(4)

RPI Department of Computer Science

Technical Report

EPL - Equational Programming Language Parsing and Dimension Propagation

Balaram Sinharoy

Project Advisor
Prof. Boleslaw Szymanski

Submitted to Information System Program
Office of Naval Research
Under Contract N00014-86-K-0442

Rensselaer Polytechnic Institute
Troy, New York 12180-3590

88-16 June, 1988

EPL - Equational Programming Language
Parsing and Dimension Propagation

by
Balaram Sinharoy

Project Advisor
Prof. Boleslaw Szymanski

Technical Report

Department of Computer Science
Rensselear Polytechnic Institute

Submitted to Information System Program
Office of Naval Research
Under Contract N00014-86-K-0442

ABSTRACT

This report gives a detailed description of parsing and dimensionality propagation for the Equational Programming Language compiler. The Equational Programming Language, called EPL, is a very high level declarative language designed to specify parallel and real-time computations. The EPL compiler translates equational specifications into procedural high level language (currently C). The compiler itself is written in C for UNIX based systems. Automated tools YACC and LEX has been used for lexical analysis and parsing. Detailed description is given of the design and implementation of the algorithms for building different structures (such as symbol table, equation trees, etc.). These structures are used by subsequent post parsing analysis and code-generation phases of the EPL compiler. The description of the algorithm for dimensionality propagation is also included.

Table of Contents

1. INTRODUCTION

1.1. Historical overview and EPL

A slowdown in the rate of growth of computing power available from single processor, a dramatic decrease in the hardware cost and a everlasting need for faster computations especially for many real-time application have inspired both users and developers of large scale computers to investigate the feasibility of parallel computation [1].

With the advent of highly parallel computers need for efficient, user-friendly compilers that help to design parallel computations became apparent. In the *traditional sequential control flow* model (often referred to as Von Neumann), there is a single thread of control which is passed from instruction to instruction [2]. In the parallel control flow model, special parallel control operators such as FORK and JOIN are used to explicitly specify parallelism. These operators allow more than one thread of control to be active at any instant and provide means for synchronizing these threads of control.

Parallel processing of a sequential algorithm can be achieved in a program explicitly or implicitly. For explicit parallelism, users must be provided with programming constructs that permit them to express parallelism in a program. Thus, the programmer is endowed with responsibility for efficiency of program execution. The programmer has to know the system reasonably well (especially the hardware and the operating system) and (s)he has to decide which portions of the program should be run in parallel. How well the underlying parallelism has been exploited entirely depends on the knowledge, skill, and judgement of the programmer. This is undesirable in most cases. Since parallel computers are mostly used by scientists and engineers of various different fields as a tool for their computations, it is a heavy burden on their part.

It is much more desirable that the compiler be able to determine the underlying parallelism in an algorithm. The programmer is then relieved from the responsibility of indicating and extracting parallelism. Thus, the parallel machine can be used more efficiently. To achieve that the programming language has to have features enabling the user's to specify his/her algorithm without prescribing the flow of control. Only necessary data dependencies will be implicitly present in the user specification. Scanning user's specification, the compiler recognizes possible control flows. Then in the optimization stage, the compiler selects that control flow which creates the most parallel code (the largest number of program units which can run in parallel). Based on the selected control flow and other considerations (such as synchronization penalty), the compiler creates blocks or units of program statements and groups them into processes. Some of these processes (the units of program statements) may be independent during execution (i.e., execution of one process does not affect the other) and thus can be run concurrently using different processors. Generating these independent processes, the compiler produces complete programs which include communication and synchronization protocols necessary to execute these processes concurrently.

In contrast, most of the present day concurrent programming languages (such as ADA, concurrent Algol, Concurrent Pascal, etc.) use explicit parallelism, so they can not be treated as user-friendly, apart from being inefficient for programs written by unsophisticated programmers.

This document describes the design of some major parts of the compiler meant to be used in highly parallel computing environment. The compiled language supports implicit parallelism and is appropriately called Equational Programming Language or EPL. To extract maximum parallelism in a user specification, the language is made non-procedural, i.e. the statements in a specification does not have any linear ordering. Shuffling the statements in a specification does not change its meaning. The advantage of this feature is that the compiler is given the opportunity to find out the sequential dependencies among the statements (or more appropriately, equations) and hence the processes that can be run concurrently, without any directive being provided by the user regarding the parallelization. If this feature is absent then different ways of coding the algorithm in a language will have different efficiency, delegating the responsibility of selecting the particular coding for the most efficient program to the user. The presence of this feature also has the added advantage of giving the opportunity to minimize the synchronization overhead, since the compiler can estimate the run-time of different processes that are to be run concurrently and adjust the processes accordingly.

1.2. EPL system

EPL system is a language translator for a non-procedural equational programming language. In a program written in EPL, there is no linear ordering among the equations, (i.e., a program obtained by permuting the statements of the original program produces the same object code on compilation). Therefore the EPL translator differs greatly in organization and scope from a conventional compiler. There are several phases of translation in the EPL system. The figure below shows the block diagram for the entire EPL system. The Lexical analyzer, Parser and the Dimension propagation phases are described in details in this document.

The EPL system may be divided into the following phases of translation:

LEXICAL ANALYZER :

Like other conventional language translators EPL also uses lexical analyzer to break down the character stream in the user's program into a stream of suitable tokens that are to be used by the parser. The parser and the lexical analyzer works interactively. The parser calls the lexical analyzer repeatedly. Each time lexical analyzer is called, it absorbs some characters from the source program and returns a token to the parser along with a value of the token. if any [7].

PARSER :

The parser checks whether the tokens returned by the lexical analyzer forms a sentence that belongs to the grammar of EPL [3, 7]. If the sentence does not belong to the EPL grammar then the parser reports syntax error the first time it detects it during parsing, and then it does error recovery and proceeds to parse the rest of the program The parser tries to report as many errors as possible (so that the user does not have to compile his/her program repeatedly) and at the same time tries to keep redundant error messages to the minimum. However like other compilers . it might give redundant error messages. Some syntax errors may be hidden by others

If no syntax error is found. the parser accepts the program and builds the symbol table, equation trees and other related global data structures that characterizes the pro-

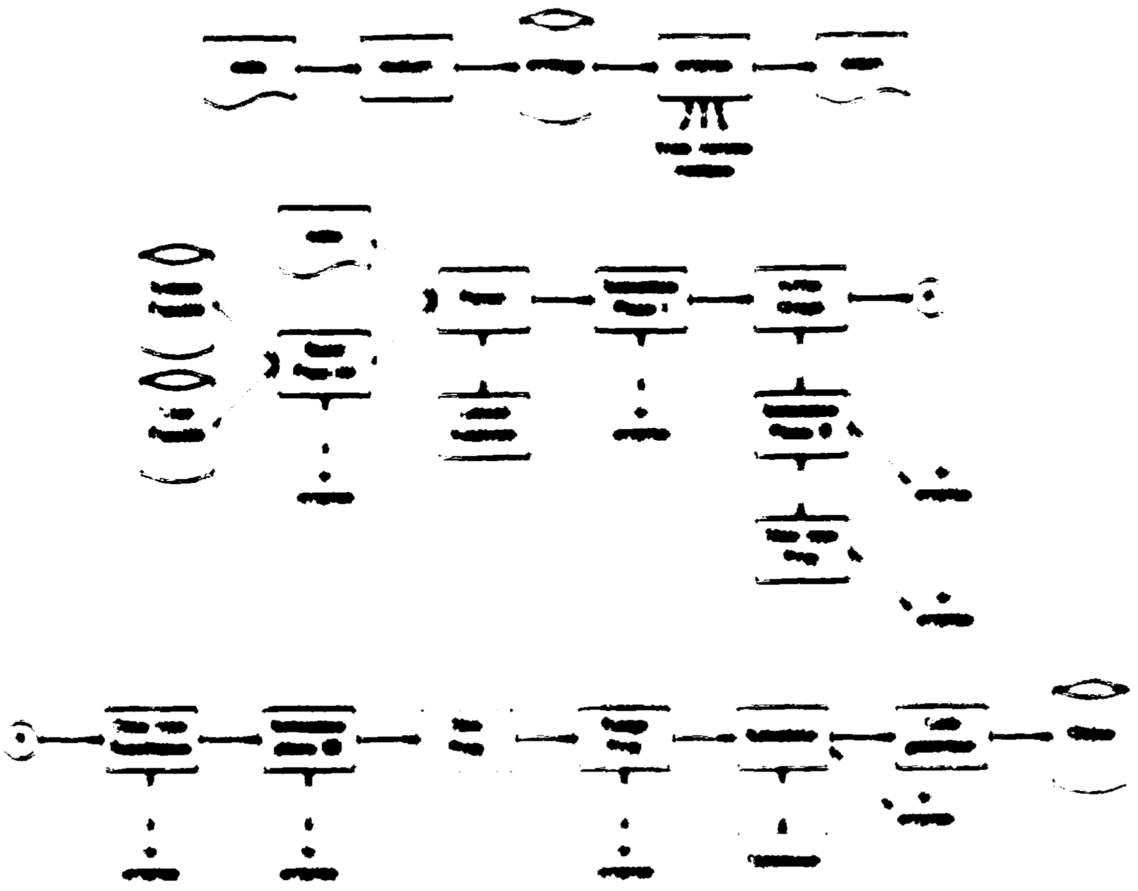


Figure 1. EPL Program Translation Flow

gram. These structures are used and manipulated by the other phases of the EPL translator.

Unlike other language translators EPL does extensive post-parsing analysis before the actual code is generated. These analysis is divided into the following translation phases :

EXTENSIVE SEMANTIC CHECKING :

Fig. 1 shows that there are three semantic checking phases in the system placed in three strategic points in the system. The greatly increased semantic checking is caused by the features of the language, which allows data structures and indices in the user's program to be left undeclared (their type will be established by data type propagation phase -- any contradiction in the implied data type for an undeclared data structure or index will be detected at this point by semantic phase III). The flexibility in the language to allow undeclared data items increases the need for detailed analysis of the declared data items to ensure that they are consistent and complete (that is data types for all undeclared data items can be established by analyzing the data types of the declared data items) [5].

DATA TYPE PROPAGATION :

As mentioned earlier, EPL allows the user to leave some data items undeclared in his/her program provided they are used, if they are done consistently, i.e., the data types of the undeclared data items can be inferred from the declared data items by analyzing the equations where they are used. Data type propagation does this propagation of data types from the fields of the declared data structures in the user's program to the undeclared data items in the program in a meaningful and consistent manner. Semantic error will be reported by the semantic checking phase if analysis finds that the declared variables are incomplete (that is they are not sufficient to determine the data types of all other undeclared data items) and/or incompatible [5, 6].

ARRAY GRAPH CONSTRUCTION :

Array graph is one of the most important structure in the EPL system. As mentioned before, an EPL program does not contain any information regarding the sequential dependencies of its equations. Though some equations can be executed concurrently, in general not all the equations can be executed at the same time. The execution of those equations which use a variable in the right hand side of the equation that is being defined by some other equation has to wait till the defining equation is executed. After analyzing the equations of a program array graph is constructed, which represents the sequential dependencies among different data structures and equations in the program which must be enforced for a meaningful computation [4,6].

DIMENSION PROPAGATION :

An important feature of EPL is that it allows the user to drop subscripting expressions consistently across the equations. This feature relieves the user from the routine job of consistently writing the subscripting expressions when they are obviously understandable, permitting him to concentrate more on the basic algorithm. This feature also becomes very useful in many scientific applications where some computation procedure need to be performed using different number of dimensions. In these cases the user needs to change only the dimensions of the declared data items and the contents of the files. He/she does not need to make any change in the algorithm (characterized by the equations) itself. Dimension propagation phase of the language translation fills in the dropped dimensions by propagating dimensions starting from the fields of the input and output file structures. The propagation is based on the analysis of the equations with the help of the array graph [6].

RANGE PROPAGATION :

To obtain independence from user specified sequentiality (which may be misleading for parallelization purpose) in an EPL program, EPL grammar does not have many conventional programming language constructs. One of the most important of such programming language construct that is absent in EPL is the iterative loop. Since any nontrivial algorithm has parts of its computation to be done iteratively, EPL must have some means to specify iteration, protecting non-procedurality at the same time. Ranged EPL data items are used to share values which are defined inside a loop, at each iteration. For more flexibility and convenience of the user, EPL allows the user to omit the upper bound of these ranges. Complex analysis of the user's declared structural ranges and run-time defined structural and subscript ranges is needed in order to determine meaningful and consistent bounding values for the ranges, The ranges are also the bounding values for the loops in the generated code [6].

OPTIMIZATION :

Most compilers written today includes an optimization phase to make the object code more efficient (in terms of time or space or both, depending on particular environment in which they are proposed to be used). However in a very few of them the optimization phase is as important as in the EPL system. There is no explicit iteration construct in the EPL grammar. Even a single valued data item that are defined in an iteration in a conventional programming language, is equivalently defined as a ranged data item in a corresponding EPL program. This range is equal to the number of iterations -- to accumulate the value of the data item at each iteration, since all or some of them may be used by some other equations in the program. Even a small EPL program that computes through a large number of implicit iterations would require a large memory. Therefore an optimization stage is performed to reduce the dimensionality of multidimensional structures that do not require any referencing of the past values (or at least do not refer to values distant from the current value) [6, 3].

SCHEDULING :

Unless the algorithm itself is written in a parallel form , the algorithm can be divided into sub-computations. At execution time, some of the sub-computations can be executed concurrently. Scheduling determines the groups of units (or processes) in an EPL program that can be run concurrently (by separate processors), and schedules these groups. Since array graph contains the dependence relationship among the assertion and data nodes, it is the major data structure used for scheduling [6].

1.3. Project Overview

1.3.1. Project Scope

The system components that have been designed and implemented in this project include :

1. Lexical Analyzer
2. Syntax Analyzer
3. Dimensionality Propagation
4. Scheduler
5. Optimization

The other components of the system are already built or in the process of being built by other members of the research group. The work for the scheduler is nearing completion and the work for the optimization phase is about to be started.

1.3.2. Computing Environment

The EPL system components described in this document are implemented in the Sequent Balance 21000 computer at the department of Computer Science in Rensselaer Polytechnic Institute running the Dynix operating system. The system is written in the programming language C. Currently the objective of the system is to translate from EPL source code to C object code using FORK and JOIN statements in the produced object code for parallel execution. System libraries used in the implementation are <stdio.h>, <strings.h>, <ctype.h> and <sys/types.h>. The code is written with portability to other UNIX based systems in mind.

2. Global Data Structures

2.1. Symbol Table

2.1.1. Formal description of the symbol table

The symbol table is organized as a hash table having 101 buckets. The hash function used for this purpose is described later in the algorithm "hash". Each entry in the symbol table has 14 fields. They are described as follows :-

1. *symbol_name

This is a pointer to the character string for the symbol for which this entry is created. A symbol here may be a variable name (see ref. 1 for legal variable names), a real or integer constant or some reserved words like LAST, RANGE, ADDRESS, PRESENT, CASE, etc. A symbol table entry is always created for the symbol PROCESS. Since PROCESS is a reserved word in equational programming language, it can not appear in the user's program to denote a data item. All the data structures in the user's program that does not belong to any file structure are made part of an intermediate file structure. The name of this file is PROCESS. In equational programming language same identifier may denote different variables (or data items). Typical example is when the same identifier is used to denote data items belonging to different file structures. So in the symbol table there might be more than one entry for the same symbol in the same bucket (all belong to different files) .

2. *parent

If the entry is made for a field, group, or record of a declared data structure in the user's program then this field contains a pointer to the parent of this data item in that data structure. If in the user's program some data structure is found that does not belong to any file structure then the parent field of the topmost data item in that data structure contains a pointer to the symbol table entry for PROCESS and it is made the next sibling of the last child added to this intermediate file structure.

If there is no possible parent for the symbol for which the entry is created then this field contains NULL. Symbols representing file name, subscripts, integer or real constant do not have any parent.

3. *old_child

This field contains a pointer to the oldest child of the symbol table entry. The oldest child is the data item that comes just after the data item declaration in the user's program for which the entry is created and is preceded by a larger integer than the integer that precedes the data item for which the entry is created.

If there is no child then this field is NULL.

4. *sibling

This field contains a pointer to the next sibling. Next sibling is the data item that is in the same level as the data item for which the entry is created and comes just after it in the user's program declaration. If there is no 'next sibling' and the entry is not for a subscript then it is NULL.

In the case of *subscript* this field contains a pointer to the head of the link list of all the pointers to the symbol table entry corresponding to the names listed in the nlist of subscript or sublinear subscript description. In other words this is the linked list of all the pointers to the symbol table entry corresponding to the data items that can be subscripted using the subscript for which the entry is created.

5. type_of_use

This field contains the 'type of use' information of the symbol. This field may contain any value between 0 and 15 depending on the type of use of the symbol in the user's program. Following is the list of all possible type of uses along with their codes :

0 = The entry represents an identifier that has not been declared , has not been used as the target of an assertion, and has not appeared in an indexing expression.
1 = Subscript
2 = Field
3 = Group
4 = Record
5 = File
6 = Last
7 = Present
8 = Range
9 = The entry is for an undeclared identifier that has been used in a subscript expression, but never used as a target of an assertion.
10= The entry is for an undeclared data name that has been found to be the target of an assertion.
11= Function
12= Address
14= Constant
15= CASE

There is no ordinary variable. For example, when a variable is declared as :

```
int : just_a_name;
```

the corresponding symbol table entry contains 2 indicating it as a field (of some interim file, the name of which is PROCESS).

6. file_type

If the entry is created for a data item then this field indicates the type of file in which the symbol for which the entry is created belongs. In equational programming language there are four different types of files. Since during parsing we might come across the name of a data item before it is declared in a file structure (if it is declared at all), we need a code to indicate that the data item has not been declared in a file structure as yet.

0 = Sequential
1 = Directed
2 = Display
3 = Port
4 = not known yet

For ordinary variable this field is 0, i.e., by default the file PROCESS is sequential.

If the symbol table entry is for the symbol "LAST" or "RANGE" then this field contains the number of repetition of "LAST." or effective number of repetition of "RANGE." (if "LAST." is preceded by "RANGE." then the "LAST."s are reckoned as "RANGE.") in the user's program.

7. type

This field contains the data type information for the fields. There are eight different data types possible in the equational programming language. For character declarations the size of a variable might not be known when the symbol table entry is created (sometimes, the size is not even at the end of parsing) . For this case code 8 is used. If the number of character is known to be "n" then this field contains "-n". As before when the entry is created for a data item, its data type may be unknown, which requires to use a code (here "0" is used) to indicate "yet unknown". If the entry is not for a "field" of a declared file structure in the user's program then also this field will contain "0".

-n	= character
0	= yet unknown
1	= short integer
2	= integer
3	= long integer
4	= real
5	= double (real)
6	= quad (real)
7	= logical
8	= char (no size is known yet).

8. io_flags

This is a field of nine subfields each of width one or two bits. They are as follows:

[1] is_input — This sub-field is of one bit and it indicates whether the entry is for a symbol of an input file or not. If the symbol for which the entry is created is declared in an input file structure then this bit is set to 1, else it is set to 0.

[2] is_output — This sub-field is also of one bit and it indicates whether the entry is for a symbol of an output file or not. If the symbol for which the entry is creates is declared in an output file structure then this bit is set to 1, otherwise it is 0.

[3] others — This sub-field has 2 bits. This sub-field is used for subscript entries and for data items that belongs to a port file structure.

If the entry is created for a subscript then this sub-field indicates whether the subscript is an ordinary subscript, in which case it contains 0, or a sublinear subscript, when it contains 1.

