
            3 pic_char                 bit(1) unaligned,
            3 pic_scale                fixed(7) unaligned,
            3 pic_size                 fixed(7) unaligned,
          2 level                      fixed(8) unaligned,
          2 boundary                   fixed(3) unaligned,
          2 size_units                 fixed(3) unaligned,
          2 scale                      fixed(7) unaligned,
          2 runtime                    bit(18) unaligned,
          2 runtime_offset             bit(18) unaligned,
          2 block_node                 ptr unaligned,
          2 token                      ptr unaligned,
          2 next                       ptr unaligned,
          2 multi_use                  ptr unaligned,
          2 cross_references           ptr unaligned,
          2 initial                    ptr unaligned,
```

```
    2 array                     ptr unaligned,
    2 descriptor                ptr unaligned,
    2 equivalence               ptr unaligned,
    2 reference                 ptr unaligned,
    2 general                   ptr unaligned,
    2 father                    ptr unaligned,
    2 brother                   ptr unaligned,
    2 son                       ptr unaligned,
    2 word_size                 ptr unaligned,
    2 bit_size                  ptr unaligned,
    2 dcl_size                  ptr unaligned,
    2 symtab_size               ptr unaligned,
    2 c_word_size               fixed(24),
    2 c_bit_size                fixed(24),
    2 c_dcl_size                fixed(24),


    2 attributes                structure aligned,
      3 data_type               structure unaligned,
        4 structure             bit(1) ,
        4 fixed                 bit(1),
        4 float                 bit(1),
        4 bit                   bit(1),
        4 char                  bit(1),
        4 ptr                   bit(1),
        4 offset                bit(1),
        4 area                  bit(1),
        4 label                 bit(1),
        4 entry                 bit(1),
        4 file                  bit(1),
        4 arg_descriptor        bit(1),
        4 storage_block         bit(1),
        4 lock                  bit(1),
        4 condition             bit(1),
        4 format                bit(1),
        4 builtin               bit(1),
        4 generic               bit(1),
        4 picture               bit(1),


      3 misc_attributes         structure unaligned,
        4 dimensioned           bit(1),
        4 initialed             bit(1),
        4 aligned               bit(1),
        4 unaligned             bit(1),
        4 connected             bit(1),
        4 precision             bit(1),
        4 varying               bit(1),
        4 local                 bit(1),
```

```
    4 decimal               bit(1),
    4 binary                bit(1),
    4 real                  bit(1),
    4 complex               bit(1),
    4 variable              bit(1),
    4 reducible             bit(1),
    4 irreducible           bit(1),
    4 returns               bit(1),
    4 position              bit(1),
    4 internal              bit(1),
    4 external              bit(1),
    4 like                  bit(1),
    4 member                bit(1),

  3 storage_class           structure unaligned,
    4 auto                  bit(1),
    4 based                 bit(1),
    4 static                bit(1),
    4 controlled            bit(1),
    4 defined               bit(1),
    4 parameter             bit(1),
    4 param_desc            bit(1),
    4 constant              bit(1),
    4 temporary             bit(1),
    4 return_value          bit(1),

  3 file_attributes         structure unaligned,
    4 print                 bit(1),
    4 input                 bit(1),
    4 output                bit(1),
    4 update                bit(1),
    4 stream                bit(1),
    4 reserved_1            bit(1),
    4 record                bit(1),
    4 sequential            bit(1),
    4 direct                bit(1),
    4 interactive           bit(1),
    4 reserved_2            bit(1),
    4 forwards              bit(1),
    4 backwards             bit(1),
    4 keyed                 bit(1),
    4 reserved_3            bit(1),
    4 environment           bit(1),

  3 compiler_developed      structure unaligned,
    4 abnormal              bit(1),
```

```
            4 packed                  bit(1),
            4 passed_as_arg           bit(1),
            4 allocate                bit(1),
            4 set                     bit(1),
            4 exp_extents             bit(1),
            4 refer_extents           bit(1),
            4 star_extents            bit(1),
            4 variable_arg_list       bit(1),
            4 non_varying             bit(1),
            4 isub                    bit(1),
            4 put_in_symtab           bit(1),
            4 contiguous              bit(1),
            4 put_data                bit(1),
            4 overlayed               bit(1),
            4 error                   bit(1),
            4 symtab_processed        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. (For further detail, see description in "Block Structure" on page 2-9.)

location - the address given to this item by the storage allocator. If this item is a parameter, "location" is the position of the parameter in first entry statement in which it appears (i.e., first entry statement processed by declare). (See "Parameter" on page 2-). If this item is controlled, location is the offset of a 3-pointer structure serving to identify the current generation of the variable. (See "Controlled" on page 2-).

allocated - indicates that storage has been allocated for this variable. Set in the case of a parameter appearing in more than one parameter position (see "Parameter" on page 2-).

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.

pix - the fields of pix record facts about a picture deduced from inspection of the picture.

pic_fixed - set if the picture is a numeric picture, not floating.

pic_float - set if the picture is numeric, floating.

pic_char - set if the picture is a character picture.

pic_scale - the scale of the (fixed) associated variable: the number of digits after the "v" in the picture, if one appears, (or zero), less the value of the picture's scale factor, if any. If the symbol is a generic arg_selector for an arithmetic argument, pic_scale is used to hold the upper limit of the scale.

pic_size - the precision for a numeric picture, the length for a character picture. If the symbol is a generic arg_selector for an arithmetic argument, pic_size is used to hold the upper limit of the precision.

level - the level number adjusted so that the level number of a member is one greater than its containing structure. Non-structure level-one variables have a level number of zero.

boundary - the storage boundary required by this item. The valid codes are given in the appendix.

size_units - used and set by the code generator only. used to keep track of the units in which the item's size is expressed.

scale - the arithmetic scale factor. If the symbol is a generic arg_selector for an arithmetic argument, scale is used to hold the lower limit of the scale.

runtime - used and set by the code generator only. Holds offset within runtime symbol table of the runtime symbol node corresponding to this symbol node.

runtime_offset - NOT USED.

block_node - points to the block_node that 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 - 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 an initialized variable this points to a list node or tree of list nodes which represents the initial attribute. If this item is a literal constant this points to the binary representation of the constant's value. If this is a level-1 "defined" variable with position attribute this points to the position expression template. In the case of a format constant, "initial" points to the format statement node. (See X.X.X.X.X)

array - points to an array node which describes the number of dimensions, the bounds, and the multipliers of this array. See "Array and Bound Nodes" on page 2-.

descriptor - points to a reference_node which points to a symbol node whose type is arg_descriptor and whose storage class is automatic, constant, controlled, temporary, 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 - in the prologue, in the allocate statement for a controlled variable, or immediately before the statement containing the call. If this is an array, the descriptor describes the entire array and the element descriptor is found in the array node.

equivalence - points to the parse of the reference given in the defined attribute or to the base constant of a group of equivalenced constants. (See "Storage Classes" on page 2-.)

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 entire 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. If this is a member of a packed structure neither this field nor its constant counterpart have any meaning, although they may contain non-empty values.

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 are the total array size, not the element size).

dcl_size - points to an expression giving the declared size of areas or the declared length of strings. If the data-type is entry this field points to the symbol node that describes the return value of the entry. In the case of a controlled variable, dcl_size points to an expression which references the runtime descriptor of the controlled variable.

symtab_size - in the case of controlled variables, set by declare_descriptor to the original (parsed only) contents of dcl_size if this is not constant. Points to an expression giving the declared size of the item. This expression is obtained by semantically translating the dcl_size expression. This pointer is null if a runtime symbol table entry is not required.

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. If the symbol is a generic arg_selector for an arithmetic argument, c_dcl_size is used to hold the lower limit of the arithmetic precision. In the case of a pictured item, the length of the pictured string.

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 - the value of this variable may change without any explicit indication in this program. A variable is abnormal if:

1. it is based, parameter, external, defined or the base of a defined variable;
2. it is used in an addr built-in function or appears in the string option of a put statement, an into or set option of a read statement, or a set option of a locate statement;
3. it is a member of an abnormal structure or is a structure containing abnormal values;
4. it is passed as an argument by reference and is static or controlled.

packed - this value is:

1. An unaligned aggregate of packed data;
2. unaligned arithmetic data;

3. unaligned non-varying string data;

4. unaligned pointer data.

passed_as_arg - set in semantics, tested by the code generator; indicates that spare bits may have been written into by the procedure called. Also set for an argument of the unspec pseudo-variable. See padded_ref in "Reference Nodes" on page 2-.

allocate - indicates that the item has been referenced; indicates that any required allocation of space may not be ommitted; inspected during preparation of the listing.

set - this item appears on the left side of an assignment, in a get list, a set() option, a keyto() option, the string() option of a put-statement, suitably as an argument to a pseudovariable operator, in an in() option, a read into() statement, or as an argument passed by reference. Defined items, and items which are the bases of defined items, are abnormal (see above) but do not inherit each other's set attribute.

exp_extents - this item has non-constant extents.

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

star_extents - this item has asterisk extents.

variable_arg_list - represents the source program construction "options(variable)".

non_varying - indicates that the item is a nonvarying character or bit string.

isub - indicates that the item is isub-defined.

put_in_symtab - this declaration must be placed in the run-time symbol table.

contiguous - used and set only by the code generator. Indicates of a string array that no element crosses a word boundary.

put_data - NOT USED

overlayed - indicates that the item is a string overlayed item.

error - would flag an inconsistent declaration: NOT USED.

symtab_processed - flag used by prepare_symbol_table. Indicates that the semantic processing needed to generate the runtime symbol table entry has already been done.

ARRAY AND BOUND NODES

The array node and its associated chain 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.

Array Nodes

Format:

```
     dcl  1 array                      based aligned,
          2 node_type                  bit(9) unaligned,
          2 reserved                   bit(34) unaligned,
          2 number_of_dimensions       fixed(7) unaligned,
          2 own_number_of_dimensions   fixed(7) unaligned,
          2 element_boundary           fixed(3) unaligned,
          2 size_units                 fixed(3) unaligned,
          2 offset_units               fixed(3) unaligned,
          2 interleaved                bit(1) unaligned,
```

```
          2 c_element_size              fixed(24),
          2 c_element_size_bits         fixed(24),
          2 c_virtual_origin            fixed(24),
          2 element_size                ptr unaligned,
          2 element_size_bits           ptr unaligned,
          2 virtual_origin              ptr unaligned,
          2 symtab_virtual_origin       ptr unaligned,
          2 symtab_element_size         ptr unaligned,
          2 bounds                      ptr unaligned,
          2 element_descriptor          ptr unaligned;
```

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.

own_number_of_dimensions - The number of dimensions declared on this item.

element_boundary - the storage boundary required by the elements of this array.

size_units - used and set by the code generator only. The units in which the element_size is expressed. See array.element_size and symbol.size_units.

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.

interleaved - This array is interleaved.

c_element_size - constant element size in words (rounded if necessary). See "size_units" above.

c_element_size_bits - constant element size in bits.

c_virtual_origin - if "virtual_origin" is null, the constant virtual origin: a virtual origin is the value (constant or variable) that must be added to the sum of the products of an item's subscripts with its multipliers to yield a correct offset relative to the beginning of the containing level-1 aggregate.

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 - if non-null, points to an expression for the virtual origin (see c_virtual_origin).

symtab_virtual_origin - points to an expression giving the virtual origin of the array. This expression is obtained by semantic translation of the "virtual_origin" expression. This pointer is null if a runtime symbol table entry is not required.

symtab_element_size - see "symtab_virtual_origin": replace "virtual_origin" by "element_size".

bounds - points to a uni-directional chain of bounds nodes each of which gives a lower bound, an upper bound, and a multiplier. These multipliers are measured in the units indicated by offset_units. The descriptor bounds are measured in bits if the item is packed, otherwise they are measured 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.

Bound Nodes

Format:

```
     dcl  1 bound                       based aligned,
          2 node_type                   bit(9),
          2 c_lower                     fixed(24),
          2 c_upper                     fixed(24),
          2 c_multiplier                fixed(24),
          2 c_desc_multiplier           fixed(24),
          2 lower                       ptr unaligned,
          2 upper                       ptr unaligned,
          2 multiplier                  ptr unaligned,
          2 desc_multiplier             ptr unaligned,
          2 symtab_lower                ptr unaligned,
          2 symtab_upper                ptr unaligned,
          2 symtab_multiplier           ptr unaligned,
          2 next                        ptr unaligned;
```

node_type - has a value of "000001001"b which identifies this node as a bound node.

c_lower - constant lower bound if "lower" is null. Used in bounds checking, to compute the range (upper-lower+1) of this dimension, and to compute multipliers for contained bound nodes.

c_upper - upper bound if "upper" is null. See "c_lower".

c_multiplier - multiplier for computing offset from subscript if "multiplier" is null. The multiplier for a bound with N contained bounds is the N+1-fold product of the ranges (upper-lower+1) of those bounds with the element size of the terminal, unsubscripted element.

c_desc_multiplier - constant descriptor multiplier if "desc_multiplier" is null. The multipliers in array descriptors, desc_multiplier's, serve the same purpose

as the more generally used multipliers, but follow different rules due to the necessity to continue the practice of EPL. The units in which desc_multiplier is expressed is bits in the case of a packed array and words in the case of an unpacked array.

lower - points to lower-bound expression tree if non-null. See "c_lower". In the case of a controlled array, points to an expression which references the runtime descriptor of the controlled variable.

upper - points to upper-bound expression tree if non-null. See "c_upper".

multiplier - points to multiplier expression tree if non-null. See "c_multiplier".

desc_multiplier - points to descriptor-multiplier expression tree if non-null. See "c_desc_multiplier".

symtab_lower - set by declare_descriptor for controlled arrays: contains the original (parsed only) tree for bound.lower if the lower bound is not constant. Otherwise used and set by the code generator only. points to an expression giving the lower bound of this dimension of the array. The expression is obtained by semantic translation of the "lower" expression. This pointer will be null if a runtime symbol table entry is not required.

symtab_upper - set by declare_descriptor in the case of controlled arrays: contains the original (parsed only) bound.upper if the upper bound is not constant. See "symtab_lower"; replace "lower" with "upper".

symtab_multiplier - set by get_array_size in the case of controlled arrays: contains the original (parsed only) bound.multiplier if the multiplier is not constant. See "symtab_lower"; replace "lower" with "multiplier".

next - if non-null, points to the immediately containing bound node. Note well that the chain of bound nodes, like most lists relating to subscripts, is kept in reversed order. Thus, "next" for a sub-array points to the bound node for the containing array.

INITIAL ATTRIBUTES

The initial attribute of PL/I 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 in Figure 2-5.

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.

STORAGE CLASSES

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

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 and the cross_references field in the symbol node is non-null (indicating one or more references to the variable). 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. But, the block node contains a number, max_display_steps, which is the maximum number of display (environment) pointers needed by the block; it is obtained from the level numbers of the block in which the reference occurs and the block in which the variable is declared.

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.

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 by the code generator.

Internal static values are initialized by the storage allocator and do not result in the creation of initialization code in the object program.

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 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. If the name contains a $, the link also includes initialization or dynamic allocation information, but the variable is allocated in the segment "name$". If the segment does not exist, it is created in the process directory.

Controlled

Controlled storage is explicitly allocated by the program at runtime. For internal controlled storage, the code generator allocates a 3-pointer block in internal static whose offset is contained in symbol.location. The first pointer points to the most recent generation of storage for the variable, the second points to the most recent generation of storage for the descriptor if the variable has expression extents, and the third points to a 3-pointer block representing the previous generation of storage. For external controlled variables, symbol.location is the offset of a link to a similar 3-pointer block in external static.

Defined

No storage is allocated for the value. The code generator develops addresses for defined references by combining the offset of the defined reference with the offset of the base reference. The qualifier field of the defined reference node points to the base-reference. The locator qualification of the base is used as

the locator qualification of the defined reference.

Parameter

Two methods are used to access a parameter and its descriptor: A reference to a parameter is always effectively qualified by a param_ptr operator. If a parameter appears in the same position within all entries in which it appears, the param_ptr operator will appear explicitly in each reference to it. Otherwise, the parameter reference is qualified by a unique automatic pointer whose value is set (via a suitable param_ptr operator) in the entry sequence of each entry in which the parameter appears.

(In the parameter's symbol node, the "location" field gives the position of the parameter within the first entry statement processed by declare. If declare finds that the parameter appears in any other position in any other entry statement, declare sets the "allocated" bit in the parameter's symbol. This all occurs in the processing of declarations in the block containing the entry labels, that is, the block father to the block containing the entry statements.

Thus, when declare processes the parameters themselves, it sets the "qualifier" field to a unique automatic pointer if "allocated" is set, or to a param_ptr expression if it is not. When the entry statements themselves are processed, the "allocated" bit may thus be inspected and suitable preparatory code inserted, if required. Refer to "Call, Save, and Return Operators" on page 2-.

Parameter-Descriptor

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.

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 compiler generated unique names and refer to the token table entry for their name just like other declarations. 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", and are linked together through the "multi_use" pointer of the symbol node. 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. The allocate bit of the base constant is on, and the allocate bits of all other equivalenced constants is off. See Figure 2-3.

Temporary Values

The compiler has need of a means to represent values which need not, and do not, correspond to generations of storage at run time. Temporaries fill this need. When a temporary generation of storage, as distinct from a temporary value, is required, an automatic variable must be declared.

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" and are linked together through the "multi_use" pointer of the symbol nodes. 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 except at the moment of evaluation in the program have a storage class of "temporary", and the "shared" bit is on in the reference node for the temporary. A shared temporary is used solely to indicate the output attributes of an operator. 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 zero "shared" bit. The "ref_count" field of the reference node indicates the number of references to this value.

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

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.

Statement_Nodes

Each statement is represented by a statement node.

Format:

```
     dcl  1 statement                  based aligned,
          2 node_type                  bit(9) unaligned,
          2 source_id                  structure unaligned,
            3 file_number              bit(8),
            3 line_number              bit(14),
            3 statement_number         bit(5),
          2 next                       ptr unaligned,
          2 back                       ptr unaligned,
          2 root                       ptr unaligned,
          2 labels                     ptr unaligned,
          2 reference_list             ptr unaligned,
          2 state_list                 ptr unaligned,
          2 reference_count            fixed(17) unaligned,
          2 ref_count_copy             fixed(17) unaligned,
          2 object                     structure unaligned,
            3 start                    fixed(17),
            3 finish                   fixed(17),
          2 source                     structure unaligned,
            3 segment                  fixed(11),
            3 start                    fixed(23),
            3 length                   fixed(11),
          2 prefix                     bit(12) unaligned,
          2 optimized                  bit(1) unaligned,
          2 free_temps                 bit(1) unaligned,
          2 LHS_in_RHS                 bit(1) unaligned,
          2 statement_type             bit(9) unaligned,
          2 processed                  bit(1) unaligned,
          2 put_in_profile             bit(1) unaligned,
          2 generated                  bit(1) unaligned;
```

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

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, the field will be zero if no original exists. (For further detail, see description in "Block Structure" on page 2-9.)

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.

state_list - used by the code generator. When the code generator processes a jump operator which references a statement not yet compiled by the code generator, it attaches a copy of the current machine state record to the state_list of the statement node referenced by the jump. If all references to a statement have been processed, the machine state available at the statement is the intersection of all of the machine states on the state_list.

reference_count - contains a count of all references to any of the labels that appeared in the label prefix of this statement. A labelled statement with no other references to its label has a count of one.

ref_count_copy - a copy of reference_count used by the optimizer and reduced by it to zero.

object - used by the code generator and the listing procedure to record the starting and finishing locations of the object code generated for the statement.

source - used by offset testing programs to locate the source text of this statement. procedure that produces the object code listing)

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

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

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

LHS_in_RHS - used in semantics to warn that portions of an aggregate target of an assignment statement are referenced in computing the right hand side and may not be changed until the whole right hand side has been computed.

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.

processed - set by semantic_translator to indicate that the statement has already been processed, so avoiding an erroneous re-processing. It may be noted that completely processed statements are created during the semantic translation, by do_semantics and io_semantics for example, and the newly created statements may be inserted after the statement currently being processed.

put_in_profile - set for the first statement among those which realize a given source language statement. If the profile option is in effect, the code generator will compile special profile code for each marked statement.

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

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.

The declaration processor constructs a reference node for each symbol node. This reference node contains the offset and locator qualifier necessary to locate the value at run-time. Each subscripted reference or substr reference results in a unique offset and a unique reference. Each locator qualified reference results in a unique reference node with its own qualifier expression. References without subscripts or locator qualification are represented by unique instances of the reference node originaly created by the declaration processor.

If the "shared" bit of a reference node is on, it indicates to the code generator and optimizer that this reference node appears as a node within more than one computation tree, and that each occurrence of this node may represent a reference to a unique value. If the "shared" bit is off, each reference to the node must represent a reference to the same value, and the "ref_count" of the reference node must indicate how many times this reference

node is referenced in the tree. The optimizer transforms the representation of the program to maximize the number of reference nodes whose shared bit is zero.

Format:

```
     dcl  1 reference                  based aligned,
          2 node_type                  bit(9) unaligned,
          2 array_ref                  bit(1) unaligned,
          2 varying_ref                bit(1) unaligned,
          2 shared                     bit(1) unaligned,
          2 put_data_sw                bit(1) unaligned,
          2 processed                  bit(1) unaligned,
          2 units                      fixed(3) unaligned,
          2 ref_count                  fixed(17) unaligned,
          2 c_offset                   fixed(24),
          2 c_length                   fixed(24),
          2 symbol                     ptr unaligned,
          2 qualifier                  ptr unaligned,
          2 offset                     ptr unaligned,
          2 length                     ptr unaligned,
          2 subscript_list             ptr unaligned,


          2 address                    structure unaligned,
            3 base                     bit(3),
            3 offset                   bit(15),
            3 op                       bit(9),
            3 no_address               bit(1),
            3 inhibit                  bit(1),
            3 ext_base                 bit(1),
            3 tag                      bit(6),
          2 info                       structure unaligned,
            3 address_in               structure,
              4 b                      dimension(0:7) bit(1),
              4 storage                bit(1),
            3 value_in                 structure,
              4 a                      bit(1),
              4 q                      bit(1),
              4 aq                     bit(1),
              4 string_aq              bit(1),
              4 complex_aq             bit(1),
              4 decimal_aq             bit(1),
              4 b                      dimension(0:7) bit(1),
              4 storage                bit(1),
              4 indicators             bit(1),
              4 x                      dimension(0:7) bit(1),
```

```
  3 skip                    bit(3),
2 data_type                 fixed(5) unaligned,
2 bits                      structure unaligned,
  3 padded_ref              bit(1),
  3 aligned_ref             bit(1),
  3 long_ref                bit(1),
  3 forward_ref             bit(1),
  3 ic_ref                  bit(1),
  3 temp_ref                bit(1),
  3 defined_ref             bit(1),
  3 evaluated               bit(1),
  3 allocate                bit(1),
  3 allocated               bit(1),
  3 abnormal                bit(1),
  3 even                    bit(1),
  3 perm_address            bit(1),
  3 aggregate               bit(1),
  3 hit_zero                bit(1),
  3 dont_save               bit(1),
  3 reserved                bit(2),
2 relocation                bit(12) unaligned,
2 last_usage                bit(18) unaligned,
2 store_ins                 bit(18) unaligned;
```

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).

shared - indicates a reference node used (potentially) in many parts of the program tree, refering to a generation of storage rather than to a value. The reference node that hangs from the symbol node has the shared bit set if there are no locator qualifier, variable length, or subscript fields needed to complete the reference.

References to such items are usually made by pointing to the symbol node's shared reference. If a reference node appears in the executable tree and has qualifier, length, or offset expressions, then it does not have the shared bit on; for a change to any such expression effectively alters the reference, and the compiler does not test for such changes.

put_data_sw - set by expression semantics when pre-processing the argument of a put_data_trans operator. It causes the subscripter to create a list of the subscripts of the scalar items and attach it at reference.subscript_list. This list is later attached to the put_data_trans operator and, ultimately, transmitted to the runtime I/O machinery.

processed - set by expression_semantics to indicate that the reference has been fully processed, so to avoid an erroneous re-processing.

units - indicates the units of the offset (bits, bytes, half_words, words).

ref_count - indicates that the reference is to a value which is referenced ref_count times (not necessarily in the current statement) without possibility of changing. (The ref_count is the number of pointers in the tree that point to this reference except in the case of a reference which is the first operand of an operator which sets its first operand; in this case, ref_count is the total number of pointers in the tree that point either to the reference or to the operator, the pointer from the operator to the reference not being counted for this purpose.) Values referenced under reference nodes with ref_count>0 may be kept in convenient registers by the code generator rather than, or as well as, in storage. The code generator reduces the ref_count after each use of the node. The optimizer tries to replace shared references with unshared references, as a means of dealing with common sub-expressions. In the case of a temporary, reduction of the ref_count to zero means that the storage or register holding the temporary may be reused.

c_offset - the constant offset. This field is meaningful whether or not the offset is variable.

c_length - the constant current length of a string value if reference.length 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. Parse uses reference.qualifier to point to a locator qualifier if one appears. In the case of a defined item, qualifier points to a reference to the base item.

offset - points to the offset expression. If the offset is entirely constant this field is null. Parse uses reference.offset to point to a list node containing the subscript expression trees, if subscripts appear; the list is in reverse order. Parse does not distinguish subscripts and arguments.

length - points to the length expression giving the current length of the string value. If the length is constant then this field is null. Parse uses reference.length to point to a reference node for the structure qualifier if any.

subscript_list - io_semantics uses this to point to a list node holding the subscripts of this reference; for put_data. The subscript expressions are listed in (forward) order. The size of this list is used by the subscripter to set the size of the block of storage (block.plio_ssl) into which the code generator will store the evaluated subscripts; subscripter sets that size, ssl_size, as max(k+1 , ssl_size), where k is the number of subscripts for the reference currently being processed.

padded_ref - indicates that the last word of the value is not shared with another value. Permits the code generator to assume, in most circumstances, that the spare bits

of the last word of storage touched by this item are zero. However, see passed_as_arg in "Symbol Nodes" on page 2-19.

abnormal - set if the symbol has the abnormal bit set or if it is a reference to a non-local automatic variable that is passed as an argument by reference.

NOTE: All other fields are set and used only by the code generator.

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() 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 aligned,
     2 node_type         bit(9) unaligned,
     2 reserved          bit(12) unaligned,
     2 number            fixed(14) unaligned,
     2 element           dimension(n refer(list.number))
                         ptr unaligned;
```

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

number - number of operands in this node.

element - pointers to the operands.

When list nodes are used to form uni-directional chains, the first "element" pointer is usually used to point to the next link in the chain.

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.