If the entry is for a data item that belongs to a PORT file then this sub-field indicates the type of the PORT file which may be any of the following three

IN --> OUT (0)
OUT --> IN (1)
nothing said (2)

The codes in the parentheses indicate the corresponding codes used

[4] rec This sub-field is used to keep the I/O information of the records of *port* files in the following manners

0 = ordinary
1 = RECORD IN
2 = RECORD OUT

The width of this sub field is 2 bits

[5] case_key This bit is set for those symbol table entry which are used as the key field for some CASE substructure. The key field is a field in a file structure involving CASE in the definition of its substructures. The structure of the record depends on the value of the key field.

[6] local_sub: This is a 1 bit sub-field. The bit is set only when the entry is for a local subscript. Local subscripts allowed in equational programming language are SUB0, SUB1, ..., SUB9.

[7] case_below:	This is a sub-field of 1 bit and it is set to 1 for those record entries for which there is a CASE entry found as a descendent of that record.
[8] string:	This is also a sub-field of 1 bit and it is set to indicate that the entry is for a character string.
[9] dim_proc:	This one bit sub-field is set to 1 only when the dimensionality for the entry has been correctly established. Needed for dimensionality propagation. At the end of parsing this bit is set only for those symbol table entries which represents a data item that belongs to an input or output file structure. After dimensionality propagation phase, this bit is set for all the symbol table entries of all the data items for which correct dimensionality can be established by propagating dimensionality from those fields which belongs to some input or output file structures.
[10] undef:	This one bit sub-field is set to "1" whenever a file is created before its definition is found in the user's program. This field is useful in semantic error reporting in the cases

when "IN=>OUT" or "OUT=>IN" is found in the definition of the file.

[11] bck_ptr: This pointer is set to 1 to indicate that there is a RANGE or LAST (or both) symbol table entry which has its other_pointers field pointing to this entry. This information is primarily used in the dimension propagation phase of the EPL system to reduce the searching time.

9. *range

This field contains a pointer to the range definition block in the RANGE table that corresponds to the ranges of the symbol for which this entry is made. The ranges of the symbol are found in the declaration of the data item for which the entry is created. The data structure for range definition block is described later. If the symbol is not subscripted then this field will contain NULL.

10. dimensionality

During parsing the symbol table entries of a data item that belongs to a file structure partially correct dimensionality value in this field (the value of the dimensionality at this time is the value found in the user's program) . At the end of parsing this field contains the correct dimensionality for data items that belongs to some input or output file structure. The correct dimensionality is the sum of all the declared dimensionalities (as found in the user's program) of all the ancestors, and the declared dimensionality of the data item under consideration.

After the dimensionality propagation phase of the EPL system, the dimensionality is propagated to all the other possible data items (on the basis of analysis of the array graph) starting from the data items for which dimensionality has been correctly established during parsing, specifically the fields that belongs to input or output file structure.

11. *other_pointers

The other_pointers field is used to store the following information :

a. If the symbol table entry is created for a sublinear subscript then this field contains a pointer to the expression tree corresponding to the expression that is associated with the declaration of the sublinear subscript.

b. If the symbol table entry is created for a field of a display file then this field contains a pointer to a character string which is the format of the data item as found in the declaration of the data item in the user's program.

c. If the symbol table entry is created for a symbol like "LAST", "RANGE", "ADDRESS", or "PRESENT" then this field contains a pointer to the symbol table entry which they qualify, which maybe an entry for a field or group or in some cases which may be another entry for "RANGE" or "ADDRESS".

d. If the symbol table entry is created for the symbol "CASE" (a new such entry is created, each in the same bucket, every time "CASE" is found in the declaration of a file in the user's program), this field contains a pointer to the Rec_case structure (described later) which keeps track of all the different sub_structures mentioned in the user's program. In this case the interpretation of the record structure of the file depends on the value of some expression (this expression appears after the keyword "WHEN" in the user's program and is evaluated each time a new record is retrieved from the disk storage. It determines which of the many alternative structures , each of which is preceded by "WHEN" or "OTHER", are to be considered for breaking up the chunk of information contained in the record being processed).

12. *ptr_arr_gr

This field contains a pointer to the node in the array graph that corresponds to the symbol whose entry is made. During the parsing this field contains NULL. This field is appropriately filled when the array graph is built. This field is needed to link the array graph nodes with the symbol table entries.

13. *vbl

This field contains a pointer to the structure "variable_block" (in the rangetab file). During parsing this field is not filled. This field is used in the Range propagation phase of the EPL system.

14. *next

This field points to the next entry in the bucket. If there is no more entry in the bucket then it is NULL.

2.1.2. Remarks

The symbol table contains all the symbols that appears in the compiled EPL program. The symbols might be variable names, string constants, integer or real constants. Each variable that belongs to DIRECT or PORT files will have two entries in the symbol table, if the file in which they belong to are declared both as input and output files. In that case one entry will have the original variable name and the other will have that name prefixed by "OUT_" . The I/O information of these two will differ. The entry with the original variable name will have is_input flag set to 1 and is_output flag set to 0. For the other entry (the one prefixed by "OUT_") the is_input will contain 0 and is_output flag will contain 1. Both of these entries will be found in the same bucket, on the basis of the original variable name .

For variables that has been prefixed by ADDRESS, RANGE, LAST or PRESENT a new entry of the same name is created in the symbol table which has its other_pointers field pointing to the symbol table entry for that variable (this new entry is created if there is no such entry already in the symbol table) . The is_input is_output flag of this new entry is made equal to the variable that they are created for and the type of use field of this new entry is set to reflect what kind of entry it is. For RANGE and LAST, the file_type field will contain the number of repetition of it as found. In the most complicated case of reference to a data item the other_pointers field of a LAST entry may point to a RANGE entry, the other_pointers field of which may point to a ADDRESS entry, the other_pointers field of which may finally point to the actual data item (the field of some file structure). If in a reference to a data item "LAST." appears after some "RANGE." in the user's program then "LAST." is replaced by another "RANGE.". Such replacement does not change the value of the prefixed variable. Furthermore, only one entry for "RANGE" with correct number of count of repetition (of RANGE and LAST) is sufficient. This is not true if "LAST." is the first prefix in the reference. In the later case entry for both "LAST" and "RANGE" has to be made in the symbol table. Example 1 through 3 in this section (shown later) shows all different cases that may arise. Example 3 is the most complicated among them.

The major difficulty in creating symbol table entries and filling their different fields is that the various attributes qualifying the symbol table entry (that are to be used to fill in the different fields) can be found scattered (in any order) throughout the user's program. This flexibility is very much welcomed from the user's point of view. On the other hand this makes the parser much more complicated. For this reason some fields of some symbol table entries are not filled before the end of the parsing phase. Because of this flexibility in EPL grammar, in some cases even at the end of parsing it is possible that some attributes of some of the symbols are not declared by the user neither can they be inferred easily from other information in the user's program.

These places are left blank by the parser and they are to be filled by other phases of the system after extensive post-parsing analysis. If after analysis some vital information still cannot be inferred from the user's program, then an error will be reported by the appropriate phase of the system. The attributes that fall in this category include data type, range, dimensionality etc. Dimensionality field for declared variable is filled during parsing with the dimensionality value found in the user's program. They are changed to the correct dimensionality at the end of parsing.

In the following pages the exact structure of the symbol table entries are given in C source code.

2.1.3. Symbol Table Structure in "C"

Following is the structure of the symbol table definition in C. It is available in the file **~epl/include/symbol_table.h** *in the system* **SEQUENT.**

```
typedef struct Sym_tab_entry {      /* The structure of a single entry
                                       of the symbol table.  */
   char *symbol_name;             /* Pointer to the symbol name */
   struct Sym_tab_entry *parent;       /* Pointer to the entry of the parent */
   struct Sym_tab_entry *old_child;  /* Pointer to the entry of the oldest
                                                                child */
   union{
   struct Sym_tab_entry *st;
   struct St_ptr_list   *l;} sibling;
```

```
                           /* Pointer to the entry of the next sibling.
                                 Pointer to a link list of names ,in
                                 case this is an entry for sublinear
                                 subscript.                   */
short type_of_use;             /* 'type of use' of the symbol:
                               0 = Means that the entry represents an
                                   identifier that has not been declared
                                   ,has not been used as the target of
                                   an assertion , and has not appeared
                                   in an indexing expression.
                               1 = subscript
                               2 = field
                               3 = group
                               4 = record
                               5 = file
                               6 = last
                               7 = present
                               8 = range
                               9 = The entry is for an undeclared
                                    identifier that has been used as a
                                    target of a assertion.
                               10= The entry is for an undeclared data
                                    name that has been found to be the
                                    target of an assertion.
                             11= Function
                               12= Address
                               14= Constant
                               15= CASE                          */

short file_type;             /* type of file in which this symbol belongs
                                0 = sequential
                                1 = direct
                                2 = display
                                3 = port
                                4 = not known yet                */

short type;                  /* type of the field:
                               -n = character (n)
                                0 = yet unknown or the entry is not for
```

```
                                          a field.
                                        1 = short (integer)
                                        2 = integer
                                        3 = long (integer)
                                        4 = real
                                        5 = double (real)
                                        6 = quad (real)
                                        7 = logical.
                                        8 = char (no size known yet)        */

struct {
   unsigned is_input : 1;    /* is the entry a part of an input file */
   unsigned is_output: 1;    /* is the entry a part of an output file*/
   unsigned others   : 2;    /* indicate the type of subscript
                                       sublinear = 1
                                   non-sublinear = 0;        OR
                              the porttype if the entry is for a symbol
                              which is a part of the port file
                                For porttype
                                       0 = IN => OUT
                                       1 = OUT => IN
                                       2 = nothing said.          */

   unsigned rec      : 2;    /* to keep the following information about
                              the record type
                                 0 = ordinary
                                 1 = RECORD IN
                                 2 = RECORD OUT                    */
   unsigned case_key : 1;    /* This will be set only for those symbol
                              table entry which are used as the key
                              field for some CASE substructure     */
   unsigned local_sub: 1;    /* This field is set only when the entry is
                              for a local subscript              */
   unsigned case_below: 1;   /* This field is set for the record entries
                              when there is a CASE entry found as a
                              descendent of that record entry      */
   unsigned string: 1;       /* This field is needed to indicate
                              whether the entry is for a string
                              or for a integer or real or variable
```

```
                        name   */
     unsigned dim_proc: 1;    /* This bit is used for dimensionality
                        propagation. When dimensionality of
                        this entry has been established this
                        bit will be set to 1  */
     unsigned undef : 1;      /* This bit is set to indicate that the
                        file (when IN=>OUT or OUT=> IN is found)
                        has been created though its definition is
                        not found as yet. */
     unsigned bck_ptr: 1;     /* This pointer when set to 1 indicates that
                        there is a RANGE or LAST symbol table
                        entry that points to this entry. */
        } io_flags;

struct Range_def_blk *range; /* Pointer to the range definition
                             block that corresponds to the
                             symbol */
short dimensionality;        /* indicates the dimensionality of the symbol.
                        If the symbol is not subscripted then it
                        contains 0 */
union{
struct Rec_case *rc;
struct Eq_tree_node *eqt;
struct Sym_tab_entry *operand;
char *name;
int record_length;
int key_offset; } other_pointers;
                    /* Pointer to the head of the when expression
                       list in case CASE entry.  Pointer from the LAST,
                       PRESENT,RANGE or ADDRESS entry to the variable.
                       pointer to the root of the expression tree
                             (for sublinear subscript entry)
                       pointer to the format string if it is an
                       entry for a field of a display.           */
struct Arry_node  *ptr_arr_gr;          /* pointer to the corresponding
                                   entry in the array graph */


                        /* pointer used during range propagation */
```

```
union {                            /* and scheduling                     */
  struct Node_sub     *nodesub;  /* node-sub for subscript entries       */
  struct Eq_tree_node *i_o_tree; /* parse tree for i/o event scheduling  */
     } range_sched_info;


union {
  struct Eqt_node_list  *node;   /* A pointer to the list of equation tree
                                    nodes.                          */
  struct Eq_tree_list *eqt;      /* A pointer to a list of equation tree
                                    heads.                          */
     } eqt_ptr;


struct Sym_tab_entry  *next;          /* pointer to the next symbol table entry  */
     } Sym_tab_entry;
```

2.1.4. Codes used for Symbol table processing

The definitions of different codes used in the symbol table are listed below. These definitions are used throughout the EPL system instead of the actual integer codes for the convenience of the designers.

	FILE TYPE	INTEGER CODE
#define	SEQ	0
#define	DIRECTED	1
#define	DISPLAY	2
#define	PORT	3

Table 1 : Table. 1: The different file types used in EPL.

	TYPE	INTEGER
#define	VOID	0
#define	SHORT	1
#define	INTEGER	2
#define	LONGINT	3
#define	REAL	4
#define	DOUBLE	5
#define	QUAD	6
#define	LOGICAL	7
#define	CHAR	8
#define	ERROR	99

Table 2 : Table showing all the different types for variables allowed in EPL.

	USE	INTEGER CODE
#define	UNDEF0	0
#define	SUBSCRIPT	1
#define	FIELD	2
#define	GROUP	3
#define	RECORD	4
#define	FILE_	5
#define	LAST	6
#define	PRESENT	7
#define	RANGE	8
#define	UNDEF9	9
#define	UNDEF10	10
#define	FUNCTION	11
#define	ADDRESS	12
#define	CASE	14

Table 3 : Table showing all possible types of symbol table entries.

2.1.5. Examples showing symbol table entries

In this section some examples are shown regarding the contents of the different symbol table entries. At the end of the description of the Rec_case and Case structures more meaningful examples involving all of these structures has been shown. The diagrams are included for clarity.

Example 1 For the equation

```
range.range(2).range.range.d = 8;
```

the corresponding symbol table entry contains (Fig. 2)

Entry is made for the symbol: RANGE
Parent: NULL
oldest child : NULL
next sibling : NULL
type of use: Range
file type contains the integer 5
type   : integer
I/O flags : 000 000000
Dimensionality : 0
Other_pointers points to the entry for D
Range : NULL

Example 2   For the equation

range.range(2).last.last(2).range.range.address.d = 5;

The corresponding symbol table entries are (Fig. 3):

Entry is made for the symbol: RANGE
Parent: NULL
oldest child : NULL
next sibling : NULL
type of use: Range
file type contains the integer 8
type   : integer
I/O flags : 100 00000
Dimensionality : 0
Other_pointers points to the entry for ADDRESS
Range : NULL

Entry is made for the symbol: ADDRESS
Parent: NULL
oldest child : NULL
next sibling : NULL
type of use: Address
file type : Sequential

type : yet unknown or the entry is not for a field
I/O flags : 100 00000
Dimensionality : 0
Other_pointers points to the entry for d
Range : NULL

Please note that in this case no separate entry for LAST has been created since semantically the above equation is same as:

range(8).address.d = 5;

Example 3 If LAST is the first item in the reference for a variable then the entry for LAST will be created as in (Fig. 4):

last.last.last(5).range.range(2).last.last(2).range.range.address.d = 5;

The corresponding symbol table entries are :

Entry is made for the symbol: LAST
Parent: NULL
oldest child : NULL
next sibling : NULL
type of use: Last
file type contains the integer 7
type : yet unknown or the entry is not for a field
I/O flags : 000 00000
Dimensionality : 0
Other_pointers points to the entry for RANGE
Range : NULL

Entry is made for the symbol: RANGE
Parent: NULL
oldest child : NULL

```
next sibling : NULL
type of use:  Range
file type contains the integer 8
type        :  integer
I/O flags  : 000 00000
Dimensionality  : 0
Other_pointers points to the entry for ADDRESS
Range :  NULL
```

```
Entry is made for the symbol:   ADDRESS
Parent: NULL
oldest child  : NULL
next sibling : NULL
type of use:  Address
file type  :  Sequential
type        :  yet unknown or the entry is not for a field
I/O flags  : 000 00000
Dimensionality  : 0
Other_pointers points to the entry for d
Range :  NULL
```

In the case of subscript we fill the type_of_use field with SUBSCRIPT and the sibling field points to a list of variables that can use that subscript. The following example illustrates this point.

Example 4 If we have the declarations as

```
input: outfile; out: outfile, dispfile;

file: outfile (direct),
 10 rec: our[*],
    20 int:   intf;
```

subs: i of (our,I , WE, YOU);

Then the symbol table of subscript i will contain (Fig. 5) :

```
Entry is made for the symbol:   i
Parent: NULL
oldest child  : NULL
Sibling :  next sibling points to the following list
                              OUT_our
                              our
                              I
                              WE
                              YOU
type of use:  Subscript
file type  :  Sequential
type        :  yet unknown or the entry is not for a field
I/O flags  : 000 00000
Dimensionality  : 0
Other_pointers is NULL
Range :  NULL
```

It is to be noted here that OUT_our and our are both in the list because our belongs to a DIRECT file which has been declared both as input and as output.

2.2. Rec_case and Case structures

Equational programming language permits vary flexible file structures. Different records in a single file can be interpreted in different manners. This feature allows the user to keep non-similar records, that are to be treated uniformly by the algorithm, in a single file. There may be various structures of these records, i.e., there may be many different ways in which the chunk of bits in a record can be broken and interpreted by the user's program. To determine which of the many possible structures are to be used for interpretation, there is a field in the record which is at a constant distance from the beginning of the record. The value of this field is used to determine the structure to be used for interpretation.

Each different structure (more appropriately, this is a substructure) is preceded by the keyword "WHEN" (or it may be preceded by the keyword "OTHER", if it is the last substructure in the list of alternative substructures) in user's program. Immediately after "WHEN" is a binary expression that may involve the structure determining field mentioned above. If this binary expression is evaluated to the logical value "true" for the record being processed then the record is to be interpreted of the structure that follows the "WHEN". If there is more than one "WHEN" for which binexp evaluates to "true" (in this case the semantic of the grammar is not strictly defined) the first structure for which the binexp is "true" will be selected. If none of the binary expression is "true" then the structure following the keyword "OTHER" (if there is an alternative preceded by "OTHER") is selected for the interpretation of the record being read. Again the semantic of the grammar is not well defined in this case if there is no "OTHER" clause.

During parsing we need to store all of these user supplied information in a convenient way. For this purpose two structures Rec_case and Case is used. Also a new symbol table entry for the symbol "CASE" is to be used for this purpose.

Each time a new CASE is found in the source code while expanding the parse tree for <declaration> a new symbol table entry for a group named CASE (thus CASE is a reserved word) is created. The "other_pointers" field of this entry will point to the structure "Rec_case" which has two fields as described below. In the Symbol Table there may be many groups in the same file having the symbol name CASE , but that will not cause any referencing problem, because the user will never refer to any variable using the symbol name CASE.

1. st	This field contains a pointer to the symbol table entry for the symbol that appears in parenthesis right after "CASE" in the source code. This is the field name in the record which is at a fixed distance (in terms of bytes) from the beginning of the record and which determines how the information contained in the record being read from the disk storage are to be interpreted.
2. ptr	This field contains a pointer to the head of the link list of "Case" structures (described below). For each "WHEN" in the source code

(which defines a new alternative substructure) a new "Case" structure is created. A "Case" structure contains all the relevant information for a "WHEN" declaration.

A "Case" structure contains four fields:

For each WHEN a "Case" structure is created where the contents of the different fields are :

eq : This field contains a pointer to the equation tree node whose label is "case_rec" (described in the equation tree description) and whose right son points to the root of the "expression" tree that corresponds to the expression right after WHEN. This is the expression which is to be tested, each time a new record is read from the disk storage, to determine whether this substructure should be selected for breaking the information in the record read in a meaningful way.

st : This field contains a pointer to the symbol table entry for the first symbol (this is like a "oldest_child") that appears after WHEN. The substructure that is associated with the "WHEN" declaration being represented by this "Case" structure is pointed to by this field.

sublinear : Associated with each of the "WHEN" declaration there must be a sublinear subscript. This sublinear subscript is used for indexing through the same type of records (that is, records that are to be interpreted in the same manner, in other words records that are to be broken into different logical parts using the same substructure -- the substructure that is being defined by the "WHEN" expression under consideration) in the file being read. This field contains a pointer to the symbol table entry of this sublinear subscript and in the user's program this sublinear subscript is to be found within the current WHEN clause.

next : In the declaration there may be many "WHEN" clauses. All of the "WHEN" clauses that are under the same "CASE" are all to be found in a single link list. This field contains a pointer to the next "Case" structure that corresponds to the next "WHEN" clause after the current one. If this is the last "WHEN" clause (representing the last alternative for interpretation), then this field contains NULL.

The substructure followed by "OTHER" is treated uniformly as "WHEN". But in this case there is no equation tree. In this case the right son of the "case_rec" equation tree node points to NULL.

2.2.1. Definition of Rec_case and Case structures in "C"

The definition of Case and Rec_Case structures in C is shown below. These definitions are available in the file ~epl/include/defn.h file in the system sequent.

```
typedef struct Case {
      struct equation_tree_node  *eq;
      struct SYM_TAB_ENTRY        *st;
      struct SYM_TAB_ENTRY        *sublinear;
      struct Case              *next;
                };


typedef struct Rec_case{
      struct SYM_TAB_ENTRY *st;
      struct Case *ptr;};
```

2.2.2. Examples

In this section some examples are shown that illustrates the contents of the symbol table in different situations. In the beginning some examples are given that involves simple file structures. At the end of this section a more complicated example has been given which involves declaration of a file using CASE to select from alternative substructures.

Example 1 In this example a simple file structure is considered. The file structure is as follows :

```
file:a,
   1 rec : b[*],
      2 group: c[5],
         3 int : d;
```

Nothing is mentioned about the I/O status of the file (The I/O definition may be available at a later stage of parsing).

The content of the symbol table entries for each of the data items in this file structure is given below. Fig. 6 depicts the file structure that is embedded in the symbol table.

```
Entry is made for the symbol:   a
Parent: NULL
Oldest child : b
next sibling : NULL
type of use:  File
file type  :  Sequential
type       :  yet unknown or the entry is not for a field
I/O flags  : 000 00000
Dimensionality  : 0
Other_pointers is NULL
Range :  NULL


Entry is made for the symbol:   b
Parent : a
Oldest child : c
next sibling : NULL
type of use:  Record
file type  :  Sequential
```

```
type        :  yet unknown or the entry is not for a field
I/O flags  : 000 00000
Dimensionality  : 1
Other_pointers is NULL



Entry is made for the symbol:   c
Parent : b
Oldest child : d
next sibling : NULL
type of use:  Group
file type  :  Sequential
type        :  yet unknown or the entry is not for a field
I/O flags  : 000 00000
Dimensionality  : 1
Other_pointers is NULL

Range : Following are the ranges defined --
         is_static: 1,  ceiling:  5,  type



Entry is made for the symbol:   d
Parent : c
Oldest child  : NULL
next sibling : NULL
type of use:  Field
file type  :  Sequential
type        :  integer
I/O flags  : 000 00000
Dimensionality  : 0
Other_pointers is NULL
Range :  NULL
```

Example 2

This example shows the content of the symbol table (relevant portions) in the case of a PORT file. Though user has declared only one file, in this case in the symbol table there will be two different file structures. The file declaration is divided into two different files for PORT files which are of the type "IN => OUT" or "OUT => IN". In these two cases the PORT file is declared both as "input" and as "output". After breaking up the file two file structures are created. One of them is treated as "input" file the other one is treated as "output" file. In the declaration of these type of file there are always exactly two record definitions. One of them is "record in" the other one is "record out". the "input" file is given the "record in" record and the "output" file is given the "record out" record. The "input" file retains the same name as that of the original file (as declared by the user), the "output" file along with all the data items that belongs to the "output" file have their name changed to another name which is obtained by prefixing "OUT_" to their original name. The DIRECT files which are declared as "input" as well as "output" are also treated in the same manner. Only difference is that both the file in the case of DIRECT file have identical file structure. The naming convention for the two files created are same for DIRECT file and PORT file. If the PORT file is not any of these types then the file will not be broken.

The example uses the following file declaration

```
file: inmes (port in=>out),
        10 rec in : inrec,
                20 int: start,stop,
        10 rec out : outrec,
                20 int: res;
```

The relevant symbol table entries are as follow (figure 7 shows the structures of the files) :

```
Entry is made for the symbol:   INMES
Parent: NULL
Oldest child : INREC
next sibling : NULL
type of use:  File
file type  :  Port
type       :  yet unknown or the entry is not for a field
I/O flags  : 100 000000
Dimensionality  : 0
Other_pointers is NULL
Range :  NULL
Eqt_ptr :  NULL


Entry is made for the symbol:   INREC
Parent : INMES
Oldest child : START
next sibling : NULL
type of use:  Record
file type  :  Port
type       :  yet unknown or the entry is not for a field
I/O flags  : 100 100000
Dimensionality  : 0
Other_pointers is NULL
Range :  NULL
Eqt_ptr :  NULL
```

```
Entry is made for the symbol:   START
Parent : INREC
oldest child  : NULL
Next sibling : STOP
type of use:  Field
file type  :  Port
type        :  integer
I/O flags  : 100 000000
Dimensionality  : 0
Other_pointers is NULL
Range :  NULL
Eqt_ptr :  NULL


Entry is made for the symbol:   STOP
Parent : INREC
oldest child  : NULL
next sibling : NULL
type of use:  Field
file type  :  Port
type        :  integer
I/O flags  : 100 000000
Dimensionality  : 0
Other_pointers is NULL
Range :  NULL
Eqt_ptr :  NULL


Entry is made for the symbol:   OUT_INMES
Parent: NULL
Oldest child : OUT_OUTREC
next sibling : NULL
type of use:  File
file type  :  Port
type        :  yet unknown or the entry is not for a field
I/O flags  : 010 000000
Dimensionality  : 0
Other_pointers is NULL
```

```
Range :  NULL
Eqt_ptr :  NULL


Entry is made for the symbol:   OUT_OUTREC
Parent : OUT_INMES
Oldest child : OUT_RES
next sibling : NULL
type of use:  Record
file type  :  Port
type       :  yet unknown or the entry is not for a field
I/O flags  : 010 200000
Dimensionality  : 0
Other_pointers is NULL
Range :  NULL
Eqt_ptr :  NULL


Entry is made for the symbol:   OUT_RES
Parent : OUT_OUTREC
oldest child  : NULL
next sibling : NULL
type of use:  Field
file type  :  Port
type       :  integer
I/O flags  : 010 000000
Dimensionality  : 0
Other_pointers is NULL
Range :  NULL
Eqt_ptr points to the equation tree node:   0X22380
```

Example 3

For a more complicated example using "CASE" in the declaration of file structures the following example is used.

The file is declared as follows :

```
FILE : geometry_file (SEQ),
  5 RECORD : geometric[* <= 500],
    10 CHAR(9) : name,
    10 LOGICAL : color,
    10 CASE (name)
         WHEN "point":
           15 REAL : xp,
         SUBLINEAR sub_point
         WHEN "plane":
           15 GROUP : XX[3],
             20 REAL : xpl,
           15 GROUP : YY[3],
             20 REAL : ypl,
         SUBLINEAR sub_plane,
    10 CASE (color)
         WHEN TRUE :
           15 INT : hue,
         SUBLINEAR sub_color;
```

The corresponding symbol table entries and the equation trees pointed to by the other_pointers of the sublinear subscript entries are as follows (Fig. 8 shows the structure of the symbol table along with the Rec_case and Case structures for this example) :

```
Entry is made for the symbol:   geometric_file
Parent: NULL
Oldest child : geometric
next sibling : NULL
```

type of use: File
file type : Sequential
type : yet unknown or the entry is not for a field
I/O flags : 100 00000
Dimensionality : 0
Other_pointers is NULL
Range : NULL

Entry is made for the symbol: geometric
Parent : geometry_file
Oldest child : name
next sibling : NULL
type of use: Record
file type : Sequential
type : yet unknown or the entry is not for a field
I/O flags : 100 00010
Dimensionality : 1
Other_pointers is NULL

Entry is made for the symbol: CASE
Parent: geometric
oldest child : NULL
next sibling : CASE
type of use: Undef 0
file type : Sequential
type : yet unknown or the entry is not for a field
I/O flags : 100 01000
Dimensionality : 0
Other_pointers points to the rec_case entry for name
Range : NULL

Entry is made for the symbol: name
Parent : geometric

```
oldest child  : NULL
Next sibling : color
type of use:  Field
file type  :  Sequential
type       :  char(9)
I/O flags  : 100 00000
Dimensionality  : 1
Other_pointers is NULL
Range :  NULL
```

```
Entry is made for the symbol:   CASE
Parent: geometric
oldest child  : NULL
next sibling : NULL
type of use:  Undef 0
file type  :  Sequential
type       :  yet unknown or the entry is not for a field
I/O flags  : 100 01000
Dimensionality  : 0
Other_pointers points to the rec_case entry for color
Range :  NULL
```

```
Entry is made for the symbol:   color
Parent : geometric
oldest child  : NULL
Next sibling : CASE
type of use:  Field
file type  :  Sequential
type       :  logical
I/O flags  : 100 00000
Dimensionality  : 1
Other_pointers is NULL
Range :  NULL
```

```
Entry is made for the symbol:   point
Parent: NULL
oldest child  : NULL
next sibling : NULL
type of use:  Undef 9
file type  :  Sequential
type        :  yet unknown or the entry is not for a field
I/O flags  : 000 00000
Dimensionality  : 0
Other_pointers is NULL
Range :  NULL


Entry is made for the symbol:   xp
Parent : CASE
oldest child  : NULL
next sibling : NULL
type of use:  Field
file type  :  Sequential
type        :  real
I/O flags  : 100 00000
Dimensionality  : 0
Other_pointers is NULL
Range :  NULL


Entry is made for the symbol:   sub_point
Parent: NULL
oldest child  : NULL
next sibling : NULL
type of use:  Subscript
file type  :  Sequential
type        :  yet unknown or the entry is not for a field
I/O flags  : 000 00000
Dimensionality  : 0
Other pointers points to the equation tree head:   0X1E160
Range :  NULL
```

```
Entry is made for the symbol:   plane
Parent: NULL
oldest child  : NULL
next sibling : NULL
type of use:  Undef 9
file type  :  Sequential
type       :  yet unknown or the entry is not for a field
I/O flags  : 000 00000
Dimensionality  : 0
Other_pointers is NULL
Range :  NULL



Entry is made for the symbol:   XX
Parent: NULL
Oldest child : xpl
Next sibling : YY
type of use:  Group
file type  :  Sequential
type       :  yet unknown or the entry is not for a field
I/O flags  : 100 00000
Dimensionality  : 1
Other_pointers is NULL

Range : Following are the ranges defined --
        is_static: 1,  ceiling:  3,  type



Entry is made for the symbol:   xpl
Parent : XX
oldest child  : NULL
next sibling : NULL
type of use:  Field
file type  :  Sequential
type       :  real
I/O flags  : 100 00000
```

```
Dimensionality  : 0
Other_pointers is NULL
Range :  NULL


Entry is made for the symbol:   YY
Parent: NULL
Oldest child : ypl
next sibling : NULL
type of use:  Group
file type  :  Sequential
type       :  yet unknown or the entry is not for a field
I/O flags  : 100 00000
Dimensionality  : 1
Other_pointers is NULL

Range : Following are the ranges defined --
        is_static: 1,  ceiling:  3,  type


Entry is made for the symbol:   ypl
Parent : YY
oldest child  : NULL
next sibling : NULL
type of use:  Field
file type  :  Sequential
type       :  real
I/O flags  : 100 00000
Dimensionality  : 0
Other_pointers is NULL
Range :  NULL


Entry is made for the symbol:   sub_plane
Parent: NULL
oldest child  : NULL
next sibling : NULL
type of use:  Subscript
```

```
file type  :  Sequential
type       :  yet unknown or the entry is not for a field
I/O flags  : 000 00000
Dimensionality  : 0
Other pointers points to the equation tree head:   0X1E200
Range :  NULL


Entry is made for the symbol:   TRUE
Parent: NULL
oldest child  : NULL
next sibling : NULL
type of use:  Undef 0
file type  :  Sequential
type       :  logical
I/O flags  : 000 00000
Dimensionality  : 0
Other_pointers is NULL
Range :  NULL


Entry is made for the symbol:   sub_color
Parent: NULL
oldest child  : NULL
next sibling : NULL
type of use:  Subscript
file type  :  Sequential
type       :  yet unknown or the entry is not for a field
I/O flags  : 000 00000
Dimensionality  : 0
Other pointers points to the equation tree head:   0X1E320
Range :  NULL



Entry is made for the symbol:   hue
Parent : CASE
oldest child  : NULL
```

```
next sibling : NULL
type of use:  Field
file type  :  Sequential
type       :  integer
I/O flags  : 100 00000
Dimensionality  : 0
Other_pointers is NULL
Range :  NULL
```

The equation trees that are being pointed to by the other_pointers field in the symbol table entries for the sublinear subscript entries are as follows:

```
------------------------------------------------------
Following are the sublinear subscripts found in the specification

 Name of the sublinear subscript:   sub_color
 The other_pointers points to the following equation tree:
------------------------------------------------------
addr-   0X1E320, label-       <literal>, left-     TRUE, right-        0,
parent-  0X21100, type-    void, cnvrt-    void
------------------------------------------------------

 Name of the sublinear subscript:   sub_plane
 The other_pointers points to the following equation tree:
------------------------------------------------------
addr-   0X1E200, label-      <variable>, left-    plane, right-        0,
parent-  0X1E600, type-    void, cnvrt-    void
------------------------------------------------------

 Name of the sublinear subscript:   sub_point
 The other_pointers points to the following equation tree:
------------------------------------------------------
addr-   0X1E160, label-      <variable>, left-    point, right-        0,
parent-  0X1E600, type-    void, cnvrt-    void
------------------------------------------------------
```

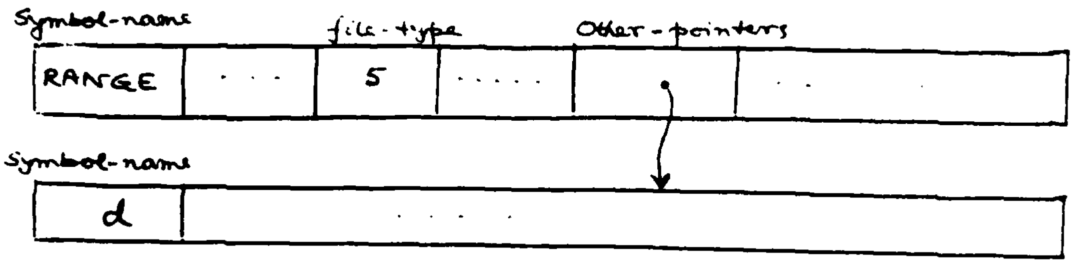


Fig. 2

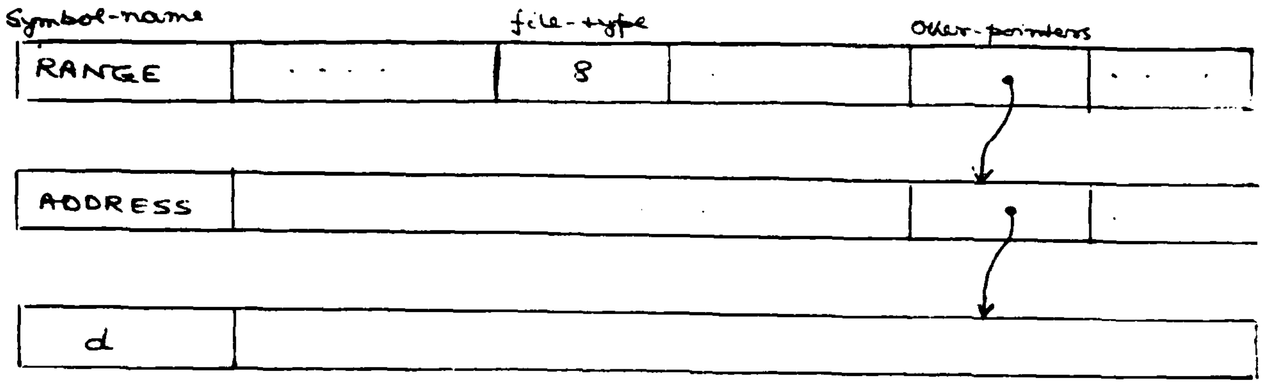


Fig. 3

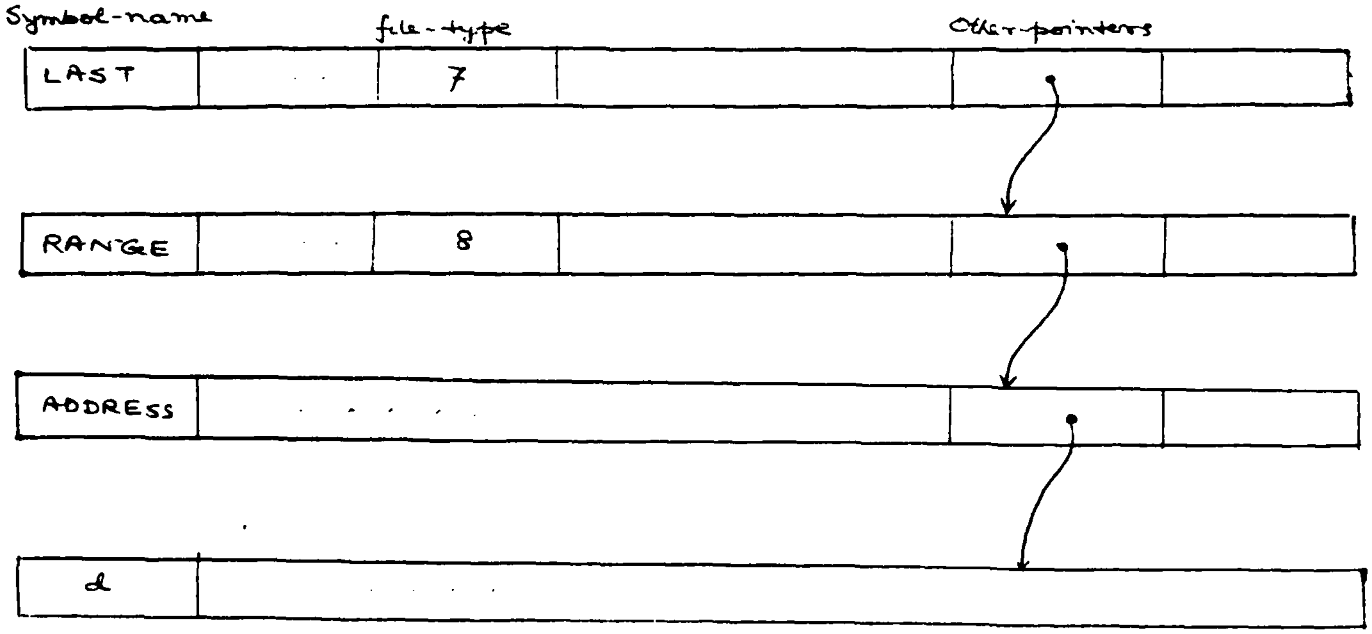
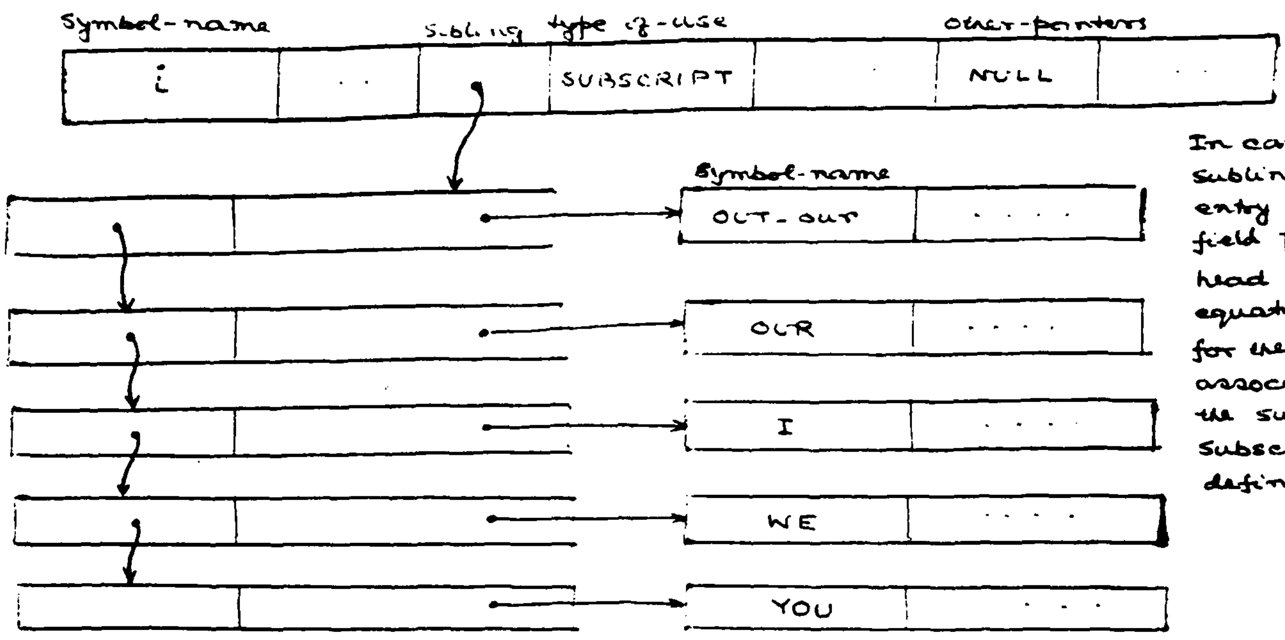


Fig. 4



In case of sublinear subscrip entry the other-point field points to the head of the equation tree for the "binexp" associated with the sublinear subscript definition.