```
dcl  1 operator          based aligned,
     2 node_type         bit(9) unaligned,
     2 op_code           bit(9) unaligned,
     2 shared            bit(1) unaligned,
     2 processed         bit(1) unaligned,
     2 optimized         bit(1) unaligned,
     2 number            fixed(14) unaligned,
     2 operand           dimension(n refer(operator.number))
                         ptr unaligned;
```

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

op_code - is one of the op codes listed in the appendix.

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.

processed - set by semantic translator to prevent erroneous re-processing of this operator tree.

optimized - this computation has been previously performed and it does not need to be re-evaluated. Operand one contains the correct value.

number - the number of operands

operand - pointers to the operands

Operators

The operators of the internal representation closely resemble the operators of the PL/I language. These operators are listed in the appendix and can be classified into distinct groups of operators having similar function. The following sections describe each class of operators.

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, and may be packed or unpacked. The desired output is defined by the attributes of operand one.

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.

ASSIGNMENT OPERATORS

The assign 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, label constant to integer, and label constant to pointer.

Assign_size_ck allows assignments between any combination of arithmetic and string data. Code is generated to check whether the receiving variable has sufficient precision or string length to hold the value to be assigned; if not, the size or stringsize condition is signaled.

The assign_zero operator requires that its operand be fixed binary aligned with a precision of <36 and a scale factor of zero.

The copy_words operator copies the storage of operand two into the storage of operand one. The number of words to be copied is given by operand three. The operator is used to implement assignment of PL/I arrays or structures. It is generated only for non-packed aggregates of identical type and aggregation.

The copy_string operator copies the storage of operand two into the storage of operand one. The number of bits to be copied is given by operand three. The operator is used to implement assignment of PL/I arrays and structures. It is generated only for packed aggregates of identical type and aggregation.

The make_desc operator is used to create a basic argument descriptor value. Operand two is a bit string value representing the left part of an argument descriptor, and operand three is an integer expression representing the size value of the basic argument descriptor. The operator combines operands two and three to produce a basic argument descriptor value.

block_assign - This operator has n operands. Operands 2 through n are integer expressions to be evaluated and stored in operand one. Operand one is a temporary or variable whose data type is block_storage and whose size is sufficient to contain the integer values. The block_assign operator is used to process the subscript list of an array element or a put data statement.

RELATIONAL OPERATORS

Operand one of the relational operators is a bit_string value of length one. The other two operands are either: both arithmetic (see "Arithmetic Operators" on page 2-50), character-string, bit-string, pointer, offset, label, entry, or file expressions.

TRANSFER OPERATORS

Operand one of a transfer operator is a label valued expression. 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.

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, and is a temporary whose storage class is block_storage. During argument list creation all argument expressions are evaluated.

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", "temporary" or a variable reference. "Return value" storage class means that the called procedure will allocate space for the return value. (Se "Temporary Values" on page 2-38.)

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

The std_call operator results in a Multics Standard 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.

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 a descriptor for each parameter.

The ex_prologue operator causes the prologue to be evaluated.

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

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 whose block node is referenced by operand three. They are used to

assign these pointers to the automatic pointers used to reference the parameter or parameter descriptor. See "Parameter" on page 2-37.

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.

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. See Figure 2-4.

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.

BUILT-IN FUNCTION OPERATORS

The built-in function operators are a miscellaneous group of operators which support PL/I 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.

INPUT/OUTPUT OPERATORS

The input/output operators may be divided into four classes.

First are the operators get_file, get_string, put_file, put_string, read_file, write_file, locate_file, delete_file, rewrite_file, open_file, and close_file. These are used by the parse to pass parsed input/output statements to the semantic phase. Each of these operators has operands enough to compass

the references and expressions occurring in the options of the statement; each has one further operand, the last, which contains a bit(36) constant which encodes the options which have appeared and also the statement type. The operands of these operators are processed, and considerably rearranged, by the semantics before the code generation phase and, with the exception of the operators open_file and close_file which are retained without operands, these operators are not passed on to the code generator.

Second are the transmission operators: get_list_trans, get_edit_trans, get_data_trans, put_list_trans, put_edit_trans, and put_data_trans. The get_data_trans operator is presented to the code generator with a single operand, a join of the items appearing in the list of the get data statement. The code generator will transform this join into a constant list of runtime-symbol-table offsets which will serve to identify the allowable runtime references. The put_data_trans operator has two operands, a list of subscript expressions and the reference with which they are associated. The code generator will see that the list of subscripts, as well as the address and runtime-symbol-table offset of the reference, are made available at runtime. Each of the other four transmission operators takes a descriptor-valued expression and the reference to which it corresponds; the code generator will see that the descriptor and the address of the item referenced are available at runtime.

Third are the special operators: record_io, stream_prep, and terminate_trans. The record_io operator takes one or two operands and the stream_prep operator takes two operands. In both cases the first operand is a bit(36) constant which is transmitted to the runtime mechanisms and defines the work to be done. In both cases, the second operand, if present, is a label (the label of a null statement following the other statements which realize the I/O statement) to which control may be transferred at runtime if the execution of the statement cannot be continued. The terminate_trans operator is always compiled, after the list items, if any, in a get or put statement and has no operands; it is compiled by the code generator into the invocation of terminating code at runtime.

Fourth is the set of format operators. The first two operands of a format operator are standard: the first identifies the next format operator (in the case of the operator l_parn, the operator identified is that following the associated r_parn); and the second is an integer expression for the repitition count. The third and other operands depend on the operator. For l_parn, the third operand identifies the first format operator of the parenthesised format list. In the r_format operator, the third

operand is a reference to a format value. In the c_format, the third and fourth operands identify the component real format operators. In all other cases, the third and subsequent operands are integer expressions. (It is to be noted that all expressions, including those involved in the format-valued reference in an r_format, are to be evaluated at runtime from the runtime procedures but are compiled, when necessary, as internal procedures of the block containing the I/O statement.)

AGGREGATE OPERATORS: LOOP AND JOIN

The loop operator takes five operands and is used for the expansion of dimensioned aggregates. Operand one points to the expression to be expanded. Operand two is a reference, the control variable in the loop. Operand three and four are the lower and upper bound expressions for the loop. Operand five is a list of those scalar expressions which have been pulled out of the loop for optimization purposes.

The join operator has a variable number of operands which it serves to present in order to the code generator. Its operands may not be null. It is used in the expansion of structured aggregates, in the presentation of data lists in get and put statements, and in the compilation of most I/O statements.

The Node Types (nodes.incl.pl1)

```
          block_node                    "000000001"b
          statement_node                "000000010"b
          operator_node                 "000000011"b
          reference_node                "000000100"b
          token_node                    "000000101"b
          symbol_node                   "000000110"b
          context_node                  "000000111"b
          array_node                    "000001000"b
          bound_node                    "000001001"b
          format_value_node             "000001010"b
          list_node                     "000001011"b
          default_node                  "000001100"b
          machine_state_node            "000001101"b
          source_node                   "000001110"b
          label_node                    "000001111"b
          cross_reference_node          "000010000"b
          sf_par_node                   "000010001"b
          temporary_node                "000010010"b
```

The Block Types (block_types.incl.pl1)

```
          root_block                    "000000001"b
          external_procedure            "000000010"b
          internal_procedure            "000000011"b
          begin_block                   "000000100"b
          on_unit                       "000000101"b
```

The Boundary and Offset Unit Values (boundary.incl.pl1)

```
          bit_                1
          character_          2
          half_               3
          word_               4
          mod2_               5
          mod4_               6
          mod8_               7
```

The Declare Types (declare_type.incl.pl1)

```
          by_declare          "001"b
          by_explicit_context "010"b
          by_context          "011"b
          by_implication      "100"b
          by_compiler         "101"b
```

The Statement Types (statement_types.incl.pl1)

```
          unknown_statement             "000000000"b
          allocate_statement            "000000001"b
          assignment_statement          "000000010"b
          begin_statement               "000000011"b
          call_statement                "000000100"b
          close_statement               "000000101"b
          declare_statement             "000000110"b
          lock_statement                "000000111"b
          delete_statement              "000001000"b
          display_statement             "000001001"b
          do_statement                  "000001010"b
          else_clause                   "000001011"b
          end_statement                 "000001100"b
          entry_statement               "000001101"b
          exit_statement                "000001110"b
          format_statement              "000001111"b
          free_statement                "000010000"b
          get_statement                 "000010001"b
          goto_statement                "000010010"b
          if_statement                  "000010011"b
          locate_statement              "000010100"b
          null_statement                "000010101"b
          on_statement                  "000010110"b
          open_statement                "000010111"b
          procedure_statement           "000011000"b
          put_statement                 "000011001"b
          read_statement                "000011010"b
          return_statement              "000011011"b
          revert_statement              "000011100"b
          rewrite_statement             "000011101"b
          signal_statement              "000011110"b
          stop_statement                "000011111"b
          system_on_unit                "000100000"b
          unlock_statement              "000100001"b
          wait_statement                "000100010"b
          write_statement               "000100011"b
          default_statement             "000100100"b
          continue_statement            "000100101"b
```

```
                    max_p_flt_bin_1     27
                    max_p_flt_bin_2     63
                    max_p_fix_bin_1     35
                    max_p_fix_bin_2     71

                    max_p_dec           61

                    min_scale           -128
                    max_scale           +127
                    max_bit_string      2359296
                    max_char_string     262144
                    max_area_size       65536
                    min_area_size       30

                    bits_per_word       36
                    bits_per_packed_ptr 36
                    bits_per_double     72
                    characters_per_half 2
                    characters_per_word 4
                    characters_per_double 8
                    words_per_label_var 4
                    words_per_entry_var 4
                    bits_per_character  9
                    bits_per_half       18

                    default_area_size   1024
                    default_flt_bin_p   27
                    default_fix_bin_p   17
                    default_flt_dec_p   10
                    default_fix_dec_p   7


                    integer_type
          "010000000000000000000010000011000000"b


                    dec_integer_type
          "010000000000000000000010000101000000"b


                    pointer_type
          "000001000000000000000010000000000000"b
```

```
          real_type
"00100000000000000000001000000011000000"b


          complex_type
"00100000000000000000001000000010100000"b


          builtin_type
"00000000000000001000000000000000000000"b


          storage_block_type
"00000000000010000000000000000000000000"b


          arg_desc_type
"00000000000100000000000000000000000000"b


          local_label_var_type
"00000000100000000000001000010000010000"b


          entry_var_type
"00000000010000000000000000000000010000"b


          bit_type
"00010000000000000000000000000000000000"b


          char_type
"00001000000000000000000000000000000000"b


          f_logical_type
"00010000000000000000010000000000000000"b


          f_dim_type
"00000000000000000000001010000000010000"b


          f_type_conflict
"11110000000000001000000000000000000000"b
```

```
          f_dim_conflict
"10000000010000001000000000000000000"b




          f_external_type
"00000000010000000000000000000000100
 01000000000100000000000000000000000"b

          f_test_type
"01110000000000000000000000000000000
 00000000000000000000000000000000000"b

          f_auto_type
"00000000000000000000001010000000010000
 10001000000000000000000000000000000"b

          f_member_type
"00000000000000000000010100000000010000
 10010010000000000000000000000100000"b

          f_common_type
"10000000000000000000000000000000000
 01000010000000000000000000000000000"b

          f_external_conflict
"10000000000000000001000000000000000
 00011000000000000000000000000000000"b

          f_auto_conflict
"10000000010000001000000000000000000
 00010000010000000000000000000000000"b

          f_common_conflict
"10000000010000001000000000000000000
 00011000010000000000000000000000000"b
```

```
          f_dat_equ_conflict
"10000000010001001000000000000000000
 00000000010000000000000000000000000"b
```

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

```
is_identifier       "100000000"b
is_isub             "010000000"b
is_delimiter        "001000000"b
is_constant         "000100000"b
is_arith_constant   "000010000"b


(FORTRAN ONLY)


label_argument                "010000001"b
hollerith_constant_header     "010000010"b
x_format_f                    "010000011"b
new_line                      "001011010"b
logical_constant              "000100001"b
```

```
     add            "000010001"b
                    opnd(1) <- opnd(2)+opnd(3)


     sub            "000010010"b
                    opnd(1) <- opnd(2)-opnd(3)


     mult           "000010011"b
                    opnd(1) <- opnd(2)*opnd(3)


     div            "000010100"b
                    opnd(1) <- opnd(2)/opnd(3)


     negate         "000010101"b
                    opnd(1) <- -opnd(2)


     exp            "000010110"b
                    opnd(1) <- opnd(2) ** opnd(3)


     and_bits       "000100001"b
                    opnd(1) <- opnd(2) & opnd(3)


     or_bits        "000100010"b
                    opnd(1) <- opnd(2)|opnd(3)


     xor_bits       "000100011"b
                    opnd(1) <- opnd(2) xor opnd(3)


     not_bits       "000100100"b
                    opnd(1) <- ^opnd(2)


     cat_string     "000100101"b
                    opnd(1) <- opnd(2)||opnd(3)
```

```
assign          "000110001"b
                opnd(1) <- opnd(2)


assign_size_ck "000110010"b
                opnd(1) <- opnd(2)


assign_zero     "000110011"b
                opnd(1) <- 0


copy_words      "000110100"b
                move opnd(2) to opnd(1) by opnd(3) words


copy_string     "000110101"b
                move opnd(2) to opnd(1) by opnd(3) units


make_desc       "000110110"b
                opnd(1) <- descriptor(opnd(2),opnd(3))


pack            "000111000"b
                opnd(1) <- encode to picture opnd(2)


unpack          "000111001"b
                opnd(1) <- decode from picture opnd(2)


less_than       "001000100"b
                opnd(1) <- opnd(2) < opnd(3)


greater_than    "001000101"b
                opnd(1) <- opnd(2) > opnd(3)


equal           "001000110"b
                opnd(1) <- opnd(2) = opnd(3)


not_equal       "001000111"b
                opnd(1) <- opnd(2) ^= opnd(3)
```

```
     less_or_equal  "001001000"b
                    opnd(1) <- opnd(2) <= opnd(3)


     greater_or_equal "001001001"b
                    opnd(1) <- opnd(2) >= opnd(3)


     jump           "001010001"b
                    go to opnd(1) unconditionally


     jump_true      "001010010"b
                    go to opnd(1) if opnd(2) is not 0


     jump_false     "001010011"b
                    go to opnd(1) if opnd(2) is all 0


     jump_if_lt     "001010100"b
                    go to opnd(1) if opnd(2) < opnd(3)


     jump_if_gt     "001010101"b
                    go to opnd(1) if opnd(2) > opnd(3)


     jump_if_eq     "001010110"b
                    go to opnd(1) if opnd(2) = opnd(3)


     jump_if_ne     "001010111"b
                    go to opnd(1) if opnd(2) ^= opnd(3)


     jump_if_le     "001011000"b
                    go to opnd(1) if opnd(2) <= opnd(3)


     jump_if_ge     "001011001"b
                    go to opnd(1) if opnd(2) >= opnd(3)


     jump_three_way "001011010"b
                    opnd(1) = expression
                       go to opnd(2) if expression < 0
                       go to opnd(3) if expression = 0
                       go to opnd(4) if expression > 0
```

```
std_arg_list   "001100001"b
               opnd(1) <- arglist(opnd(2) desclist(opnd(3)))


return_words   "001100010"b
               return aggregate opnd(1), opnd(2) is length
               in words


std_call       "001100011"b
               opnd(1) <- call opnd(2) with opnd(3)


return_bits    "001100100"b
               return aggregate opnd(1), opnd(2) is length
               in bits


std_entry      "001100101"b
               entry(opnd(1)... opnd(n))


return_string  "001100110"b
               return string opnd(1)


ex_prologue    "001100111"b
               execute the prologue -no operands-


allot_auto     "001101000"b
               opnd(1) <- addrel(stack,opnd(2))


param_ptr      "001101001"b
               opnd(1) <- ptr to opnd(2) in  block opnd(3)


param_desc_ptr "001101010"b
               opnd(1) <- ptr to opnd(2) in block opnd(3)


std_return     "001101011"b
               return -no arguments-


allot_ctl      "001101100"b
               allocate opnd(1) and its  desc opnd(2)
```

```
free_ctl        "001101101"b
                free opnd(1)


bit_to_char     "010000000"b
                opnd(1) <- (opnd(2)+8)/9


bit_to_word     "010000001"b
                opnd(1) <- (opnd(2)+35)/36


char_to_word    "010000010"b
                opnd(1) <- (opnd(2)+3)/4


half_to_word    "010000011"b
                opnd(1) <- (opnd(2)+1)/2


word_to_mod2    "010000100"b
                opnd(1) <- (opnd(2)+1)/2*2


word_to_mod4    "010000101"b
                opnd(1) <- (opnd(2)+3)/4*4


word_to_mod8    "010000110"b
                opnd(1) <- (opnd(2)+7)/8*8


rel_fun         "010000111"b
                opnd(1) <- rel(opnd(2))


baseno_fun      "010001000"b
                opnd(1) <- baseno(opnd(2))


desc_size       "010001001"b
                opnd(1) <- substr(opnd(2),13,24)


ceil_fun        "010010000"b
                opnd(1) <- ceil(opnd(2))
```

```
floor_fun      "010010001"b
               opnd(1) <- floor(opnd(2))


round_fun      "010010010"b
               opnd(1) <- round(opnd(2))


sign_fun       "010010011"b
               opnd(1) <- sign(opnd(2))


abs_fun        "010010100"b
               opnd(1) <- abs(opnd(2))


trunc_fun      "010010101"b
               opnd(1) <- trunc(opnd(2))


tran_sign_fun  "010010110"b
               opnd(1) <- abs(opnd(2))
               with the sign of opnd(3)


index_fun      "010100000"b
               opnd(1) <- index(opnd(2),opnd(3))


off_fun        "010100001"b
               opnd(1) <- offset(opnd(2),opnd(3))


complex_fun    "010100010"b
               opnd(1) <- complex(opnd(2),opnd(3))


conjg_fun      "010100011"b
               opnd(1) <- conjg(opnd(2),opnd(3))


mod_fun        "010100100"b
               opnd(1) <- mod(opnd(2),opnd(3))


repeat_fun     "010100101"b
               opnd(1) <- repeat(opnd(2),opnd(3))
```

```
     verify_fun     "010100110"b
                    opnd(1) <- verify(opnd(2),opnd(3))


     translate_fun  "010100111"b
                    opnd(1) <- translate(opnd(2),opnd(3))


     lock_fun       "010101000"b
                    opnd(1) <- stac(opnd(2),opnd(3))


     real_fun       "010101001"b
                    opnd(1) <- real(opnd(2))


     imag_fun       "010101010"b
                    opnd(1) <- imag(opnd(2))


     length_fun     "010101011"b
                    opnd(1) <- length(opnd(2))


     pl1_mod_fun    "010101100"b
                    opnd(1) <- mod(opnd(2))


     search_fun     "010101101"b
                    opnd(1) <- search(opnd(2),opnd(3))


     allocation_fun "010101110"b
                    opnd(1)<-allocation(opnd(2))


     reverse_fun    "010101111"b
                    opnd(1) <- reverse(opnd(2))


     addr_fun       "010110000"b
                    opnd(1) <- addr(opnd(2))


     addr_fun_bits  "010110001"b
                    opnd(1) <- addr(opnd(2))
```

```
ptr_fun         "010110010"b
                opnd(1) <- ptr(opnd(2),opnd(3))


baseptr_fun     "010110011"b
                opnd(1) <- baseptr(opnd(2))


addrel_fun      "010110100"b
                opnd(1) <- addrel(opnd(2),opnd(3))


min_fun         "011000000"b
                opnd(1) <- min(opnd(1),opnd(2),...)


max_fun         "011000001"b
                opnd(1) <- max(opnd(1),opnd(2),...)


pos_dif_fun     "011000010"b
                opnd(1) <- opnd(2) - min(opnd(2),opnd(3))


enable_on       "011010100"b
                opnd(1) is the cond name
                opnd(2) is the file name
                opnd(3) is the block


revert_on       "011010101"b
                opnd(1) is the cond name,
                opnd(2) is the file name


signal_on       "011010110"b
                opnd(1) is the cond name
                opnd(2) is the file name


bound_ck        "011100000"b
                opnd(1) <- opnd(2)
                if opnd(3) <= opnd(2) <= opnd(4)


range_ck        "011100001"b
                opnd(1) <- opnd(2)
                if opnd(3) <= opnd(2) <= opnd(4)
```

```
     loop           "011100010"b
                    do opnd(1) for opnd(2) from opnd(3)
                    to opnd(4) by 1 , opnd(5) being
                    a list of scalar expressions removed
                    from the loop for optimization purposes.


     join           "011100011"b
                    compile in sequence:
                    opnd(1), opnd(2) ... opnd(n)


     r_parn         "011110001"b


     l_parn         "011110010"b
                    opnd(1) is format operator after
                    parenthesized format list, opnd(2)
                    is repitition count, opnd(3) is
                    first format operator of
                    parenthesized format list


     r_format       "011110011"b
                    opnd(1) is next format operator
                    opnd(2) is repitition count
                    opnd(3) is format-valued reference


     c_format       "011110100"b
                    opnd(1) is next format operator
                    opnd(2) is repitition count
                    opnd(3) is real format operator
                    opnd(4) is real format operator


     f_format       "011110101"b
                    opnd(1) is next format operator
                    opnd(2) is repitition count
                    opnd(3) is field size
                    opnd(4) is default decimal position
                    opnd(5) is scale factor


     e_format       "011110110"b
                    opnd(1) is next format operator
                    opnd(2) is repitition count
                    opnd(3) is field size
                    opnd(4) is default decimal position
```

```
                opnd(5) is total precision


     b_format       "011110111"b
                    opnd(1) is next format operator
                    opnd(2) is repitition count
                    opnd(3) is field size


     a_format       "011111000"b
                    opnd(1) is next format operator
                    opnd(2) is repitition count
                    opnd(3) is field size


     x_format       "011111001"b
                    opnd(1) is next format operator
                    opnd(2) is repitition count
                    opnd(3) is field size


     skip_format    "011111010"b
                    opnd(1) is next format operator
                    opnd(2) is repitition count
                    opnd(3) is skip count


     column_format  "011111011"b
                    opnd(1) is next format operator
                    opnd(2) is repitition count
                    opnd(3) is target column


     page_format    "011111100"b
                    opnd(1) is next format operator
                    opnd(2) is repitition count


     line_format    "011111101"b
                    opnd(1) is next format operator
                    opnd(2) is repitition count
                    opnd(3) is target line number


     picture_format "011111110"b
                    opnd(1) is next format operator
                    opnd(2) is repitition count
                    opnd(3) is picture constant
```

```
     get_list_trans "100000000"b
                    getlist(opnd(2))
                    with opnd(1)=desc(opnd(2))


     get_edit_trans "100000001"b
                    getedit(opnd(2))
                    with opnd(1)=desc(opnd(2))


     get_data_trans "100000010"b
                    opnd(1) is join of items (references)
                    in data list.


     put_list_trans "100000011"b
                    putlist(opnd(2))
                    with opnd(1)=desc(opnd(2))


     put_edit_trans "100000100"b
                    putedit(opnd(2))
                    with opnd(1)=desc(opnd(2))


     put_data_trans "100000101"b
                    putdata(opnd(2))
                    where opnd(1) points to list-node of
                    subscript expressions (or is null)


     terminate_trans "100000110"b
                    terminate stream transmission


     stream_prep    "100000111"b
                    initiate stream transmission
                    opnd(1) is description of statement
                    opnd(2) is label for abnormal return


     record_io      "100001000"b
                    perform record_i/o operation
                    opnd(1) is description of statement
                    and options; opnd(2), if present, is
                    label for abnormal return
```

```
     open_file      "100011001"b
                    opnd(1) is linesize
                    opnd(2) is file
                    opnd(3) is title
                    opnd(4) is pagesize
                    opnd(5) is attribute-bits
                    opnd(6) is job-bits


     close_file     "100011010"b
                    opnd(2) is file
                    opnd(3) is job-bits
```

These operators are produced by the parse but are not used as input to the code generator.

They are processed by the semantic translator.

```
     return_value   "100010010"b
                    return(opnd(1))


     allot_based    "100010011"b
                    allot opnd(1) in opnd(2)


     free_based     "100010100"b
                    free opnd(1) out of opnd(2)


     get_file       "100010101"b
                    opnd(1) is copy
                    opnd(2) is file
                    opnd(3) is skip
                    opnd(4) is list
                    opnd(5) is job-bits
```

```
get_string     "100010110"b
               opnd(1) is copy
               opnd(2) is string
               opnd(4) is list
               opnd(5) is job-bits


put_file       "100010111"b
               opnd(1) is line
               opnd(2) is file
               opnd(3) is skip
               opnd(4) is list
               opnd(5) is job-bits


put_string     "100011000"b
               opnd(2) is string
               opnd(4) is list
               opnd(5) is job-bits


read_file      "100011011"b
               opnd(1) is set, into, or ignore
               opnd(2) is file
               opnd(3) is key or keyto
               opnd(4) is job-bits


write_file     "100011100"b
               opnd(1) is from
               opnd(2) is file
               opnd(3) is keyfrom
               opnd(4) is job-bits


locate_file    "100011101"b
               opnd(2) is file
               opnd(3) is keyfrom
               opnd(4) is variable to be located
               opnd(5) is job-bits


do_fun         "100011110"b
               opnd(1) is join of a list
               opnd(2) is control variable ref
               opnd(3) is specification operator
```

```
do_spec         "100011111"b
                opnd(1) to opnd(2) by opnd(3)
                repeat opnd(4) while opnd(5)
                opnd(6) is next specification


rewrite_file    "100100000"b
                opnd(1) is from
                opnd(2) is file
                opnd(3) is key
                opnd(4) is job-bits


delete_file     "100100001"b
                opnd(2) is file
                opnd(3) is key
                opnd(4) is job-bits


refer           "100100101"b
                opnd(1) refer(opnd(2))


prefix_plus     "100100110"b
                opnd(1) <- +opnd(2)


nop             "100100111"b
                no-op
```

n

SECTION III

SYNTACTIC TRANSLATION

Syntactic translation is the process of disassembling the source program into its consituent parts called tokens, building an internal representation of the program, and putting information into the symbol table and other tables. The syntactic translator consists of two modules called the lexical analyser and the parse.

LEXICAL_ANALYSIS

The lexical analyser scans the characters of the source program from left to right and organizes the characters into groups of tokens which represent a statement. It creates the source listing file, it also builds a token table which contains the source representation of all tokens used in the source program. The lexical analyser is called by the parse each time the parse needs a new statement.

The token table produced by the lexical analyser contains a single entry for each unique token in the source program. Searching of the token table is done using a hash coded scheme that provides quick access to the table.

Each token table entry contains a pointer which may eventually point to a declaration of the token, that is, the symbol node. For each statement, the lexical analyzer builds a vector of pointers to the tokens which were found in the statement. This vector is the input to the parse.

NAME: lex

Function:

1. It maintains an internal static running character index to the source segment that shows at any instant the beginning of the source that the lexical analyser has yet to process.

2. It scans the source segment until it reaches the next semicolon, and groups the characters it has scanned into a set of lexical units called tokens. The order of tokens is kept in an internal static array of pointers called the token list. When lex returns, the character index is pointing at the character immediately following the semicolon that it has just scanned.

3. When an include statement is found in the text, lex treats the include segment as the current source segment and goes on processing, until it reaches the end of the include segment. Then it reverts to the original source segment.

4. If a listing is required, lex writes the source into the listing segment.

Entry:

lex

Usage:

```
declare lex entry;

call lex;
```

Programs that invoke this entry:

procedure_parse
do_parse
if_parse

Entry:

lex$write_last_line

This entry checks that no text follows the logical end of the program. This entry writes the last line of the source into the listing segment. It also writes the list of all include files used by the program into the listing segment.

Usage:

```
declare lex$write_last_line entry;

call lex$write_last_line;
```

Programs that invoke this entry:

parse

Entry:

lex$terminate_source

This entry terminates the source segment.

Usage:

```
          declare lex$terminate_source entry;

          call lex$terminate_source;


Programs that invoke this entry:

pl1


Entry:

          lex$scan_token_table

     This entry goes down the hash table and checks for duplicate
declarations.