Fig - 5

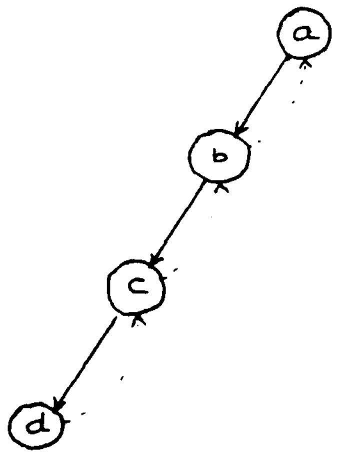


Fig-6 :
Solid arrows - pointer to old-child
dashed arrows - parent pointers.

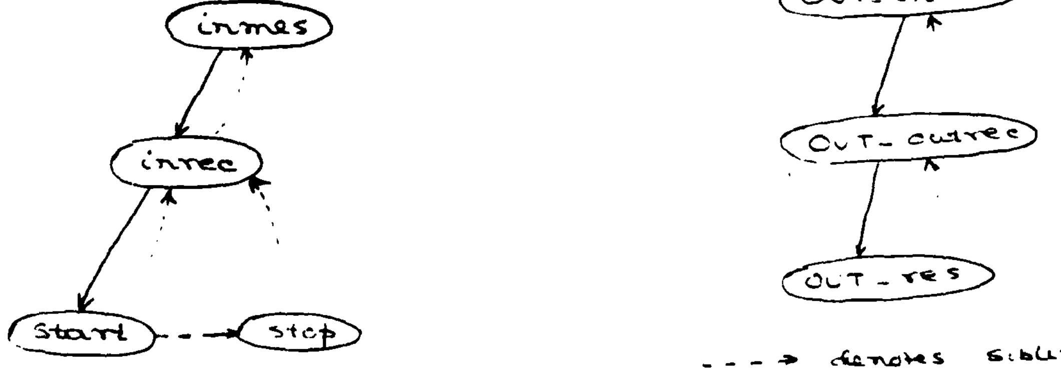


Fig. 7

- - -> denotes sibling pointer
↙ denotes old-child pointer.
↑ denotes parent pointer.

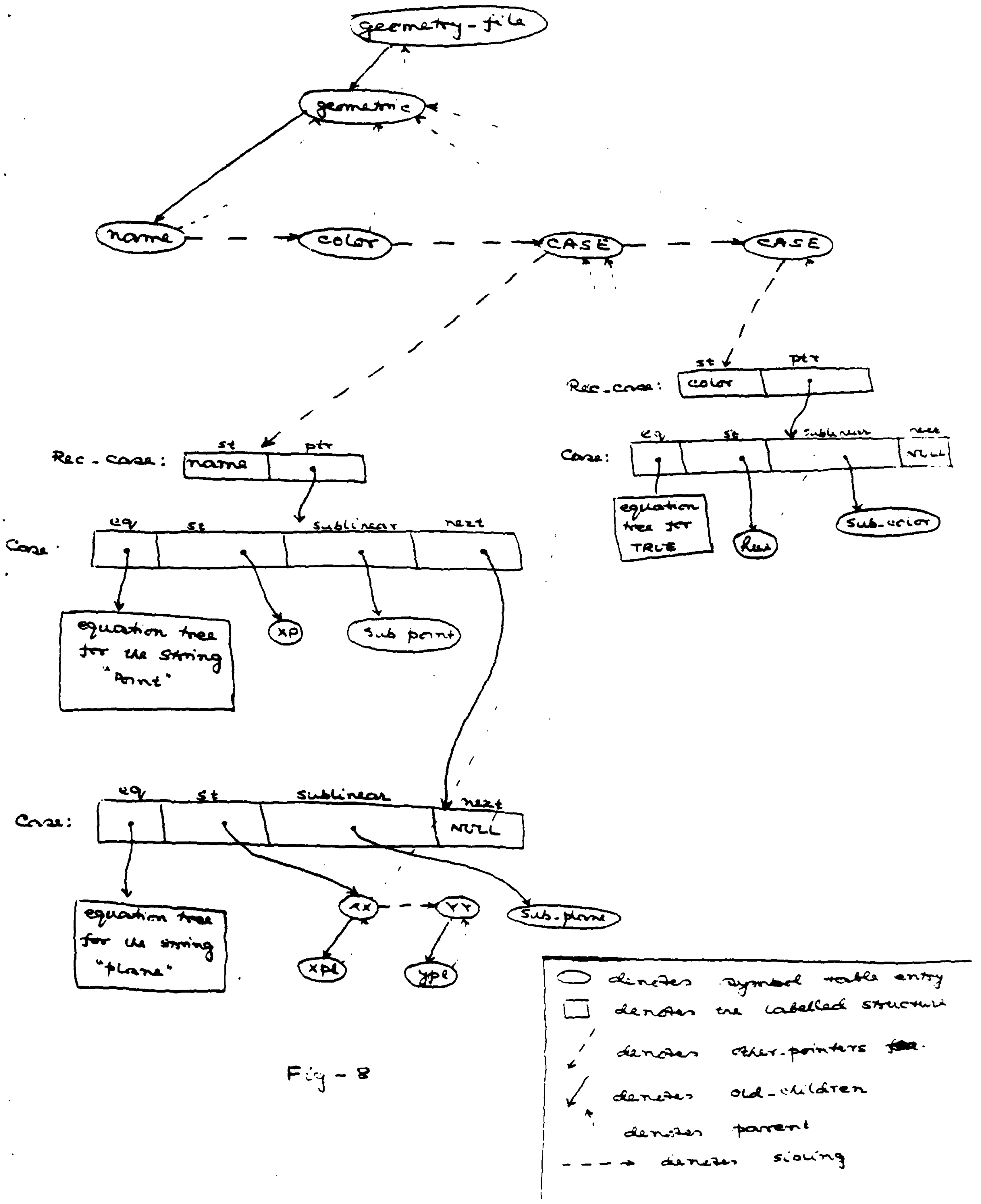

geometry-file
geometric
name
color
CASE
CASE
Rec-case:
st
ptr
name
Case:
eq
st
sublinear
next
equation tree for the string "Point"
xp
sub point
Case:
eq
st
sublinear
next
NULL
equation tree for the string "plane"
xx
yy
sub-plane
xpl
ypl
Rec-case:
st
ptr
color
Case:
eq
st
sublinear
next
NULL
equation tree for TRUE
Red
sub-color
denotes symbol table entry
denotes the labelled structure
denotes other-pointers
denotes old-children
denotes parent
denotes sibling

Fig - 8

2.3. Equation Trees

2.3.1. General Structure

The equation trees generated by the parser are teh linked lists of equation tree roots with a scalar pointer variable indicating the first tree on the list. The equation trees on this list will be made up of a collection of tree nodes connected together as a binary tree. The logical structure of a single tree node is described below.

The fields of this structure are defined as follows:

label :	The "label" field will be used to indicate what semantic entity the node represents. The field will be an integer and the semantic entities are assigned specific codes, given in table two, to be used in this field.
left :	The field labeled "left" will be a pointer to the left child of the equation tree node. The pointer can be a pointer to another tree node, a pointer to a symbol table entry, or it can be nil.
right :	The "right" field will be a pointer to the right child of the node. The right child of an equation tree node will always be another equation tree node or nil.

parent : A pointer to the parent of each node will be recorded in the "parent" field. The <assertion> node will have a pointer to the tree list node which points directly to it. The <sublin> node will have a pointer to the symbol table entry for the sublinear subscript that the tree is a condition of. The <case_rec> node will have a pointer to the symbol table entry for the record name that the case expression is a condition of.

type : The "type" field will indicate the data type of the expression or literal that the node represents. The values that can be entered in this field are the integer encodings of the EPL data types plus codes to indicate when an error has been encountered during data type propagation. Initially this field will be zero for all nodes. The data type encodings are given in table one.

convert_from : The "convert_from" field will be used to alert the code generator that the expression or literal represented by the node must be converted from some type indicated by this field to the type indicated by the field previously described. Initially this field will be zero for all nodes. During type checking the value of "type" and "convert_from" will be changed whenever necessary to reflect type conversion necessary for expression evaluation.

The logical structure for the nodes used to hold individual equation trees in the list of equation tress is described below.

The fields of this structure are defined as follows:

tree :	The "tree" field is a pointer to the root of the equation tree that the node links in the list.
is_range :	If the variable defined by the assertion(s) is a "RANGE" prefixed item, then the "is_range" flag will be set to true to denote this fact. For normal variables "is_range" is always false.
target :	A pointer to the symbol table entry for the variable defined by the assertion is loaded into the "target" field of the structure.
multi_def :	If some variable or range is specified as the target of more than one assertion, all definitions of the quantity after the first will be linked to the first definition through the "multi_def" pointer field. Duplicate definitions will not appear in the main equation tree list.
assertion :	The relative position of the assertion within the source file will be

used to mark the assertion when identifying error and warning messages to the user.

next :	The pointer to the next tree in the list of trees will be held in "next." An empty "next" pointer indicates the end of the list.

The definition of the tree nodes, linked list nodes and the linked list head pointer will be as follows:

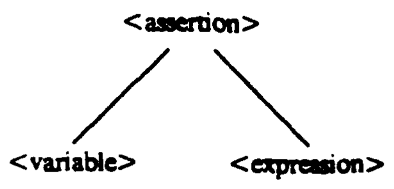
```
typedef struct Eq_tree_node {              /* binary tree node template */
    short                   label;       /* type of node              */
    union {
     Sym_tab_entry             *symbol;     /* For child from symtab     */
     struct Eq_tree_node *tree;            /* for child that is tree    */
        }                   left;        /* left child pointer        */
    struct Eq_tree_node  *right;           /* right child pointer       */
    union {
     struct Eq_tree_node *tree;            /* for normal parent         */
     struct Eq_tree_list *list;          /* for parent of <assertion> */
     Sym_tab_entry             *symbol;     /* for parent of <sublin>
                                              or <case_rec>  */
        }                   parent;      /* parent pointer            */
    short                   type;        /* arithmetic type of result */
    short                   cnvrt_from;  /* type to convert from      */
  } Eq_tree_node;

typedef struct Eq_tree_list {              /* node to hold tree in list */
    Eq_tree_node               *tree;      /* pointer to head of tree   */
    short                   is_range:1;  /* flag for range definition */
    Sym_tab_entry              *target;    /* entity defined            */
    struct Eq_tree_list      *multi_def;  /* multiple defintion chain  */
    short                   assertion;   /* assertion label           */
    struct Eq_tree_list      *next;       /* next node in list         */
  } Eq_tree_list;

extern Eq_tree_list *equation_trees;
```

	LABEL	INTEGER CODE
#define	ASSERTION	3
#define	SUBLIN	4
#define	CASE_REC	5
#define	IF_EXP	6
#define	IF_ACTION	9
#define	CASE_EXP	12
#define	WHEN_TEST	15
#define	WHEN_ACTION	18
#define	OTHER_ACTION	21
#define	VARIABLE	24
#define	INDEX	27
#define	BINARY_OR	33
#define	BINARY_AND	34
#define	BINARY_EQ	35
#define	BINARY_NE	36
#define	BINARY_GE	37
#define	BINARY_LE	38
#define	BINARY_GT	39
#define	BINARY_LT	40
#define	BINARY_CONCAT	41
#define	BINARY_PLUS	42
#define	BINARY_MINUS	43
#define	BINARY_TIMES	44
#define	BINARY_DIVIDE	45
#define	BINARY_POWER	46
#define	UNARY_PLUS	51
#define	UNARY_MINUS	52
#define	UNARY_NOT	53
#define	FCALL	60
#define	PARMS	63
#define	LITERAL	66
#define	SUB	69

Table 4 : Table showing the integer codes used to label equation tree nodes.

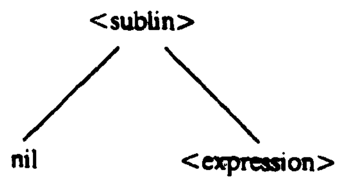
2.3.2. Node Label Definitions

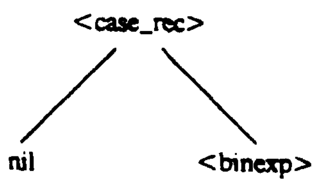
The labels for the various nodes that the equation trees will be constructed from are defined as follows.



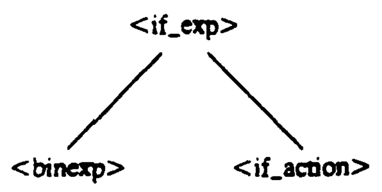
<expression> ::= <if_exp> | <case_exp> | <binexp>

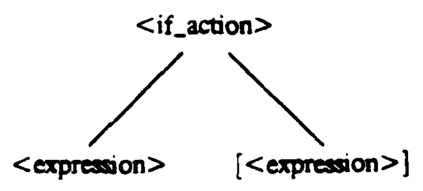
The BNF-like notation above is intended to indicate that the symbol <expression> is actually a generic form for any expression, including if-then constructs, case constructs, or simple expressions.





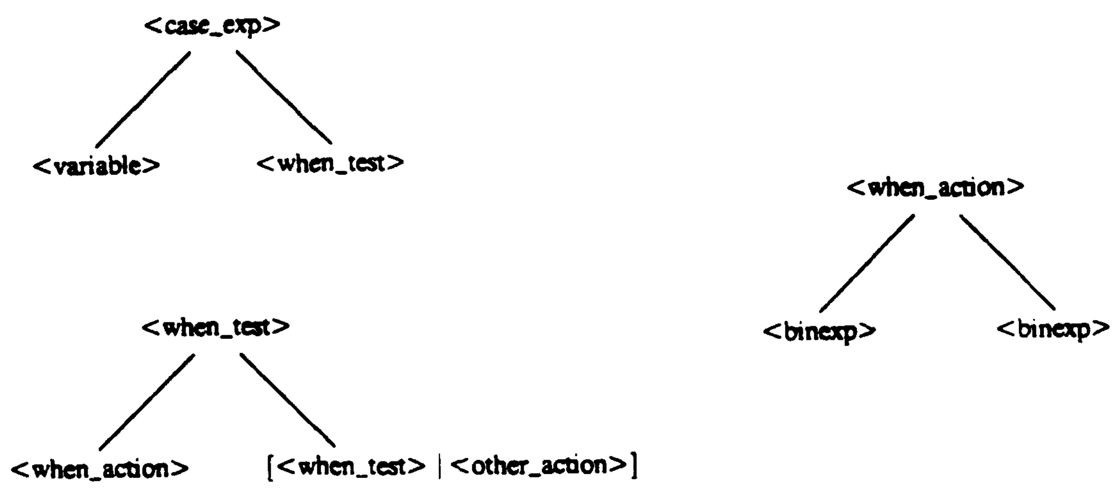
The <assertion> node will be used to link assertions into the equation tree list. The <sublin> and <case_rec> nodes will be used to link sublinear subscript expressions and record level case expressions into the symbol table. When the trees are being scanned from the bottom up, these three node types will indicate what type of parent the tree has so that information can be drawn from the equation tree header block or symbol table entry when necessary.



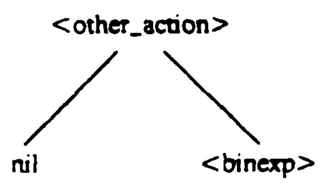


The <expression> on the left is evaluated if the <binexp> of the parent <if_exp> is true. If the right <expression> pointer is not nil this expression is evaluated if the <binexp> of the parent <if_exp> is false.

Note that there is no unique node for ELSEIF. Since many languages do not provide an "elseif" clause for their "if-else" constructs it is best to convert the "if-else-endif" into nested "if"'s at the time the source program is parsed to reduce the work in code generation and specification analysis.

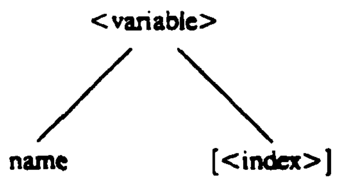


The <binexp> on the left is the case-expression that is to be tested for "<variable> == <binexp>." The <binexp> on the right is the expression that is to be evaluated if the above test is true.

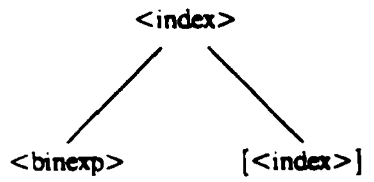


The <binexp> on the right is the expression to be evaluated if all previous case expressions were false. The <binexp> is placed on the right and the left link remains nil for the sake of consistency with <when_action>.

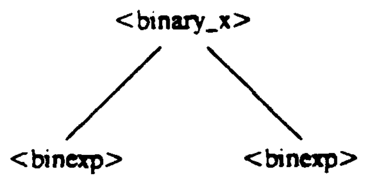
```
<binexp> ::= <variable> | <binary_|> | <binary_&> | <binary_==> | <binary_!=> |
             <binary_>=> | <binary_<=> | <binary_>> | <binary_<> |
             <binary_|> | <binary_+> | <binary_-> | <binary_*> |
             <binary_/> | <binary_**> | <unary_+> | <unary_-> |
             <unary_~> | <fcall> | <literal> | <SUB>
```

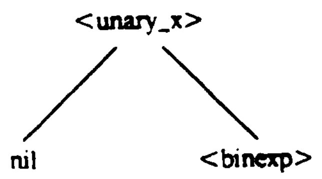


Where "name" is a pointer to the symbol table entry for the name referenced.

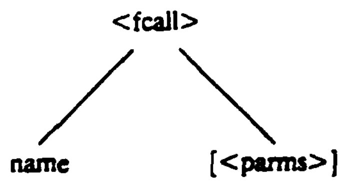


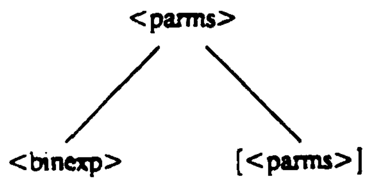
The generic form for binary and unary expressions will be as follows:

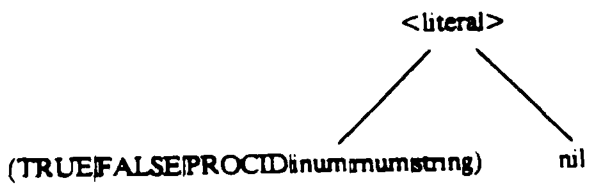


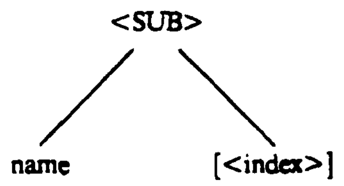


The <binexp> is placed on the right to allow the same in-order traversal to be used for both binary and unary sub-trees.









All variables that are found to be subscripts during analysis will be converted to <SUB> nodes and reinserted in the tree. The keywords SUB0 through SUB9 (if used in the user's program) will appear in the symbol table suffixed by an underscore followed by the equation number in which it is used (for example, if SUB5 is used in the equation number 4 then its symbol table entry will be SUB5_4). By making this change it is made sure that the different use of the same SUB0 (or other SUB) will have seperate entries, so the equation tree will point to the correct reference.

The codes that will be loaded into the "label" field of the equation tree nodes are defined in the following table. Note that code numbers are sparsely assigned so that future enhancements to the language can be inserted into the table in logical locations.

Type	value
<assertion>	3
<sublin>	4
<case_rec>	5
<if_exp>	6
<if_action>	9
<case_exp>	12
<when_test>	15
<when_action>	18
<other_action>	21
<variable>	24
<index>	27
<binary_|>	33
<binary_&>	34
<binary_==>	35
<binary_!=>	36
<binary_>=>	37
<binary_<=>	38
<binary_>>	39
<binary_<>	40
<binary_|>	41
<binary_+>	42
<binary_->	43
<binary_*>	44
<binary_/>	45
<binary_**>	46
<unary_+>	51
<unary_->	52
<unary_~>	53
<fcall>	60
<parms>	63
<literal>	66
<SUB>	69

Table 5: Table showing the integer codes used for different equation tree nodes.

2.4. Range Table

2.4.1. General Structure

In our attempt to develop consistent range specifications from the program written by the user there will be three major phases. The first phase will take place at the time the user's specification is parsed as we draw from the user's declarations as much information as we can regarding the structures defined by the user. The second phase begins while we are constructing the array graph, as we mark the nodes in the array graph with the subscript and range information that can be inferred from the user's equations. Once the nodes in the array graph have all been set in place and decorated with the range information obtained from the assertions, range propagation procedures will begin combining all the information gathered into sets of ranges that are required to execute the user's program.

To fulfill the requirements of the first phase, we will make definition blocks that can be pointed to by any structure in the symbol table that is declared with ranges, or any structure that is not declared but is discovered to have ranges during dimension propagation. Definition blocks will be linked together in a list with the head of the list being a pointer in the symbol table for an entry that has ranges. Blocks will be placed in the list in the same order that they are declared, with the leftmost range of an entity being the first range on the list. Each block will have fields containing information found at compile time, as well as fields that are intended to be filled during later analysis and optimization phases.

Another set of range definition blocks will be constructed later, after range propagation is complete. For each range set that is formed for the specification, a single range definition block will be created. This block will be pointed to by the "def_list.rdb" pointer in the range set header structure (described below). The block will contain all information relevant to the range, including all dynamic definitions and the single ceiling that pertains to all subscripts indexing the range.

The logical structure for a range definition block is given below.

The fields of this structure are defined as follows:

def :	The first field in the structure will be a pointer to the head of a possible list of dynamic definition blocks (described below) that apply to the range in question. The "def" field will not be filled until the initial phases of range propagation associate RANGE assertions and end-of-file items with the ranges that they define. If, after range propagation, the "def" field is nil, the range has no dynamic definitions.
sub_name :	The "sub_name" field is a field that is provided for use in the code

	generation phase for holding temporary subscript names that apply to ranges. This field will not be loaded with any meaningful data until code generation begins.
is_static :	If the user defined the range as having an explicit upper bound, the range is considered static and the "is_static" flag is set to true. If the range was defined by an asterisk, or an asterisk and an upper limit, the dimension is considered dynamic and the "is_static" flag is set to false.
ceiling :	Whenever a numeric bound is defined for a range the "ceiling" field will contain the value of that bound. If the range has neither a static bound nor a dynamic bound the value of this field will be zero.
window :	If the optimization phase of translation windows a dimension it will load the length of the window into the "window" field. The "window" field will only be meaningful if the value of the "type" field (described below) is 'W.'
type :	Once optimization is complete, the "type" field for each dimension will hold a single character constant indicating what degree of static/dynamic character the range has. The value 'P' will indicate a

physical dimension, 'V' will indicate a virtual dimension and 'W' will indicate that the dimension has been windowed. If the range is windowed, the window length will be loaded into the "window" field. The "type" field will be loaded with a blank when the block is initially constructed.

range_set : To allow easy access to the range set that any range definition block is a member of, the "range_set" pointer will be used. This pointer will be nil initially, and range propagation will load it with a meaningful value once the range sets are created.

next : The next range definition block in the linked list of range definition blocks attached to a symbol table entry will be pointed to by the "next" field. If the field is empty the list ends. This field will always be empty when the block is used as the common block for a range definition at the close of range propagation.

3. ALGORITHMS FOR LEXICAL ANALYSIS

3.1. The Objective

The lexical analysis is the first phase of the EPL system. As mentioned before, the purpose of this phase is to break the input stream of characters (i.e., the user-specification into a stream of tokens. By doing this we reduce the volume of information processing required by the later phases of the translator. This phase works interactively with the parser. Each time parser needs another token for building the parse tree it calls the lexical analyzer, which in turn absorbs some of the characters from the input stream (keeping a pointer, to indicate where to start reading, in the input stream next time it is called by the parser) and returns the token that causes the largest number of characters to be absorbed from the input stream. To be more specific, in cases when the characters at the top of the input stream can produce more than one token the decision is made in favor of the token that corresponds to the largest number of input characters. If there are more than one token that corresponds to the largest number of input characters then the token that is defined earlier among these are chosen.

For the lexical analysis purpose LEX has been used. Since there are some features in the grammar of EPL that causes the grammar not to belong to LALR (1), some tricks has been used through the lexical analyzer to resolve this problem (since we are using YACC for parsing, we need to have our grammar in LALR (1)). The parser is confused when faced with ',' in a few places (i.e., it has more than one alternative to proceed with parsing). For that reason lookahead is done for ',' so that different tokens are supplied to the parser in these situations. Following is the LEX specification for the EPL system.

3.2. The LEX specification.

```
   %{
#include <stdio.h>

extern FILE *yyin;
static int line=1;
int i;
%}

%START  format imply ordinary

letter  [A-Za-z]
digit   [0-9]
space   [ 0]
comment "/*"("/"|"*"*[^"*""/"])*"*"*"*/"
comma ","({space}|{comment})*

A [Aa]
B [Bb]
C [Cc]
D [Dd]
E [Ee]
F [Ff]
G [Gg]
```

```
H [Hh]
I [Ii]
J [Jj]
K [Kk]
L [Ll]
M [Mm]
N [Nn]
O [Oo]
P [Pp]
Q [Qq]
R [Rr]
S [Ss]
T [Tt]
U [Uu]
V [Vv]
W [Ww]
X [Xx]
Y [Yy]
Z [Zz]

%p 9999
%a 4000


%%
";"   {  return(';');}
":"   {  return(':');}
{comma}{S}{U}{B}{L}{I}{N}({E}{A}{R})?{space}    {
                                        return(_COM_SUBL); }
{comma}/{digit}                            {
                                        return(_COMMA_BEFORE_INUM);  }
","   {  return(',');}
"["   {  return('[');}
"]"   {  return(']'); }
"("   {  return('('); }
")"   {  return(')'); }
"|"  {  return('|');}
"&"   {  return('&');}
"~"   {  return('~');}
```

```
"+"  { return('+');}
"/"  { return('/'); }
"-"  { return('-');}
"*"  { return('*'); }
"="  { return('='); }
"."  { return('.');}
"**"  { return(_POWER);}
"|"  { return(_CONCAT);}
<imply>"=>"  { return(_IMPLY);}
<imply>"<="  { return(_REV_IMPLY);}
<format>"E"  { return('E');}


{P}{R}{O}{C}({E}{S}{S})?                { return(_PROCESS);}
{F}{U}{N}{C}({T}{I}{O}{N})?              { return(_FUNCTION);}
{S}{E}{Q}({U}{E}{N}{T}{I}{A}{L})?         { return(_SEQUENTIAL);}
{I}{N}/({space})*"."                  { return(_IN);}
{O}{U}{T}/({space})*"."                 { return(_OUT);}
{I}{N}({P}{U}{T})?                     { return(_INPUT);}
{O}{U}{T}({P}{U}{T})?                    { return(_OUTPUT);}
{D}{I}{R}{E}{C}{T}                      { return(_DIRECT);}
{P}{O}{R}{T}                          { return(_PORT);}
{D}{I}{S}{P}({L}{A}{Y})?                 { return(_DISPLAY);}
{A}{D}{D}{R}({E}{S}{S})?                  { return(_ADDRESS);}
{F}{I}{L}{E}                          { return(_FILE);}
{W}{H}{E}{N}                           { return(_WHEN);}
{G}{R}({O}{U})?{P}                      { return(_GROUP); }
{R}{E}{C}({O}{R}{D})?                    { return(_RECORD);}
{D}({O}{U})?{B}{L}({E})?                  { return(_DOUBLE);}
{I}{N}{T}({E}{G}{E}{R})?                  { return(_INTEGER);}
{C}{H}{A}{R}                           { return(_CHAR); }
{S}{H}{O}{R}{T}                         { return(_SHORT);}
{L}{O}{N}{G}                           { return(_LONG);}
{Q}{U}{A}{D}                           { return(_QUAD); }
{L}{O}{G}{I}{C}({A}{L})?                  { return(_LOGICAL);}
{S}{U}{B}{L}{I}{N}({E}{A}{R})?              { return(_SUBLINEAR); }
{R}{E}{A}{L}                           { return(_REAL); }
{S}{U}{B}{S}({C}{R}{I}{P}{T})?              { return(_SUBSCRIPT); }
{C}{A}{S}{E}                           { return(_CASE); }
```

```
{O}{T}{H}{E}{R}{S}              { return(_OTHERS);}
{R}{A}{N}{G}{E}                 { return(_RANGE); }
{T}{H}{E}{N}                    { return(_THEN); }
{E}{L}{S}{E}                    { return (_ELSE); }
{E}{N}{D}{I}{F}                 { return(_ENDIF); }
{E}{L}{S}({E})?{I}{F}           { return(_ELSEIF); }
{T}{R}{U}{E}                    { return(_TRUE); }
{F}{A}{L}{S}{E}                 { return(_FALSE); }
({P}{R}{O}{C})?{I}{D}           { return(_PROCID); }
{L}{A}{S}{T}                    { return(_LAST); }
{P}{R}{E}{S}{E}{N}{T}           { return(_PRESENT); }
{O}{F}                          { return(_OF); }
{I}{F}                          { return(_IF); }

{S}{U}{B}{digit}                  { /* local subscript */
                                Set the global variable "name" to _SUB
                                followed by the digit, followed by
                                underscore, followed by the equation
                                number in which it is found.
                                return(_SUB);  }

{letter}({letter}|{digit}|"_")*  {/* valid symbol */
                                Change all the letters in the array
                                "private_copy" to capital letters.
                                return(_NAME);
                               }
({digit})+                      {/* integer constant */
                                Set global array "inum_copy" to the
                                string for the integer.
                                return(_INUM);}
"=="                            { yylval.inum = 0;
                                return (_RELOP); }
(">="|"=>")                     { yylval.inum = 1;
                                return(_RELOP); }
("=<"|"<=")                     { yylval.inum = 2;
                                return(_RELOP); }
("<>"|"~=")                     { yylval.inum = 3;
                                return(_RELOP);  }
">"                             { yylval.inum = 4;
```

```
                                   return(_RELOP);  }
"<"                                { yylval.inum = 5;
                                   return(_RELOP); }


<ordinary>((({digit})+("."{digit}+)?)|(({digit})*("."{digit}+)))("E"("+"|"-")?{digit}+)?
{
                                   /* real constant */
                          yylval.st = (struct Sym_tab_entry *)install(yytext,0);
                                   return(_RNUM); }


"'"([^']|"''")*"'"            {  /* string constant */
                              Create appropriate symbol table entry for the
                              string, if it is not already created.
                              return(_STRING); }


"/*"("/"|"*"*[^"*""/"])*"*"*"*/"        { /* Do Nothing for the Comment */ }

"0 line++; { /* increment line number */}
[ ] ;       {/* ignore white spaces */}
%%
```

4. ALGORITHMS FOR SYNTAX ANALYZER

4.1. The Objective

The objective of the syntax analyzer is to parse the user's program using the EPL grammar. If at the time of parsing any syntax error is found it is reported to the user along with useful error diagnostics. The parser does not abort the processing in the event of syntax error. It does error recovery and proceeds to parse the rest of the user's program, so that more syntax errors can be reported. This reduces the number of time the user needs to recompile to rectify all the syntax error. If any syntax error is found then the parser returns '0' to the calling program (the main routine of the EPL system) to indicate that the processing of the latter phases need not be initiated. If there is no syntax error found in the user's program then the parser returns '1' to the calling program to indicate that the parsing has been done successfully, so the later phases of the system should be initiated.

Besides checking for the syntax errors the parser also builds many of the necessary global data structures which are to be used and modified appropriately by different parts of the EPL system. The two most important data structures that are built by the parser are the symbol table and the equation tree. The description of these two data structures has been given already in the preceding sections. Some parts of some of these data structures may not be filled during parsing, because of insufficient information available in the user's program at the time of parsing. If the information can be inferred from other information in the user's program then in some later phase of the EPL system, by extensive analysis, these attributes are determined and appropriate places in the data structures are filled. Some examples of these kind of attributes are data types, dimensions , etc. As mentioned before, EPL grammar allows the user to use a data item without defining it. In the data type propagation phase of the EPL system the type of these undeclared data items are determined (if possible) and the data type fields of the symbol table entries for these data items are filled in. The dimensions of many data items also can be left unspecified in an EPL program. The dimensionality propagation phase of the EPL system propagates the dimension from the data items for which the dimension is known to those for which it is not known.

Besides symbol table and equation trees, the other important global data structures built by the syntax analyzer which are used by some later phase of the EPL system includes the Case structures, the Rec_case structures, the range definition blocks , list of data items that has appeared as fields of some input or output file structures, list of files defined in the user's program, list of sublinear subscripts and the module structure.

The range definition block, Case structure and Rec_case structure has already been described in detail in an earlier section. List of data items that has appeared as fields of some input or output file structure is required for the dimension propagation phase. The dimension propagation starts from this list, because everything in this list has their dimension correctly defined in the user's program (otherwise it is a semantic error). The dimension of all the other data items are established from these data items. The global pointer "io_list" points to the head of this list. The global pointer "top_files" points to the list of pointers to the symbol table entries that corresponds to the files declared in the user's program. The global pointer "top_subl_lst" points to the list of sublinear lists. The module structure contains the information regarding the program module that are needed for outside communication, especially by the configurator system. The module structure keeps the name of the program module being compiled, the type of the module (that is, whether the module is a function or a procedure), and the range of the program module.

4.2. The Grammar

The parser has been built using YACC. The language that can be parsed using YACC are of type LALR(1). The EPL language is more complex than LALR(1). For this reason the EPL specification in EBNF form does not have any direct translation to the LÁLR(1). Some changes are made in the grammar specification so that the syntax analyzer can be written using YACC while at the same time it will be equivalent to the EBNF specification for the grammar. Most of the changes made are in the grammar rules. In a few cases more than one character lookahead was needed. Since YACC does only one character lookahead, when more than one character lookahead is needed for parsing it is done through the lexical analyzer. These situation arises when "," is found. The lexical analyzer looks ahead to determine whether "sublinear" or integer number follows "," or not. If "sublinear" follows "," then it returns the token "_COM_SUBL" , if some integer number follows "," then it returns the token "_COMMA_BEFORE_INUM" (of course, intervening spaces and comments are ignored). If neither follow "," then the lexical analyzer returns ',' to the syntax analyzer. In the following the integer values used for representing the tokens and the LALR(1) grammar used for YACC are shown.

4.2.1. The integer values used for representing the tokens

YACC assigns the following integer values for the token used in the yacc specification.

	Token	Integer Code
# define	_PROCESS	257
# define	_FUNCTION	258
# define	_SEQUENTIAL	259
# define	_INPUT	260
# define	_OUTPUT	261
# define	_DIRECT	262
# define	_PORT	263
# define	_DISPLAY	264
# define	_ADDRESS	265
# define	_FILE	266
# define	_WHEN	267
# define	_GROUP	268
# define	_RECORD	269
# define	_DOUBLE	270
# define	_INTEGER	271
# define	_CHAR	272
# define	_SHORT	273
# define	_LOGICAL	274
# define	_SUBLINEAR	275
# define	_REAL	276
# define	_SUBSCRIPT	277
# define	_CASE	278
# define	_OTHERS	279
# define	_RANGE	280
# define	_THEN	281
# define	_ELSE	282
# define	_ENDIF	283
# define	_ELSEIF	284

Table 6 : Table showing the integer values assigned to the tokens returned by the lexical analyzer (continued ..).

	Token	Integer Code
# define	_TRUE	285
# define	_FALSE	286
# define	_PROCID	287
# define	_LAST	288
# define	_PRESENT	289
# define	_OF	290
# define	_IF	291
# define	_SUB	292
# define	_NAME	293
# define	_INUM	294
# define	_RELOP	295
# define	_RNUM	296
# define	_STRING	297
# define	_IMPLY	298
# define	_REV_IMPLY	299
# define	_IN	300
# define	_OUT	301
# define	_LONG	302
# define	_QUAD	303
# define	_DIGIT	304
# define	_COMMA_BEFORE_INUM	305
# define	_COM_SUBL	306
# define	_CONCAT	307
# define	_POWER	308

Table 6 (continued) : Table showing the integer values assigned to the tokens returned by the lexical analyzer.