Usage:

          declare lex$scan_token_table entry;

          call lex$scan_token_table;


Programs that invoke this entry:

pl1


Entry:

          lex$initialize_lex




DRAFT: SUBJECT TO CHANGE      3-84                  order number
```

This entry initializes the data$data pointer once per process, and initializes the hash table once per compilation.

Usage:

```
declare lex$initialize_lex entry;

call lex$initialize_lex;
```

Programs that invoke this entry:

parse

Entry:

lex$meter

This entry gathers some statistics about the hash table.
1. Number of empty buckets in the hash table.
2. Total number of tokens used in the program.
3. Maximum number of tokens in a single bucket of hash table.
4. Total storage used by all the token nodes for the program.

Usage:

```
declare lex$meter entry ( token_count, token_words,
empty_buckets, maximum );

call lex$meter ( fixed bin(15), fixed bin(15), fixed
bin(15), fixed bin(15) );
```

1.	token_count	total number of tokens used in the program. (output)

2.	token_words	total number of words of storage in the tree segment used by all the token nodes in the program. (output)
3.	empty_buckets	total number of empty buckets in the hash table. (output)
4.	maximum	the maximum number of tokens in a single bucket. (output)

Programs that invoke this entry:

none

Internal Procedures:

create_source
an internal procedure to create a source node for each of the include file used in the source program.

lex_create_token
an internal function used to create a token node for the token represented by the token_string. This function does essentially the same things as the external procedure create_token. The reason for this internal function is to save the large number of calling sequence lex would have to made to call the more expensive external procedure.

lex_err
an internal procedure used to call the error message program error_.

External Variables:

data$data
pl1_stat_$cur_statement
pl1_stat_$hash_table
pl1_stat_$last_source
pl1_stat_$line_count
pl1_stat_$listing_on
pl1_stat_$node_uses
pl1_stat_$seg_name
pl1_stat_$source_index
pl1_stat_$source_list_ptr
pl1_stat_$source_ptr
pl1_stat_$source_seg
pl1_stat_$st_length
pl1_stat_$st_start
pl1_stat_$statement_id
tree_$

Internal Static Variables:

bitcount	bit_count of an include file.
dataptr	pointer to the data$ segment that contains the driving table for lex.
end_of_file	bit indicating end of segment is reached.
file_ptr	pointer to an include file.
file_stack	array of structure that contains the information of the source segment and all the include files used in the source.
file_token	pointer to the token node created for the name of an include file.
filename_length	length of the include file name.
first_time	bit indicating whether lex$initialize_lex has been previously called in the same process.
index	the running character index to the source segment.
line_size	length of the current source line being processed by lex.

listing_on	bit indicating whether a listing is needed for this compilation. It has the same value as pl1_stat_$listing_on.
old_file_token	pointer to the old token node created for the name of an include file.
saved_index	saved running character index.
saved_length	saved length of current line.
saved_source_line	saved total length of current source line.
saved_tindex	saved length of the token string.
seg_ptr	pointer to an include file.
semi_colon_ptr	pointer to the token node ";".
source_depth	number of include files used.
source_files	total number of include files used.
source_line	total length of current source line.
source_string_length	length of the source segment.

Programs Called:

bindec
bindec$vs
create_token
error_
error_$no_text
find_include_file_$initiate_count
hcs_$terminate_noname
pl1_get
pl1_print$for_lex
pl1_print$non_varying
pl1_print$non_varying_nl
pl1_print$varying_nl
token_to_binary
translator_info_$get_source_info
tree_$

Include Files used:

rename
create_token
language_utility
source_id_descriptor
nodes
token
token_types
token_list
source_list
declare_type
symbol
system

Errors Diagnosed:

Error 76
Error 99
Error 100
Error 101
Error 103
Error 104
Error 105
Error 106
Error 107
Error 108
Error 109
Error 110
Error 111
Error 112
Error 125
Error 151
Error 152
Error 153
Error 154
Error 155
Error 156
Error 157
Error 158
Error 159
Error 441

NAME: data

Function:

This is a data segment that contains the driving table for the lexical analyzer. It consists of a two dimensional matrix of the form matrix(1:31,0:29). The lexical analyzer is an approximation of a finite state machine with 31 states. The input to the lexical analyzer is a character string. The character set used to construct the string can be loosely classified into 29 types. By a simple transformation, the matrix is declared as matrix(0:929). Each element of the matrix is a 36 bit bitstring containing four 9 bit substrings. The first nine bits give the token type of a resulting group of characters, the second nine bits are currently not used, the third nine bits give the action to take in lex, and the last nine bits give the next state.

THE_PARSE

The parse gets the statement represented by the vector of token pointers from the lex and proceeds to analyze the statement, and transform the statement into an appropriate internal representation. The completed internal representation is a program tree that contains all the relationships between all the components of the original source program.

NAME: parse

Function:

1. It initializes various static variables and modules used for the parse.

2. It creates the root block node as the basis for the whole tree segment for the program.

3. It calls lex for the first statement of the program, and subsequently invokes procedure_parse to parse the remaining statements of the program.

Entry:

parse

Usage:

```
declare parse entry ( ptr, ptr, fixed bin (15) );

call parse ( root, source_ptr, source_length );
```

1.	root	pointer to the root node block created by parse. (output)
2.	source_ptr	pointer to the base of the segment containing the source program. (input)
3.	source_length	length in characters of the source program. (input)

Programs that invoke this entry:

pl1

Internal Procedures:

none

External Variables:

pl1_stat_$compiler_created_index
pl1_stat_$error_memory
pl1_stat_$one
pl1_stat_$util_abort

Internal Static Variables:

none

Programs Called:

create_block
create_token
error_$initialize_error
lex$initialize_lex
lex$write_last_line
parse_error
procedure_parse
reserve$clear
statement_type

Include Files used:

block
block_types
language_utility

```
parse
source_id_descriptor
statement_types
token_types


Errors Diagnosed:


Error 180
Error 417
```

NAME: procedure_parse

Function:

1. It processes all statements occurring in begin blocks and procedures.

   By processing a statement is meant the following steps:
   a. calling lex to get the statement.
   b. calling statement_type to determine the type of the statement.
   c. calling an appropriate procedure to parse the statement into its proper internal representation.

2. It creates a block node for the begin block or the procedure.

3. It calls itself recursively to handle nested blocks.

4. It attempts to match end statements to the proper procedure statement or begin statement.

Entry:

procedure_parse

Usage:

declare procedure_parse entry (fixed bin(15), ptr, bit(12) aligned, ptr, ptr, bit(9) aligned, bit(1) aligned);

call procedure_parse (token_list_index, entry_ptr, conditions, father_block_ptr, end_ptr, block_type, return_flag);

1.	token_list_index	index of the token_list for the statement. (input/output)
2.	entry_ptr	pointer to the list of labels. (input)

3.	conditions	conditions for the block. (input)
4.	father_block_ptr	pointer to the block node containing this block. (input)
5.	end_ptr	pointer to the token that ends the block. (output)
6.	block_type	type of this block. (input)
7.	return_flag	bit indicating if there is a return statement in this block. (output)

Programs that invoke this entry:

parse
procedure_parse
do_parse
on_parse
if_parse

Internal Procedures:

none

External Variables:

pla_stat_$cur_statement
tree_$

Internal Static Variables:

none

Programs Called:

create_block
create_operator
create_statement
declare_label
declare_parse
default_parse
do_parse
if_parse
io_statement_parse
lex
on_parse
parse_error
procedure_parse
process_entry
statement_parse
statement_type

Include Files used:

parse
language_utility
source_id_descriptor
token_list
block
declare_type
op_codes
statement
token
block_types
statement_types
token_types
list

Errors Diagnosed:

Error 410
Error 411
Error 412
Error 416

NAME: do_parse

Function:

1. It parses the do statement.

2. It processes all statements following the do statement until a matching end statement is found.

3. It may call itself recursively to process other do statements.

Entry:

do_parse

Usage:

```
declare do_parse entry ( fixed bin(15), ptr, bit(12) aligned, ptr, ptr, bit(1) aligned, bit(1) aligned, bit(1) aligned );
```

```
call do_parse ( token_list_index, entry_ptr, conditions, father_block_ptr, end_ptr, entry_flag, return_flag, iterative_do_flag );
```

1.	token_list_index	index of the token_list for the statement. (input/output)
2.	entry_ptr	pointer to the list of labels. (input)
3.	conditions	conditions for the block. (input)
4.	father_block_ptr	pointer to the block node containing this block. (input)

5.	end_ptr	pointer to the token node that ends the block. (output)
6.	entry_flag	bit indicating whether there is any entry statement within this block. (output)
7.	return_flag	bit indicating whether there is any return statement within this block. (output)
8.	iterative_do_flag	bit indicating whether an iterative do group has been found. (output)

Programs that invoke this entry:

procedure_parse
do_parse
if_parse

Internal Procedures:

print
an internal procedure used to call the error message program parse_error.

External Variables:

pl1_stat_$cur_statement
tree_$

Internal Static Variables:

none

Programs Called:

create_label
create_list
create_operator
create_statement
declare_label
declare_parse
default_parse
do_parse
expression_parse
free_node
if_parse
io_statement_parse
lex
on_parse
parse_error
procedure_parse
process_entry
reference_parse
statement_parse
statement_type

Include Files used:

parse
language_utility
source_id_descriptor
token_list
block
op_codes
operator
statement
token
block_types
statement_types
token_types
list
label
reference
declare_type

Errors Diagnosed:

Error 404
Error 405
Error 406
Error 407
Error 408
Error 409
Error 411
Error 413
Error 416
Error 418
Error 419
Error 424
Error 425
Error 426
Error 429
Error 433

NAME: on_parse

Function:

1. It parses the on statement.

2. It processes all statements in the on unit.

3. It creates a block node for the on unit.

Entry:

on_parse

Usage:

```
declare on_parse entry ( fixed  bin(15),  ptr,  bit(12)
aligned, ptr, ptr );
```

```
call    on_parse    (    token_list_index,   entry_ptr,
conditions, father_block_ptr, end_ptr );
```

1.	token_list_index	index of the token_list for the statement. (input/output)
2.	entry_ptr	pointer to the list of labels. (input)
3.	conditions	conditions for the block. (input)
4.	father_block_ptr	pointer to the block node containing this block. (input)
5.	end_ptr	pointer to the token that ends the block. (output)

Programs that invoke this entry:

procedure_parse
do_parse
if_parse

Entry:

on_parse$revert

This entry parses the revert statement and the signal statement.

Usage:

```
declare on_parse$revert entry ( fixed bin(15), ptr, ptr
);
```

```
call on_parse$revert( token_list_index, statement_ptr,
father_block_ptr );
```

1.	token_list_index	index of the token_list for the statement. (input/output)
2.	statement_ptr	pointer to the statement node for the revert statement or the signal statement. (input)
3.	father_block_ptr	pointer to the block node that contains this block. (input)

Programs that invoke this entry:

statement_parse

Internal Procedures:

get_condition
this internal function ascertains if the condition name is valid, and records the condition context for the name.

External Variables:

pl1_stat_$condition_index
tree_$

Internal Static Variables:

none

Programs Called:

bindec$vs
context
create_block
create_list
create_operator
create_statement
create_symbol
create_token
declare_label
free_node
io_statement_parse
parse_error
procedure_parse
reference_parse
statement_parse
statement_type

Include Files used:

parse
language_utility
source_id_descriptor
block
block_types
context_codes
declare_type
list
nodes
op_codes
operator
reference
statement
statement_types
symbol
token
token_list
token_types

Errors Diagnosed:

Error 1
Error 42
Error 420
Error 421
Error 422
Error 423

NAME: statement_type

Function:

1. It parses the condition prefix for the statement.

2. It parses the label prefix for the statement.

3. It determines the type of statement returned by lex.

Entry:

statement_type

Usage:

```
declare statement_type entry ( fixed bin(15), ptr,
bit(12) aligned) returns (fixed bin(15));

type = statement_type ( token_list_index, label_ptr,
conditions );
```

1.	token_list_index	index of the token_list for the statement. (input/output)
2.	label_ptr	pointer to the list of labels for the statement. (output)
3.	conditions	conditions for the statement. (output)
4.	type	type of statement found by this procedure. (output)

Programs that invoke this entry:

procedure_parse
parse
do_parse
on_parse
if_parse

Internal Procedures:

has_equal
: an internal function to advance the token_list_index to search for an equal token.

print
: an internal procedure to call the error message program parse_error.

skip_parens
: an internal procedure to advance the token_list_index until it matches a corresponding right parenthesis.

External Variables:

tree_$

Internal Static Variables:

none

Programs Called:

create_list
create_reference
create_token
parse_error

Include Files used:

language_utility
source_id_descriptor
token_list
list
reference
nodes
token_types
statement_types

Errors Diagnosed:

Error 2
Error 43
Error 44
Error 45
Error 95
Error 96

NAME: statement_parse

Function:

1. The following statements are parsed by this program:
   allocate statement
   assignment statement
   call statement
   free statement
   goto statement
   null statement
   return statement

Entry:

   statement_parse

Usage:

   declare statement_parse entry (fixed bin(15), ptr, bit(12) aligned, ptr, fixed bin(15));

   call statement_parse (token_list_index, label_ptr, conditions, cur_block, type);

1.	token_list_index	index of the token_list for the statement. (input/output)
2.	label_ptr	pointer to the list of labels for the statement. (input)
3.	conditions	conditions for the statement. (input)
4.	cur_block	pointer to the block node containing this statement. (input)

5. type   type of statement to be parsed by
                            this program. (input)

Programs that invoke this entry:

procedure_parse
do_parse
on_parse
if_parse

Internal Procedures:

print
                  an internal procedure used to call the error
                  message program parse_error.

External Variables:

pl1_stat_$cur_statement
tree_$

Internal Static Variables:

          none

Programs Called:

context
create_list
create_operator
create_reference
create_statement
create_symbol
declare_label

```
expression_parse
on_parse$revert
parse_error
reference_parse


Include Files used:


parse
language_utility
source_id_descriptor
block
declare_type
context_codes
label
list
nodes
op_codes
operator
reference
statement
statement_types
symbol
token
token_list
token_types


Errors Diagnosed:


Error 1
Error 5
Error 49
Error 150
Error 444
Error 446
Error 447
Error 450
Error 451
Error 452
Error 453
Error 454
Error 455
Error 456
Error 460
```

NAME: if_parse

Function:

1. It parses the if statement.

2. If the then clause is an independent statement, this program will parse the then clause.

3. If the then clause is a group or a begin block, this program will process all the statements in the then clause.

4. It also processes all the statements in the else clause if there is an else clause.

Entry:

if_parse

Usage:

```
declare if_parse entry ( fixed bin(15), ptr, bit(12) aligned, ptr, ptr, bit(1) aligned );

call if_parse ( token_list_index, entry_ptr, conditions, father_block, end_ptr, return_flag );
```

1.	token_list_index	index to the token_list for the statement. (input/outut)
2.	entry_ptr	pointer to the list of labels for this statement. (input)
3.	conditions	conditions for this statement. (input)
4.	father_block	pointer to the block node containing this statement. (input)

5. end_ptr pointer to the token that ends the block. (output)

6. return_flag bit indicating whether there is a return statement in this statement. (output)

Programs that invoke this entry:

procedure_parse
do_parse
if_parse

Internal Procedures:

print
an internal procedure used to call the error message program parse_error.

External Variables:

pl1_stat_$cur_statement
tree_$

Internal Static Variables:

none

Programs Called:

create_label
create_list
create_operator
create_statement

declare_label
do_parse
expression_parse
if_parse
io_statement_parse
lex
on_parse
parse_error
procedure_parse
reference_parse
statement_parse
statement_type

Include Files used:

parse
language_utility
source_id_descriptor
token_list
token
token_types
op_codes
block
block_types
statement
statement_types
nodes
reference
operator
list
label
symbol
declare_type

Errors Diagnosed:

Error 1
Error 412
Error 430
Error 431
Error 432
Error 446

NAME: io_statement_parse

Function:

1. It parses the following input/output statements:
   get statement
   put statement
   read statement
   write statement
   rewrite statement
   locate statement
   delete statement
   open statement
   close statement

2. It calls format_list_parse to parse the format statement.

Entry:

io_statement_parse

Usage:

declare io_statement_parse entry (fixed bin(15), ptr, bit(12) aligned, ptr, ptr, bit(1) aligned, bit(9) aligned);

call io_statement_parse (token_list_ptr, entry_ptr, conditions, father_block, end_ptr, return_flag, statement_type);

1.	token_list_index	index to the token_list for the statement. (input/output)
2.	entry_ptr	pointer to the list of labels fo this statement. (input)
3.	conditions	conditions for this statement. (input)

4.	father_block	pointer to the block node containing this statement. (input)
5.	end_ptr	pointer to the token that ends the block. (output)
6.	return_flag	bit indicating whether there is a return statement in this block. (input)
7.	statement_type	type of statement to be parsed by this program. (input)

Programs that invoke this entry:

procedure_parse
do_parse
on_parse
if_parse

Internal Procedures:

none

External Variables:

pl1_stat_$cur_statement
tree_$

Internal Static Variables:

none

Programs Called:

context
create_operator
create_statement
create_symbol
create_token
data_list_parse
declare_label
expression_parse
format_list_parse
parse_error

Include Files used:

parse
language_utility
source_id_descriptor
list
block_types
label
block
context_codes
nodes
declare_type
operator
op_codes
statement
statement_types
symbol
token_list
token_types

Errors Diagnosed:

Error 169
Error 237
Error 238
Error 239
Error 240
Error 241
Error 243
Error 245
Error 247
Error 254

Error 257
Error 288
Error 289
Error 290
Error 293
Error 428

NAME: format_list_parse

Function:

1. It parses the format list in a format statement.

2. It parses the format list in a get (edit) statement or a put (edit) statement.

Entry:

format_list_parse

Usage:

declare format_list_parse entry (fixed bin(15), ptr, ptr, ptr) returns (bit(1) aligned);

success_bit = format_list_parse (token_list_index, cur_block, statement_ptr, format_tree);

1.	token_list_index	index to the token list for the statement. (input)
2.	cur_block	pointer to the block node containing the format list. (input)
3.	statement_ptr	pointer to the statement node containing the format list. (input)
4.	format_tree	pointer to the format list returned by this program. (output)
5.	success_bit	bit indicating if the list of tokens does indeed parse into a format list. (output)

Programs that invoke this entry:

io_statement_parse
format_list_parse

Internal Procedures:

none

External Variables:

tree_$

Internal Static Variables:

none

Programs Called:

create_operator
create_symbol
declare_picture
expression_parse
format_list_parse
free_node
parse_error
reference_parse

Include Files used:

parse
language_utility
source_id_descriptor

block
declare_type
label
list
nodes
operator
op_codes
picture_image
reference
statement
statement_types
token_list
token_types
symbol

Errors Diagnosed:

Error 278
Error 427
Error 439

NAME: data_list_parse

Function:

1. It parses the data list in an input/output statement.

Entry:

data_list_parse

Usage:

```
declare data_list_parse entry ( fixed bin(15), ptr,
ptr) returns ( bit(1) aligned );

success_bit = data_list_parse ( token_list_index,
cur_block, data_tree );
```

1.	token_list_index	index to the token list for the statement. (input)
2.	cur_block	pointer to the block node containing the statement. (input)
3.	data_tree	pointer to the data list returned by this program. (output)
4.	success_bit	bit indicating if the list of tokens does indeed parse into a data list. (output)

Programs that invoke this entry:

io_statement_parse

Internal Procedures:

none

External Variables:

tree_$

Internal Static Variables:

none

Programs Called:

create_operator
expression_parse
parse_error
reference_parse

Include Files used:

parse
language_utility
source_id_descriptor
operator
op_codes
token_list
token_types

Errors Diagnosed:

Error 255
Error 256

Error 258
Error 404
Error 405
Error 406
Error 407
Error 408
Error 409
Error 418
Error 419
Error 424
Error 426

NAME: expression_parse

Function:

1. This procedure parses expressions using a simple operator procedence technique. The syntax parsed is:

```
<expression> ::= <primitive> [ <operator> <primitive> ]
...
```

where the nth operator and its operands are stacked if the n+1st operator has higher precedence. The primitive is parsed by the intenal procedure "primitive".

Entry:

```
expression_parse
```

Usage:

```
declare expression_parse entry ( fixed bin(15), ptr )
returns (ptr);

expression_tree = expression_parse ( token_list_index,
cur_block );
```

1.	token_list_index	index to the token list for the statement. (input/output)
2.	cur_block	pointer to the block node containing this expression. (input)
3.	expression_tree	pointer to the expression returned by this program. (output)

Programs that invoke this entry:

attribute_parse
data_list_parse
default_parse
do_parse
expression_parse
format_list_parse
if_parse
io_statement_parse
reference_parse
statement_parse

Internal Procedures:

primitive
: an internal procedure used to parse expressions, exponentiation operators, and parenthesized expressions.

External Variables:

tree_$

Internal Static Variables:

t
: pointer used to get better accessing to the list of tokens.

Programs Called:

create_operator
create_token
evaluate
expression_parse
reference_parse

Include Files used:

parse
language_utility
source_id_descriptor
token_list
token
nodes
operator
op_codes
token_types

Errors Diagnosed:

none

NAME: reference_parse

Function:

1. It parses the list of tokens into a reference node whenever possible.

2. The reference may be locator qualified, structure qualified, subscripted, or any combination thereof.

3. The reference may also be a function reference.

Entry:

reference_parse

Usage:

```
declare reference_parse entry ( fixed bin(15), ptr )
returns (ptr);

reference_tree = reference_parse ( token_list_index,
cur_block );
```

1.	token_list_index	index to the token list for the statement. (input/output)
2.	cur_block	pointer to the block node containing this operand. (input)
3.	reference_tree	pointer to the operand representing the result of reference_parse. (output)

Programs that invoke this entry:

attribute_parse
data_list_parse
do_parse
expression_parse
format_list_parse
if_parse
io_statement_parse
on_parse
statement_parse

Internal Procedures:

atom
an internal procedure to test and parse the list of tokens into an expression. Expressions of the form
(reference)
is parsed into
temporary_node = reference

External Variables:

tree_$

Internal Static Variables:

none

Programs Called:

context
create_list
create_operator
create_reference
create_symbol
expression_parse

Include Files used:

parse
language_utility
source_id_descriptor
context_codes
declare_type
list
nodes
op_codes
operator
reference
symbol
token
token_list
token_types

Errors Diagnosed:

none

NAME: declare_parse

Function:

1. It parses the declare statement.

Entry:

declare_parse

Usage:

```
declare declare_parse entry ( fixed bin(15), ptr, ptr
);

call declare_parse ( token_list_index, cur_block,
labelptr );
```

1.	token_list_index	index to the token_list for the statement. (input/output)
2.	cur_block	pointer to the block node containing this statement. (input)
3.	labelptr	pointer to the list of labels to this statement. (input)

Programs that invoke this entry:

procedure_parse
do_parse

Entry:

declare_parse$abort

This entry calls the error message program parse_error. It also attempts to resume parse at the first comma after the error token not contained in parentheses.

Usage:

```
declare declare_parse$abort entry( fixed bin(15), ptr );

call declare_parse$abort ( error_number, error_pointer );
```

1.	error_number	the error number. (input)
2.	error_pointer	pointer to the operand that causes the error. (input)

Programs that invoke this entry:

attribute_parse
declare_parse
descriptor_parse

Internal Procedures:

declare_parse_factored
is called to parse all the tokens in the declare statement between "declare" and the semicolon. It calls attribute_parse to process the attributes, and it calls itself recursively to process factored attribute lists when it encounters a left parenthesis.

link_symbol
an internal procedure used to link up members of a structure.

External Variables:

pl1_stat_$cur_statement
pl1_stat_$statement_id
pl1_stat_$unwind
tree_$

Internal Static Variables:

cblock	pointer to the block node containing this declare statement.
factored_level	number indicating the depth of structure level the current symbol is in.
k	used to show the position of the token_list_index.
l	used to show the position of the token_list_index.
previous_symbol	pointer to the symbol node of the structure containing the current symbol.

Programs Called:

attribute_parse
create_statement
create_symbol
create_token
declare_label
declare_parse$abort
free_node
merge_attributes
parse_error
token_to_binary

Include Files used:

parse
language_utility
source_id_descriptor
block
token_types
statement_types
symbol
token_list
token
declare_type
reference
link_symbol

Errors Diagnosed:

Error 3
Error 27

NAME: attribute_parse

Function:

1. It parses the attribute set occurring in declare statements, in the returns(), entry() attributes, and in the when() clause of then generic () attribute.

Entry:

attribute_parse

Usage:

declare attribute_parse entry (ptr, ptr, fixed bin(15), bit(1) aligned);

call attribute_parse (cur_block, symbol_ptr, token_list_index, generic_bit);

1.	cur_block	pointer to the block node containing this declaration. (input)
2.	symbol_ptr	pointer to the symbol node for which the attributes are declared for. (input)
3.	token_list_index	index to the token list for the statement. (input/output)
4.	generic_bit	bit indicating that the procedure is called in the generic attribute context, which allows the declaration of precision attribute to range from low precision to high precision and the scale attribute to range from low scale to high scale. (input)

Programs that invoke this entry:

declare_parse
default_parse
descriptor_parse

Internal Procedures:

get_scale
an internal procedure to get the scale of a fixed or precision attribute.

initial_list
an internal procedure to parse the initial attribute.

print
an internal procedure used to call the error message program declare_parse$abort.

refer_exp
an internal procedure to get the size or the bound of an item. In particular, if the size or bound has refer_extents declaration, it will be parsed.

External Variables:

pl1_stat_$one
tree_$

Internal Static Variables:

none

Programs Called:

context
create_array
create_bound
create_list
create_operator
create_token
declare_parse$abort
descriptor_parse
expression_parse
reference_parse
token_to_binary

Include Files used:

parse
language_utility
source_id_descriptor
attribute_table
block
token_list
reference
context_codes
token_types
symbol
array
operator
op_codes
list
nodes

Errors Diagnosed:

Error 6
Error 7
Error 8
Error 9
Error 10
Error 11
Error 12
Error 13

Error 14
Error 15
Error 17
Error 18
Error 19
Error 20
Error 22
Error 23
Error 24
Error 26
Error 57
Error 138
Error 192
Error 193

NAME: default_parse

Function:

1. It parses the default statement.

Entry:

default_parse

Usage:

```
declare default_parse entry ( fixed bin(15), ptr, ptr
);

call default_parse ( token_list_index, cur_block,
label_ptr );
```

1.	token_list_index	index to the token list for the statement. (input/output)
2.	cur_block	pointer to the block node containing this statement. (input)
3.	label_ptr	pointer to the list of labels for this statement. (input)

Programs that invoke this entry:

procedure_parse
do_parse

Internal Procedures:

none

External Variables:

pl1_stat_$cur_statement
pl1_stat_$statement_id
pl1_stat_$unwind
tree_$

Internal Static Variables:

none

Programs Called:

attribute_parse
create_default
create_statement
create_symbol
declare_label
expression_parse
free_node
parse_error

Include Files used:

parse
language_utility
source_id_descriptor
default
symbol
block
token_list
token_types

statement_types
declare_type

Errors Diagnosed:

Error 48

NAME: descriptor_parse

Function:

1. It parses descriptor lists. Descriptor lists occur in the following three contexts:
   entry (descriptior list) in the entry attribute,
   returns (descriptior list) in the returns attribute,
   when (descriptior list) in the when clause of the generic attribute.

Entry:

descriptor_parse

Usage:

declare descriptor_parse entry (ptr, ptr, fixed bin(15)) returns (ptr);

return_ptr = descriptor_parse (cur_block, token_ptr, token_list_index);

1.	cur_block	pointer to the block node containing this declaration. (input)
2.	token_ptr	pointer to the token node for which the attribute is declared for. (input)
3.	token_list_index	index to the token_list for the statement. (input/output)
4.	return_ptr	pointer to the chain of list nodes returned by this program. (output)

Programs that invoke this entry:

attribute_parse
process_entry

Internal Procedures:

link_symbol
an internal procedure used to link up members of a structure.

External Variables:

tree_$

Internal Static Variables:

none

Programs Called:

attribute_parse
bindec$vs
create_list
create_symbol
create_token
declare_parse$abort
parse_error
token_to_binary

Include Files used:

```
parse
language_utility
source_id_descriptor
symbol
token_list
token_types
declare_type
list
link_symbol


Errors Diagnosed:


Error 16
```

NAME: process_entry

Function:

1. It parses the procedure statement and the entry statement.

Entry:

process_entry

Usage:

declare process_entry entry (fixed bin(15), bit(9) aligned, ptr, ptr, bit(12) aligned);

call process_entry (token_list_index, statement_type, cur_block, entry_ptr, conditions);

1.	token_list_index	index to the token_list for the statement. (input/output)
2.	statement_type	type of statement. (input)
3.	cur_block	pointer to the block node containing this statement. (input)
4.	entry_ptr	pointer to the list of labels for this statement. (input)
5.	conditions	conditions for this statement. (input)

Programs that invoke this entry:

procedure_parse
do_parse

Internal Procedures:

print
an internal procedure used to call the error message program parse_error.

External Variables:

cg_static_$support
pl1_stat_$cur_statement
pl1_stat_$root
pl1_stat_$statement_id
pl1_stat_$unwind
pl1_stat_$validate_proc
tree_$

Internal Static Variables:

none

Programs Called:

context
create_cross_reference
create_list
create_operator
create_statement
create_symbol
create_token
descriptor_parse
parse_error
reserve$rename_parse

Include Files used:

parse
language_utility
source_id_descriptor
token_list
context_codes
nodes
token
statement_types
statement
cross_reference
symbol
declare_type
operator
token_types
op_codes
list
block
block_types

Errors Diagnosed:

Error 34
Error 35
Error 36
Error 37
Error 38
Error 39
Error 40
Error 41
Error 46

NAME: context

Function:

1. It records the context of certain identifiers found during the parse.

Entry:

context

Usage:

```
declare context entry ( ptr, ptr, fixed bin(15) );

call context ( identifier, block_ptr, context_type );
```

1.	identifier	pointer to the token node representing the identifier. (input)
2.	block_ptr	pointer to the block node containing this token. (input)
3.	context_type	type of cntext to be recorded for the identifier. (input)

Programs that invoke this entry:

attribute_parse
io_statement_parse
on_parse
process_entry
reference_parse
statement_parse

Internal Procedures:

none

External Variables:

none

Internal Static Variables:

none

Programs Called:

create_context

Include Files used:

language_utility
source_id_descriptor
context
context_codes
nodes
block

Errors Diagnosed:

none

NAME: evaluate

Function:

1. It examines an expression involving two token constants and decides if they can be simplified into one token constant.

Entry:

evaluate

Usage:

declare evaluate entry (bit(9) aligned, ptr, ptr) retruns (ptr);

return_ptr = evaluate (op_code, first_ptr, second_ptr);

1.	op_code	indicates the kind of operation is involved. (input)
2.	first_ptr	pointer to the first token constant. (input)
3.	second_ptr	pointer to the second token constant. (input)
4.	return_ptr	pointer to the token node representing the resulting operand. (output)

Programs that invoke this entry:

expression_parse

Internal Procedures:

none

External Variables:

none

Internal Static Variables:

none

Programs Called:

bindec
create_operator
create_token
token_to_binary

Include Files used:

op_codes
operator
token
token_types
language_utility
source_id_descriptor

Errors Diagnosed:

none

SECTION IV

DECLARATION PROCESSING

THE_CONTEXT_PROCESSOR

The context processor scans all the context nodes containing contextually derived attributes recorded during the parse. The context processor either augments the partial symbol table node created from declaration statements or creates new declarations. This activity constitutes the contextual and implicit declarations.

NAME: context_processor

Function:

1. It does the context processing of all the context entries on a block node.

2. For each context entry in the block, it will try to match a previous declared symbol.

3. If a previous declaration is found, the context declaration will be overwritten except for the parameter context. If no previous declaration is found, a symbol node will be created, and the context declaration copied on to the symbol node.

4. If a condition context entry is found to match with a declaration not in the same block, a new declaration will be made.

5. This program also expands the like attribute appearing anywhere in the block.

Entry:

context_processor

Usage:

```
declare context_processor entry ( ptr );

call context_processor ( block_ptr );
```

1. block_ptr — pointer to the block node whose block.context chain is to be scanned. (input)

Programs that invoke this entry:

context_processor
semantic_translator

Internal Procedures:

found
an internal procedure to match a context entry with a previously declared symbol node entry.

print
an internal procedure to call the error message program error_$no_text.

process_like
an internal procedure to process and expand the like attribute in a symbol node.

External Variables:

pl1_stat_$root

Internal Static Variables:

none

Programs Called:

context_processor
copy_expression$copy_sons
create_symbol
error_$no_text
lookup

Include Files used:

semant
language_utility
source_id_descriptor
block
nodes
reference
context
declare_type
symbol
token

Errors Diagnosed:

Error 69
Error 74
Error 74
Error 75
Error 119
Error 120
Error 133
Error 189
Error 214

After contextual and implicit declarations have been processed, the declaration processor scans all the symbol table nodes to develop additional information about each variable. These include the preparation of accessing code: transforming parameters and automatic adjustible arrays into based references, calculation of boundary requirements, offset expressions, and array multipliers and virtual origins; the computation of storage requirements for each variable; and the generation of initialization code for some variables.

NAME: declare

Function:

1. This program establishes complete declarations for all the names used in the program.

2. It calls declare_structure to establish the complete declaration for all the members of the structure.

3. It calls validate to get the default attributes, and to check for correctness of all the declared attributes.

4. It creates descriptors for parameters and controlled variables.

5. It calls get_size to determine the storage size and boundary requirement for the declaration.

6. It generates a character string constant for condition constants.

7. It establishes the complete declaration for the returns descriptor and the parameter descriptor for an entry declaration.

8. For all the return values of all the entry constants in the block, it determines whether the attributes associated with the return values are the same. An integer will be created for use in the semantic translator if the attributes associated with the return values are not the same.

9. Pointers are created for parameters appearing in more than one position in any entry statement.

10. Allot_auto operators will be created in the prologue sequence for the block, for automatic variables with adjustible sizes.

11. It calls expand_initial to do the initialization of variables if necessary.

Entry:

declare

Usage:

declare declare entry (ptr);

call declare (symbol_ptr);

1. symbol_ptr pointer to the symbol node to be processed by this program. (input)

Programs that invoke this entry:

builtin
declare
declare_structure
defined_reference
expand_assign
expand_primitive
expression_semantics
io_semantics
operator_semantics
semantic_translator

Internal Procedures:

none

External Variables:

pl1_stat_$eis_mode

Internal Static Variables:

none

Programs Called:

compare_declaration
copy_expression
create_list
create_operator
create_statement$prologue
declare
declare_constant$char
declare_constant$integer
declare_descriptor
declare_descriptor$parm
declare_integer
declare_pointer
declare_structure
expand_initial
get_size
lookup
semantic_translator$abort
semantic_translator$error
validate

Include Files used:

semant
language_utility
source_id_descriptor
symbol
block
reference
list
operator
statement
op_codes
statement_types
nodes
token
token_types

declare_type
boundary
system

Errors Diagnosed:

Error 98
Error 149
Error 194
Error 196
Error 213

NAME: compare_declaration

Function:

1. It compares the data type and the size of two declarations.

2. If the two declarations are arrays, or structures, it calls itself recursively to compare the array dimensions, bounds, or attributes of members of the structure.

Entry:

compare_declaration

Usage:

declare compare_declaration entry (ptr, ptr) returns (bit(1) aligned);

success_bit = compare_declaration (first_ptr, second_ptr);

1.	first_ptr	pointer to either a reference node or a symbol node. (input)
2.	second_ptr	pointer to a symbol node. (input)
3.	success_bit	bit indicating if the comparison is successful. (output)

Programs that invoke this entry:

compare_declaration
declare
expand_assign

operator_semantics

Internal Procedures:

none

External Variables:

none

Internal Static Variables:

none

Programs Called:

compare_declaration
compare_expression

Include Files used:

semant
language_utility
source_id_descriptor
array
nodes
picture_image
reference
symbol

Errors Diagnosed:

none

NAME: validate

Function:

1. It validates that all attributes on a declaration is compatible.

2. It applies the default attributes to every declaration.

3. It checks for completeness of certain attributes.

4. It develops the packed attribute and the abnormal attribute.

5. It validates that precision, scale, string size, and area size are within proper range.

Entry:

validate

Usage:

```
declare validate entry ( ptr );

call validate ( symbol_ptr );
```

1. symbol_ptr — pointer to the symbol node to be processed by this program. (input)

Programs that invoke this entry:

declare_
declare_structure
expression_semantics

Internal Procedures:

evaluate
an internal procedure to evaluate the predicate of a default statement.

inconsistent
an internal procedure to check for incompatible attributes in the same declaration.

print
an internal procedure to call the error message program semantic_translator$error.

system
an internal procedure to evaluate the system defaults.

External Variables:

none

Internal Static Variables:

none

Programs Called:

error_$no_text
merge_attributes
propagate_bit
semantic_translator$error
token_to_binary

Include Files used:

```
semant
language_utility
source_id_descriptor
default
symbol
symbol_bits
reference
operator
token
token_types
decoded_token_types
list
block
op_codes
nodes
system
attribute_table
declare_type


Errors Diagnosed:


Error 97
Error 113
Error 200
Error 201
Error 204
Error 205
Error 206
Error 207
Error 208
Error 209
Error 211
Error 212
Error 215
Error 216
Error 217
Error 218
Error 219
Error 220
Error 222
Error 279
Error 280
Error 281
Error 282
Error 283
Error 284
```

Error 285
Error 357
Error 360
Error 367

NAME: merge_attributes

Function:

1. It merges attributes from a template declaration into a target declaration.

Entry:

merge_attributes

Usage:

```
declare merge_attributes entry ( ptr, ptr ) returns (
bit(1) aligned );
```

```
success_bit = merge_attributes ( target_symbol_ptr,
template_symbol_ptr );
```

1.	target_symbol_ptr	pointer to the symbol node of the declaration to which the attributes are merged into. (input)
2.	template_symbol_ptr	pointer to the symbol node of the declaration of the template. (input)
3.	success_bit	bit indicating if the merging process is successful. (output)

Programs that invoke this entry:

declare_parse
lang_util_
validate

Internal Procedures:

none

External Variables:

none

Internal Static Variables:

none

Programs Called:

copy_expression
create_token

Include Files used:

symbol
reference
token
token_types
language_utility
source_id_descriptor

Errors Diagnosed:

none

NAME: get_size

Function:

1. It creates statements in the prologue sequence for adjustible bounds or adjustible sizes.

2. It turns on the varying_ref bit in the reference node for varying strings.

3. It fills the length and c_length fields in the reference node for areas.

4. It fills in the word_size and c_word_size fields in the symbol node.

5. If the declaration is a picture, it calls declare_picture to check the syntax of the picture string and to develop all its attributes.

6. It calculates the boundary requirement for each declaration.

7. If the declaration is an array, it calls get_array_size to find the total size and to compute the multipliers and virtual origin used by subscripted references to the array elements.

8. If the declaration is a member of the structure, it stores the offset units in the c_length field of the reference node temporarily.

9. If the declaration is a structure, it tries to improve the offset units to the best possible unit.

Entry:

get_size

Usage:

```
          declare get_size entry (ptr);

          call get_size ( symbol_ptr );
```

1. symbol_ptr — pointer to the symbol node to be processed by this program. (input/output)

Programs that invoke this entry:

declare
declare_structure
declare_temporary
expand_initial
lang_util_
operator_semantics

Internal Procedures:

addf
: an internal procedure to create an add operator.

multf
: an internal procedure to create a mult operator.

External Variables:

pl1_stat_$eis_mode
pl1_stat_$util_abort

Internal Static Variables:

none

Programs Called:

create_operator
create_statement$prologue
declare_constant$integer
declare_integer
declare_picture
get_array_size

Include Files used:

language_utility
source_id_descriptor
symbol
block
statement
statement_types
reference
token
operator
op_codes
boundary
system

Errors Diagnosed:

Error 414
Error 434
Error 440
Error 457
Error 458
Error 459

NAME: get_array_size

Function:

1. It fills in the element size fields of the array node and expresses them in the best unit.

2. It walks down the bound pairs and construct two multipliers for each bound pair. The descriptor multiplier is used only when the array is accessed as a parameter. It is expressed in bits if the array is packed, and in words if it is unpacked. The other multiplier is used by this procedure and is expressed in the unit given by offset_units.

3. Multipliers are computed by the following rule:

```
m(n)   = element_size
m(n-1) = (hb(n)-lb(n)+1) * m(n)
m(n-2) = (hb(n-1)-lb(n-1)+1) * m(n-1)
   .
   .
   .
m(1)   = (hb(2)-lb(2)+1) * m(2)
```

4. The address of a subscripted element is:

   addr(a(i(1),i(2),...,i(n))) = B - V + (i(1)*m(1) + i(2)*m(2) + ... + i(n)*m(n))

   where

   B = the beginning of storage for the array, that is, the offset of the first element, addr(a(i(lb),i(lb(2)),...,i(lb(n)))

   and

   V = the virtual origin, that is, the offset of the 0th element, addr(a(0,0,...,0))

5. The first multiplier is the element size. It is converted to bits when used as the descriptor multiplier of a packed, array.

6. It loops down the bound pairs and develop the other multiplirs.

7. It creates statements in the prologue sequence if any multiplier is an expression.

8. The last multiplier gives the total size of the array, this total size is recorded in the symbol node.

Entry:

get_array_size

Usage:

```
declare get_array_size entry (ptr);

call get_array_size ( symbol_ptr, offset_unit );
```

1.	symbol_ptr	pointer to the symbol node with the dimensioned attribute. (input)
2.	offset_unit	unit in which the offset is expressed. (input)

Programs that invoke this entry:

get_size

Internal Procedures:

addf
: an internal procedure to create an add operator.

assignf
: an internal procedure to create an assign operator in the prologue sequence.

interleaved
: an internal procedure to distribute the bounds, multipliers, and virtual origins of a dimensional structure onto all its contained members at every level.

multf
an internal procedure to create a mult operator.

subf
an internal procedure to create a sub operator.

virtue
an internal procedure to add a term to the virtual origin.

External Variables:

pl1_stat_$eis_mode
pl1_stat_$util_error

Internal Static Variables:

none

Programs Called:

copy_expression
create_array
create_bound
create_operator
create_statement$prologue
declare_constant$integer
declare_integer
token_to_binary

Include Files used:

language_utility
source_id_descriptor
array

reference
symbol
token
token_types
block
operator
op_codes
statement
statement_types
boundary
nodes
system

Errors Diagnosed:

Error 168

NAME: declare_structure

Function:

1. It scans the structure to determine the boundary, packing, and size required fby each member.

2. It computes the boundary, packing, and size required by the level one structure.

3. It then computes the offset for each member of the structure.

Entry:

declare_structure

Usage:

declare declare_structre entry (ptr);

call declare_structure (symbol_ptr);

1. symbol_ptr pointer to the symbol node to be processed by this program. (input)

Programs that invoke this entry:

declare

Internal Procedures:

get_structure_size
an internal procedure to compute the offset of each structure member, to determine the level one structure size, and to call the internal procedure initialize to initialize each structure member, if necessary.

initialize
an internal procedure to initialize all members of the structure, if necessary.

structure_scan
an internal procedure to propagate the refer_extent, exp_extent, and star_extent bits upward, to determine the boundary required by each structure member, and the packing of the structure.

External Variables:

pl1_stat_$eis_mode

Internal Static Variables:

none

Programs Called:

copy_expression
create_operator
create_statement$prologue
declare_
declare_constant$integer
declare_descriptor
declare_descriptor$param
declare_pointer
expand_initial
get_size
offset_adder
semantic_translator$error

validate

Include Files used:

semant
language_utility
source_id_descriptor
symbol
array
block
reference
operator
statement
op_codes
nodes
statement_types
boundary
list
system

Errors Diagnosed:

Error 210

INITIALIZATION

The declaration processor creates statements in the prologue sequence of the declaring blocks to do the initialization of variables. Variables that require initialization includes file constants, varying strings, areas, in addition to variables with the initial attribute.

NAME: expand_initial

Function:

1. It initializes a file constant by creating an internal static file state block, and a file attribute block.

2. It initializes varying strings to null strings.

3. It initializes areas to "empty".

4. It creates a statement to initialize scalar variables.

5. For array initialization, it creates a subscript. For one dimension arrays, it creates codes to initialize the subscript to zero, increments it, and uses it as a subscript of the array, while the initial values are assigned one by one to the elements of the array.

6. For multi-dimensional arrays, a one dimensional vector whose number of elements is equal to the number of dimensions of the multi-dimensional array is created. Initialization is done in two steps. First the one dimensional array is initialized, then loop and join operators are created to initialize the multi-dimensional array.

Entry:

expand_initial

Usage:

declare expand_initial entry (ptr, ptr, ptr);

call expand_initial (symbol_ptr, statement_ptr, locator_qualifier);

1.	symbol_ptr	pointer to the symbol node to be processed by this program. (input)
2.	statement_ptr	pointer to the statement node or block node containing this declaration. (input)
3.	locator_qualifier	locator qualifier of the expression, if any. (input)

Programs that invoke this entry:

alloc_semantics
declare
declare_structure

Internal Procedures:

addf
: an internal procedure to create an add operator.

assign_initial
: an internal procedure to assign the values of an initial attribute to a vector.

assignf
: an internal procedure to create an assign operator.

link_father
: an internal procedure to create a list node to structure qualify members of the structure.

make_statement
: an internal procedure to create a statement node either in the prologue sequence or in the main sequence of the block.

multf
: an internal procedure to create a mult operator.

subf
an internal procedure to create a sub operator.

External Variables:

none

Internal Static Variables:

none

Programs Called:

copy_expression
create_array
create_bound
create_cross_reference
create_label
create_list
create_operator
create_reference
create_statement
create_statement$prologue
create_symbol
create_token
declare_constant$bit
declare_constant$char
declare_constant$integer
declare_integer
declare_pointer
get_size
semantic_translator$abort
token_to_binary

Include Files used:

semant
language_utility
source_id_descriptor
cross_reference
symbol
boundary
system
label
reference
token
token_types
declare_type
statement
block
statement_types
op_codes
operator
array
list
nodes

Errors Diagnosed:

Error 264
Error 292
Error 442

SECTION V

SEMANTIC TRANSLATION

The semantic translator scans over the internal representation of the program and transforms the internal representation to reflect the attributes declared with each variable. Thus the semantics of the variables will be used by this phase of the compiler to produce a more sophisticated and meaningful internal representation of the program ready for the optimizer and the code generator.

NAME: semantic_translator

Function:

1. It calls the context_processor to process all the context information recorded during the parse.

2. For each block, starting from pl1_stat_$root, going down for its son block and then its brother block, the program performs the following jobs:
   a. It collects all the information necessary to determine whether a block can be quick.
   b. It goes down the chain block.declaration and calls declare to process all the symbols in the chain.
   c. It calls expression_semantics to process all the statements in the main sequence of the block, and then all the statements in the prologue sequence of the block.

3. It goes over the block nodes and determine if they are quick.

Entry:

semantic_translator

Usage:

declare semantic_translator entry;

call semantic_translator;

Programs that invoke this entry:

pl1
v2pl1_semant_

Entry:

semantic_translator$abort

This entry is called when a fatal error occurs in declaration processing or semantic translation. Recovery consists of deleting the offending statement from the program by transforming it into a null statement. Illegal declaration remain in the program. The error message program error_ or error_$no_text is called, and control is transferred to start process the next statement or the next symbol.

Usage:

```
declare semantic_translator$abort entry ( fixed bin(15), ptr );

call semantic_translator$abort ( error_number, error_pointer );
```

1. error_number — error number. (input)
2. error_pointer — pointer to an operand used by the error message program. (input)

Programs that invoke this entry:

alloc_semantics
builtin
declare
defined_reference
do_semantics
expand_assign
expand_infix
expand_initial
expand_primitive
expression_semantics
function
generic_selector

lookup
match_arguments
operator_semantics
semantic_translator
subscripter
v2pl1_semant_

Entry:

semantic_translator$error

This entry is called when a non-fatal error occurs during the semantic translation or declaration processing. The error message program error_ or error_$no_text is called to issue a warning, and control is transferred to continue process the same statement or the same symbol.

Usage:

declare semantic_translator$error entry (fixed bin(15), ptr);

call semantic_translator$error (error_number, error_pointer);

1. error_number — error number. (input)

2. error_pointer — pointer to an operand used by the error message program. (input)

Programs that invoke this entry:

builtin
declare
declare_structure
defined_reference
expression_semantics

function
io_data_list_semantics
io_semantics
semantic_translator
v2pl1_semant_
validate

Entry:

semantic_translator$call_es

This entry is called by prepare_symbol_table in the code generator, when it wants to process an expression hanging off a symbol node.

Usage:

declare semantic_translator$call_es entry (ptr, ptr, ptr, label) returns (ptr);

return_tree = semantic_translator$call_es (cur_block, statement_ptr, input_tree, abort_label);

1.	cur_block	pointer to the block node containing this operand. (input)
2.	statement_ptr	pointer to the statement node containing this operand. (input)
3.	input_tree	pointer to the operand to be processed by this program. (input)
4.	abort_label	the label to be transferred to if this program is aborted for any reason. (input)
5.	return_tree	pointer to the operand returned by this program. (output)

Programs that invoke this entry:

prepare_symbol_table
v2pl1_semant_

Internal Procedures:

process_label
an internal procedure to process all the labels in the label list, and to issue warnings if the previous statement is a goto statement and there are no labels on the current statement.

External Variables:

pl1_stat_$LHS
pl1_stat_$abort_label
pl1_stat_$cur_statement
pl1_stat_$debug_semant
pl1_stat_$error_flag
pl1_stat_$index
pl1_stat_$last_severity
pl1_stat_$multi_type
pl1_stat_$node_uses
pl1_stat_$profile_length
pl1_stat_$quick_pt
pl1_stat_$root
pl1_stat_$st_length
pl1_stat_$st_start
pl1_stat_$statement_id
pl1_stat_$stop_id
pl1_stat_$util_abort
pl1_stat_$util_error

Internal Static Variables:

abort
a label indicating where the control should go if there is a fatal error occuring

	anywhere in the declaration processing of symbols, or the semantic processing of statements.
had_error	a bit indicating if an error has occurred in the processing. It is used only by the semantic_translator$call_es entry.

Programs Called:

context_processor
convert
debug
declare
error
error_
error_$no_text
expression_semantics
ioa_
semantic_translator$abort
semantic_translator$error

Include Files used:

quick_info
semant
language_utility
source_id_descriptor
block
block_types
declare_type
operator
semantic_bits
list
symbol
reference
statement
statement_types
nodes
token
token_types
system

Errors Diagnosed:

Error 56

When an operator is encountered, the attributes of the operands are examined, and from these attributes, the attributes of the result of the operation are derived. The result of an operator is represented in the program as a temporary node. These temporary nodes may be operands of other operators, and the attributes of these temporary nodes may in turn be used to derive the properties of yet other temporary nodes.

Some operators may be modified, and some operators may be changed to a std_call operator to invoke a library routine if the semantics warrants it.

NAME: operator_semantics

Function:

1. It goes down the operator node and extracts the data types from the operands.

2. For most operators, it determines the type, precision, scale of the result, and creates a temporary node to hold the result. It also converts each operand to the appropriate type, precision, and scale in order to produce the result.

3. For the exponentiation operator, it determines from the operands either to pass along the exponentiation operator, or to create a std_call operator to call a libraray subroutine.
   cxp1_
   dcxp1_
   cxp2_
   dcxp2_
   decimal_exp_
   xp22_
   dxp12_
   cxp12_
   dcxp12_

4. Fr the assignment operator, the following steps are taken:
   a. If the right side is a constant, convert it to the type of the left side, unless the left side has no type. Then the right side is converted to the type represented by the constant itself.
   b. If the left side has no type, it is converted to the type of the right side.
   c. If the assignment is to a char(*) or bit(*) return parameter, a statement will be created to make a descriptor for the return parameter.
   d. In certain cases assignments of x=0 are transformed into an operator assign_zero(x).
   f. If the right side is an operator whose output temporary has the same attributes as the left side, replace the temporary with a reference to the left side.
   g. Assignments of a pointer to an offset and vice versa are transformed into off_fun operator or ptr_fun operator.
   h. Area assignment is converted into a call to area_$assign (addr(a1), addr(a2));

5. For the std_call operator, the procedure function will be invoked.

6. For the std_entry operator, a goto statement is created before and a null statement is created after the statement containing the std_entry operator. If any parameter or return value appears in a differenct position in another entry statement, then an assignment statement will be created so that the parameter or return value are made to be qualified by automatic pointers. If the block has multiple return types, an assignment statement is created so that it is possible to determine by means of an automatic integer which entry is invoked. An ex_prologue operator is created with every std_entry operator.

7. For a return_value operator with multiple return values, it is necessary to create a number of statements best illustrated by the following sequence:

```
          if entry_indicator ^= 1 then goto label1;
          entry_1_return_value = return_operand;
          return;
label1:   ;

          :
          :
          :

          if entry_indicator ^= n then goto labeln;
          entry_n_return_value = return_operand;
          return;
labeln:   ;
```

   It is sometimes possible to cause a fatal error by the processing of one of the generated statements, in that case, that statement will be transformed into a signal statement.

8. For input/output operators, the procedure io_semantics will be invoked.

9. For do_fun operators, the procedure do_semantics will be invoked.

10. For allot_based and free_based operators, the procedure alloc_semantics will be invoked.

Entry:

operator_semantics

Usage:

```
declare operator_semantics entry ( ptr, ptr, ptr,
bit(36) aligned ) returns (ptr);
```

```
return_tree = operator_semantics ( block_ptr,
statement_ptr, input_tree, context_bits );
```

1.	block_ptr	pointer to the block node containing this statement. (input)
2.	statement_ptr	pointer to the statement node containing this operator. (input)
3.	input_tree	pointer to the operator node that is to be processed by operator_semantics. (input)
4.	context_bits	bits containing special information about this operator node. (input/output)
5.	return_tree	pointer to the operator node returned by operator_semantics. (output)

Programs that invoke this entry:

alloc_semantics
builtin
do_semantics
expand_infix
expand_prefix
expression_semantics
operator_semantics

Internal Procedures:

convert_relationals
: an internal procedure used to force proper conversions of operands of relational operators.

converter
: an internal procedure used to convert operand(2) and operand(3) of the operator node to their appropriate type.

extract
: an internal procedure used to extract data types and useful pointers of all the operands of the operator node.

make
: an internal procedure used to create an operator node and a statement node, and attach the operator node to the root of the statement node.

prepare
: an internal procedure used to create statements for any expression found in the return_value operator, when there are multiple return types.

print
: an internal procedure used to call the error message program semantic_translator$abort.

External Variables:

pl1_stat_$abort_label
pl1_stat_$cur_statement
pl1_stat_$error_flag
pl1_stat_$multi_type
pl1_stat_$root

Internal Static Variables:

none

Programs Called:

```
alloc_semantics
compare_declaration
convert
convert$to_target
convert$validate
copy_expression
create_label
create_list
create_operator
create_reference
create_statement
create_symbol
create_token
declare
declare_constant
declare_constant$integer
declare_temporary
do_semantics
expand_assign
expression_semantics
free_node
function
get_size
io_semantics
operator_semantics
refer_extent
reserve$declare_lib
semantic_translator$abort
share_expression
```

Include Files used:

```
semant
language_utility
source_id_descriptor
array
symbol
symbol_bits
operator
```

mask
label
list
block
block_types
statement
reference
semantic_bits
op_codes
statement_types
nodes
system
token
token_types
declare_type
decoded_token_types

Errors Diagnosed:

Error 50
Error 51
Error 52
Error 53
Error 78
Error 134
Error 135
Error 180
Error 198
Error 223
Error 227
Error 229
Error 435

OPERAND_PROCESSING

Operands may be constants, or references. References may be simple references, subscripted references, structure qualified references, locator qualified references, or function references. References may further be defined on other references. The semantic translator finds the correct declaration for each variable, builds and processes the length expression, offset expression and qualifier expression for each variable. When these accessing expressions are fully processed, the code generator can produce codes to access the data at runtime.

NAME: expression_semantics

Function:

1. It processes the operator nodes in the following manner:
   a. It calls io_semantics for io opcodes.
   b. It calls format_list_semantics for format opcodes.
   c. It gets the proper pointer for locator qualification for refer and bit_pointer opcodes.
   d. It calls itself recursively to process all the operands of the operator node. After all the operands are processed, it calls operator_semantics to produce the appropriate temporary result. If any of the operands is an aggregate reference or aggregate expression, it will invoke the aggregate package expand_assign, expand_infix or expand_prefix to do further processing of the operator.

2. It processes the token node and the reference nodes in the following manner:
   a. It converts the constants if there are default statements in the block. Otherwise, it leaves the constants alone.
   b. It calls lookup to get the proper symbol node pointer.
   c. If the symbol has the builtin attribute, it calls builtin.
   d. If the symbol has the generic attribute, it calls generic_selector.
   e. It processes the qualifier.
   f. It processes the subscripts. It determines if the reference is a scalar, a cross-section, or an array reference. It calls subscripter to compute the offset. If the symbol has the defined attribute, it calls defined_reference to compute the offset.
   g. It processes the offset field of the reference node.
   h. It processes the length field of the reference node.
   i. If the symbol has the entry attribute, it calls function.
   j. It turns on the aggregate bit in context_bits if the reference is a structure or an array. It then goes through an algorithm to determine whether the LHS_in_RHS bit in the statement node should be turned on.

Entry:

Usage:

```
declare expression_semantics entry ( ptr, ptr, ptr, bit
(36) aligned ) returns (ptr);
```

```
return_tree = expression_semantics ( block_ptr,
statement_ptr, input_tree, context_bits );
```

1.	block_ptr	pointer to the block node containing this statement. (input)
2.	statement_ptr	pointer to the statement node containing this operand. (input)
3.	input_tree	pointer to the operand that is to be processed by expression_semantics. (input)
4.	context_bits	bits containing special information about this operand. (input/output)
5.	return_tree	pointer to the operand returned by expression_semantics. (output)

Programs that invoke this entry:

alloc_semantics
builtin
declare_descriptor
defined_reference
expand_assign
expand_infix
expand_initial
expand_primitive
expression_semantics
function
generic_selector
io_data_list_semantics
io_semantics
operator_semantics

semantic_translator
subscripter
v2pl1_semant_

Internal Procedures:

print
an internal procedure used to call the error message program semantic_translator$abort.

External Variables:

pl1_data$builtin_name
pl1_stat_$LHS
pl1_stat_$index
pl1_stat_$locator
pl1_stat_$root

Internal Static Variables:

none

Programs Called:

builtin
convert
convert$to_integer
convert$to_target
copy_expression
create_cross_reference
create_list
create_operator
create_reference
create_symbol
create_token
declare
defined_reference

expand_assign
expand_infix
expand_prefix
expand_primitive
expression_semantics
format_list_semantics
free_node
function
generic_selector
io_semantics
lookup
operator_semantics
propagate_bit
semantic_translator$abort
semantic_translator$error
share_expression
simplify_offset
subscriptor
validate

Include Files used:

semant
language_utility
source_id_descriptor
block
block_types
builtin_table
cross_reference
declare_type
label
list
nodes
op_codes
operator
reference
semantic_bits
statement
symbol
symbol_bits
system
token
token_types

Errors Diagnosed:

Error 63
Error 64
Error 65
Error 66
Error 67
Error 68
Error 70
Error 71
Error 72
Error 73
Error 77
Error 80
Error 83
Error 102
Error 121
Error 137
Error 145
Error 291

NAME: simplify_offset

Function:

1. It attempts to reduce the precision of the length expression, if possible.

2. It attempts to simplify the offset expression into an expression part and a constant part. The expression part will be stored in reference.offset, and the constant part will be stored in reference.c_offset.

3. The expressions of the form
   constant
   expression + constant
   expression - constant
   constant + expression
   constant1 * constant2
   constant1 * (expression + constant2)
   constant1 * (expression - constant2)
   constant1 * (constant2 + expression)
   will be simplified by this program.

Entry:

simplify_offset

Usage:

```
declare simplify_offset entry ( ptr );

call simplify_offset ( tree );
```

1.	tree	pointer to the reference node whose offset expression and length expression are to be processed by this program. (input)

Programs that invoke this entry:

builtin
expand_primitive
expression_semantics
function

Internal Procedures:

check_addr
: an internal procedure to improve code generated for an unaligned item locator qualified by the addr of another item.

check_char_units
: an internal procedure to ensure unaligned binary numbers or pointers not to have character offset units.

check_exp
: an internal procedure to determine whether an offset expression occurs as part of the length expression.

fb1_const
: an internal procedure to determine if a declaration is a single word fixed binary constant.

fb_value
: an internal procedure to determine if a declaration is a fixed binary real constant or fixed binary real aligned variable.

fix_exp
: an internal procedure to reduce precision of the temporary of an expression to default precision, if possible.

free_exp
: an internal procedure to free the storage for an expression.

free_op
: an internal procedure to free an operator

node.

in_expression
an internal procedure to the internal procedure check_exp to determine whether an expression appears as part of another expression.

External Variables:

none

Internal Static Variables:

none

Programs Called:

compare_expression
convert$to_integer
copy_expression
create_operator
declare_constant$integer
declare_temporary
free_node
share_expression

Include Files used:

semant
language_utility
source_id_descriptor
operator
reference
symbol
array
op_codes

nodes
system
boundary

Errors Diagnosed:

none

NAME: offset_adder

Function:

1. It combines one set of offset with another set of offset.

Entry:

offset_adder

Usage:

declare offset_adder entry (ptr, fixed bin(31), fixed bin(3), ptr, fixed bin(31), fixed bin(3), bit(1));

call offset_adder (offset_1, c_offset_1, unit_of_offset_1, offset_2, c_offset_2, unit_of_offset_2, no_improve_bit);

1.	offset_1	pointer to the first offset expression. (input/output)
2.	c_offset_1	first constant offset. (input/output)
3.	unit_of_offset_1	unit in which offset_1 and c_offset_1 are measured. (input/output)
4.	offset_2	pointer to the second offset expression. (input)
5.	c_offset_2	second constant offset. (input)
6.	unit_of_offset_2	unit in which offset_2 and c_offset_2 are measured. (input)

7. no_improve_bit — bit indicating whether the offsets should be improved to the best unit. (input)

Programs that invoke this entry:

builtin
declare_structure
defined_reference
subscripter

Internal Procedures:

get_ptr
an internal procedure to eliminate the mod_bit and mod_byte operators, and to modifiy the mod_word operator before combining the two offsets.

External Variables:

pl1_stat_$eis_mode

Internal Static Variables:

none

Programs Called:

create_operator
declare_constant$integer
free_node

Include Files used:

semant
language_utility
source_id_descriptor
operator
nodes
op_codes
boundary
system

Errors Diagnosed:

none

NAME: lookup

Function:

1. Given an identifier, it searches through the list of symbol nodes to find the applicable declaration associated with the identifier. This list of symbol nodes are chained first through token.declaration, and thereafter through symbol.multi_use.

2. Fully qualified references are considered applicable.

3. Partially qualified references are considered applicable if no better reference or no other partially qualified references can be found.

4. It creates a cross reference node for the identifier.

Entry:

lookup

Usage:

```
declare lookup entry ( ptr, ptr, ptr, ptr, bit (36) aligned ) returns (bit(1) aligned);
```

```
success_bit = lookup ( block_ptr, statement_ptr, input_tree, symbol_ptr, context_bits );
```

1.	block_ptr	pointer to the block node containing this statement. (input)
2.	statement_ptr	pointer to the statement node conatining this operand. (input)
3.	input_tree	pointer to the operand to be processed by lookup. (input)

4.	symbol_ptr	pointer to the symbol node for the operand. (output)
5.	context_bits	bits containing the special information about this operand. (output)
6.	success_bit	bit indicating if lookup has successfully found the symbol node corresponding to the input tree. (output)

Programs that invoke this entry:

context_processor
declare
defined_reference
expression_semantics
function
prepare_symbol_table
v2pl1_semant_

Internal Procedures:

none

External Variables:

none

Internal Static Variables:

none

Programs Called:

```
create_cross_reference
semantic_translator$abort
```

Include Files used:

```
semant
language_utility
source_id_descriptor
symbol
label
reference
semantic_bits
block
statement
token
list
cross_reference
nodes
```

Errors Diagnosed:

Error 221

NAME: subscripter

Function:

1. It gathers all the subscripts from the subscript list. If the subscript is a constant, it gets its value and ascertain that the constant is within the subscript range. If the subscript is a variable or expression, it converts the result to integer type.

2. If the subscriptrange prefix is on, it creates a bound_ck operator.

3. If all the subscripts are constants, it will yield a constant offset as the partial result, otherwise it will yield an expression offset as the partial result.

4. It calls offset_adder to combine the partial offset with the offset produced by the declaration processor.

Entry:

subscripter

Usage:

```
declare subscripter entry ( ptr, ptr, ptr, ptr, ptr )
returns (ptr);
```

```
return_ptr = subscripter ( cur_block, statement_ptr,
input_tree, subscript_ptr, symbol_ptr );
```

1.	cur_block	pointer to the block node containing this operand. (input)
2.	statement_ptr	pointer to the statement node containing this operand. (input)

3.	input_tree	pointer to the operand to be processed by subscripter. (input)
4.	subscript_ptr	pointer to the list of subscripts. (input)
5.	symbol_ptr	pointer to the symbol node for the operand. (input)
6.	return_tree	pointer to the operand returned by subscripter. (output)

Programs that invoke this entry:

defined_reference
expand_primitive
expression_semantics
function

Internal Procedures:

addf
: an internal procedure to create an add operator.

multf
: an internal procedure to create a mult operator.

print
: an internal procedure used to call the error message program semantic_translator$abort.

subf
: an internal procedure to create a sub operator.

External Variables:

pl1_stat_$eis_mode

Internal Static Variables:

none

Programs Called:

convert$to_integer
copy_expression
create_bound
create_list
create_operator
declare_constant$integer
expression_semantics
offset_adder
semantic_translator$abort
token_to_binary

Include Files used:

semant
language_utility
source_id_descriptor
block
label
symbol
array
reference
statement
list
token
operator
op_codes
boundary
nodes
token_types
declare_type
semantic_bits
system

Errors Diagnosed:

Error 81
Error 82
Error 84
Error 184

NAME: function

Function:

1. It does the semantic processing of all the arguments.

2. It determines whether descriptors are needed for the arguments.

3. It determines whether an argument should be passed by-value or by-reference.

4. It has an algorithm to handle the special case when an argument is a cross section reference.

5. It does the semantic processing of the returns argument.

6. It creates a desc_size operator for the returns argument if necessary.

7. It creates a statement for the std_call operator if the returns parameter has the star_extents and/or the varying attribute.

Entry:

function

Usage:

declare function entry (ptr, ptr, ptr, ptr, bit(36) aligned) returns (ptr);

return_tree = function (cur_block, statement_ptr, input_tree, symbol_ptr, context_bits);

1. cur_block pointer to the block node containing this operand. (input)

2.	statement_ptr	pointer to the statement node containing this operand. (input)
3.	input_tree	pointer to the operand to be processed by function. (input)
4.	symbol_ptr	pointer to the symbol node for the operand. (input)
5.	context_bits	bits containing special information about this operand. (input/output)
6.	return_tree	pointer to the operand returned by this program. (output)

Programs that invoke this entry:

expression_semantics
operator_semantics

Internal Procedures:

print
an internal procedure used to call the error message programs semantic_translator$abort and semantic_translator$error.

prop_bit
an internal procedure used to turn on an attribute bit throughout a structure.

External Variables:

pl1_stat_$node_uses
pl1_stat_$quick_pt

Internal Static Variables:

none

Programs Called:

check_star_extents
copy_expression
create_array
create_bound
create_list
create_operator
create_reference
create_statement
create_symbol
declare_constant$integer
declare_descriptor
declare_temporary
expression_semantics
lookup
match_arguments
semantic_translator$abort
semantic_translator$error
share_expression
simplify_offset
subscripter

Include Files used:

semant
language_utility
source_id_descriptor
array
block
declare_type
list
nodes
op_codes
operator
quick_info
reference
semantic_bits
statement
statement_types
symbol

```
symbol_bits
system
token
token_types


Errors Diagnosed:


Error 47
Error 85
Error 86
Error 88
```

NAME: generic_selector

Function:

1. It does the semantic processing of all the arguments of the generic reference and gets the symbol pointer for each argument.

2. It calls the internal procedure compare_generic for each argument for each corresponding argument selector in every alternative.

3. It selects the proper entry reference when all the arguments match a particular selector.

Entry:

generic_selector

Usage:

declare generic_selector entry (ptr, ptr, ptr, ptr, bit(36) aligned) returns (ptr);

return_tree = generic_selector (cur_block, statement_ptr, input_tree, subscript_ptr, context_bits);

1.	cur_block	pointer to the block node containing this operand. (input)
2.	statement_ptr	pointer to the statement node containing this operand. (input)
3.	input_tree	pointer to the operand to be processed by this program. (input)
4.	subscript_list	pointer to the list of subscripts. (input)

5. context_bits — bits containing special information about this operand. (input)

6. return_tree — pointer to the operand returned by this program. (output)

Programs that invoke this entry:

expression_semantics

Internal Procedures:

compare_generic
an internal procedure to determine if an argument matches the description for a specific argument selector in the generic declaration.

External Variables:

none

Internal Static Variables:

none

Programs Called:

copy_expression
create_operator
create_symbol
expression_semantics
semantic_translator$abort

Include Files used:

semant
language_utility
source_id_descriptor
semantic_bits
list
symbol
reference
token
token_types
nodes
statement
statement_types
operator
op_codes
array
declare_type
picture_image

Errors Diagnosed:

Error 65

NAME: match_arguments

Function:

1. It is called by the procedure function to determine if an argument matches the corresponding parameter description, so that the argument can be passed by-reference instead of by-value.

2. It may call itself recursively if both the argument and the parameter are aggregate references so that lower level mismatches are also taken into consideration.

Entry:

match_arguments

Usage:

```
declare match_arguments entry ( ptr, ptr ) returns
(bit(1) aligned);
```

```
success_bit = match_arguments ( first_ptr, second_ptr
);
```

1.	first_ptr	pointer to the first operand. (input)
2.	second_ptr	pointer to the symbol node of the second operand. (input)
3.	success_bit	bit indicating if the two operands match. (output)

Programs that invoke this entry:

function
match_arguments

Internal Procedures:

none

External Variables:

none

Internal Static Variables:

int_index
number indicating the depth of a structure the program is operating on.

parent_is_scalar
bit indicating if the parent is a scalar.

Programs Called:

compare_expression
match_arguments
semantic_translator$abort

Include Files used:

semant
language_utility
source_id_descriptor
array
nodes
picture_image

reference
symbol

Errors Diagnosed:

Error 269

NAME: make_non_quick

Function:

1. It walks through an expression tree, if it finds a function reference to an internal procedure, it makes the internal procedure non-quick.

Entry:

make_non_quick

Usage:

```
declare make_non_quick entry (ptr) ;

call make_non_quick ( tree );
```

1. tree — pointer to the expression to be processed by this program. (input)

Programs that invoke this entry:

check_star_extents
io_data_list_semantics

Internal Procedures:

none

External Variables:

none

Internal Static Variables:

none

Programs Called:

none

Include Files used:

reference
list
operator
symbol
block
nodes
op_codes

Errors Diagnosed:

none

NAME: builtin

Function:

1. It does the semantics processing of all builtin functions.

2. It checks whether a builtin function is called with an acceptable number of arguments.

3. It processes all the arguments and extracts the data type and the pointer of all the arguments.

4. If an aggregate reference is found among any of the arguments, it determines if the result of the builtin should be an aggregate.

5. It calls expand_arguments, an internal procedure to handle those aggregate builtin references.

6. It checks to make sure whether all the arguments have acceptable data types. converting them if necessary.

7. For individual builtin functions, the work is rather straight forward, it creates either an operator node with the appropriate temporary, or it creates a std_call operator to call a runtime library subroutine.

Entry:

builtin

Usage:

declare builtin entry (ptr, ptr, ptr, ptr, ptr, bit(36) aligned) returns (ptr);

return_tree = builtin (cur_block, statement_ptr, input_tree, subscript_list, builtin_symbol, context_bits);

1.	cur_block	pointer to the block containing this builtin function. (input)
2.	statement_ptr	pointer to the statement node containing this builtin function. (input)
3.	input_tree	pointer to the builtin function to be processed. (input)
4.	subscript_list	pointer to the list of arguments for this builtin function. (input)
5.	builtin_symbol	pointer to the symbol node for this builtin function. (input)
6.	context_bits	bits containing special information for this builtin function. (input/output)
7.	return_tree	pointer to the operand returned by this procedure. (output)

Programs that invoke this entry:

builtin
expression_semantics

Internal Procedures:

check_strings
: an internal procedure to make sure that all the members of the structure used as the argument to the string builtin have the same kind of string.

convert_arg
: an internal procedure to convert an operand to a certain data type.

expand_arguments
: an internal procedure to expand all the aggregate arguments to the builtin function.

make_assignment
an internal procedure to create an operator node and a statement node and to attach the operator node to the root of the statement.

merge
an internal procedure to combine the results of the expanded arguments of the builtin function.

External Variables:

pl1_stat_$builtin_name
pl1_stat_$cur_statement
pl1_stat_$eis_mode

Internal Static Variables:

none

Programs Called:

builtin
check_star_extents
compare_expression
convert
convert$from_builtin
convert$to_integer
convert$to_target
convert$to_target_fb
copy_expression
create_list
create_operator
create_reference
create_statement
create_symbol
create_token
declare
declare_constant
declare_constant$bit

declare_constant$char
declare_constant$integer
declare_descriptor
declare_integer
declare_temporary
defined_reference
expand_assign
expand_infix
expand_primitive
expression_semantics
fill_refer
offset_adder
operator_semantics
propagate_bit
reserve$declare_lib
semantic_translator$abort
semantic_translator$error
share_expression
simplify_offset

Include Files used:

semant
language_utility
source_id_descriptor
array
block
boundary
builtin_table
decoded_token_types
declare_type
label
list
mask
nodes
operator
op_codes
reference
semantic_bits
statement
statement_types
symbol
symbol_bits
system
token
token_types

Errors Diagnosed:

Error 121
Error 122
Error 123
Error 124
Error 126
Error 127
Error 128
Error 131
Error 132
Error 139
Error 141
Error 142
Error 146
Error 147
Error 148
Error 160
Error 167
Error 168
Error 187
Error 188
Error 190
Error 436
Error 437
Error 438

NAME: initialize_builtin

Function:

1. It initializes the external static data block pl1_data$ that contains information about all the builtin functions.

Entry:

initialize_builtin

Usage:

declare initialize_builtin

call initialize_builtin;

Programs that invoke this entry:

none

Internal Procedures:

none

External Variables:

pl1_data_image$builtin_name

Internal Static Variables:

none

Programs Called:

write_list_

Include Files used:

mask
op_codes
system

Errors Diagnosed:

none

NAME: pl1_data

Function:

This data segment contains information of all the builtin functions. For each builtin function, it describes: the name of the builtin function; whether the builtin function will produce an aggregate result if some of its arguments are aggregates; the opcode if the builtin function is to result in an operator; the procedure to invoke if the builtin function is to result in a std_call operator; the label to transfer to in the procedure builtin; the number of arguments expected for the builtin function; and the data type expected of these arguments.

This data segment is used extensively by the procedure builtin.

NAME: reserve

Function:

1. This program maintains a list of names of all the library subroutines that the resulting object program may invoke.

2. It calls reserve$read_lib to create a token node with a specific name.

3. It declares the name as an entry constant.

Entry:

reserve$declare_lib

Usage:

declare reserve$declare_lib entry (fixed bin(15)) returns (ptr) ;

entry_ptr = reserve$declare_lib (subroutine_number);

1.	subroutine_number	number on the reserved list for library subroutines. (input)
2.	entry_ptr	pointer to the reference node representing the entry. (output)

Programs that invoke this entry:

alloc_semantics
builtin
convert_chars
lang_util_

operator_semantics

Entry:

reserve$read_lib

This entry is used to create a token node for a specific library subroutine name.

Usage:

```
declare reserve$read_lib entry ( fixed bin(15) )
returns (ptr) ;

token_ptr = reserve$read_lib ( subroutine_number );
```

1.	subroutine_number	number on the reserved list for library subroutines. (input)
2.	token_ptr	pointer to the token node returned by this program. (output)

Programs that invoke this entry:

compile_link
lang_util_
reserve$declare_lib

Entry:

reserve$clear

This entry clears the renamed_array and the declared_array used in this program.

Usage:

```
declare reserve$clear entry ( ) returns (ptr) ;

null_ptr = reserve$clear ( );
```

1.	null_ptr	null pointer returned by this program. (output)

Programs that invoke this entry:

lang_util_
parse

Entry:

reserve$rename_parse

This entry is used to implement the rename option used in a procedure statement. By this option, the name of a specific library subroutine may be changed.

Usage:

```
declare reserve$rename_parse entry ( fixed bin(15),
bit(1) aligned );

call reserve$rename_parse ( subroutine_number,
success_bit );
```

1.	subroutine_number	number on the reserved list for library subroutines. (input)
2.	success_bit	bit indicating if the renaming step is successful. (output)

Programs that invoke this entry:

lang_util_
process_entry

Internal Procedures:

none

External Variables:

pl1_stat_$root
tree_$

Internal Static Variables:

declared_array	an array of bits to indicate whether a particular library subroutine name has already been declared as an entry constant.
parallel_ptr	an array of pointers used to indicate the new name to use if the particular library subroutine name has been renamed in a rename option.
parallel_ptr_number	a number showing an empty slot in the parallel_ptr array.
renamed_array	an array of bits to indicate whether a particular library subroutine name has

already been renamed in a rename option.

Programs Called:

create_symbol
create_token
parse_error
reserve$read_lib

Include Files used:

language_utility
source_id_descriptor
boundary
declare_type
op_codes
operator
parameter
reference
symbol
system
token
token_list
token_types

Errors Diagnosed:

none

NAME: defined_reference

Function:

1. Given a defined reference node and a subscript list, this procedure determines whether the defined reference is properly declared.

2. It forms the proper offset expression for the defined reference.

Entry:

defined_reference

Usage:

```
declare defined_reference entry ( ptr, ptr, ptr, ptr,
ptr, bit(36) aligned) returns (ptr);
```

```
return_tree = defined_reference ( block_ptr,
statement_ptr, input_tree, subscript_list, symbol_ptr,
context_bits );
```

1.	block_ptr	pointer to the block node containing this statement. (input)
2.	statement_ptr	pointer to the statement node containing this operand. (input)
3.	input_tree	pointer to the operand that is to be processed by program. (input)
4.	subscript_list	pointer to the list of subscripts for this defined reference. (input)
5.	symbol_ptr	pointer to the symbol node of this defined reference. (input)

6. context_bits — bits containing special information about this operand. (input/output)

7. return_tree — pointer to the operand returned by this program. (output)

Programs that invoke this entry:

builtin
expand_primitive
expression_semantics
io_data_list_semantics

Internal Procedures:

find
: an internal procedure to find and replace asterisks and isubs in a subscript list.

find_r
: an internal procedure to find and replace only isubs in a subscript list.

isubs_or_stars
: an internal procedure to find asterisks and isubs in the subscript list of based reference offsets and to replace and form the proper offset expression for the defined reference.

match
: an internal procedure to match the defined item's father against its base to determine the suitability for simple defining or isub defining.

print
: an internal procedure used to call the error message program semantic_translator$abort.

string_overlay
: an internal procedure to determine the suitability of a reference being string

overlayed defining.

External Variables:

pl1_stat_$eis_mode
pl1_stat_$root

Internal Static Variables:

none

Programs Called:

convert
copy_expression
create_operator
create_symbol
decbin
declare
declare_constant$integer
declare_temporary
expression_semantics
lookup
offset_adder
propagate_bit
semantic_translator$abort
semantic_translator$error
subscripter
token_to_binary

Include Files used:

semant
language_utility
source_id_descriptor
symbol
symbol_bits

block
reference
semantic_bits
token
statement
array
list
oO
op_codes
token_types
nodes
system
declare_type
boundary

Errors Diagnosed:

Error 77
Error 81
Error 82
Error 175
Error 176
Error 177
Error 178
Error 179
Error 181
Error 183
Error 185

Special tools are needed to handle aggregate references and aggregate expressions in a pl1 program. Aggregate references and aggregate expressions are recognized by expression_semantics. This information is transmitted back to the caller, who now recognizes that some or all of the operands of an operator are aggregates, and who will invoke expand_assign, expand_infix, or expand_prefix to do the processing depending on whether the operator is an assign operator, an infix operator or a prefix operator.

NAME: expand_assign

Function:

1. This procedure looks at the left side and the right side of the assign operator, and transforms the operator into loop and join operators.

2. If the left side is already a loop operator or join operator, then expand_infix is called to merge the left side and the right side.

3. If the right side is a constant, it is converted into the type it represents.

4. If the LHS_in_RHS bit in the statement node is on, assignment must be done in two steps.

5. If the left side is a temporary with no data type, it is replaced with a temporary whose type and extents are given by the right side.

6. If an optimization can be found, the assignment is transformed into a copy_string or copy_word operator. Otherwise expand_infix is called to merge the left side and the right side.

Entry:

expand_assign

Usage:

```
declare expand_assign entry ( ptr, ptr, ptr, bit (36) aligned, ptr) returns (ptr);

return_tree = expand_assign ( block_ptr, statement_ptr, input_tree, context_bits, aggregate_reference );
```

1.	block_ptr	pointer to the block node containing this statement. (input)
2.	statement_ptr	pointer to the statement node containing this operand. (input)
3.	input_tree	pointer to the operand to be processed by this program. (input)
4.	context_bits	bits containing special information about this operand. (input/output)
5.	aggregate_reference	pointer to the aggregate reference node, sometimes served as the secondary return value. (output)
6.	return_tree	pointer to the operand returned by this program. (output)

Programs that invoke this entry:

builtin
expand_assign
expression_semantics
operator_semantics

Internal Procedures:

declare_expression
: an internal procedure used to create a declaration which represents the result of an aggregate reference.

fill
: an internal procedure of fill_desc used to create assignments to descriptors to individual member or bound of an aggregate reference.

fill_desc
: an internal procedure used to create assignments to descriptors of an aggregate expression when used as a return value.

make_copy
an internal procedure to create a copy_string operator or a copy_word operator.

maker
an internal procedure to create a source like declaration of a temporary.

print
an internal procedure used to call the error message program semantic_translator$abort.

size
an internal procedure to determine the size of a string array temporary.

External Variables:

none

Internal Static Variables:

none

Programs Called:

compare_declaration
convert
copy_expression
create_array
create_bound
create_operator
create_statement
create_symbol
create_token
declare
declare_constant$integer
declare_temporary
expand_assign
expand_infix

```
expression_semantics
refer_extent
semantic_translator$abort
simplify_expression
subscriptor


Include Files used:


semant
language_utility
source_id_descriptor
array
block
boundary
declare_type
decoded_token_types
list
nodes
op_codes
operator
reference
semantic_bits
statement
statement_types
symbol
symbol_bits
system
token
token_types


Errors Diagnosed:


Error 90
Error 91
Error 93
Error 195
```

NAME: expand_prefix

Function:

1. It is used to expand a unary operator when its operand is an aggregate reference, or an aggregate expression.

2. It calls expand_primitive to expand the aggregate reference.

3. It calls an internal procedure to apply the unary operation to each member of the aggregate reference.

Entry:

expand_prefix

Usage:

declare expand_prefix entry (ptr, ptr, ptr) returns (ptr);

return_tree = expand_prefix (block_ptr, statement_ptr, input_tree);

1.	block_ptr	pointer to the block node containing this statement. (input)
2.	statement_ptr	pointer to the statement node containing this operand. (input)
3.	input_tree	pointer to the operand to be processed by this program. (input)
4.	return_tree	pointer to the operand returned by this program. (output)

Programs that invoke this entry:

expression_semantics

Internal Procedures:

apply_prefix
: an internal procedure to apply the unary operation to each member of the aggregate expression. A call is made to operator_semantics to process the unary operators thus formed.

External Variables:

none

Internal Static Variables:

none

Programs Called:

create_operator
expand_primitive
operator_semantics

Include Files used:

semant
language_utility
source_id_descriptor
operator

semantic_bits
op_codes
nodes

Errors Diagnosed:

none

NAME: expand_infix

Function:

1. It is used to expand an infix operator when some of its operands are aggregate references or aggregate expressions.

2. It calls expand_primitive to expand any aggregate reference.

3. It calls an internal procedure to locally optimize any scalar expression found in any operand.

4. It calls the internal procedure walk or match to apply the binary operation to the expanded operands.

Entry:

expand_infix

Usage:

declare expand_infix entry (ptr, ptr, ptr) returns (ptr);

return_tree = expand_infix (block_ptr, statement_ptr, input_tree);

1.	block_ptr	pointer to the block node containing this statement. (input)
2.	statement_ptr	pointer to the statement node containing this operand. (input)
3.	input_tree	pointer to the operand to be processed by this program. (input)
4.	return_tree	pointer to the operand returned by this program. (output)

Programs that invoke this entry:

builtin
expand_assign
expression_semantics

Internal Procedures:

match
: an internal procedure to match the expanded parts of aggregate references and to combine them.

simplify_scalar
: an internal procedure to extract scalar subexpressions so that it is evaluated only once outside the loop.

walk
: an internal procedure to walk down the loop and join operator of one aggregate reference and to apply the binary operation to the expanded member and a scalar.

External Variables:

pl1_stat_$LHS

Internal Static Variables:

none

Programs Called:

compare_expression
create_operator

```
create_statement
create_symbol
declare_temporary
expand_primitive
expression_semantics
operator_semantics
semantic_translator$abort
share_expression


Include Files used:


semant
language_utility
source_id_descriptor
declare_type
nodes
op_codes
operator
reference
semantic_bits
statement
statement_types
symbol
system


Errors Diagnosed:


Error 79
```

NAME: expand_primitive

Function:

1. It determines from the subscript list the number of additional subscripts that needs be created.

2. It calls the internal procedure expander to do the expansion.

3. Depending on the declaration of the aggregate reference, it returns a series of loop and join operator to represent the expansion of the aggregate reference.

Entry:

expand_primitive

Usage:

declare expand_primitive entry (ptr, ptr, ptr) returns (ptr);

return_tree = expand_primitive (block_ptr, statement_ptr, input_tree);

1.	block_ptr	pointer to the block node containing this statement. (input)
2.	statement_ptr	pointer to the statement node containing this operand. (input)
3.	input_tree	pointer to the operand to be processed by this program. (input)
4.	return_tree	pointer to the operand returned by this program. (output)

Programs that invoke this entry:

builtin
expand_infix
expand_prefix
expression_semantics

Internal Procedures:

addf
: an internal procedure to create an add operator.

bit_ptr
: an internal procedure to search and replace the bit_pointer operator with the proper locator qualifier.

declare_index
: an internal procedure to declare a control index of the form "s.n" used in the loop operators.

expander
: an internal procedure to create a join operator for structure reference, and to create a loop operator for array reference. It may call itself recursively if the sublevel member of an aggregate reference is again an aggregate reference.

make_loop
: an internal procedure to create a loop operator.

process_subscripted_reference
: an internal procedure to do the semantics processing of a scalar subscripted reference produced by the expansion of the aggregate reference.

subf
: an internal procedure to create a sub operator.

External Variables:

none

Internal Static Variables:

none

Programs Called:

```
bindec$vs
copy_expression
create_bound
create_list
create_operator
create_reference
create_symbol
create_token
declare
declare_constant$integer
declare_temporary
defined_reference
expression_semantics
refer_extent
semantic_translator$abort
share_expression
simplify_expression
simplify_offset
subscripter
```

Include Files used:

```
semant
language_utility
source_id_descriptor
array
declare_type
label
list
```

nodes
op_codes
operator
reference
semantic_bits
symbol
system
token
token_types

Errors Diagnosed:

Error 81

NAME: simplify_expression

Function:

1. It walks through the expression and simplify all constant expressions of the form:
   constant1 + constant2
   constant1 - constant2
   constant1 * constant2

Entry:

simplify_expression

Usage:

declare simplify_expression (ptr, fixed bin, bit(1) aligned) returns (ptr);

return_tree = simplify_expression (input_tree, constant_value, modified_bit);

1.	input_tree	pointer to the expression to be simplified. (input/output)
2.	constant_value	value of the expression if the entire expression can be reduced to a constant. (output)
3.	modified_bit	bit indicating if the entire expression is reduced to a constant. (output)
4.	return_tree	pointer to the modified expression. (output)

Programs that invoke this entry:

expand_assign
expand_primitive

Internal Procedures:

none

External Variables:

none

Internal Static Variables:

none

Programs Called:

declare_constant$integer

Include Files used:

language_utility
source_id_descriptor
nodes
op_codes
operator
reference
symbol
system

Errors Diagnosed:

none

SPECIAL_STATEMENTS

Certain operators representing allocate statements, do statements, or input/output statements undergo considerable modifications. Many new statements and operators may be created to fully implement their meaning.

NAME: alloc_semantics

Function:

1. It transforms the allot_based and free_based operators into calls to the runtime routines alloc_, alloc_$storage, and freen_.

2. If the allocation reference has the control attribute, the allot_based operator is transformed into the allot_ctl operator and the free_based operator is transformed into the free_ctl operator.

3. If the set reference is an unaligned pointer or an offset, statements will be created after or before the call to do the conversion between the different data types.

4. If the allocation reference is an aggregate reference with refer_extents, statements will be created to assign the expression value to the refer reference in the refer option.

5. If the allocation reference has the initial attribute, the procedure expand_initial will be invoked to do the initialization of the based allocated reference.

Entry:

alloc_semantics

Usage:

declare alloc_semantics entry (ptr, ptr, ptr);

call alloc_semantics (block_ptr, statement_ptr, input_tree);

1. block_ptr pointer to the block node containing this statement. (input)

2.	statement_ptr	pointer to the statement node containing this operand. (input)
3.	input_tree	pointer to the operand to be processed by this program. (input/output)

Programs that invoke this entry:

operator_semantics

Entry:

alloc_semantics$init_only

This entry is called by io_semantics in the processing of a locate statement.

Usage:

```
declare alloc_semantics$init_only entry ( ptr, ptr, ptr
);
```

```
call alloc_semantics$init_only ( locator,
statement_ptr, input_tree );
```

1.	locator	locator qualifier of the allocation reference. (input)
2.	statement_ptr	pointer to the statement node containing this operand. (input)
3.	input_tree	pointer to the operand to be processed by this program. (input)

Programs that invoke this entry:

io_semantics

Internal Procedures:

build_assignment
: an internal procedure to create statements to assign expression values to the refer reference in a refer option.

getsize
: an internal procedure to get the number of storage words to be allocated or freed.

External Variables:

none

Internal Static Variables:

none

Programs Called:

copy_expression
create_list
create_operator
create_reference
create_statement
create_symbol
declare_constant$integer
declare_descriptor$ctl
declare_pointer
declare_temporary
expand_initial

```
expression_semantics
operator_semantics
propagate_bit
refer_extent
reserve$declare_lib
semantic_translator$abort
share_expression


Include Files used:


semant
language_utility
source_id_descriptor
array
boundary
list
nodes
operator
op_codes
reference
semantic_bits
statement
statement_types
symbol
symbol_bits
system


Errors Diagnosed:


Error 114
Error 115
Error 116
Error 117
Error 118
```

NAME: do_semantics

Function:

1. It does the semantics processing of the do statement.

2. If the control variable of the do statement is locator qualified, subscript qualified, or has length expressions, these qualifiers will be extracted out of the do loop to prevent their values from being reset accidentally.

3. Depending on the existence of to-clause, by-clause, repeat-clause, and while-clause in the do-specification, statements will be created to represent their logic.

4. If the do statement is a multiple specification do loop, a label variable will be created to control the flow of logic.

Entry:

do_semantics

Usage:

declare do_semantics entry (ptr, ptr, ptr);

call do_semantics (block_ptr, statement_ptr, input_tree);

1.	block_ptr	pointer to the block node containing this statement. (input)
2.	statement_ptr	pointer to the statement node containing this operand. (input)
3.	input_tree	pointer to the operand that is to be processed by program. (input/output)

Programs that invoke this entry:

operator_semantics

Internal Procedures:

copy_ref
: an internal procedure to determine whether a reference should be shared.

make_operator
: an internal procedure to create an operator node.

make_statement
: an internal procedure to create a statement node.

External Variables:

none

Internal Static Variables:

none

Programs Called:

copy_expression
create_label
create_list
create_operator
create_reference
create_statement
create_symbol
create_token

declare_integer
declare_pointer
free_node
operator_semantics
semantic_translator$abort
share_expression

Include Files used:

semant
language_utility
source_id_descriptor
block
declare_type
label
list
nodes
operator
op_codes
reference
semantic_bits
statement
statement_types
symbol
system
token
token_types

Errors Diagnosed:

Error 140
Error 143
Error 144

NAME: io_semantics

Function:

1. io_semantics handles both the major io operators compiled by the parse as the root nodes of I/O statements, and the minor io operators (transmission operators), provided, for the most part, by io_data_list_semantics in the compilation of the data lists of get and put statements. For the handling of transmission operators, see step 15, below.

2. The parse attaches operands of two types to io operators: reference and expression operands from the various options of io statements, attached in canonical positions known to parse and semantics alike; and a special final operand which is, in effect, a 36 bit bitstring. This last operand has a bit position for every option and statement type recognizable by the parse; the bits are set to describe the particular statement observed by the parse and serve importantly to drive the compilation of the statement by io_semantics (see step 13).

3. The design of the compiled procedure provides that I/O statements are almost entirely executed out-of-line by the PL/1 runtime I/O routines, PLIO. The work of io_semantics is, then, to provide for suitable invocations of PLIO and to provide for the transfer of information between the compiled procedure and PLIO. The general design of the compiled procedure is as follows:

   a. each block containing an I/O statement or format statement is non-quick; that is, it has a stack frame distinct from that of its parent block (if any).

   b. each stack frame corresponding to a block containing an I/O statement has a workspace, PS, reserved in it for use by PLIO during the execution of any I/O statement and for passing information from the compiled procedure to PLIO (and sometimes back again).

   c. the location of this workspace is known to PL/1 operators by a convention between the code generator and the PL/1 operators; all invocations of PLIO are accomplished by PL/1 operator invocations - rather than by full PL/1 calls - the PL/1 operators pass a single argument to PLIO in every case, namely the address of PS.

Accordingly, the compiled procedure must, either by direct code or with the help of the PL/1 operators, store into PS (and, occasionally, elsewhere) all information which the invoked entry in PLIO will require to complete its work.

4. The work of io_semantics thus consists chiefly of compiling assignments to PS and invocations of PLIO. This is accomplished by the creation of assignment operators and of special io-operators which the code generator compiles into invocations of "transfer vector" entries in PL/1 operators. Certain of the jobs of assigning to PS are done by the code generator as part of its work in compiling the special io-operators. Some of the information that appears in PS is constant through the life of the stack frame containing the PS (for example, the stack frame pointer, the runtime symbol table pointer) and is put into PS by code supplied by the code generator on its own motion (see PLM for io_op) rather than as the compilation of operators generated by io_semantics.

5. Assignment to or from PS is tricky, an anomaly in the compiler. Although at runtime PS is a structure containing pointers, integers, character and bit strings, a label, etc., at compile time PS is simply an unstructured "storage block". Assignment to PS makes use of the fact that the code generator will in effect take unspec of the object being assigned and will put it, as a bit string, at a position in PS depending on the offset relative to PS. It is thus necessary to convert the object being assigned into the exact form which it will have in PS prior to assigning it to PS. Extracting information from PS (as with the string returned for a KEYTO option) requires use of a defined reference whose qualifier points at the right spot in PS.

6. A source_io_statement is compiled into a list of statements, as follows:

   a. a labelled null statement (if the source statement was labelled);

   b. an assignment statement whose root is a join operator all but the last of whose operands are assignment operators each of which assigns an argument to its proper place in PS (or the like); and the last operand of which invokes PLIO (to do preparatory work in the stream case, to do the main work in the record case.);

c. in the case of most get and put statements, a list of statements implementing the implied DO's in the LIST, EDIT, or DATA option and having transmission operators for each scalar list item;

d. in the case of a read statement with a keyto option, a statement to assign to the target of the keyto option;

e. in many cases a null statement to which PLIO is to pass control if the remainder of the statement's execution is to be aborted.

7. So that arguments to PLIO may be stored in the proper form and in the proper place (chiefly in PLIO), io_semantics maintains an "assign-list", of length "lal", which is a list of assign operators each of which makes such an assignment. The operators in this list are created by an internal procedure of io_semantics, assign_ps, which creates the operator and in many cases inserts conversion operators or operators to create a pointer to a given argument.

8. The io operator is processed as follows. First, if the io statement is labelled, a null statement is inserted after the labelled statement, and the root nodes of the labelled and null statements are interchanged so that a labelled null statement precedes an unlabelled io statement.

9. The length, "lal", of the "assign-list" is initialized to zero and the existence of PS is provided for. The last operand of the io-operator is converted to a 36 bit bitstring item, "job", which shows the options processed by the parse. Additional bits will be set in "job" by io_semantics and "job" will be passed to PLIO via PS where it will be interpretted as specifying the work to be done at runtime.

10. If a DATA, LIST, or EDIT option appears, io_data_list_semantics is called. This has the effect of appending statements after the io statement, statements which implement implied DO's, transmissions of all list elements, and the establishment of format lists. For a "get data" statement, no statements are created. Instead, a get_data_trans operator is compiled containing the list of allowed targets (a list of zero length for the source statement "get data;"); the code generator will translate this operator into a constant list of runtime symbol table offsets and the address of this constant list will be put into PS.

11. If the io statement is a LOCATE statement, then the reference in the statement is checked for conformance to the language, the pointer to be SET is established, and the size of the generation to be located is computed and assigned in PS.

    References to the variable to be allocated and to the pointer to be set are preserved in the local variables "locate_var" and "locate_set", respectively. The unprocessed reference to the pointer (if it appears) and to the variable to be allocated are removed from among the operands of the io operator.

12. The oerands now attached to the io operator are processed by expression_semantics. They are processed as, and are required to turn out to be, scalar, except in the two cases: the operands for the FROM and INTO options.

13. The bits of the "job" (see step 2) now drive the further processing of the operands of the io operator, the presence of the i-th bit of "job" causing the code at the label "action(i)" to be invoked relative to the appropriate operand of the io operator. In most cases the work of the code so invoked is to check the semantic correctness of the program element and then cause one or more assignments to PS (or the like) to be compiled and put on the "assign-list" (see step 7) by calling the internal procedure "assign_ps". Most of these actions require no documentation here. A few special actions will be considered.

    INTO , FROM (actions 25,27): Storage in PS of the address and bitlength of the generation appearing in the option would suffice but for two points. First, the compiler's addressing of varying strings and of arrays of varying strings has to be considered. The compiler will take the address of the first data word (i.e., the second word) of a varying string or array of varying strings; and will calculate the bitlength only on the data portion of a scalar varying string. Accordingly, bit(3) of "job" is set to indicate that the generation is varying and bit(35) of "job" is set to indicate a varying array. Second, as an optimization in consideration of the fact that the runtime I/O mechanism expects byte-aligned and byte-lengthed generations of storage, the compiler will set bit(34) of "job" to indicate that byte-alignment and byte-length of the generation must be checked at runtime; if the byte-alignment and byte-length can be assured at compile time, then this bit will not be set.

KEYTO (action 22): The reference in the KEYTO option (the keyto target) is checked to see that it is a character string reference (pseudovariables not being allowed). An assignment statement is created before the statement following the read statement. This assignment statement will pick up the value obtained at runtime and assign it to the keyto target. A labelled null statement is then created before the statement originally following the read statement (and, thus, after the assignment statement) whose label is assigned to PS as an abnormal return label.

OPEN (action 34): A structure, FAB2, is created in the stack frame to receive the attributes specified in the open statement. A template is created to initialize FAB2; constants for title, pagesize, and linesize are written into the template. An assignment of the template to FAB2 is placed in the "assign-list". Assignments of variable values for title, pagesize, and linesize are compiled and placed into the "assign-list". An assignment of the address of FAB2 to PS is compiled and placed in the "assign-list".

14. After the "job"-dictated actions are done, the "job" word is corrected for use at runtime and placed in the record_io or stream_prep operator, if any. An assignmant statement is created before the current statement (which has been made a null statement) to which is attached a join operator joining the operators in the "assign-list".

15. The transmission operators (see step 1), as originally created by data_list_parse and as transformed by io_data_list_semantics, are of three kinds.

    The get_data_trans operator has as its single operand a join of the references appearing in the list of the get data statement. This operator is not processed further by io_semantics.

    The put_data_trans operator is received by io_semantics with one operand, a reference containing a subscript list. io_semantics moves this subscript list to the first operand position of the put_data_trans operator. The code generator will make the runtime symbol table offset for the reference and the evaluated subscript values available at runtime.

    The remaining transmission operators, get_list_trans, put_list_trans, get_edit_trans, and put_edit_trans, are treated as a class. To each is attached a descriptor valued expression whose value describes the item being transmitted (this item is always scalar at this point, aggregates having

been expanded by expand_prefix - see io_data_list_semantics). This descriptor is a trivially determined constant in the cases of numeric or pictured items, but may be complicated in the case of string items which may be adjustable, have refer extents, etc.

Entry:

io_semantics

Usage:

declare io_semantics entry (ptr, ptr, ptr);

call io_semantics (block_ptr, statement_ptr, input_tree);

1. block_ptr — pointer to the block node containing this statement. (input)

2. statement_ptr — pointer to the statement node containing this operand. (input)

3. input_tree — pointer to the operand that is to be processed by program. (input)

Programs that invoke this entry:

expression_semantics
operator_semantics

Internal Procedures:

assign_ps
an internal procedure whose princial use is the assignment with coercive conversion of

some element to PS. It has been extended to do addressing and to assign to storage blocks other than PS.

io_semantics_util
a dummy entry point, never called.

io_semantics_util$keys
an internal procedure to extend the size of PS to 48 + 65 words long to accommodate the new key, which is declared as char(256) varying. It also sets list.element(50) to the defined new key, whose qualifier is PS|48.

io_semantics_util$make_fa
an internal procedure to create a work space of 122 words to store the format stack in the use of a "get edit" or "put edit" statement.

io_semantics_util$make_fab2
an internal procedure to create a work space of 14 words to accommodate the title option, page size, and line size in an open statement.

io_semantics_util$make_ffsb
an internal procedure to create a fake FSB block for the use of string option in a get statement or put statement.

io_semantics_util$make_ps
an internal procedure to create a 48 word work space for the PS used by all io statements.

io_semantics_util$make_ssl
an internal procedure to create a work space for the subscript list used in a "put data" statement.

External Variables:

pl1_stat_$generate_symtab

Internal Static Variables:

none

Programs Called:

alloc_semantics$init_only
convert
convert$to_target
copy_expression
create_label
create_list
create_operator
create_reference
create_statement
create_symbol
create_token
declare
declare_constant
declare_constant$bit
declare_constant$integer
declare_descripter
declare_temporary
expression_semantics
io_data_list_semantics
ioa_
operator_semantics
propagate_bit
refer_extent
semantic_translator$error
share_expression

Include Files used:

semant
language_utility
source_id_descriptor
nodes
block
list
operator
op_codes

semantic_bits
symbol
array
system
reference
token
token_types
statement
statement_types
declare_type
label
ps_map
symbol_bits
boundary

Errors Diagnosed:

Error 62
Error 114
Error 115
Error 461
Error 462
Error 263
Error 464
Error 465
Error 466
Error 467
Error 468
Error 471
Error 472
Error 474
Error 475

NAME: io_data_list_semantics

Function:

1. It processes the data list of a stream-io statement.

2. It turns on the set bit of the symbol node for an item in a get statement data list.

3. It turns on the get_data bit in the block node for "get data;" or "put data;" statements.

4. Items in a "get data" statement data list will be put on the pl1_stat_$ok_list;

5. Items in a "put data" statement data list will have their symbol.put_in_symtab bit turned on.

6. It calls the internal procedure io_join_semantics to process the items on the data list of a "get/put list/edit" statement.

7. It calls the entry format_list_semantics to process the format list in a "get edit" or "put edit" statement.

Entry:

io_data_list_semantics

Usage:

declare io_data_list_semantics entry (ptr, ptr, ptr);

call io_data_list_semantics (block_ptr, statement_ptr, input_tree);

1. block_ptr pointer to the block node containing this statement. (input)

2. statement_ptr — pointer to the statement node containing this operand. (input)

3. input_tree — pointer to the operand to be processed by this program. (input/output)

Programs that invoke this entry:

io_semantics

Entry:

format_list_semantics

It processes the format list of a format statement, or the format list in "get edit" or "put edit" statements. It may call itself recursively to process format items and format lists.

Usage:

```
declare format_list_semantics entry ( ptr, ptr, ptr  );
```

```
call  format_list_semantics ( block_ptr, statement_ptr, input_tree );
```

1. block_ptr — pointer to the block node containing this statement. (input)

2. statement_ptr — pointer to the statement node containing this operand. (input)

3. input_tree — pointer to the operand to be processed by this program. (input)

Programs that invoke this entry:

expression_semantics
io_data_list_semantics

Internal Procedures:

down
: an internal procedure to turn on the set bit and the put_in_symtab bit in the symbol node and all the lower level members.

io_join_semantics
: The internal procedure io_join_semantics processes an item list, which may contain simple items such as references and expressions as well as complex items - implied do groups -, by creating statements and inserting them just before the statement that originally followed the io statement being compiled. These statements control the do-groups and contain the transmission operators which io_semantics later processes. The join seen by io_join_semantics contains simple items and/or do_fun operators (corresponding to implied do groups). io_join_semantics collects the maximum number of consecutive simple items, replaces each with the appropriate transmission operator containing the simple item, and creates a statement whose root node is a join containing these transmission operators (if there are more than one) or containing the transmission operator itself (if there is exactly one). Each do_fun operator is processed by the creation of a do statement containing, as its root, the do_fun operator, next followed by the result of invoking io_join_semantics recursively to process the item list associated with the do_fun operator, and finally followed by a labelled null statement whose label is associated with the do statement as if it were the associated end statement.

label_of_statement
an internal procedure to create a label to attach to a null statement created by this program.

walk
an internal procedure to turn on the set bit and the put_in_symtab bit in the symbol node and all its fathers and sons and brothers.

External Variables:

pl1_stat_$ok_list

Internal Static Variables:

none

Programs Called:

convert$to_target
create_label
create_list
create_operator
create_statement
create_symbol
declare_constant$integer
declare_temporary
defined_reference
expression_semantics
format_list_semantics
make_non_quick
semantic_translator$abort

Include Files used:

semant
language_utility
source_id_descriptor
nodes
system
mask
reference
block
token
token_types
semantic_bits
symbol
declare_type
label
list
op_codes
operator
statement
statement_types
ps_map

Errors Diagnosed:

Error 170
Error 171
Error 469
Error 470
Error 473

SECTION VIII

AN_OVERVIEW

The procedures described in this section deals with many of the utility functions not limited to use by any phase of the compiler.

The scheme used for the allocation and freeing of the nodes used by the compiler is simple. When a node is needed, it is allocated in the tree_$ segment -- sometimes in the xeq_tree_$ segment. When a node is to be freed, generally no action is taken. But because of the frequency of allocating and freeing certain nodes like the operator node (2 or 3 operands), list node (2 or 3 elements), reference node, and statement node, a pool is maintained to keep track of the freed nodes. On subsequent allocation of the same type of node, this pool is examined for the existence of a freed and reuseable node before attempting to allocate a fresh node in the tree_$ segment (or xeq_tree_$ segment).

NAME: create_block

Function:

1. It creates and initializes a block node.

Entry:

create_block

Usage:

```
declare create_block entry  (  bit(9)  aligned,  ptr  )
returns (ptr) ;

block_ptr = create_block ( block_type, father_block_ptr
);
```

1.	block_type	type of block node to be created. (input)
2.	father_block_ptr	pointer to the block node containing this block. (input)
3.	block_ptr	pointer to the block node returned by this program. (output)

Programs that invoke this entry:

code_generator
lang_util_
on_parse
parse
prepare_symbol_table
procedure_parse

Internal Procedures:

none

External Variables:

pl1_stat_$node_uses
pl1_stat_$statement_id

Internal Static Variables:

none

Programs Called:

pl1_get
tree_$

Include Files used:

rename
block
block_types
nodes

Errors Diagnosed:

none

NAME: create_statement

Function:

1. It creates and initializes a statement node.

Entry:

create_statement

Usage:

declare create_statement entry (bit(9) aligned, ptr, ptr, bit(12) aligned) returns (ptr);

statement_ptr = create_statement (statement_type, father_ptr, label_ptr, conditions);

1.	statement_type	type of statement to be created. (input)
2.	father_ptr	either a pointer to the block node containing this statement, or a pointer to the statement node preceding this statement. (input)
3.	label_ptr	pointer to the list of labels for this statement. (input)
4.	conditions	conditions for this statement. (input)
5.	statement_ptr	pointer to the statement node created by this program. (output)

Programs that invoke this entry:

alloc_semantics
builtin
code_generator
declare_descriptor
declare_parse
default_parse
do_parse
do_semantics
expand_assign
expand_infix
expand_initial
function
if_parse
io_data_list_semantics
io_semantics
io_statement_parse
lang_util_
on_parse
operator_semantics
prepare_symbol_table
procedure_parse
process_entry
statement_parse
statement_recognizer

Entry:

create_statement$prologue

This entry is used to create a statement node in the prologue sequence instead of the main sequence of the block.

Usage:

declare create_statement$prologue entry (bit(9) aligned, ptr, ptr, bit(12) aligned) returns (ptr);

```
          statement_ptr     =     create_statement$prologue     (
statement_type, father_ptr, label_ptr, conditions );


1.  statement_type            type of statement  to  be  created.
                              (input)

2.  father_ptr                either a pointer to the block  node
                              containing  this  statement,  or  a
                              pointer  to  the   statement   node
                              preceding  this statement.  (input)

3.  label_ptr                 pointer to the list of  labels  for
                              this statement.  (input)

4.  conditions                conditions  for   this   statement.
                              (input)

5.  statement_ptr             pointer  to  the   statement   node
                              created  by this program.  (output)


Programs that invoke this entry:


declare
declare_descriptor
declare_structure
expand_initial
get_array_size
get_size
lang_util_


Internal Procedures:


          none


External Variables:


pl1_stat_$cur_statement
pl1_stat_$tree_ptr
pl1_stat_$node_uses



DRAFT: SUBJECT TO CHANGE      8-295                 order number
```

pl1_stat_$source_seg
pl1_stat_$st_length
pl1_stat_$st_start
pl1_stat_$statement_id
tree_$

Internal Static Variables:

none

Programs Called:

pl1_get
xeq_tree_$

Include Files used:

rename_xeq
token_list
label
reference
list
statement
block
nodes
statement_types

Errors Diagnosed:

none

NAME: create_operator

Function:

1. It creates and initializes an operator node.

Entry:

create_operator

Usage:

```
declare create_operator entry ( bit(9) aligned, fixed
bin(15) ) returns (ptr);

operator_ptr = create_operator ( op_code, arg_number );
```

1.	op_code	operator code for this operator. (input)
2.	arg_number	number of arguments for this operator. (input)
3.	operator_ptr	pointer to the operator node created by this program. (output)

Programs that invoke this entry:

alloc_semantics
attribute_parse
builtin
convert
copy_expression
data_list_parse
declare

```
declare_descriptor
declare_structure
defined_reference
do_parse
do_semantics
evaluate
expand_assign
expand_infix
expand_initial
expand_prefix
expand_primitive
expression_parse
expression_semantics
format_list_parse
function
generic_selector
get_array_size
get_size
if_parse
io_data_list_semantics
io_semantics
io_statement_parse
lang_util_
offset_adder
on_parse
operator_semantics
prepare_symbol_table
procedure_parse
process_entry
reference_parse
simplify_offset
statement_parse
subscripter


Internal Procedures:

          none


External Variables:


pl1_stat_$tree_ptr
pl1_stat_$node_uses
```

Internal Static Variables:

none

Programs Called:

pl1_get
xeq_tree_$

Include Files used:

rename_xeq
operator
nodes

Errors Diagnosed:

none

NAME: create_reference

Function:

1. It creates and initializes a reference node.

Entry:

create_reference

Usage:

```
declare create_reference entry ( ptr )  returns  (ptr);

reference_ptr = create_reference ( token_ptr );
```

1.	token_ptr	pointer to the token node or symbol node for this reference (input)
2.	reference_ptr	pointer to the reference node created by this program. (output)

Programs that invoke this entry:

alloc_semantics
builtin
code_generator
copy_expression
declare_descriptor
do_semantics
expand_initial
expand_primitive
expression_semantics
fill_refer
function

get_reference
io_semantics
lang_util_
operator_semantics
refer_extent
reference_parse
share_expression
statement_parse
statement_type

Entry:

create_reference$for_symbol

This entry is called so that the reference node created will be allocated in the xeq_tree_ segment instead of the tree_ segment.

Usage:

declare create_reference entry (ptr) returns (ptr) ;

reference_ptr = create_reference (token_ptr);

1.	token_ptr	pointer to the token node or symbol node for this reference. (input)
2.	reference_ptr	pointer to the reference node created by this program. (output)

Programs that invoke this entry:

create_symbol

Internal Procedures:

none

External Variables:

pl1_stat_$free_ptr
pl1_stat_$node_uses
xeq_tree_$

Internal Static Variables:

none

Programs Called:

pl1_get
tree_$

Include Files used:

rename
nodes
reference

Errors Diagnosed:

none

NAME: create_token

Function:

1. It prepares to create a token node for the token represented by the given string.

2. It tries to find the token node in the hash table.

3. If it succeeds, it returns the pointer to the token node found.

4. If it fails, it creates a new token node, puts the pointer in the appropriate slot in the hash table, and returns.

Entry:

create_token

Usage:

```
declare create_token entry ( char (*) aligned, bit (9) aligned ) returns (ptr) ;

token_ptr = create_token ( token_string, token_type );
```

1.	token_string	string for which the token is made. (input)
2.	token_type	type of token to be created. (input)
3.	token_ptr	pointer to the token node returned by this program. (output)

Programs that invoke this entry:

attribute_parse
builtin
convert
create_identifier
declare_parse
descriptor_parse
do_semantics
evaluate
expand_assign
expand_initial
expand_primitive
expression_parse
expression_semantics
initialize_int_static
io_semantics
io_statement_parse
lang_util_
lex
merge_attributes
on_parse
operator_semantics
parse
process_entry
reserve
statement_type

Internal Procedures:

none

External Variables:

pl1_stat_$hash_table
pl1_stat_$node_uses

Internal Static Variables:

none

Programs Called:

pl1_get
tree_$

Include Files used:

rename
nodes
token
create_token

Errors Diagnosed:

none

NAME: create_symbol

Function:

1. It creates and initializes a symbol node.

Entry:

create_symbol

Usage:

```
declare  create_symbol entry ( ptr, ptr, bit(3) aligned
) returns (ptr);

symbol_ptr  =  create_symbol  (  block_ptr,  token_ptr,
create_type );
```

1.	block_ptr	pointer to the block node containing this symbol. (input)
2.	token_ptr	pointer to the token node for which the symbol node is created. (input)
3.	create_type	bits indicating whether the symbol node is created by declaration, by context, by implication, or by the compiler. (input)
4.	symbol_ptr	pointer to the symbol node returned by this program. (output)

Programs that invoke this entry:

```
alloc_semantics
builtin
context_processor
copy_expression
declare_constant
declare_descriptor
declare_integer
declare_parse
declare_pointer
declare_temporary
default_parse
defined_reference
descriptor_parse
do_semantics
expand_assign
expand_infix
expand_initial
expand_primitive
expression_semantics
format_list_parse
function
generate_constant
generic_selector
get_variable
io_data_list_semantics
io_semantics
io_statement_parse
lang_util_
on_parse
operator_semantics
process_entry
reference_parse
reserve
statement_parse


Internal Procedures:

          none


External Variables:


pl1_stat_$free_ptr
pl1_stat_$node_uses



DRAFT: SUBJECT TO CHANGE      8-307                order number
```

tree_$

Internal Static Variables:

none

Programs Called:

create_identifier
create_reference$for_symbol
pl1_get
tree_$

Include Files used:

rename
symbol
block
token_list
nodes

Errors Diagnosed:

none

NAME: create_context

Function:

1. It creates and initializes a context node.

Entry:

create_context

Usage:

```
declare create_context entry ( ptr, ptr ) returns
(ptr);

context_ptr = create_context ( block_ptr, token_ptr );
```

1.	block_ptr	pointer to the block node containging this token. (input)
2.	token_ptr	pointer to the token node for which the context is to be recorded. (input)
3.	context_ptr	pointer to the context node returned by this program. (output)

Programs that invoke this entry:

context
lang_util_

Internal Procedures:

none

External Variables:

pl1_stat_$node_uses

Internal Static Variables:

none

Programs Called:

pl1_get
tree_$

Include Files used:

rename
context
nodes
block

Errors Diagnosed:

none

NAME: create_array

Function:

1. It creates and initializes an array node.

Entry:

create_array

Usage:

declare create_array entry ( ) returns (ptr);

array_ptr = create_array ( );

1. array_ptr pointer to the array node returned by this program. (output)

Programs that invoke this entry:

attribute_parse
copy_expression
expand_assign
expand_initial
function
get_array_size
lang_util_

Internal Procedures:

none

External Variables:

pl1_stat_$node_uses

Internal Static Variables:

none

Programs Called:

pl1_get
tree_$

Include Files used:

rename
nodes
array

Errors Diagnosed:

none

NAME: create_bound

Function:

1. It creates and initializes a bound node.

Entry:

create_bound

Usage:

```
declare create_array entry ( ) returns (ptr);

array_ptr = create_array ( );
```

1. array_ptr pointer to the array node returned by this program. (output)

Programs that invoke this entry:

attribute_parse
copy_expression
expand_assign
expand_initial
expand_primitive
function
get_array_size
lang_util_
subscripter

Internal Procedures:

none

External Variables:

pl1_stat_$node_uses

Internal Static Variables:

none

Programs Called:

pl1_get
tree_$

Include Files used:

rename
nodes
array

Errors Diagnosed:

none

NAME: create_list

Function:

1. It creates and initializes a list node.

Entry:

create_list

Usage:

```
declare  create_list  entry  (  fixed bin(15) ) returns
(ptr);

list_ptr = create_list ( number );
```

1.	number	number of elements for this list node. (input)
2.	list_ptr	pointer to the list node returned by this program. (output)

Programs that invoke this entry:

alloc_semantics
assign_storage
attribute_parse
builtin
check_o_and_s
compile_entry
compile_statement
convert_chars
copy_expression
declare

descriptor_parse
do_parse
do_semantics
expand_initial
expand_primitive
expression_semantics
function
gen_pl1_linkage
get_reference
if_parse
io_data_list_semantics
io_semantics
lang_util_
mst
name_assign
on_parse
operator_semantics
optimizer
process_entry
reference_parse
statement_parse
statement_type
subscripter

Internal Procedures:

none

External Variables:

pl1_stat_$free_ptr
pl1_stat_$node_uses

Internal Static Variables:

none

Programs Called:

pl1_get
xeq_tree_$

Include Files used:

rename
nodes
list

Errors Diagnosed:

none

NAME: create_default

Function:

1. It creates and initializes a default node.

Entry:

create_default

Usage:

```
declare create_default entry ( ) returns (ptr);

default_ptr = create_default ( );
```

1. default_ptr pointer to the default node created by this program. (output)

Programs that invoke this entry:

default_parse
lang_util_

Internal Procedures:

none

External Variables:

pl1_stat_$node_uses

Internal Static Variables:

none

Programs Called:

pl1_get
tree_$

Include Files used:

rename
default
nodes

Errors Diagnosed:

none

NAME: create_label

Function:

1. It creates and initializes a label node.

Entry:

create_label

Usage:

```
declare  create_label  entry ( ptr, ptr, bit(3) aligned
);

label_ptr = ( block_ptr, token_ptr, create_type );
```

1.	block_ptr	pointer to the block node containing this label. (input)
2.	token_ptr	pointer to the token node for which the label node is created. (input)
3.	create_type	bits indicating whether the label node is created by declaration, by context, by implication, or by the compiler. (input)
4.	label_ptr	pointer to the label node returned by this program. (output)

Programs that invoke this entry:

code_generator
compile_block

compile_statement
compile_tree
convert_chars
declare_label
do_parse
do_semantics
expand_initial
if_parse
io_data_list_semantics
io_semantics
lang_util_
operator_semantics
set_indicators

Internal Procedures:

none

External Variables:

pl1_stat_$node_uses
pl1_stat_$statement_id
tree_$

Internal Static Variables:

none

Programs Called:

create_identifier
pl1_get
tree_$

Include Files used:

rename
nodes
block
label
token_list
token

Errors Diagnosed:

none

NAME: create_cross_reference

Function:

1. It creates and initializes a cross_reference node.

Entry:

create_cross_reference

Usage:

```
declare create_cross_reference entry ( ) returns (ptr);

cross_reference_ptr = create_cross_reference ( );
```

1. cross_reference_ptr pointer to the cross_reference node returned by this program. (output)

Programs that invoke this entry:

expand_initial
expression_semantics
lang_util_
lookup
process_entry

Internal Procedures:

none

External Variables:

pl1_stat_$node_uses

Internal Static Variables:

none

Programs Called:

pl1_get
tree_$

Include Files used:

rename
cross_reference
nodes

Errors Diagnosed:

none

NAME: create_identifier

Function:

1. It fabricates a compiler-created unique name.

2. It creates a token node for that name.

Entry:

create_identifier

Usage:

```
declare create_identifier entry ( ) returns (ptr);

token_ptr = create_identifier ( );
```

1. token_ptr pointer to the token node returned by this program. (output)

Programs that invoke this entry:

create_label
create_symbol
lang_util_

Internal Procedures:

none

External Variables:

pl1_stat_$compiler_created_index

Internal Static Variables:

none

Programs Called:

bindec$vs
create_token

Include Files used:

token_types

Errors Diagnosed:

none

NAME: create_storage

Function:

1. It allocates a block of words.

Entry:

create_storage

Usage:

declare create_storage entry (fixed bin) returns (ptr);

storage_ptr = create_storage (number);

1.	number	number of words to be allocated. (input)
2.	storage_ptr	pointer to the block of storage returned by this program. (output)

Programs that invoke this entry:

declare_constant
lang_util_

Internal Procedures:

none

External Variables:

pl1_stat_$node_uses

Internal Static Variables:

none

Programs Called:

pl1_get
tree_$

Include Files used:

rename

Errors Diagnosed:

none

NAME: pl1_get

Function:

1. It calls tree_manager$get_free to get a free area.

Entry:

pl1_get

Usage:

```
declare  pl1_get  entry  ( fixed bin(15), ptr ) returns
(ptr);

return_ptr = pl1_get ( size, area_ptr );
```

1.	size	number of words to be allocated. (input)
2.	area_ptr	pointer to the area inside which space is to be allocated. (input)
3.	return_ptr	pointer to the space just allocated. (output)

Programs that invoke this entry:

assign_storage
cg_error
compile_formats
copy_temp
create_array
create_block
create_bound

create_context
create_cross_reference
create_default
create_label
create_list
create_operator
create_reference
create_statement
create_storage
create_symbol
create_token
e_v
generate_constant
lang_util_
lex
mst
pl1_signal_catcher
stack_temp
state_man

Entry:

pl1_put

This entry is used for freeing an area. But currently this entry does nothing.

Usage:

```
declare pl1_put entry;

call pl1_put;
```

Programs that invoke this entry:

none

Internal Procedures:

none

External Variables:

none

Internal Static Variables:

none

Programs Called:

none

Include Files used:

none

Errors Diagnosed:

none

NAME: tree_manager

Function:

1. It manages the use of multiple free storage segments used by the compiler during compilation.

Entry:

tree_manager$init

It creates a tree_$ segment, and a xeq_tree_$ segment.

Usage:

```
declare tree_manager$init entry ( label );

call tree_manager$init ( abort_label );
```

1.	abort_label	label indicating where the transfer is to go if all the storage space is exhausted. (input)

Programs that invoke this entry:

lang_util_
v2pl1

Entry:

tree_manager$truncate

It truncates the tree_$ segment as well as the xeq_tree_$ segment.

Usage:

```
declare tree_manager$truncate entry ( );

call tree_manager$truncate;
```

Programs that invoke this entry:

lang_util_
v2pl1

Entry:

tree_manager$get_free

This entry makes a call to the Multics system routine hcs_$make_seg to allocate a free segment in the process directory of the user.

Usage:

```
declare tree_manager$get_free entry ( fixed bin(24), ptr, ptr );

call tree_manager$get_free ( size, area_ptr, unused_ptr );
```

1.	size	number of words to be allocated. (input)
2.	area_ptr	pointer to the area inside which space is to be allocated. (input)
3.	unused_ptr	dummy argument, currently not being used for any purpose.

Programs that invoke this entry:

lang_util_

Internal Procedures:

none

External Variables:

pl1_stat_$condition_index
pl1_stat_$free_ptr
pl1_stat_$root
pl1_stat_$source_list_ptr
pl1_stat_$tree_vec_index
tree_$tree_

Internal Static Variables:

abort_label	label indicating where the transfer is to go if all the free segments are exhausted.
tree_vec	array of pointers to the free segments it has allocated.
xeq_ptr	pointer to the xeq_tree area it has allocated.

Programs Called:

hcs_$make_seg
hcs_$truncate_seg
ioa_

Include Files used:

source_list

Errors Diagnosed:

none

NAME: free_node

Function:

1. Given a pointer to a node, it will determine the type of node to be freed.

2. If the node is an operator node, a list node, a reference node, or a symbol node, the node will be saved on a free-list. Future creations of the same type of node can pick it up from the free-list, without having to allocate a new node.

Entry:

free_node

Usage:

```
declare free_node entry ( ptr );

call free_node ( node_ptr );
```

1. node_ptr — pointer to the node to be freed by this program. (input)

Programs that invoke this entry:

declare_parse
default_parse
do_parse
do_semantics
expression_semantics
format_list_parse
lang_util_

offset_adder
on_parse
operator_semantics
optimizer
prepare_symbol_table
simplify_offset

Internal Procedures:

          none

External Variables:

pl1_stat_$free_ptr

Internal Static Variables:

          none

Programs Called:

          none

Include Files used:

rename
nodes
symbol
token
block
statement
reference
array
list

context
label
operator

Errors Diagnosed:

none

It is often necessary for the compiler to declare a constant or a variable with some specific attributes to facilitate the processing of other references and expressions. This function is accomplished by the following procedures.

NAME: declare_integer

Function:

1. It creates a symbol node and makes a fixed binary real declaration of default precision and automatic storage class.

Entry:

declare_integer

Usage:

```
declare declare_integer entry ( ptr );

return_ptr = declare_integer ( block_ptr );
```

1.	block_ptr	pointer to the block node for which the integer is declared. (input)
2.	return_ptr	pointer to the reference node representing the integer declared by this program. (output)

Programs that invoke this entry:

builtin
declare
do_semantics
expand_initial
get_array_size
get_size
lang_util_

Internal Procedures:

none

External Variables:

Internal Static Variables:

none

Programs Called:

create_symbol

Include Files used:

language_utility
source_id_descriptor
boundary
declare_type
symbol
system

Errors Diagnosed:

none

NAME: declare_pointer

Function:

1. It creates a symbol node and makes apointer declare_constantl withautomatic storage class.

Entry:

declare_pointer

Usage:

declare declare_pointer

return_ptr = declare_pointer (block_ptr);

1.	block_ptr	pointer to the block node for which the pointer is declare. (input)
2.	return_ptr	pointer to the reference node representing the pointer declared by this program. (output)

Programs that invoke this entry:

alloc_semantics
declare
declare_descriptor
declare_structure
do_semantics
expand_initial
lang_util_
prepare_symbol_table

Internal Procedures:

none

External Variables:

none

Internal Static Variables:

none

Programs Called:

create_symbol

Include Files used:

language_utility
source_id_descriptor
boundary
declare_type
symbol
system

Errors Diagnosed:

none

NAME: declare_temporary

Function:

1. It searches through the list of temporary nodes already created for an identical declaration. If the search is successful, that temporary will be returned.

2. If the search fails, it creates a symbol node and makes a new declaration with temporary storage class.

3. The new temporary node created witll be chained onto the list of temporary nodes.

Entry:

declare_temporary

Usage:

```
declare declare_temporary entry ( bit(36) aligned,
fixed bin(31), fixed bin(15), ptr ) returns (ptr) ;
```

```
return_ptr = declare_temporary ( data_type, precision,
scale, length );
```

1.	data_type	data type of the temporary. (input)
2.	precision	precision of the temporary if the data type is arithmetic, otherwise the string length. (input)
3.	scale	scale of the temporary if the data type is arithmetic. (input)
4.	length	length expression of the string if the data type is a string. (input)

5.	return_ptr	pointer to the reference node representing the temporary. (output)

Programs that invoke this entry:

alloc_semantics
builtin
convert
decimal_op
declare_descriptor
defined_reference
expand_assign
expand_infix
expand_primitive
function
io_data_list_semantics
io_semantics
lang_util_
operator_semantics
prepare_symbol_table
simplify_offset

Internal Procedures:

none

External Variables:

pl1_stat_$temporary_list

Internal Static Variables:

none

Programs Called:

create_symbol
get_size

Include Files used:

language_utility
source_id_descriptor
symbol
boundary
mask
reference
declare_type

Errors Diagnosed:

none

NAME: declare_label

Function:

1. Given a list of labels, this program will get the token representing each label.

2. For each token, if a declaration has already been made, it will check if the attributes are consistent. For constant label arrays, it will update the high bound and the low bound.

3. If no declaration has been made, a label node will be created.

Entry:

declare_label

Usage:

declare declare_label entry (ptr, ptr, ptr, bit(3) aligned);

call declare_label (block_ptr, statement_ptr, label_ptr, declare_type);

1.	block_ptr	pointer to the block node containing this label. (input)
2.	statement_ptr	pointer to the statement node containing this label. (input)
3.	label_ptr	pointer to the list node of labels. (input)
4.	declare_type	bits indicating whether the declare is by context, by implicating, or

by the compiler.

Programs that invoke this entry:

declare_parse
default_parse
do_parse
if_parse
io_statement_parse
on_parse
procedure_parse
statement_parse

Internal Procedures:

none

External Variables:

none

Internal Static Variables:

none

Programs Called:

create_label
parse_error
token_to_binary

Include Files used:

language_utility
source_id_descriptor
block
label
list
nodes
reference
token

Errors Diagnosed:

Error 31
Error 54

NAME: declare_descriptor

Function:

1. It creates a descriptor for the argument of a call.

2. It determines if the descriptor has already been made for the argument.

3. It creates a parameter descriptor pointer if necessary.

Entry:

declare_descriptor

Usage:

```
declare declare_descriptor entry ( ptr, ptr, ptr,  ptr,
bit(1) aligned ) returns (ptr) ;
```

```
descriptor_ptr   =   declare_descriptor   (  block_ptr,
statement_ptr,          symbol_ptr,          locator_qualifier,
array_descriptor_bit );
```

1.	block_ptr	pointer to the block node containing this declaration. (input)
2.	statement_ptr	pointer to the statement node containing this operand. (input)
3.	symbol_ptr	pointer to the symbol node for which the descriptor is to be made. (input)
4.	locator_qualifier	locator qualifier expression for this operand. (input)

5. array_descriptor_bit — bit indicating if an array descriptor is required. (input)

6. descriptor_ptr — pointer to the descriptor created by this program. (output)

Programs that invoke this entry:

builtin
declare
declare_structure
function
io_semantics
lang_util_

Entry:

declare_descriptor$ctl

This special entry point is used to make assignments to controlled descriptors at allocation time.

Usage:

```
declare  declare_descriptor$ctl  entry ( ptr, ptr, ptr, ptr, bit(1) aligned ) returns (ptr) ;
```

```
descriptor_ptr =  declare_descriptor$ctl  (  block_ptr, statement_ptr,           symbol_ptr,           locator_qualifier, array_descriptor_bit );
```

1. block_ptr — pointer to the block node containing this declaration. (input)

2. statement_ptr — pointer to the statement node containing this operand. (input)

3. symbol_ptr — pointer to the symbol node for which the descriptor is to be made. (input)

4. locator_qualifier — locator qualifier expression for this operand. (input)

5. array_descriptor_bit — bit indicating if an array descriptor is required. (input)

6. descriptor_ptr — pointer to the descriptor created by this program. (output)

Programs that invoke this entry:

alloc_semantics
lang_util_

Entry:

declare_descriptor$param

This entry point is used to indicate that all the extents and bounds have already been computed by get_size.

Usage:

```
declare declare_descriptor$param entry ( ptr, ptr, ptr,
ptr, bit(1) aligned ) returns (ptr) ;
```

```
descriptor_ptr = declare_descriptor$param (  block_ptr,
statement_ptr,          symbol_ptr,          locator_qualifier,
array_descriptor_bit );
```

1. block_ptr — pointer to the block node containing this declaration. (input)

2.	statement_ptr	pointer to the statement node containing this operand. (input)
3.	symbol_ptr	pointer to the symbol node for which the descriptor is to be made. (input)
4.	locator_qualifier	locator qualifier expression for this operand. (input)
5.	array_descriptor_bit	bit indicating if an array descriptor is required. (input)
6.	descriptor_ptr	pointer to the descriptor created by this program. (output)

Programs that invoke this entry:

declare_
declare_structure
lang_util_

Internal Procedures:

assignf
: an internal procedure to create a statement for the assignment to the descriptor.

assignm
: an internal procedure to create a statement for generating multiplier assignments to controlled descriptors.

builder
: an internal procedure to build a descriptor from the symbol node.

copy
: an internal procedure to call copy_expression for a reference node if the reference node has offset expression, length expression, or qualifier expression.

set_star
                    an internal procedure to propagate the
                    star_extents bit upward.

External Variables:

pl1_stat_$util_abort

Internal Static Variables:

          none

Programs Called:

copy_expression
create_operator
create_reference
create_statement
create_statement$prologue
create_symbol
declare_constant$desc
declare_constant$integer
declare_pointer
declare_temporary
expression_semantics
refer_extent
token_to_binary

Include Files used:

semant
language_utility
source_id_descriptor
semantic_bits
symbol
array
reference

statement
block
operator
statement_types
op_codes
system
declare_type
boundary
nodes
token
token_types

Errors Diagnosed:

Error 28
Error 29

NAME: declare_picture

Function:

1. It calls picture_info_ to ascertain that the picture string is valid.

2. It fills in the attributes of the picture as determined by picture_info_.

3. It declares the picture_constant, and puts it in symbol.general.

Entry:

declare_picture

Usage:

declare declare_picture entry (char(*) aligned, ptr, fixed bin(15));

call declare_picture (picture_string, symbol_ptr, error_code);

1.	picture_string	character string representing the picture. (input)
2.	symbol_ptr	pointer to the symbol node with the picture attribute. (input)
3.	error_code	error number returned by picture_info_. (output)

Programs that invoke this entry:

format_list_parse
get_size
lang_util_

Internal Procedures:

none

External Variables:

none

Internal Static Variables:

none

Programs Called:

declare_constant$bit
picture_info_

Include Files used:

language_utility
source_id_descriptor
picutre_constant
picutre_image
picutre_types
reference
symbol

Errors Diagnosed:

none

NAME: declare_constant

Function:

1. It computes the boundary requirement and the bit size needed to declare the constant.

2. It searches throught the chain of constants to find a constant with the same value.

3. It creates a new symbol node to represent the value if no other constant has the same value.

4. If another constant value can be found but with different attributes, then an equivalence declaration will be made.

5. The new constant will be linked to the constant chain.

Entry:

declare_constant

Usage:

declare declare_constant entry (bit(*) aligned, bit(36) aligned, fixed bin(31), fixed bin(15)) returns (ptr) ;

return_ptr = declare_constant (value, data_type, precision, scale);

1.	value	value of the constant to be declared. (input)
2.	data_type	data type of the constant. (input)
3.	precision	precision of the constant if the data type is arithmetic, otherwise the string length. (input)

4. scale — scale of the constant if the data type is fixed. (input)

5. return_ptr — pointer to the reference node representing the constant declared by this program. (output)

Programs that invoke this entry:

builtin
convert
declare_constant
io_semantics
lang_util_
operator_semantics

Entry:

declare_constant$bit

This entry is used to declare a bit string constant.

Usage:

```
declare declare_constant$bit entry ( bit(*)  aligned  )
returns (ptr) ;

return_ptr = declare_constant$bit ( bit_string );
```

1. bit_string — bit string value of the constant to be declared. (input)

2. return_ptr — pointer to the reference node representing the bit constant declared by this program. (output)

Programs that invoke this entry:

builtin
declare_picture
expand_initial
io_semantics
lang_util_

Entry:

declare_constant$char

This entry is used to declare a character string constant.

Usage:

```
declare declare_constant$char entry ( char(*) aligned )
returns (ptr) ;

return_ptr = declare_constant$char ( char_string );
```

1.	char_string	char string value of the constant to be declared. (input)
2.	return_ptr	pointer to the reference node representing the character constant declared by this program. (output)

Programs that invoke this entry:

builtin
declare
expand_initial
lang_util_

Entry:

declare_constant$desc

This entry is used to declare a constant descriptor.

Usage:

```
declare  declare_constant$desc entry ( bit(*) aligned )
returns (ptr) ;

return_ptr = declare_constant$desc ( desc_bit_string );
```

1.	desc_bit_string	bit string value of the descriptor constant to be declared. (input)
2.	return_ptr	pointer to the reference node representing the descriptor constant declared by this program. (output)

Programs that invoke this entry:

declare_descriptor
lang_util_

Entry:

declare_constant$integer

This entry is used to declare a fixed binary constant.

Usage:

```
          declare declare_constant$integer entry ( fixed  bin(31)
) returns (ptr) ;


          return_ptr = declare_constant$integer ( value );
```

1.	value	value of the integer constant to be declared. (input)
2.	return_ptr	pointer to the reference node representing the integer constant declared by this program. (output)

Programs that invoke this entry:

alloc_semantics
builtin
declare
declare_descriptor
declare_structure
defined_reference
expand_assign
expand_initial
expand_primitive
function
get_array_size
get_size
io_data_list_semantics
io_semantics
lang_util_
offset_adder
operator_semantics
prepare_symbol_table
simplify_offset
subscripter

Internal Procedures:

none

External Variables:

pl1_stat_$constant_list

Internal Static Variables:

none

Programs Called:

create_storage
create_symbol
declare_constant

Include Files used:

language_utility
source_id_descriptor
symbol
reference
system
boundary
declare_type

Errors Diagnosed:

none

CONVERSION_PROGRAMS

Conversion between data types is done by the following utility procedures.

NAME: convert

Function:

1. It gets the input type, input precision, and input scale.

2. It gets the output type; and output precision and output scale if possible.

3. It checks the validity of this attempted conversion.

4. If the input and the output have identical data types, no conversion is done.

5. If the input is not a constant, an assign operator will be created, so that conversion will be done at run time.

6. If the input is a constant, conversion is done at compile time.

Entry:

convert

Usage:

```
declare  convert entry ( ptr, bit(36) aligned ) returns
(ptr) ;

return_ptr = convert ( input_tree, target_type );
```

1.	input_tree	operand to be converted by this program. (input)
2.	target_type	data type to which the operand is to be converted. (input)

3. return_ptr pointer to the result returned by this program. (output)

Programs that invoke this entry:

builtin
defined_reference
expand_assign
expression_semantics
io_semantics
lang_util_
operator_semantics
semantic_translator

Entry:

convert$from_builtin

This entry is used to suppress warning diagnostics that may normally be given, because the user does an explicit conversion using a builtin function.

Usage:

declare convert$from_builtin entry (ptr, bit(36) aligned) returns (ptr) ;

return_ptr = convert$from_builtin (input_tree, target_type);

1. input_tree operand to be converted by this program. (input)

2. target_type data type to which the operand is to be converted. (input)

3. return_ptr pointer to the result returned by this program. (output)

Programs that invoke this entry:

builtin
lang_util_

Entry:

convert$to_integer

This entry is used to convert an operand to a fixed binary integer value with no scale factors.

Usage:

```
declare convert$to_integer entry ( ptr, bit(36) aligned
) returns (ptr) ;

return_ptr = convert$to_integer ( input_tree,
target_type );
```

1. input_tree operand to be converted by this program. (input)
2. target_type data type to which the operand is to be converted. (input)
3. return_ptr pointer to the result returned by this program. (output)

Programs that invoke this entry:

builtin
expression_semantics
lang_util_
simplify_offset
subscripter

Entry:

convert$to_target

This entry is used to convert an operand to the data type, precision, scale or length specified by a target declaration.

Usage:

declare convert$to_target entry (ptr, ptr) returns (ptr) ;

return_ptr = convert$to_target (input_tree, target_reference);

1.	input_tree	operand to be converted by this program. (input)
2.	target_reference	pointer to the target reference node. (input)
3.	return_ptr	pointer to the result returned by this program. (output)

Programs that invoke this entry:

builtin
expression_semantics
io_data_list_semantics
io_semantics

lang_util_
operator_semantics

Entry:

convert$to_target_fb

This entry is used to suppress warning diagnostics that may normally be given when an operand is converted to the data type, precision, scale or length specified by the target declaration because the user does an explicit conversion using a builtin function.

Usage:

```
declare convert$to_target_fb entry ( ptr, ptr ) returns (ptr) ;

return_ptr = convert$to_target_fb ( input_tree, target_reference );
```

1.	input_tree	operand to be converted by this program. (input)
2.	target_reference	pointer to the target reference node. (input)
3.	return_ptr	pointer to the result returned by this program. (output)

Programs that invoke this entry:

builtin
lang_util_

Entry:

convert$validate

This entry is used to find out whether two sides of an assign operator is compatible.

Usage:

```
declare convert$validate entry ( ptr, ptr ) returns (ptr) ;

return_ptr = convert$validate ( input_tree, target_reference );
```

1.	input_tree	operand to be converted by this program. (input)
2.	target_reference	pointer to the target reference node. (input)
3.	return_ptr	pointer to the result returned by this program. (output)

Programs that invoke this entry:

lang_util_
operator_semantics

Internal Procedures:

ceil
an internal procedure to perform the ceiling function.

desc_type
an internal procedure to convert the data type and precision into a descriptor type code.

get_target_size
an internal procedure to compute the output precision, scale and length, when the input type, input precision, scale, length and output type is known.

print
an internal procedure to call the error message program pl1_stat_$util_abort or pl1_stat_$util_error.

External Variables:

pl1_stat_$util_abort
pl1_stat_$util_error

Internal Static Variables:

none

Programs Called:

assign_
char_to_numeric_
create_operator
create_token
declare_constant
declare_temporary
share_expression

Include Files used:

language_utility
source_id_descriptor
declare_type
desc_dcls
desc_types
mask
nodes
op_codes
operator
reference
symbol
system
token
token_types

Errors Diagnosed:

Error 223
Error 224
Error 225
Error 226
Error 227
Error 228
Error 229
Error 230
Error 231
Error 232
Error 233
Error 234
Error 235
Error 236
Error 246
Error 248
Error 249
Error 250
Error 251
Error 252
Error 253
Error 443

NAME: bindec

Function:

1. It converts a fixed binary number to a fixed decimal number.

Entry:

bindec

Usage:

declare bindec entry (fixed bin) returns (char(12) aligned);

character_result = bindec (binary_number);

1.	binary_number	binary number to be converted. (input)
2.	character_result	decimal result expressed in characters. (output)

Programs that invoke this entry:

display_pl1_map
display_pl1_text
evaluate
lang_util_
lex
pl1_print
v2pl1

Entry:

bindec$vs

This entry is used to return a varying character string instead of a nonvarying character string.

Usage:

declare bindec$vs entry (fixed bin) returns (char(12) varying);

character_result = bindec$vs (binary_number);

1.	binary_number	binary number to be converted. (input)
2.	character_result	decimal result expressed in characters. (output)

Programs that invoke this entry:

cg_error
create_identifier
decode_node_id
descriptor_parse
display_pl1_text
display_text
expand_primitive
lang_util_
lex
on_parse
pl1_error_print
prepare_symbol_map_

Internal Procedures:

none

External Variables:

none

Internal Static Variables:

none

Programs Called:

none

Include Files used:

eis_bits
eis_micro_ops

Errors Diagnosed:

none

NAME: binoct

Function:

1. It converts a bit string to an octal string.

Entry:

binoct

Usage:

```
declare  binoct  entry  (  bit(36)  aligned ) returns (
char(12) aligned );

character_result = binoct ( bit_string );
```

1.	bit_string	bit string to be converted. (input)
2.	character_result	octal result expressed in characters. (output)

Programs that invoke this entry:

display_pl1_map
display_text
lang_util_
pl1_symbol_print

Internal Procedures:

none

External Variables:

none

Internal Static Variables:

none

Programs Called:

none

Include Files used:

none

Errors Diagnosed:

none

NAME: binary_to_octal_string

Function:

1. It converts a fixed binary constant into a octal string.

Entry:

binary_to_octal_string

Usage:

declare binary_to_octal_string entry (fixed bin, char(12) aligned);

call binary_to_octal_string (integer, octal_string);

1.	integer	fixed binary constant to be converted. (input)
2.	octal_string	character string representation of the octal value. (output)

Programs that invoke this entry:

none

Entry:

binary_to_octal_var_string

This entry returns a varying octal string instead of a nonvarying octal string.

Usage:

```
declare binary_to_octal_var_string entry ( fixed bin, char(12) varying );

call binary_to_octal_var_string ( integer, octal_var_string );
```

1.	integer	fixed binary constant to be convertd. (input)
2.	octal_string	character string representation of the octal value. (output)

Programs that invoke this entry:

pl1_error_print

Internal Procedures:

none

External Variables:

none

Internal Static Variables:

none

Programs Called:

none

Include Files used:

none

Errors Diagnosed:

none

NAME: decbin

Function:

1. It converts a character string representing a signed or unsigned decimal constant to a fixed binary value.

Entry:

decbin

Usage:

```
declare decbin entry ( char(*) aligned ) returns ( fixed bin );

value = decbin ( decimal_string );
```

1.	decimal_string	character string representing a signed or unsigned decimal constant. (input)
2.	value	value returned by this program. (output)

Programs that invoke this entry:

defined_reference
lang_util_

Internal Procedures:

none

External Variables:

none

Internal Static Variables:

none

Programs Called:

none

Include Files used:

none

Errors Diagnosed:

none

NAME: token_to_binary

Function:

1. It gets the value of a constant token node.

Entry:

token_to_binary

Usage:

```
declare token_to_binary entry ( ptr ) returns ( fixed
bin );

value = token_to_binary ( token_ptr );
```

1.	token_ptr	pointer to the token node to be converted. (input)
2.	value	value returned by this program. (output)

Programs that invoke this entry:

attribute_parse
declare_descriptor
declare_label
declare_parse
defined_reference
descriptor_parse
evaluate
expand_initial
get_array_size
initialize_int_static

lang_util_
lex
subscriptre
validate

Internal Procedures:

none

External Variables:

none

Internal Static Variables:

none

Programs Called:

none

Include Files used:

none

Errors Diagnosed:

none

The following procedures duplicates expressions or nodes so that the same expression or reference may be used or shared by different parts of the program.

NAME: copy_expression

Function:

1. It duplicates a node and its components.

Entry:

copy_expression

Usage:

```
declare copy_expression entry ( ptr unaligned ) returns
(ptr) ;

return_ptr = copy_expression ( operand_ptr );
```

1.	operand_ptr	pointer to the operand to be duplicated. (input)
2.	return_ptr	pointer returned by this program. (output)

Programs that invoke this entry:

alloc_semantics
builtin
copy_expression
declare
declare_descriptor
declare_structure
defined_reference
do_semantics
expand_assign
expand_initial

expand_primitive
expression_semantics
fill_refer
function
generic_selector
get_array_size
io_semantics
lang_util_
merge_attributes
operator_semantics
optimizer
prepare_symbol_table
refer_extent
share_expression
simplify_offset
subscripter

Entry:

copy_sons

This entry is used to duplicate all the symbol nodes of the members of a structure.

Usage:

```
declare copy_sons entry ( ptr, ptr );

call copy_sons ( father_ptr, stepfather_ptr );
```

1.	father_ptr	pointer to the symbol node to be duplicated. (input)
2.	stepfather_ptr	pointer to the new symbol node. (output)

Programs that invoke this entry:

context_processor
lang_util_

Internal Procedures:

copy_symbol
an internal procedure to create a symbol node, and to duplicate all the fields in the symbol node.

External Variables:

pl1_stat_$util_abort

Internal Static Variables:

previous
pointer set to remember the original symbol.next when a symbol node is to be duplicated.

Programs Called:

copy_expression
create_array
create_bound
create_list
create_operator
create_reference
create_symbol

Include Files used:

language_utility
source_id_descriptor

array
symbol
declare_type
list
nodes
operator
op_codes
reference

Errors Diagnosed:

Error 32

NAME: share_expression

Function:

1. It determines whether a reference node or an operator node can be shared, and increments the reference count.

2. It calls copy_expression if these nodes are not sharable.

Entry:

share_expression

Usage:

```
declare  share_expression entry ( ptr ) returns (ptr) ;

return_ptr = share_expression ( operand_ptr );
```

1.	operand_ptr	pointer to the operand to be shared. (input)
2.	return_ptr	pointer returned by this program. (output)

Programs that invoke this entry:

alloc_semantics
builtin
call_op
convert
do_semantics
expand_infix
expand_primitive
expression_semantics

function
io_semantics
lang_util_
operator_semantics
simplify_offset
string_temp

Internal Procedures:

none

External Variables:

none

Internal Static Variables:

none

Programs Called:

copy_expression
create_reference

Include Files used:

language_utility
source_id_descriptor
nodes
operator
reference
symbol

Errors Diagnosed:

none

The following procedures are used to print the error messages either on the user's console or in the program listing.

NAME: parse_error

Function:

1. It calls the error message program error_.

Entry:

parse_error

Usage:

declare parse_error entry (fixed bin(15), ptr);

call parse_error (error_number, error_ptr);

1.	error_number	error number. (input)
2.	error_ptr	pointer to the node exhibiting the error. (input)

Programs that invoke this entry:

data_list_parse
declare_label
declare_parse
default_parse
descriptor_parse
do_parse
format_list_parse
if_parse
io_statement_parse
lang_util_
on_parse
parse

procedure_parse
process_entry
reserve
statement_parse
statement_type

Entry:

parse_error$no_text

This entry is called when the error is caused not as a result of processing the statements in the block.

Usage:

```
declare parse_error$no_text entry ( fixed bin(15),  ptr
);

call  parse_error$no_text  ( error_number, error_ptr );
```

1.	error_number	error number. (input)
2.	error_ptr	pointer to the node exhibiting the error. (input)

Programs that invoke this entry:

none

Internal Procedures:

none

External Variables:

pl1_stat_$cur_statement
pl1_stat_$source_seg
pl1_stat_$st_length
pl1_stat_$st_start
pl1_stat_$statement_id
tree_$

Internal Static Variables:

none

Programs Called:

error_

Include Files used:

source_id
language_utility
source_id_descriptor
token_list
statement

Errors Diagnosed:

none

NAME: pl1_error_print

Function:

1. It gets the error message from the error message segment.

2. It constructs the file number, line number and statement number, and the statement that causes the error.

3. It prints the complete message on the user's console.

Entry:

pl1_error_print$write_out

Usage:

declare pl1_error_print$write_out entry (fixed bin(15), 1 unaligned, 2 bit(8), 2 bit(14), 2 bit(5), ptr, fixed bin(11), fixed bin(31), fixed bin(31), fixed bin(15));

call pl1_error_print$write_out (error_number, statement_id, token_ptr, source_seg, source_start, source_length, source_line);

1.	error_number	error number. (input)
2.	statement_id	a substructure containing the file number, line number, and statement number where the error occurred. (input)
3.	token_ptr	pointer to the identifier causing the error. (input)
4.	source_seg	pointer to the source segment. (input)

5.	source_start	index showing the start of the statement causing the error. (input)
6.	source_length	length of the statement causing the error. (input)
7.	source_line	not being used.

Programs that invoke this entry:

error_

Entry:

pl1_error_print$listing_segment

This entry is used to dump the error message on the listing segment rather than the user's console.

Usage:

```
declare pl1_error_print$listing_segment entry ( fixed bin(15), 1 unaligned, 2 bit(8), 2 bit(14), 2 bit(5), ptr );

call pl1_error_print$listing_segment ( error_number, statement_id, token_ptr );
```

1.	error_number	error number. (input)
2.	statement_id	a substructure containing the file number, line number, and statement number where the error occurred. (input)
3.	token_ptr	pointer to the identifier causing the error. (input)

Programs that invoke this entry:

error_

Internal Procedures:

next_string
an internal procedure to get the error message from the error message segment.

quote_token
an internal procedure to replace the "$" in the error message text with the corresponding identifier string.

External Variables:

cg_static_$debug
pl1_stat_$abort_label
pl1_stat_$brief_error_mode
pl1_stat_$err_stm
pl1_stat_$error_memory
pl1_stat_$error_messages
pl1_stat_$error_width
pl1_stat_$greatest_severity
pl1_stat_$last_severity
pl1_stat_$last_statement_id
pl1_stat_$severity_plateau
pl1_stat_$source_list_ptr
tree_$

Internal Static Variables:

none

Programs Called:

```
binary_to_octal_var_string
bindec$vs
decode_source_id
ios_$write_ptr
pl1_print$varying
pl1_print$varying_nl


Include Files used:


language_utility
source_id_descriptor
token
token_types
token_list
source_list
source_id


Errors Diagnosed:

          none
```

NAME: error

Function:

1. It calls the error message program error_.

Entry:

error

Usage:

```
declare error entry ( fixed bin(15), ptr, ptr );

call  error ( error_number, statement_ptr, token_ptr );
```

1.	error_number	error number. (input)
2.	statement_ptr	pointer to the statement node containing this error. (input)
3.	token_ptr	pointer to the token node causing this error. (input)

Programs that invoke this entry:

adjust_ref_count
aq_man
assign_op
assign_storage
cg_error
compile_statement
eval_exp
expmac
gen_pl1_symbol

jump_op
lang_util_
m_a
mst
pl1_signal_catcher
prepare_operand
prepare_symbol_table
semantic_translator
stack_temp
xr_man

Entry:

error$omit_text

This entry calls error_$no_text instead of error_.

Usage:

```
declare error$omit_text entry ( fixed bin(15), ptr, ptr
);

call  error$omit_text  (  error_number,  statement_ptr,
token_ptr );
```

1.	error_number	error number. (input)
2.	statement_ptr	pointer to the statement node containing this error. (input)
3.	token_ptr	pointer to the token node causing this error. (input)

Programs that invoke this entry:

none

Internal Procedures:

none

External Variables:

pl1_stat_$err_stm

Internal Static Variables:

none

Programs Called:

error_
error_$no_text

Include Files used:

language_utility
source_id_descriptor
source_id
statement
source_list

Errors Diagnosed:

none

NAME: error_

Function:

1. This is an error message buffering program.

2. If the listing option is on in the compilation, up to 100 error messages and related information are saved in the internal static buffer error_info.

3. It then calls pl1_error_print$write_out to print the error message on the user's console.

Entry:

error_

Usage:

declare error_ entry (fixed bin(15), 1 unaligned, 2 bit(8), 2 bit(14), 2 bit(5), ptr, fixed bin(8), fixed bin(23), fixed bin(11), fixed bin(31));

call error_ (error_number, statement_id, token_ptr, source_seg, source_start, source_length, source_line);

1.	error_number	error number. (input)
2.	statement_id	a substructure containing the file number, line number, and statement number where the error occurred. (input)
3.	token_ptr	pointer to the identifier causing the error. (input)
4.	source_seg	pointer to the source segment. (input)

5.	source_start	index showing the start of the statement causing the error. (input)
6.	source_length	length of the statement causing the error. (input)
7.	source_line	not being used.

Programs that invoke this entry:

error
lang_util_
lex
parse_error
semantic_translator

Entry:

error_$no_text

This entry is called when it is not possible to determine the specific statement causing the error.

Usage:

```
declare error_$no_text entry ( fixed bin(15), 1
unaligned, 2 bit(8), 2 bit(14), 2 bit(5), ptr);
```

```
call error_$no_text entry ( error_number, statement_id,
token_ptr );
```

1.	error_number	error number. (input)
2.	statement_id	a substructure containing the file number, line number, and statement number where the error occurred.

(input)

3. token_ptr pointer to the identifier causing the error. (input)

Programs that invoke this entry:

context_processor
error
initialize_int_static
lang_util_
lex
semantic_translator
validate

Entry:

error_$finish

This entry is called to sort the error messages in the buffer by statement number, and then dump them onto the listing segment.

Usage:

```
declare error_$finish entry;

call error_$finish;
```

Programs that invoke this entry:

lang_util_
v2pl1

Entry:

error_$initialize_error

This entry is used to initialize the internal static running index ei.

Usage:

```
declare error_$initialize_error entry;

call error_$initialize_error;
```

Programs that invoke this entry:

lang_util_
parse

Internal Procedures:

none

External Variables:

pl1_stat_$error_width
pl1_stat_$listing_on

Internal Static Variables:

ei	running index into the error_info array.

error_info	array of structure serving as the buffer for up to 100 error messages diagnosed by the program.
error_number	number of an individual error.
file_number	file number of an individual error.
line_number	line number of an individual error.
statement_id	substructure of error_info.
statement_number	statement number of an individual error.
token_pt	pointer to record the identifier causing an individual error.

Programs Called:

pl1_error_print$listing_segment
pl1_error_print$write_out
pl1_print$non_varying_nl

Include Files used:

language_utility
source_id_descriptor
nodes
operator
op_codes
reference
symbol
source_id

Errors Diagnosed:

none

NAME: decode_node_id

Function:

1. It decodes the source_id of a node into its component parts of file number, line number, and statement number.

Entry:

decode_node_id

Usage:

```
declare decode_node_id entry ( ptr, bit(1) aligned )
returns ( char(120) varying );

source_id_string = decode_node_id ( node_ptr,
capital_bit );
```

1.	node_ptr	pointer to a node whose source_id is to be decoded. (input)
2.	capital_bit	bit indicating whether to return upper case characters. (input)
3.	source_id_string	character string returned by this program. (output)

Programs that invoke this entry:

compile_statement
lang_util_
optimizer
pl1_signal_catcher

Entry:

decode_source_id

This entry supplies a 27-bit bitstring instead of a pointer to a node.

Usage:

declare decode_source_id entry (1 structure unaligned, 2 bit(8), 2 bit(14), 2 bit(5), bit(1) aligned) returns (char(120) varying);

source_id_string = decode_source_id (source_id, capital_bit);

1.	source_id	source_id to be decoded. (input)
2.	capital_bit	bit indicating whether to return upper case characters. (input)
3.	source_id_string	character string returned by this program. (output)

Programs that invoke this entry:

lang_util_
pl1_error_print

Internal Procedures:

none

External Variables:

pl1_stat_$source_list_ptr

Internal Static Variables:

none

Programs Called:

bindec$vs

Include Files used:

nodes
source_id
source_list
token

Errors Diagnosed:

none

The compiler is grouped and bound into four distinct segments in the Multics system. These bound segments are called bound_parse_, bound_semant_, bound_lang_util_, and bound_cg_. Each of these may invoke procedures bound in the same segment or procedures in other bound segments. To facilitate cross-segment procedure invocation and to reduce the names appearing on a bound segment, each bound segment has a entry vector program and a transfer vector program. The entry vector program introduces entry names in its own bound segment invoked by other segments; while the transfer vector introduces entry names on other bound segments invoked by some procedures in its own bound segment. All the entry vector programs and transfer vector programs are written in the assembly language ALM.

Note: There is no entry vector program for bound_parse_ because none of its components are invoked by procedures in the other bound segments.

NAME: v2pl1_semant_

Function:

1. This is the entry vector program for the bound segment bound_semant_.

Entry:

abort
call_es
error
expression_semantics
lookup
prepare_symbol_table
semantic_translator

NAME: lang_util_

Function:

1. This is the entry vector program for the bound segment bound_lang_util_.

Entry:

pl1_signal_catcher
generate_definition
end_symbol
beg_symbol
init_linkage
gen_pl1_linkage
compile_link
assign_storage
compile_formats
mst
by_size
display_pl1_text
display_pl1_map
merge_attributes
unaligned_nl
for_lex
string_ptr_nl
string_ptr
non_varying_nl
non_varying
varying_nl
varying
initialize_error
finish
no_text
error_
error
decode_source_id
decode_node_id
parse_error
decbin
share_expression
to_target
to_target_fb

```
to_integer
validate
from_builtin
convert
rename_parse
read_lib
clear
declare_lib
copy_sons
copy_expression
compare_expression
optimizer
declare_temporary
declare_pointer
declare_picture
declare_integer
param
ctl
declare_descriptor
char
bit
desc
integer
declare_constant
refer_extent
get_size
free_node
get_free
truncate
init
pl1_get
prologue
create_storage
create_statement
create_reference
create_operator
create_list
token_to_binary
vs
binoct
bindec
create_identifier
create_token
create_symbol
create_label
create_default
create_cross_reference
create_context
create_bound
```

```
create_block
create_array
```

TRANSFER_VECTOR_PROGRAMS

Please refer to the previous subsection "Entry Vector Programs" for the description of transfer vector programs.

NAME: parse_xfer_vector

Function:

1. This is the transfer vector program for the bound segment
    bound_parse_.

Entry:

pl1_signal_catcher
string_ptr_nl
merge_attributes
prepare_symbol_table
truncate
init
token_to_binary
semantic_translator
rename_parse
declare_picture
declare_lib
clear
varying_nl
non_varying_nl
non_varying
for_lex
pl1_get
parse_error
optimizer
free_node
no_text
initialize_error
finish
error_
create_token
create_symbol
create_statement
create_reference
create_operator
create_list
create_label
create_default
create_cross_reference
create_context

```
create_bound
create_block
create_array
copy_expression
vs
binoct
bindec
```

NAME: semant_xfer_vector

Function:

1. This is the transfer vector program for the bound segment bound_semant_.

Entry:

error
create_block
token_to_binary
share_expression
declare_lib
refer_extent
merge_attributes
get_size
free_node
no_text
error_
declare_temporary
declare_pointer
declare_integer
param
ctl
declare_descriptor
integer
char
bit
declare_constant
decbin
create_token
create_symbol
prologue
create_statement
create_reference
create_operator
create_list
create_label
create_cross_reference
create_bound
create_array
copy_sons

```
copy_expression
validate
to_target_fb
to_target
to_integer
from_builtin
convert
compare_expression
vs
```

NAME: util_xfer_vector

Function:

1. This is the transfer vector program for the bound segment
    bound_lang_util_.

Entry:

cg_error
l_v
e_v
prepare_operand
call_es
lookup
expression_semantics

DATA_SEGMENTS

The pl1_stat_ data segment contains the external static variables used by all phases of the compiler.

LHS	A pointer to the symbol node of the left hand side of an assignment statement currently being processed by the semantic translator. This pointer is set by expression_semantics, and reset by semantic_translator. This pointer is used by expand_infix and expression_semantics to decide whether an aggregate expression may be simplified.
abort_label	This label field is set by the procedure semantic_translator. Transferring to this label results in unwinding the compiler, printing an error message informing the user that the compilation has been aborted, and executing the cleanup handler.
apostrophe_mode	not used.
brief_error_mode	This bit(1) field is set to "1"b if the brief option is specified. This field controls the amount of text to be printed when an error occurred.
card_input	not used.
char_pos	This field contains an approximate character count for the current listing segment. It is approximate because it is always one larger than the actual character count. If the listing file is a multisegment file, this field only contains the character count of the active component.
check_bounds	not used.
compiler_created_index	Initialized to 0, this is a count of the compiler generated symbol names. The names are of the form "cp.n", where n is the value of compiler_created_index.
compiler_name	This character field is the compiler name to be stored in the object segment by the code generator. The name of this compiler is "pl1".

condition_index	Initialized to 0, this is a count of the compiler generated condition na,mes. The names are of the form "condition.n", where n is the value of condition_index.
constant_list	The root of the chain of all constants created by the compiler.
convert_len	not used.
convert_ptr	not used.
convert_switch	not used.
cur_block	not used.
cur_level	not used.
cur_statement	A pointer to the statement node currently being processed by the semantic translator.
debug_semant	This bit(1) field is set to "1"b if the debug_semant option is specified.
dummy_block	Initialized to a null pointer, this pointer is used by the code generator.
eis_mode	This bit(1) field is used to indicate whether the extended instuction code is desired by this compilation. In the current Multics system, this bit is always on.
equivalence_base	not used.
err_stm	A pointer pointing to the statement node in which an error has been discovered.
error_flag	This bit(1) field is used to indicate whether an error has occurred in the processing of compiler generated statements for the return statement in a multiple-entried program.
error_memory	The procedure error_ remembers the first 100 errors, so they can be sorted by line number before being placed in the listing segment.
error_messages	a pointer to the segment containing the text for all error messages.

error_width	The line length for the I/O stream user_output. If user_output does not have a line length, the value 120 is used.
expl_continuation_count	not used.
format_list	not used.
free_ptr	array of headers of free reuseable nodes saved in the allocation pool.
generate_symtab	This bit(1) field is set to "1"b if there is a "get/put data;" statement in the program.
greatest_severity	This field is initialized to 0 at the beginning of a compilation and will indicate the error level high water mark at the end of the compilation. In other words, the highest severity error recorded for this compilation.
had_data_io	not used.
hash_table	The token node hash table.
hollerith_mode	not used.
index	A number indicating the current locater qualifier in the external static array pl1_stat_$locator.
last_severity	A number indicating the severity of error encountered, used to set the had_error bit in the procedure semantic_translator$call_es.
last_source	The number of include files used in this compilation.
last_statement_id	not used.
line_count	At the end of a compilation this field is set to the number of newline characters in the source segment.
list3_node	not used.
list5_node	not used.

list_ptr	A pointer to the current listing segment.
listing_on	This bit(1) field is set to "1"b if a listing segment is to be produced.
locator	An array of pointers to keep track of the locator qualifiers occurring at different levels of an expression.
max_list_size	This field is the max_seg_size of the current listing segment.
max_node_type	This indicates the number of different types of nodes used by the compiler.
modetable	not used.
multi_type	This bit(1) indicates that the semantic translator is currently processing a return statement in a multiple-entried program.
no_quick_blocks	not used.
node_name	An array of character(12) containing the names of different nodes used by the compiler.
node_sizes	An array of numbers showing the sizes of different nodes used by the compiler.
node_uses	An array of counters, one for each node length. The appropriate counter is bumped whenever a node is created. The length of the operator is based on the number of words allocated for it. This information is provided for metering purposes.
ok_list	The root of the chain of OK lists. One OK list is created for each "get data" statement.
one	A pointer to the token "1", a decimal integer.
optimize	This bit(1) field indicates whether an optimize option is used in the compilation.
options	A character string representation of all options specified in the compilation. This

	character string will appear in the listing segment.
pathname	The absolute pathname of the source segment.
phase	The current compilation phase.
print_cp_dcl	If the cpdcl option is specified, this field is set to "1"b.
profile_length	The number of words to be allocated to implement the profile feature of the compiler. This value will approximate the number of statements in the subprogram.
quick_pt	A pointer to the bit array real_quick_info in the procedure semantic_translator. The bit array contains information on whether each block can be quick.
root	A pointer to the root block node.
seg_name	The entryname of the source segment but without the final component ".pl1".
severity_plateau	This field is initially one but can be set by the user to any value from one to four. This field implements the severity option by specifying the minimum error level of error messages to be printed.
source_index	A running index to the source segment currently working on by the procedure lex.
source_list_ptr	A pointer to the array of structures source_list that contains information about the source segment and all the include files used in the compilation.
source_ptr	A pointer to the source segment.
source_seg	An index used to indicate the source segment or include file currently working on by the procedure lex.
st_length	Current length of the statement being compiled. It is updated every time another token is parsed.

st_start	Character offset of the beginning of the current statement relative to the base of the source segment.
statement_id	The line number, statement number, and file number of the current statement.
stop_id	If the debug_semant option or debug_cg option is used, this field is compared to pl1_stat_$statement_id. If they are equal, then the procedure debug will be invoked.
table	Set to "1"b if the table option is specified in the compilation.
temporary_list	The root of the chain of the temporary nodes created during the compilation.
tree_vec_index	This field specifies how many additional segments are being used by the compiler to accommodate all the nodes used for the internal representation of the program. Its value will be zero if only tree_ and xeq_tree_ are being used. The value is maintained dynamically and reflects only the current storage requirements.
unwind	A label variable set to a label constant in the procedure process_entry. This label is used as the point of transfer in case an error occurs in the procedures declare_parse or default_parse.
user_id	The Person.Project.instance tag for the current compilation.
util_abort	An entry variable used by the utility procedures to unwind after a level 3 error. It is assigned the value semantic_translator$abort. Transferring to this label results in unwinding the compiler sufficiently to continue compilation.
util_error	An entry variable used by the utility procedures to unwind after an error. It is assigned the value semantic_translator$error. No unwinding results from transferring to this label.

validate_proc	A pointer to the symbol node of the validating procedure when the validate option is used in a procedure statement or entry statement.

OTHER_MISCELLANEOUS_PROGRAMS

Some procedures deal with other miscellaneous functions, and do not fall into any category described earlier.

NAME: refer_extent

Function:

1. It scans a reference node or an operator node to find a refer operator among some of its components.

2. It replaces all the refer operators by the refer target, qualified with a proper locator qualifier.

Entry:

refer_extent

Usage:

```
declare refer_extent entry ( ptr, ptr );

call  refer_extent ( expression_tree, locator_qualifier
);
```

1.	expression_tree	pointer to the operator node or reference node to be processed by this program. (input/output)
2.	locator_qualifier	pointer to be used as the locator qualifier. (input)

Programs that invoke this entry:

alloc_semantics
declare_descriptor
expand_assign
expand_primitive
io_semantics

lang_util_
operator_semantics
refer_extent

Internal Procedures:

none

External Variables:

none

Internal Static Variables:

none

Programs Called:

copy_expression
create_reference
refer_extent

Include Files used:

language_utility
source_id_descriptor
nodes
reference
operator
op_codes

Errors Diagnosed:

none

NAME: fill_refer

Function:

1. It scans a reference node or an operator node to find a refer operator among some of its components.

2. It replaces all the refer operators by the refer target, qualified with a proper locator qualifier.

3. It has an argument which indicates whether the locator qualifier needs be duplicated.

Entry:

fill_refer

Usage:

declare fill_refer entry (ptr, ptr, bit(1) aligned) returns (ptr) ;

return_ptr = fill_refer (expression_tree, locator_qualifier, copy_switch);

1.	expression_tree	pointer to the operator node or reference node to be processed by this program. (input)
2.	locator_qualifier	pointer to be used as the locator qualifier. (input)
3.	copy_switch	bit indicating whether copy_expression should be invoked. (input)
4.	return_ptr	pointer to the operator or reference node returned by this

program. (output)

Programs that invoke this entry:

builtin
prepare_symbol_table

Internal Procedures:

none

External Variables:

none

Internal Static Variables:

none

Programs Called:

copy_expression
create_reference

Include Files used:

language_utility
source_id_descriptor
nodes
reference
operator

op_codes

Errors Diagnosed:

none

NAME: check_star_extents

Function:

1. It scans through all the arguments of a call for a length expression appearing in a position corresponding to a star extents parameter.

2. It calls make_non_quick if the search is successful.

Entry:

check_star_extents

Usage:

```
declare check_star_extents entry ( ptr, ptr );

call check_star_extents ( symbol_ptr, argument_list  );
```

1. symbol_ptr — pointer to the symbol node of the entry. (input)

2. argument_list — list of arguments for the entry. (input)

Programs that invoke this entry:

builtin
function

Internal Procedures:

none

External Variables:

none

Internal Static Variables:

none

Programs Called:

make_non_quick

Include Files used:

nodes
block
symbol
reference
operator
list

Errors Diagnosed:

none

NAME: propagate_bit

Function:

1. It turns on a specified bit in the symbol node.

2. It also turns on the corresponding bit in the symbol node for all members of the structure.

Entry:

propagate_bit

Usage:

```
declare propagate_bit entry ( ptr, fixed bin(15) );

call propagate_bit ( symbol_ptr, bit_position );
```

1.	symbol_ptr	pointer to the symbol node whose attribute is to be propagated. (input)
2.	bit_position	bit position of the attribute in the symbol node that is to be propagated. (input)

Programs that invoke this entry:

alloc_semantics
builtin
defined_reference
expression_semantics
validate

Internal Procedures:

propagate
an internal procedure to turn on a specified bit throughout a structure.

External Variables:

none

Internal Static Variables:

none

Programs Called:

none

Include Files used:

symbol

Errors Diagnosed:

none

Historical Background

This edition of the Multics software materials and documentation is provided and donated to Massachusetts Institute of Technology

by Group BULL including BULL HN Information Systems Inc. as a contribution to computer science knowledge. This donation is made also to give evidence of the common contributions of Massachusetts Institute of Technology, Bell Laboratories, General Electric, Honeywell Information Systems Inc., Honeywell BULL Inc., Groupe BULL and BULL HN Information Systems Inc. to the development of this operating system. Multics development was initiated by Massachusetts Institute of Technology Project MAC (1963-1970), renamed the MIT Laboratory for Computer Science and Artificial Intelligence in the mid 1970s, under the leadership of Professor Fernando Jose Corbato. Users consider that Multics provided the best software architecture for managing computer hardware properly and for executing programs. Many subsequent operating systems incorporated Multics principles. Multics was distributed in 1975 to 2000 by Group Bull in Europe , and in the U.S. by Bull HN Information Systems Inc., as successor in interest by change in name only to Honeywell Bull Inc. and Honeywell Information Systems Inc. .