4.2.2. The LALR(1) grammar used by the YACC for parsing

```
    %{
#include <stdio.h>
#include "symbol_table.h"
#include "defn.h"
#include "defn1.h"
#include "eqtrees.h"
#include "arry_grph.h"
#include "rangetab.h"
#include "st_manipulator.h"
#include "file_lst.h"
#include "subl_lst.h"
#include "routines.h"
#include "itoa.h"
#include "copy_name.h"
#include "file_dump.h"
#include "pars_rout.h"
extern int debug;
extern int b_debug;
struct Sym_tab_entry *temporary;
struct Case *whn_ptr;
%}

%token <inum> _PROCESS _FUNCTION _SEQUENTIAL _INPUT
%token <inum> _DIRECT _PORT _CASE _DOUBLE _INTEGER
%token <inum> _DISPLAY _ADDRESS _FILE _WHEN _GROUP
%token <inum> _CHAR _SHORT _LOGICAL _SUBLINEAR _REAL
%token <inum> _OTHERS _RANGE _THEN _ELSE _ENDIF
%token <inum> _OUTPUT _SUBSCRIPT _TRUE _FALSE _RELOP
%token <inum> _PROCID _LAST _PRESENT _OF _IF _SUB
%token <inum> _RNUM _STRING  _IMPLY _REV_IMPLY _IN
%token <inum> _NAME _INUM _ELSEIF _RECORD _OUT
%token <inum> _LONG _QUAD  _DIGIT
%token <inum> _COMMA_BEFORE_INUM
%token <inum> _COM_SUBL
```

```
%nonassoc <inum> '='
%left <inum> '+' '-' '|'
%left <inum> '*' '/' '&' _CONCAT
%right <inum> _POWER

%type <inum> func_proc opt_fattr fattr porttype type rec

%type <name> format fflag opt_minus opt_E

%type <st> nlist substructure simpstructure

%type <eqt>

%type <rd> opt_range range

%type <cs> opt_others when_exp

%type <sp> nlist2 name1 opt_nlist

%union
{
 short sh;
 int inum;
 char *name;
 struct Sym_tab_entry *st;
 struct Range_def_blk *rd;
 struct Eq_tree_node *eqt;
 struct Rec_case  *rc;
 struct Case *cs;
 struct St_ptr_list *sp;
 struct Parm_list *pl;
}

%%
specification    : func_proc ':'  _NAME opt_range sem_col statement
                 ;

func_proc         : _PROCESS
                 | _FUNCTION
```

```
                ;

statement         : statement  header sem_col
                | statement declaration sem_col
                | statement equation sem_col
                | header sem_col
                | declaration sem_col
                | equation sem_col
                ;

header            : _INPUT ':' nlist
                | _OUTPUT ':' nlist
                ;

nlist           : nlist ',' _NAME
                | _NAME
                ;

declaration       : _FILE ':' _NAME opt_fattr  substructure_list
                | _GROUP ':' _NAME opt_range simpstructure_list
                | type ':' ndxlist
                | subscript
                ;

ndxlist          : _NAME opt_range
                | ndxlist ',' _NAME opt_range
                ;

opt_fattr         : '(' fattr  par
                |
                ;

simpstructure_list: simpstructure_list _COMMA_BEFORE_INUM  simpstructure
                | _COMMA_BEFORE_INUM simpstructure
                ;
```

```
substructure_list : substructure_list  _COMMA_BEFORE_INUM substructure
                  | _COMMA_BEFORE_INUM substructure
                  ;

fattr             : _SEQUENTIAL
                  | _DIRECT
                  | _DISPLAY
                  | _PORT
                  | _PORT
                      {BEGIN imply;}
                    porttype
                     {
                       BEGIN ordinary;
                     }
                  ;

porttype            : _INPUT _IMPLY _OUTPUT
                  | _OUTPUT _IMPLY _INPUT
                  ;

type               : _CHAR
                  | _CHAR '(' _INUM par
                  | _SHORT
                  | _INTEGER
                  | _LONG
                  | _REAL
                  | _DOUBLE
                  | _QUAD
                  | _LOGICAL
                  ;

rec               : _RECORD ':'
                  | _RECORD _INPUT ':'
                  | _RECORD _OUTPUT ':'
                  ;

opt_range            : '['
                       {
                         BEGIN imply;
```

```
                   }
                 range
                   {
                   BEGIN ordinary;
                   }
                bracket
              |
              ;


opt_others        : opt_WHEN_OTHERS ':' simps _COM_SUBL _NAME
              |
              ;

opt_WHEN_OTHERS   : _WHEN  _OTHERS
              | _OTHERS
              ;

when_exp          : when_exp  _WHEN binexp ':' simps  _COM_SUBL _NAME
              | _WHEN binexp ':' simps  _COM_SUBL _NAME
              ;

simps             : simps  _COMMA_BEFORE_INUM  simpstructure
              | simpstructure
              ;

substructure      : keep_num  rec   _NAME opt_range
              | case_proc '(' keep_name par  when_exp opt_others
              | case_proc '(' keep_name par  opt_others
              | simpstructure
              ;


simpstructure     : _INUM type ':' substr_rest_rep
              | keep_num _GROUP ':' keep_name opt_range
              | keep_num keep_name  opt_range
              ;
```

```
substr_rest_part  :  _NAME opt_range
                  ;

substr_rest        : substr_rest_part
                  | substr_rest_part format
                      {
                        BEGIN ordinary;
                      }
                  ;

substr_rest_rep   : substr_rest_rep ',' substr_rest
                  | substr_rest
                  ;

range              : range  ',' _INUM
                  | range  _COMMA_BEFORE_INUM _INUM
                  | range ',' '*'
                  | range _COMMA_BEFORE_INUM '*'
                  | range ',' _INUM _IMPLY '*'
                  | range _COMMA_BEFORE_INUM _INUM _IMPLY '*'
                  | range ',' '*' _REV_IMPLY _INUM
                  | _INUM
                  | '*'
                  | _INUM _IMPLY '*'
                  | '*' _REV_IMPLY _INUM
                  ;


format             : paren opt_minus _INUM  fflag  par
                  | paren  opt_minus '*' fflag  par
                  ;

opt_minus            : '-'
                  |
                  ;

fflag              : '.' _INUM '+' opt_E
                  | '.' _INUM opt_E
                  | '+' opt_E
```

```
                 | opt_E
                 ;

opt_E              : 'E'
                 |
                 ;

subs_part          : _SUBSCRIPT ':' keep_name _OF '(' prefnlist par
                 ;

subs_part2          :  _SUBSCRIPT ':' _NAME
                 ;

subscript          : subs_part subs_rest
                 | subs_part
                 | subs_part2 subs_rest
                 | subs_part2
                 | subs_part2 _SUBLINEAR '(' nlist2  par  binexp
                 ;

opt_nlist          : _OF '(' prefnlist  par
                 |
                 ;

prefnlist          :  prefnlist ',' prefname
                 |  prefname
                 ;

nlist2             : nlist2 ',' name1
                 | name1
                 ;

name1               : _NAME  '.' _NAME
                 | _NAME
                 ;
```

```
subs_rest      : subs_rest ',' _NAME opt_nlist
               | ',' _NAME  {$<name>$ = strsave(private_copy); } opt_nlist
               ;

equation       : variable '=' expression
               ;

variable       : range_rep varbase
               | varbase
               ;

range_rep      : range_rep1 _ADDRESS '.'
               | _ADDRESS '.'
               | range_rep1
               ;

range_rep1     : range_rep1 _LAST opt_inum '.'
               | range_rep1 _RANGE opt_inum '.'
               | _RANGE opt_inum '.'
               | _LAST opt_inum '.'
               ;


opt_inum       : '(' _INUM   par
               |
               ;

varbase        : prefname index
               | prefname
               ;

prefname       : _NAME '.' _NAME
               | _IN '.' _NAME '.' _NAME
               | _OUT '.' _NAME '.' _NAME
               | _NAME
               | _IN '.' _NAME
               | _OUT '.' _NAME
               ;
```

```
index           : '[' aexp_list bracket
                ;

aexp_list       : aexp_list ',' aexp
                | aexp_list _COMMA_BEFORE_INUM aexp
                | aexp
                ;

expression      : _IF binexp _THEN expression elseif_rep endif
                | _IF binexp _THEN expression endif
                | _IF binexp _THEN expression elseif_rep
                                             _ELSE expression endif
                | _IF binexp _THEN expression _ELSE expression endif
                | _CASE '(' variable  par  when_rep
                | _CASE '(' variable  par  when_rep opt_WHEN_OTHERS ':' binexp
                | _CASE '(' variable  par  opt_WHEN_OTHERS ':' binexp
                | binexp
                ;

elseif_rep      : elseif_rep _ELSEIF binexp _THEN expression
                | _ELSEIF binexp _THEN expression
                ;

when_rep        : when_rep _WHEN binexp ':' binexp
                | _WHEN binexp ':' binexp
                ;

binexp          : binexp '|' bterm
                | bterm
                ;

bterm           : bterm '&' bfactor
                | bfactor
                ;

bfactor         : strexp _RELOP {relop_val = yylval.inum; } strexp
                | strexp
                ;
```

```
strexp          : strexp _CONCAT sterm
                | sterm
                ;

sterm           : aexp
                | _STRING
                ;

aexp            : aexp sign term
                | sign term                 %prec _POWER
                | term
                ;

sign            : '+'
                | '-'
                ;

term            : term '*' factor
                | term '/' factor
                | factor
                ;

factor          : factor _POWER prim
                | '~' prim
                | prim
                ;

prim            : _RNUM
                | _INUM
                | '(' binexp  par
                | fcall
                | _PROCID
                | _PRESENT '.' varbase
                | variable
                | _TRUE
                | _FALSE
                | _SUB
                ;
```

```
fcall              : keep_name '(' opt_binexp_list  par
                   | keep_name '(' par
                   ;

opt_binexp_list    : binexp opt_binexp_rest
                   ;

opt_binexp_rest    : ',' binexp  opt_binexp_rest
                   | _COMMA_BEFORE_INUM  binexp  opt_binexp_rest
                   |
                   ;

endif              : _ENDIF
                   ;

bracket            : ']'
                   ;

par                : ')'
                   ;


sem_col            : ';'
                   ;

keep_num           : _INUM
                   ;

case_proc          : _INUM _CASE
                   ;

paren              : '('
                   ;

%%
```

4.2.3. The action routines used in the YACC specification.

The action routines used are in most cases quite complex. Here a simplified very high level description of the action routines is given. For more details please see the appendix. The action routines are placed in the proper places within the YACC specification. The description given here is not complete and somewhat ambiguous. It is written that way to keep the description short and simple. The basic approach towards solving different situations is focused upon. Following is the description of the action routines used in the YACC specification.

```
%{
#include <stdio.h>
#include "symbol_table.h"
#include "defn.h"
#include "defn1.h"
#include "eqtrees.h"
#include "arry_grph.h"
#include "rangetab.h"
#include "st_manipulator.h"
#include "file_lst.h"
#include "subl_lst.h"
#include "routines.h"
#include "itoa.h"
#include "copy_name.h"
#include "file_dump.h"
#include "pars_rout.h"
extern int debug;
extern int b_debug;
struct Sym_tab_entry *temporary;
struct Case *whn_ptr;
%}

%token <inum> _PROCESS _FUNCTION _SEQUENTIAL _INPUT
%token <inum> _DIRECT _PORT _CASE _DOUBLE _INTEGER
%token <inum> _DISPLAY _ADDRESS _FILE _WHEN _GROUP
%token <inum> _CHAR _SHORT _LOGICAL _SUBLINEAR _REAL
%token <inum> _OTHERS _RANGE _THEN _ELSE _ENDIF
%token <inum> _OUTPUT _SUBSCRIPT _TRUE _FALSE _RELOP
%token <inum> _PROCID _LAST _PRESENT _OF _IF _SUB
%token <inum> _RNUM _STRING  _IMPLY _REV_IMPLY _IN
%token <inum> _NAME _INUM _ELSEIF _RECORD _OUT
%token <inum> _LONG _QUAD  _DIGIT
%token <inum> _COMMA_BEFORE_INUM
%token <inum> _COM_SUBL

%nonassoc <inum> '='
%left <inum> '+' '-' '|'
%left <inum> '*' '/' '&' _CONCAT
%right <inum> _POWER

%type <inum> func_proc opt_fattr fattr porttype type rec

%type <name> format fflag opt_minus opt_E

%type <st> nlist substructure simpstructure

%type <eqt>

%type <rd> opt_range range

%type <cs> opt_others when_exp

%type <sp> nlist2 name1 opt_nlist

%union
{
 short sh;
 int inum;
```

```
    char *name;
    struct Sym_tab_entry *st;
    struct Range_def_blk *rd;
    struct Eq_tree_node *eqt;
    struct Rec_case  *rc;
    struct Case *cs;
    struct St_ptr_list *sp;
    struct Parm_list *pl;
   }

  %%
specification      : func_proc ':'  _NAME opt_range sem_col
                    {
                     Create the module structure for the program module.
                     create a new entry for the symbol PROCESS (PROCESS
                     is the name of the interim file).
                     Set the global variables as
                        under_file = 0;
                        porttype = 2;
                    }
                   statement
                    {
                     If debug flag is set then call the following routines:
                           print_files();
                           print_subl();
                           print_def();
                           print_iol();
                           print_st();
                           print_all_eq_trees();
                    }
                  ;

func_proc           : _PROCESS
                  | _FUNCTION
                  ;

statement           : statement  header sem_col
                  | statement declaration sem_col
                  | statement equation sem_col
```

```
                | header sem_col
                | declaration sem_col
                | equation sem_col
                ;

header            : _INPUT ':' nlist
                | _OUTPUT ':' nlist
                ;

nlist           : nlist ',' _NAME
                   {
                    Create the file list (if it is not yert created) and
                    install the symbol (the file names) in the file list
                    if it is not yet installed. Check for possible
                    redefinition. Also create the symbol table entries
                    for these symbols and fill the type_of_use ,
                    io_flags.input and io_flags.output fields of the
                    symbol table entry.
                   }
                | _NAME
                   {
                    Do the same thing as done for the previous grammer rule.
                   }
                ;


declaration       : _FILE ':' _NAME
                   {
                    Create a symbol table entry for the file name if it
                    is not already created. Check for redeclaration. Set
                          under_file = 1;
                    to indicate that all the substructures now belongs to
                    a file structures and not to the interim file PROCESS.

                    Insert the symbol table entry of the file in the
                    Relation_q table.
                   }
                  opt_fattr  substructure_list
                   {
```

```
          Fill up different fields of the symbol table entry
          for the file. By now the table relation_q  are filled
          so call
             insert_relations();
          to insert the relationship of all the data items that
          belongs to the file among each other.

          If it is a DIRECT or a simple PORT file (i.e., without
          IN => OUT or OUT => IN) then make two identical copy
          of the same file. One of these is of type output.
          The name of all the data items (including the file name)
          of the output file are prefixed with "OUT_".

          If it is a PORT file with IN => OUT or OUT => IN then
          there are always exactly two record  structures defined
          under the file structure definition. One is for the
          input "RECORD INPUT" and the other is for "RECORD OUTPUT"
          In this case create two file structures, one for the
          input and the other for the output. The input file has
          only one record , the input record, with its record
          structure.

          The output file has also one record , the output record,
          with its record structure. Moreover, prefix the name of
          all entries that belongs to this output file with "OUT_".

          While doing all these check for possible semantic errors.
        }

      | _GROUP ':' _NAME opt_range
        {
          Create a symbol table entry for the group name and fill
          in as many fields of that entry as possible. Insert a
          pointer to the symbol table entry in the beginning
          of the relation_q array. The group entry along
          with all the data items that belongs to this group are
          made part of the interim file.
        }
        simpstructure_list
```

```
                        {
                          Call the routine
                              insert_relations();
                          to insert the sibling, child , parent relationship for
                          all the data items that belongs to the group. Call the
                          routine
                                actual_insert(pointer to the
                                        symbol table of the group);
                          to insert the I/O information and file_type information
                          of all the entries under the group.
                        }

                      | type ':' ndxlist
                        {
                          Have the inherited attribute pass down the parse tree
                          of the non-terminal "ndxlist".
                        }
                      | subscript
                      ;

ndxlist                 : _NAME
                        {
                          create symbol table entry for the name and make it a
                          childs of the interim file. Fill some of the fields
                          of this entry as per information available.
                        }
                        opt_range
                        {
                          Set the range and dimensionality field of the symbol
                          table entry.
                        }
                      | ndxlist ',' _NAME opt_range
                        {
                          Do similar thing as done for the previous grammer rule.
                        }
                      ;


opt_fattr               : '(' fattr  par
                          {
                            pass the synthesized attribute (the file type
                            information) up the parse tree.
                          }
                        | { same as for the previous rule}
                        ;


simpstructure_list: simpstructure_list _COMMA_BEFORE_INUM  simpstructure
                        | _COMMA_BEFORE_INUM simpstructure
                        ;

substructure_list : substructure_list  _COMMA_BEFORE_INUM substructure
                        | _COMMA_BEFORE_INUM substructure
                        ;

fattr                   : _SEQUENTIAL
                             { Pass the file type information up the parse tree }
                        | _DIRECT     { same }
                        | _DISPLAY    { same }
                        | _PORT       { same }
                        | _PORT       { same }
                            {BEGIN imply;}
                          porttype
                           {
                             BEGIN ordinary;
                             Pass the file type information up the parse tree.
                           }
                        ;

porttype                 : _INPUT _IMPLY _OUTPUT
                             { Pass the file type information up the parse tree }
                        | _OUTPUT _IMPLY _INPUT
                             { Pass the file type information up the parse tree }
                        ;

type                     : _CHAR
                             { Pass the file type information up the parse tree }
                        | _CHAR '(' _INUM par  { Same }
                | _SHORT            { Same }
                | _INTEGER           { Same }
                | _LONG             { Same }
                | _REAL             { Same }
                | _DOUBLE            { Same }
                | _QUAD             { Same }
                | _LOGICAL           { Same }
                ;

rec             : _RECORD ':'
                    { Pass the file type information up the parse tree }
                | _RECORD _INPUT ':'       { same }
                | _RECORD _OUTPUT ':'      { same }
                ;

opt_range         : '['
                    {
                      BEGIN imply;
                      dimensionality = 0;
                    }
                  range
                    {
                    BEGIN ordinary;
                    }
                 bracket
                |
                ;


opt_others        : opt_WHEN_OTHERS ':' simps _COM_SUBL _NAME
                 {
                   Create the Case structure and fill in all the fields of
                   that structure with appropriate values.

                   Adjust the top_subl_list.
                 }
                |
                 {
                  Return NULL to indicate that no case structure has been
```

```
                        created.
                      }
                    ;

opt_WHEN_OTHERS   : _WHEN  _OTHERS
                  | _OTHERS
                    ;

when_exp            : when_exp  _WHEN binexp ':' simps  _COM_SUBL _NAME
                      {
                        Create the Case structure and fill in all the fields of
                        that structure with appropriate values.

                        Adjust the top_subl_list.
                      }
                  | _WHEN binexp ':' simps  _COM_SUBL _NAME
                      {
                        Same as for the previous grammer rule.
                      }
                    ;

simps               : simps  _COMMA_BEFORE_INUM  simpstructure
                  | simpstructure
                    ;

substructure        : keep_num  rec   _NAME opt_range
                      {
                        Create a symbol table entry for the record entry and
                        insert inthe relation_q. Fill in the fields of the entry.

                        Set range and dimensionality field and set the dim_proc
                        flag to 1.
                      }
                  | case_proc '(' keep_name par  when_exp opt_others
                      {
                        Install a new symbol table entry for the CASE. Fill
                        its fields and adjust the relation_q table for correct
                        insertion of the relations.
                      }
                  | case_proc '(' keep_name par  opt_others
                     {
                       Similar as done for the previous grammer rule.
                     }
                  | simpstructure
                  ;



simpstructure      : _INUM type ':' substr_rest_rep
                  | keep_num _GROUP ':' keep_name opt_range
                     {
                      Insert the name in the symbol table and fill its
                      different fields. Adjust the relation_q table.
                     }
                  | keep_num keep_name  opt_range
                     {
                      Same as for the previous rule.
                     }
                  ;

substr_rest_part  :   _NAME opt_range
                     {
                      similar as the previous grammer rule.
                     }
                  ;

substr_rest        : substr_rest_part
                  | substr_rest_part format
                       {
                        BEGIN ordinary;
                       }
                  ;

substr_rest_rep    : substr_rest_rep ','  substr_rest
                  | substr_rest
                  ;

range              : range  ',' _INUM
```

```
                 {
                  Create range_def_blk and fill in the is_static, ceiling
                  and next fields of it. Count the dimensionality.
                 }
                |range  _COMMA_BEFORE_INUM _INUM
                 {
                  Similar as the previous grammer rule.
                 }
                |range ',' '*'                          { same }
                |range _COMMA_BEFORE_INUM '*'           { same }
                |range ',' _INUM _IMPLY '*'             { same }
                |range _COMMA_BEFORE_INUM _INUM _IMPLY '*' { same }
                |range ',' '*' _REV_IMPLY _INUM         { same }
                | _INUM                                 { same }
                | '*'                                   { same }
                | _INUM _IMPLY '*'                      { same }
                | '*' _REV_IMPLY _INUM                  { same }
                ;


format           : paren opt_minus _INUM fflag par
                 {
                  Retrieve the format string and store in a safe place.
                 }
                |paren opt_minus '*' fflag par
                 {
                  Retrieve the format string and store in a safe place.
                 }
                ;

opt_minus         : '-'
                |
                ;

fflag            : '.' _INUM '+' opt_E
                |'.' _INUM opt_E
                |'+' opt_E
                |opt_E
                ;
```

```
opt_E          : 'E'
               |
               ;

subs_part       : _SUBSCRIPT ':' keep_name _OF '(' prefnlist par
                {
                 Install the subscript entry in the symbol table.
                }
               ;

subs_part2      :  _SUBSCRIPT ':' _NAME
                {
                 Install the subscript entry in the symbol table.
                }
               ;

subscript       : subs_part subs_rest
               | subs_part
               | subs_part2 subs_rest
               | subs_part2
               | subs_part2 _SUBLINEAR '(' nlist2  par  binexp
                 {
                   Install the sublinear subscript in the symbol table
                   and add it in the sublinear list.
                 }
               ;

opt_nlist       : _OF '(' prefnlist  par
               |
               ;

prefnlist      :  prefnlist ',' prefname
                {
                 Make a list of all the names that can be referred to by
                 the subscript (it may include two copy for a single
                 name if the name has two entries in the symbol table for
                 input and output).
```

```
                  }
                | prefname
                  {
                    Similar as before.
                  }
                ;


nlist2          : nlist2 ',' name1
                  {
                    Make a list of the symbol table entries referred and
                    pass a pointer to the head of the list up the parse tree.
                  }
                | name1
                  {
                    Do similar thing.
                  }
                ;

name1             : _NAME  '.' _NAME
                  {
                  Return a pointer to the appropriate entry of the symbol
                  table. Install a new entry if the entry is not already
                  there.
                  }
                | _NAME
                   {
                     Do similar thing.
                   }
                ;

subs_rest         : subs_rest ',' _NAME opt_nlist
                   {
                     Create a symbol table entry for the subscript.
                   }
                | ',' _NAME  {$<name>$ = strsave(private_copy); } opt_nlist
                   {
                     Create a symbol table entry for the subscript.
                   }
                ;

equation        : variable '=' expression
                  {
                   Change the type_of_use field of the appropriate symbol
                   table entry.
                   Create an equation tree node with label ASSERTION and
                   fill in its other fields appropriately.
                   Check for multiple definition of the variable. If there
                   is a multiple definition then adjust the pointers of the
                   involved equations  as required.
                  }
                ;

variable        : range_rep varbase
                  {
                   Maintain a stack for keeping the range, last, address
                   and present information. Report stack error if found.
                   Create appropriate RANGE, LAST, PRESENT or ADDRESS
                   entry if needed and if they are not already found
                   in the symbol table. Have the other_pointer fields
                   of all these symbol table entries appropriately
                   linked.
                   Return a pointer to the appropriate symbol table entry
                   for use in the equation tree.
                  }

                | varbase
                  {
                    Pop the stack.
                  }
                ;


range_rep       : range_rep1 _ADDRESS '.'
                  {
                     Indicate that the ADDRESS is found.
                  }
                  | _ADDRESS '.'
                    { Same as before. }
                  | range_rep1
                  ;

range_rep1          : range_rep1 _LAST opt_inum '.'
                    {
                      Increment range_counter or last_counter as appropriate
                      by the value returned by the opt_inum.
                    }
                  | range_rep1 _RANGE opt_inum '.'
                    {
                      Increment the range_counter by the value returned by
                      opt_inum.
                    }
                  | _RANGE opt_inum '.'
                    {
                      Set range_counter to the value returned by opt_inum.
                    }
                  | _LAST opt_inum '.'
                    {
                      Set last_counter to the value returned by opt_inum.
                    }
                  ;

opt_inum            : '(' _INUM   par
                    {
                     Return the value of the _INUM.
                    }
                  | { Return 1. }
                  ;

varbase            : prefname
                     {
                      push(range_counter,last_counter,is_address);
                      range_counter = last_counter = is_address = 0;
                     }
                   index
```

```
                          {
                           Adjust the pointers of the equation tree nodes as
                           required.
                          }
                        | prefname
                          {
                           Similar action as for the previous rule.
                          }
                        ;

prefname                 : _NAME '.' _NAME
                          {
                           IN_OUT = 0;
                           Set st_ptr to the symbol table entry for the variable.
                           Call the routine pref1 () .
                          }
                        | _IN '.' _NAME '.' _NAME
                          {
                           IN_OUT = 1;
                           Rest same as before.
                          }
                        | _OUT '.' _NAME '.' _NAME
                          {
                           IN_OUT = 2;
                           Rest same as before.
                          }
                        | _NAME
                          {
                           IN_OUT = 0;
                           Rest same as before.
                          }
                        | _IN '.' _NAME
                          {
                           IN_OUT = 1;
                           Rest same as before.
                          }
                        | _OUT '.' _NAME
                          {
                           IN_OUT = 2;
```

```
                   Rest same as before.
                 }
               ;

index          : '[' aexp_list bracket
               ;

aexp_list      : aexp_list ',' aexp
                 {
                   Create appropriate equation tree nodes (as described in
                   the Global data structure section) and link them.
                 }
               | aexp_list _COMMA_BEFORE_INUM aexp
                 {
                   Similar as before.
                 }
               | aexp
                 {
                   Similar as before.
                 }
               ;

expression     : _IF binexp _THEN expression elseif_rep endif
               | _IF binexp _THEN expression endif
               | _IF binexp _THEN expression elseif_rep _ELSE expression endif
               | _IF binexp _THEN expression _ELSE expression endif
                 {
                   For each of the above rule create a equation tree node
                   and fill its different fields appropriately.
                 }
               | _CASE '(' variable par when_rep
                 {
                   Create an appropriate CASE_EXP equation tree node.
                 }
               | _CASE '(' variable par when_rep opt_WHEN_OTHERS ':' binexp
                 {
                   Create appropriate OTHER_ACTION and CASE_EXP
                   equation tree node and link them.
```

```
                      }
                    | _CASE '(' variable  par  opt_WHEN_OTHERS ':' binexp
                      {
                        Create an appropriate CASE_EXP and OTHER_ACTION
                        equation tree node and link them.
                      }
                    | binexp
                    ;

elseif_rep            : elseif_rep _ELSEIF binexp _THEN expression
                    | _ELSEIF binexp _THEN expression
                      {
                        For each of the above rule, create appropriate
                        IF_ACTION and IF_EXP nodes and link them properly.
                      }
                    ;

when_rep              : when_rep _WHEN binexp ':' binexp
                    | _WHEN binexp ':' binexp
                      {
                        For each of the above rule, create appropriate
                        WHEN_ACTION and WHEN_TEST nodes and link them properly.
                      }
                    ;

binexp                : binexp '|' bterm
                      {
                       Create an appropriate BINARY_OR equation tree node.
                      }
                    | bterm
                    ;

bterm                 : bterm '&' bfactor
                      {
                       Create an appropriate BINARY_AND equation tree node.
                      }
                    | bfactor
                    ;
```

```
bfactor         : strexp _RELOP {relop_val = yylval.inum; } strexp
                  {
                    Create an appropriate equation tree node with label
                BINARY_EQ, BINARY_GE, BINARY_LE, BINARY_NE, BINARY_GT,
                    or BINARY_LT depending on the value of the relop_val.
                  }
                | strexp
                ;

strexp          : strexp _CONCAT sterm
                  {
                   Create an appropriate BINARY_CONCAT equation tree node.
                  }
                | sterm
                ;

sterm           : aexp
                | _STRING
                  {
                   Create an appropriate LITERAL equation tree node.
                  }
                ;

aexp            : aexp sign term
                  {
                   Create an appropriate BINARY_PLUS or BINARY_MINUS
                   equation tree node depending on the "sign".
                  }
                | sign term                %prec _POWER
                  {
                   Create an appropriate UNARY_PLUS or UNARY_MINUS
                   equation tree node depending on the "sign".
                  }
                | term
                ;

sign            : '+'
                | '-'
                ;
```

```
term            : term '*' factor
                  {
                    Create an appropriate BINARY_TIMES equation tree node.
                  }
                | term '/' factor
                  {
                    Create an appropriate BINARY_DIVIDE equation tree node.
                  }
                | factor
                ;

factor          : factor _POWER prim
                  {
                    Create an appropriate BINARY_POWER equation tree node.
                  }
                | '~' prim
                  {
                    Create an appropriate UNARY_NOT equation tree node.
                  }
                | prim
                ;

prim            : _RNUM
                  {
                    Create an appropriate LITERAL equation tree node.
                    Set the type of the symbol table entry for the real
                    number to REAL.
                  }
                | _INUM
                  {
                    Create an appropriate LITERAL equation tree node.
                    Set the type of the symbol table entry for the integer
                    number to INTEGER.
                  }
                | '(' binexp  par
                | fcall
                | _PROCID
                  {
                    Create an appropriate LITERAL equation tree node.
                    }
                  | _PRESENT '.' varbase
                    {
                      Create an appropriate symbol table entry if required
                      for the symbol PRESENT whose other_pointers field
                      points to the symbol table entry for "varbase".
                  | variable
                  | _TRUE
                    {
                      Create a symbol table entry for the symbol TRUE if it
                      is not already present in the symbol table.
                      Create an appropriate LITERAL equation tree node.
                    }
                  | _FALSE
                    {
                      Create a symbol table entry for the symbol FALSE if it
                      is not already present in the symbol table.
                      Create an appropriate LITERAL equation tree node.
                    }
                  | _SUB
                    {
                      Create a symbol table entry for the unique symbol
                      created by the lexical analyzer using the itoa routine
                      for this purpose. The symbol is for a subscript.
                      Create an appropriate LITERAL equation tree node.
                    }
                  ;

fcall               : keep_name '(' opt_binexp_list  par
                  | keep_name '(' par
                    {
                      For both of the above rules :
                      create an appropriate FCALL equation tree node.
                    }
                  ;

opt_binexp_list     : binexp opt_binexp_rest
                    {
                      Create an appropriate PARMS equation tree node.
```

```
                   }
                 ;

opt_binexp_rest   : ',' binexp  opt_binexp_rest
                 | _COMMA_BEFORE_INUM  binexp  opt_binexp_rest
                   {
                    For bothe of the above rules :
                    create an appropriate PARMS equation tree node.
                   }
                 |
                 ;

endif             : _ENDIF
                 ;

bracket           : ']'
                 ;

par               : ')'
                 ;


sem_col           : ';'
                 ;

keep_num          : _INUM
                 ;

case_proc         : _INUM _CASE
                   {
                    Create a new symbol table entry for the symbol CASE.
                    Add this entry to the case record list. The head of
                    this list is pointed to by case_rec_list_head.
                    Add this entry appropriately to the relation_q.
                   }
                 ;


paren             : '('
                    ;
%%
```

4.2.4. The routines used in the Syntax Analyzer

The syntax analyzer uses several routines. In this section the purpose and the calling format of these routines are described. The algorithms are not given, but the actual source codes can be found in the appendix.

4.2.4.1. hash

Routine name: hash

File name : source/st_manipulator.h

Routine type: integer function.

Author: Balaram Sinharoy

Calling Format: hash (s)

Arguments:

char *s -- The Symbol name.

Calling routines: yyparse,lookup

Global variable/structures used:

HASHSIZE
Sym_tab_entry

Purpose:

This routine returns the bucket (out of 101 buckets) in which the symbol "s" will be placed.

Example:

The call hash ("our") will return 39 indicating that the entry for the symbol "our" is to made in the bucket 39.

Algorithm:

Input: A pointer to a character string for denoting the name of a symbol.

Output: An integer in between 0 and 100 which denoted the bucket in which symbol belongs.

```
{

  Add all the characters (the integer values of their representation) in the string.
  Divide the sum by the constant HASHSIZE (which in our case is 100) and return
the remainder.

}
```

4.2.4.2. strsave

Routine name: strsave

File name : source/st_manipulator.h

Routine type: char function.

Author: Balaram Sinharoy

Calling Format: strsave (s)

Arguments:

char *s -- The Symbol name.

Calling routines: yyparse,do_install,install

Global variable/structures used:

None.

Purpose:

This routine will allocate a new location for the symbol name pointed to by s and return a pointer to this location to the calling routine.

4.2.4.3. lookup

Routine name: lookup

File name : source/st_manipulator.h

Routine type: struct Sym_tab_entry

Author: Balaram Sinharoy

Calling Format: lookup (s)

Arguments:

char *s -- The Symbol name.

Calling routines: yyparse, install

Global variable/structures used:

Prl_struct
Sym_tab_entry

Purpose:

The purpose of this routine is to lookup for the entry of the symbol "s" in the symbol table. It first calls the routine hash to find out in which bucket the symbol belongs to. Then it searches along the bucket to find out the first entry of that symbol in that bucket.

Example:

lookup ("our") will return a pointer to a symbol table entry in the bucket 39.

Algorithm:

Input : A pointer to a character string "s" which denotes a symbol.

Output: A pointer to a symbol table entry if the entry is already there
in the symbol table else it returns NULL.

```
{
```

I = hash(s)

Search through the link list of the symbols in the I-th bucket of the symbol table to find out the first entry in the bucket that has the same symbol name.

If such an entry is found in the symbol table then return a pointer to that entry of the symbol table else return NULL

}

4.2.4.4. do_conditional_install

Routine name: do_conditional_install

File name : source/st_manipulator.h

Routine type: struct Sym_tab_entry

Author: Balaram Sinharoy

Calling Format: do_conditional_install (st_ptr,symbol,np,nd,type)

Arguments:

char *symbol -- The Symbol name.
st_ptr,np -- Sym_tab_entry
int type
nd -- Prl_struct

Calling routines: install

Global variable/structures used:

Prl_struct
Sym_tab_entry

Purpose:

This routine goes through the appropriate bucket and tries to find an entry in the symbol table that has the same name (a character string) and whose other_pointers points to the entry pointed to by the argument st_ptr. This routine is called from the routine install. When the I flag in install is set to 4,5,6 or 7, entry for RANGE, ADDRESS,PRESENT or LAST is searched accordingly.

~~~~~~

4.2.4.5. do_install

Routine name: do_install

File name : source/st_manipulator.h

Routine type: struct Sym_tab_entry

Author: Balaram Sinharoy

Calling Format: do_install(symbol_name)

Arguments:

char *symbol_name -- The Symbol name.

Calling routines: yyparse,install
~~~~~~

Global variable/structures used:

brn_ptr
Prl_struct
Sym_tab_entry

Purpose:

The purpose of this routine is to actually install the symbol name symbol_name. When called it creates a new entry in the symbol table allocating space for it, for the symbol name pointed to by the pointer symbol_name.

4.2.4.6. *install*

Routine name: install

File name : source/st_manipulator.h

Routine type: struct Sym_tab_entry

Author: Balaram Sinharoy

Calling Format: install (symbol_name , I)

Arguments:

char *symbol_name -- The Symbol name.

int I -- A switch to determine which type of entry we are looking for.

Calling routines: yyparse

Global variable/structures used:

Prl_struct
Sym_tab_entry
already_installed, install_always.
last_st_ptr,address_st_ptr,present_st_ptr,range_st_ptr

Purpose:

The primary purpose of this routine is to lookup for the entry of the symbol "symbol_name" in the symbol table. If the entry is not found then a new symbol table entry in the appropriate bucket will be made. The searching for the symbol_name is directed by the flag I.

Depending on the value of I the search is done:

I = 0: The first entry in the bucket that has the same name is returned, if there is no entry found then it creates another entry and returns a pointer tio that entry.

I = 1: An entry which belongs to the file that has a name same as the name pointed to by filename is found out, if such an entry is not found then another entry is created .

I = 2: Find an entry with the name symbol_name and which is a FILE.

I = 3: Find an entry symbol_name which is a SUBSCRIPT .

I = 4: Find out the entry in the symbol table which is a RANGE and whose other_pointers pointer points to the appropriate symbol table entry.

I = 5: Find out the entry in the symbol table entry which is a ADDRESS and whose other_pointers pointer points to the apprpriate symbol table entry.

I = 6: Find out the entry in the symbol table entry which is a PRESENT

and whose other_pointers pointer points to the apprpriate
symbol table entry.

I = 7: Find out the entry in the symbol table entry which is a LAST
and whose other_pointers pointer points to the apprpriate
symbol table entry.

I = 8: Find the symbol table entry which is a function name.

I = 9: If the use of a variable occurs before its declaration then
that variable is already in the symbol table and so we have
to refer to that place. Looking through the bucket find out
the entry that has the same name and which does not have any
parent, which means that it is not attached to any
structure

Example:

lookup ("our" ,0) will return a pointer to the first symbol table entry in the bucket 39. If it is found that there is no entry in that bucket then it will make a new entry for "our" in the symbol table.

Algorithm:

Input: A pointer "symbol_name" to a character string denoting a symbol name
and an integer "I" which dictates how the search and installation of
the symbol name will be done. A global variable "install_always" is used
which is "1" to indicate that the symbol name is to be installed no
matter what is the value of "I".

Output: A pointer to the symbol table entry , which may be created by this
algorithm or which may have been already present in the symbol table.

A global variable "already_installed" is set to "1" if the particular variable which is represented by the symbol (it is to be noted here that many variable may be represented by the same symbol) is already found in the symbol table.

```
{

np = lookup (symbol_name);
already_installed = 1;
if (install_always == 1)
  install the symbol in the symbol table by calling the function "do_install"
else
  do the following depending on the value of "I" :

I = 0:

  if ("np" points to an entry of a string)
    search the rest of the current bucket for another entry
    of the same symbol which is not a string.
  if no such entry is found
    call do_install(symbol_name);

I = 1:
  If (np == NULL)
    call do_install(symbol_name);
  else
    while ("np" does not point to the symbol name "symbol_name" | "np"
          does not belong to the file "filename");
        /* filename is a global variable pointing to a character string
          for a filename */
    {
      np = np->next;
      if (np == NULL)
          break;
    }.
  if (np == NULL)
    np = do_install(symbol_name);

I = 2:
  while ("np" does not point to the symbol name "symbol_name" |
          "np" is not an entry for a FILE)
   {
      np = np->next;
      if (np == NULL)
          break;
   }
  if (np == NULL)
    np = do_install(symbol_name);

I = 3:
  while ("np" does not point to the symbol name "symbol_name" |
          "np" is not an entry for a SUBSCRIPT)
   {
      np = np->next;
      if (np == NULL)
          break;
   }
  if (np == NULL)
    np = do_install(symbol_name);

I = 4:
  np = do_conditional_install(range_st_ptr,symbol_name,np,nd,RANGE);

I = 5:
  np=do_conditional_install(address_st_ptr,symbol_name,np,nd,ADDRESS);

I = 6:
  np=do_conditional_install(present_st_ptr,symbol_name,np,nd,PRESENT);

I = 7:
  np = do_conditional_install(last_st_ptr,symbol_name,np,nd,LAST);

I = 8:
  while ("np" does not point to the symbol name "symbol_name" |
          "np" is not an entry for a FUNCTION)
   {
      np = np->next;
      if (np == NULL)
```

```
            break;
    }
   if (np == NULL)
    {
     np = do_install(symbol_name);
     np->type_of_use = FUNCTION;
    }

 I = 9:
   while("np" does not point to the symbol name "symbol_name" |
         "np" has a parent | "np" is an entry for a string)
    {
       np = np->next;
       if (np == NULL) break;
    }
```

4.2.4.7. itoa

Routine name: itoa

File name : source/itoa.h

Routine type: a pointer to a character string.

Author: Balaram Sinharoy

Calling Format: itoa(inum)

Arguments:

inum -- integer

Calling routines: lex.yy.c

Global variable/structures used:

None.

Purpose:

This routine returns a pointer to a character string that contains the character string for the integer inum. This is used for appending the integer (in character) with the SUB.

Example:

itoa(23) will return a pointer to the character string "23".

4.2.4.8. insert_relations

Routine name: insert_relations

File name : source/routines.h

Routine type: void

Author: Balaram Sinharoy

Calling Format: insert_relations()

Arguments:

None.

Calling routines: yyparse

Global variable/structures used:

filetype
relation_q[]
io_inf
io_inf2
Prl_struct
Sym_tab_entry

Purpose:

Before calling the routine insert_relations the table relation_q has been filled by the yyparse. The first entry in the relation_q is the filename. Each entry in the relation_q table has two parts. The first part contains a pointer to the symbol table entry and the second part contains the integer that has been found in front of the symbol name in the definition of the file in which the name has been found. To seperate the WHENs in the CASES number 0 is used. To indicate the end of definition of case the integer -2 has been used.

Given with this table this routine inserts all the relations (i.e. it fills the entries for the parent sibling and old_child filds for all the variables that has been found in the definition of that file).

Example:

For the file declaration :

```
file: commands,
 10 rec: inrec[*],
   20 char(2): code,
   20 case(code)
      when single: 30 int: arg1, sublin j1
      when 'MV'  : 30 int: arg21,
                   30 int: arg22, sublin j2;
```

The content of the relation_q is:

```
commands 0
inrec 10
code 20
CASE 20
arg1 30
 0
arg21 30
arg22 30
 0
 -2
```

The corresponding entries in the symbol table are:

```
This is an entry in the 42 bucket :

    Entry is made for the symbol:   commands
    Parent: NULL
    Oldest child : inrec
    next sibling : NULL
    type of use:  File
    file type  :  Sequential
    type       :  yet unknown or the entry is not for a field
    I/O flags  : 010 00000
    Dimensionality  : 0
    Other_pointers is NULL
    Range :  NULL

This is an entry in the 24 bucket :

    Entry is made for the symbol:   inrec
    Parent : commands
    Oldest child : code
    next sibling : NULL
    type of use:  Record
    file type  :  Sequential
    type        : yet unknown or the entry is not for a field
    I/O flags  : 010 00010
    Dimensionality  : 1
    Other_pointers is NULL

    Range : Following are the ranges defined --
            is_static: 0,  ceiling:  0,  type


This is an entry in the 7 bucket :

    Entry is made for the symbol:   code
    Parent : inrec
    oldest child  : NULL
    Next sibling : CASE
    type of use:  Field
    file type  :  Sequential
    type        :  char(2)
    I/O flags  : 010 00000
    Dimensionality  : 0
    Other_pointers is NULL
    Range :  NULL


This is an entry in the 82 bucket :

    Entry is made for the symbol:   CASE
    Parent: NULL
    oldest child  : NULL
    next sibling : NULL
    type of use:  Undef 0
    file type  :  Sequential
    type        :  yet unknown or the entry is not for a field
    I/O flags  : 010 01000
    Dimensionality  : 0
    Other_pointers points to the rec_case entry for code
    Range :  NULL
```

```
This is an entry in the 60 bucket :

    Entry is made for the symbol:   arg1
    Parent : CASE
    oldest child  : NULL
    next sibling : NULL
    type of use:  Field
    file type  :  Sequential
    type       :  integer
    I/O flags  : 010 00000
    Dimensionality  : 0
    Other_pointers is NULL
    Range :  NULL


This is an entry in the 9 bucket :

    Entry is made for the symbol:   arg21
    Parent: NULL
    oldest child  : NULL
    Next sibling : arg22
    type of use:  Field
    file type  :  Sequential
    type       :  integer
    I/O flags  : 010 00000
    Dimensionality  : 0
    Other_pointers is NULL
    Range :  NULL


This is an entry in the 10 bucket :

    Entry is made for the symbol:   arg22
    Parent: NULL
    oldest child  : NULL
    next sibling : NULL
    type of use:  Field
    file type  :  Sequential
```

type : integer
I/O flags : 010 00000
Dimensionality : 0
Other_pointers is NULL
Range : NULL

4.2.4.9. insert_parent

Routine name: insert_parent

File name : source/copy_name.h

Routine type: void

Author: Balaram Sinharoy

Calling Format: insert_parent (oldest_child)

Arguments:

oldest_child -- pointer to symbol table entry.

Calling routines: copy_file

Global variable/structures used:

None.

Purpose:

This routine will insert the parent of all the entries which are sibling of other entries and the oldest of them have their parent defined. This also works recursively.

4.2.4.10. insert_io

Routine name: insert_io

File name : source/routines.h

Routine type: void

Author: Balaram Sinharoy

Calling Format: insert_io()

Arguments:

None

Calling routines: yyparse

Global variable/structures used:

io_inf -- contains the information for io_flags.is_input
io_inf2 -- contains the inforrmation for io_flags.is_output

Purpose:

This routine inserts all the I/O information for all the nodes in the symbol table. It starts from the file list, and assuming that the file name has correct I/O information it goes through all the entries that belongs to this tree and inserts the correct I/O information.

4.2.4.11. io_entry

Routine name: io_entry

File name : source/routines.h

Routine type: integer function.

Author: Balaram Sinharoy

Calling Format: io_entry (st_ptr)

Arguments:

st_ptr -- a pointer to the symbol table.

Calling routines: yyparse

Global variable/structures used:

None.

Purpose:

This routine is used to determine the I/O information that the entry st_ptr is going to get. The call to this routine is very rare. In those occassions when we need to know the I/O information before we have inserted it by calling io_insert explicitly we call this routine. This routine finds out the I/O information by finding the file in which it belongs, if the I/O information is not already found in the entry for that symbol.

4.2.4.12. add_dimn

Routine name: add_dimn(st_ptr)

File name : source/routines.h

Routine type: void

Author: Balaram Sinharoy

Calling Format: add_dimn(st_ptr)

Arguments:

st_ptr -- pointer to symbol table entry.

Calling routines: correct_dimn

Global variable/structures used:

None

Purpose:

This routine does the actual addition. This routine also calls itself recursively to do the addition successively for all the entries that belongs to the subtree rooted at st_ptr.

4.2.4.13. correct_dimn

Routine name: correct_dimn

File name : source/routines.h

Routine type: integer function.

Author: Balaram Sinharoy

Calling Format: correct_dimn

Arguments:

None.

Calling routines: yyparse

Global variable/structures used:

files_ptr,top_ptr -- pointers to Files

Purpose:

The routine correct_dimn does the dimensionality correction at the end of parsing. The dimensionality that has been inserted in the place of the corresponding field is what it was defined in the specification of the program. The correct dimensionalty of a variable is the sum of the dimensionality of its and all of its predecessor and this is true for all the descendent of a file. In this routine starting from the first direct descendent of file we add the dimensionality information of the parent to all of its children.

Example:

For the declaration:

```
file : a (direct),
  10 rec : our [*],
    20 int : c[*,7,*], d[8,9];
```

the dimensionalities that will be found after the parsing are:

```
our -- 1;
c   -- 4;
d   -- 3;
```

4.2.4.14. pop

Routine name: pop

File name : source/routines.h

Routine type: void function.

Author: Balaram Sinharoy

Calling Format: pop()

Arguments:

None.

Calling routines: yyparse

Global variable/structures used:

stack_counter, stack[]
range_counter,last_counter,is_address

Purpose:

This routine pops the value of the range_counter ,is_address, last_counter from the stack. This popped values are the current values for the variable that is being processed currently and the corresponding range , address and last information are stored in the symbol table by creating appropriate last, range and address entries in the symbol table.

4.2.4.15. push

Routine name: push

File name : source/routines.h

Routine type: void function.

Author: Balaram Sinharoy

Calling Format: push(range,last,addr)

Arguments:

range,last,addr -- integer

Calling routines: yyparse

Global variable/structures used:

stack_counter, stack[]

Purpose:

This routine pushes the range,addr and last infomation that associates with the "prefname" in the stack and adjusts the stack counter for later retrieval.

4.2.4.16. get_node

Routine name: get_node

File name : source/routines.h

Routine type: void

Author: Balaram Sinharoy

Calling Format: get_node (label,left,right,parent,type,left_type)

Arguments:

label,type,left_type -- integer
left -- can be pointer to Sym_tab_entry or Eq_tab_entry
parent,right -- pointer to Eq_tab_entry

Calling routines: yyparse

Global variable/structures used:

None

Purpose:

This routine creates a new entry in the in the equation tree whose label is "label" left child is "left", right child is "right", parent is "parent",and which is of type "type". left_type is used as a flag to indicate that the left child is a symbol table pointer (when left_type is 1) or a pointer to equation tree (when left_type is 0).

4.2.4.17. copy_entry

Routine name: copy_entry

File name : source/copy_name.h

Routine type: void

Author: Balaram Sinharoy

Calling Format: copy_entry (ptr)

Arguments:

ptr -- pointer to symbol table entry.

Calling routines: yyparse,copy_file

Global variable/structures used:

None.

Purpose:

This routine copies all the entries that belong to the subtree pointed to by ptr into a seperate place. While copying it will find the appropriate loose variable (i.e., not attached to any file structure) with compatible I/O information whenever possible.If ptr is NULL then it does nothing. It also calls recursively to copy all the variables that is descendent of ptr.

4.2.4.18. copy_file

Routine name: copy_file

File name : source/copy_name.h

Routine type: a pointer to a character string.

Author: Balaram Sinharoy

Calling Format: copy_file (parent)

Arguments:

parent -- pointer to Sym_tab_entry

Calling routines: yyparse,create_duplicate_file

Global variable/structures used:

None.

Purpose:

This routine will make a duplicate copy of the file pointed to by parent. While copying it will make a scan through the symbol table to find whether any more of the loose variables can also be attached to the same file. For that purpose it goes through the bucket in which it is supposed to belong and tries to find the loose variable (that is, a variable which is the same as the variable whose copy we are making , only that it has different I/O information.

4.2.4.19. dump

Routine name: dump

File name : source/copy_name.h

Routine type: void

Author: Balaram Sinharoy

Calling Format: dump(ptr1,ptr2)

Arguments:

ptr1,ptr2 -- pointer to symbol table entry.

Calling routines: yyparse,copy_file

Global variable/structures used:

None.

Purpose:

This routine dumps all the fields of the entry pointed to by ptr1 into the fields of the entry pointed to by the pointer ptr2.

4.2.4.20. change_out_name

Routine name: change_out_name

File name : source/routines.h

Routine type: void

Author: Balaram Sinharoy

Calling Format: change_out_name ()

Arguments:

None.

Calling routines: yyparse

Global variable/structures used:

top_files is referenced to know the list of files.

Purpose:

This function will change the name of all those PORT files which are output (in this case there are two copies of the same file in the symbol_table. One copy is for the input and the other c copy is for the output. Before this routine is called (this routine is called only when the parser has completed its task) the name of this input and the output file is the same, after this routine is called the name of all the entries in the output file (only for the above kind of otuput files are treated here) will be prefixed by OUT_.

4.2.4.21. rec_change_name

Routine name: rec_change_name

File name : source/routines.h

Routine type: void function.

Author: Balaram Sinharoy

Calling Format: rec_change_name (symbol)

Arguments:

symbol -- a pointer to the symbol table.

Calling routines: change_out_name

Global variable/structures used:

None.

Purpose:

This routine recursively goes through all the symbol table entries that belongs to the subtree rooted at "symbol" and changes their symbol name by prefixing the existing name with "OUT_". This routine calls itself recursively.

4.2.4.22. print_files

Routine name: print_files

File name : source/file_dump.h

Routine type: void

Author: Balaram Sinharoy

Calling Format: print_files ()

Arguments:

None.

Calling routines: yyparse

Global variable/structures used:

None.

Purpose:

This routine will dump the files that has been declared in the program. Since at the time of insertion a new file name was inserted on top of another old name that are there already in the list (and so it is a stack) so the print out will be in the reverse order.

4.2.4.23. print_rel

Routine name: print_rel

File name : source/routines.h

Routine type: void

Author: Balaram Sinharoy

Calling Format: print_rel (s)

Arguments:

char *s -- The Symbol name.

Calling routines: yyparse

Global variable/structures used:

None

Purpose:

This routine prints the entries of the relation_q that corresponds for the file pointed to by s.

4.2.4.24. print_subl

Routine name: print_subl

File name : source/file_dump.h

Routine type: void function

Author: Balaram Sinharoy

Calling Format: print_subl ()

Arguments:

None.

Calling routines: yyparse

Global variable/structures used:

None.

Purpose:

The routine print_subl prints the sublinear subscript list .The head of the sublinear subscript list is pointed to by the pointer top_subl_list. The printing of the sublinear subscript will be in the opposite order in which they are found in the specification. This routine also prints the associated equation tree pointed to by the other_pointers field of the sublinear subscript entry in the symbol table.

----

5. Dimensionality propagation

5.1. The objective

The purpose of the dimensionality propagation is to assign correct dimensionality to all the variables that are used in the assertions. The basic algorithm is as follows. We know that all the variables that are fields of some input or output file structure has their dimensionality correctly defined in the definition of that file. We start our dimension propagation from these fields. We go through the array graph for this purpose. We begin with the list of variables which belongs to some input or output file. Looking at the array graph we know which equations has used this variable (in the right hand side or on the left hand side). We find out the dimensionality that has been used for that variable in that equation. We find out the difference in dimensionality in the use of that variable in that equation and the definition (or in later stage, the established dimensionality in earlier iteration) of that variable. We call this difference DIMDIF. Now we know that all the variables that has been used in that equation has the actual dimensionality equal to the number of index used for that variable in this particular equation plus DIMDIF. Thus we establish the actual dimensionality of each variable in this equation and then we keep variables for which the dimensionality has been established into a queue. In the next iteration of the same process we can start with the new queue of variables and propagate the dimensionality to other variables that are used in other equationother equations. When we have established the dimensionality of some variable, we keep track of the assertion which has caused the dimensionality of that variable to be established. We also set a flag to indicate that the dimensionality has been established. In later iteration if we find that we are attempting to establish the dimensionality of some variable whose dimensionality has already been found, we check whether these two values of the dimensionality agrees. If they do not agree we know that the user has made some mistake in dropping the dimensionality of a variable in some equation. We can not be absolutely sure exactly in which equation the user has made the mistake. So in case any error is found we first report to the user the error. To help the user to find out exactly where he has made the mistake we list also some equations. These are all the equations that has been used in establishing the dimensionality of this variable. When listing, we go back upto the equation which is the common root (if any found) in the tree of assertions. The tree nodes are the assertions and there is an edge from each assertion whose dimensionality has been defined by the use of a variable, to the assertion which has caused the dimensionality of that variable to be established. If there is no common root in this tree, then we end up with two input or output variables which caused us to establish different dimensionalities of the same variable.

The algorithm is divided into a few routines. The algorithm is described below.

5.2. The algorithm for dimensionality propagation.

The main routine of the EPL system calls dim_prop for dimensionality propagation. No argument need to be passed to the routine. The algorithm for dimensionality propagation can be described as follow.

```
Algorithm dim_prop ()

{
 /* Initialization */

 Q <- all array grap nodes that denote a data element
      which belongs to an input or output file.

 level = 0;

 /* Iteration  */

 prop ();  /* this propagates the dimensionality information
              for the level 0 and sets the QP queue to the
              set of data elements for whom the dimensionality
              has been established  */

 while (QP != NULL)  /* QP contains all the data elements for which
                        dimensionality has just been established but
                        dimensionality has not been propagated yet
                        using them.  */
   {
    Q <- QP;
    prop ();          /* Propagate dimensionality using this nes data
                         elements   */
   }

}
```

```
Algorithm prop ()

{

  QP = NULL;

  For each data element "S" in the queue Q do

    {

     for each assertion arry_node "N" in the owner_list and dep_list
       of "S" do

       {

         If the assertion has not been processed yet (a bit is set
          always when an assertion is processed) do

           {

             node = dim_search (N->ptr.ptp,S->ptr.str);

            /* Search the equation tree for this assertion
               to find out the first occurence of this
               variable in that equation tree (so that
               the symbol table pointer for that entry in the
```

```
                equation tree matches with the symbol table
                entry for the data element "S".)  */

              count = count_index (node);  /* Count the number of index */

              DIMDIF = established number of index for node "S" -
                    count;

              dim_insert (N,DIMDIF,N);

              set the dim_defn field of the array graph node N to node S;

            }

         }

      }

   }
-----------------------------------------------------------------------------



Algorithm dim_search(ASS,NODE)

                  /* This algorithm searches for the first occurence
                     of the equation tree node NODE in the equation tree
                     pointed to by ASS  */

if (ASS == NODE)

   return (ASS);

else if (ASS == NULL)

   return (NULL),
else

  {

    newnode = search (ass->left_child,NODE);

    if (newnode == NULL) then

        search(ass->right_child,NODE);

    return (newnode);

  }

}
```

```
Algorithm  dim_insert (N, DIMDIF,ASS)

{
 if (N == NULL)
   return ();

 else
  {
  if ( N is a variable node ) then
    {
     count = count_index (N);
     dimn = count + DIMDIF;
     if (dim_proc field of the  symbol table entry for N is 0) then
       {
        set the dimensionality field to dimn;
        set dim_proc to 1;
        set dim_defn field of the array graph entry for N to ASS;
```

```
            insert the array graph node for N in QP;
           }
          else
           {
             if (dimn != the dimensionality field of the symbol table
                 entry for N)
             dim_error (ASS, dim_defn field of array graph entry for N);
           }
          }

      dim_insert (left child of N in equation_tree, DIMDIF, ASS);
      dim_insert (right child of N in equation_tree, DIMDIF, ASS);
    }
```

```
Algorithm count_index (node);

{

  count = 0;

  new_node = node->left_child;

  while (new_node != NULL)

   {

     count ++;

     new_node = new_node->left_child;

   }

}
```

```
Algorithm dim_error (this_ass, prev_ass)
 {
   dim_list(this_ass , table1, count1);
   dim_list (prev_ass, table2, count2);
   dim_cmp (table1, table2, count1, count2);
   print error messages along with the assertion numbers contained in the
   adjusted contents of table1 and table2;
 }
```

```
Algorithm dim_cmp (table1,table2,count1,count2)
{
   Find the first assertion number in table1 that is also in
   table2;
   set count1 to include elements in table1 upto this
```

```
    assertion number;

    set count2 to include elements in table2 upto this

    assertion number;

}
```

```
Algorithm dim_list ( ass,table,count)
{
  count = 0;

  defn_ptr = ass;

  while (defn_ptr != NULL) do

    {

      if (defn_ptr points to an assertion node in the array graph)

          {

              insert defn_ptr in the table;

              defn_ptr = defn_ptr->dim_defn;

              count++;

          }

    }
```

}

---

6. References

[1] Hwang, Kai and Briggs, Faye, *Computer Architecture and Parallel Processing* McGraw-Hill Book Company, New York, 1984.

[2] Parallel Algorithms and Architectures : International Workshop, Suhl, GDR, May 25-30, 1987 proceedings.

[3] Aho A., Sethi R., Ullman J. *Compilers: Principles, Techniques, and Tools* Addison Wesley, New York, 1986.

[4] J. Bruno. Array Graph descriptions

[5] D. Clarke. Master's project in the Dept. of Comp. Sc. in RPI.

[6] Kang-sen Lu. Model Program Generator: System and Programming Documentation. Technical Report, Dept. of Comp. Sc., University of Pennsylvania.

[7] Axel T. Schriener, H. George Friedman, Jr. *Introduction to Compiler Construction using UNIX*. Englewood Cliffs, N.J. : prentice Hall.