COMMON SUBEXPRESSION DETECTION IN DATAFLOW PROGRAMS

by

Philip E.C. Jones, BSc.(Hons), ADEL

A thesis submitted for the degree of
Master of Science
in the Department of Computer Science
at the University of Adelaide

November 1989

CONTENTS

LIST OF FIGURES

LIST OF TABLES

SUMMARY

On a data driven dataflow machine, instructions are executed by many asynchronous processors, the sequencing constraints of execution of these instructions being determined solely by the data dependencies between them. In this mode of execution, the parallelism is fine grained and can be exploited at the level of machine instructions. However, execution requires considerable overhead peculiar to the dataflow environment; a high proportion of the executed instructions are involved in duplicating and switching data tokens, and on a tagged token machine, in the maintenance of token labels or tags. Execution time of a dataflow program depends upon the number and cost of the instructions and the degree to which parallelism can be exploited with a given number of processors; the efficiency will also depend upon the token traffic in the system. The aims of this project were firstly to develop a compiler for a significant subset of the programming language VAL (a dataflow language developed at M.I.T.) to generate object code for the University of Manchester dataflow machine. The second aim was to investigate the effects of applying some conventional compiler optimization techniques to the compilation of dataflow programs.

A compiler has been developed which generates an intermediate code structure designed to allow the generation of efficient dataflow target code. As indicated above, the target language used for the experiments in this project is that for the University of Manchester dataflow machine. Programs were executed on a locally developed simulator.

The main optimization investigated in depth deals with the detection and removal of common subexpressions. The applicative nature of the language VAL means that within the one scope of nomenclature,

different occurrences of the same expression always yield the same value. The compiler is designed to enable the maximum sharing of common subexpressions with full advantage being taken of the dataflow environment. This has enabled common subexpressions to be shared even when one or both are conditionally executed. The approach taken here is to detect all common subexpressions and then determine whether their relative positioning within program structures, which may be conditionally executed, allow their values to be shared. Shareable common subexpressions are defined in terms of formally defined selection rules. Run time statistics from executing a variety of programs on the simulator provide a basis for assessing the effects of common subexpression elimination; in summary, the results indicate a marked improvement in program execution performance.

DECLARATION

This thesis contains no material which has been accepted for the award of any other degree or diploma in any University and, to the best of my knowledge and belief, contains no material previously published or written by any other person, except where due reference is made in the text of the thesis.

If this thesis is accepted for the award of the degree, permission is granted for it to be photo-copied.

P.E.C. Jones

November 8th 1989

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Dr. Barbara Kidman, for all the help and encouragement she has provided, and for the many thought provoking discussions which have lead to many of the results presented in this thesis. I would also like to thank Dr. Andrew Wendelborn for his help and for taking over the supervision of this thesis while Dr. Kidman was on study leave.

I am also grateful to Mr. Roberto Morello for making his simulator available for the project.

This research was supported in part by a University of Adelaide Postgraduate Research Award.

1. INTRODUCTION

1.1 Background

Research on multiprocessor systems is aimed at improving machine performance by allowing parallel processing of independent processes. Parallel processing systems based on the so-called dataflow principle [Dennis, 1974], have multiple processors executing small processes asynchronously. Sequencing of these processes (and their operations) is determined by the data dependencies between operations *not* by the order in which they appear in the program source code, so that the responsibility for control flow management is removed from the programmer. The order in which operations are performed is determined by the availability of data values passed between processes at run time as operands to their operations. Thus in dataflow, there is no concept of assigning a value to a memory cell.

If more than one process requires the same data value, that value is duplicated, so that each process has its own copy. In a pure dataflow computation the execution of basic operations is *data driven*; this automatically allows concurrency at all levels from that of operations upwards, but at the expense of duplicating and switching operations. It is also possible to consider *demand driven* dataflow systems [Treleaven, Brownbridge & Hopkins, 1982; Wendelborn, 1985], but attention here is focused entirely on the more conventional data driven dataflow principle.

1.1.1 Dataflow graphs

A dataflow program written in two-dimensional graphical form [Dennis, 1974] (for example Figure 1.1), consists of a set of nodes representing instructions, interconnected by directed arcs representing the transfer of

data between operations. It is conventional to represent a sequence of instructions by nodes one under the other, and instructions available for parallel execution written alongside one another [Gurd, Kirkham & Watson, 1985]. An instruction is ready for execution when its inputs or operands are available; it will execute as soon as a processor becomes available. At execution time, data values are passed between nodes along the arcs from output points of one node to input points of another. The data values are packaged into tokens each of which consists of a typed data value and a destination address. Static dataflow systems do not allow concurrent reactivation of code and hence are not able to handle recursion. Dynamic systems allow recursive reactivation by some means.

In the tagged token dataflow model [Arvind & Gostelow, 1977; Gurd, Watson & Glauert, 1980], a label is also associated with each token. The label is used to distinguish operands when several tokens are on the same arc. Operands to an instruction must have matching labels before they can be used in the execution of the corresponding instruction. Instructions are executed asynchronously, and during iteration, recursion and array processing, the same instruction may be executed many times simultaneously with different label values identifying the particular iteration, recursion or array index; each execution will require its own set of input tokens. The label provides the necessary information required to perform the matching of data for the one operation. Note that all required operands must be present to enable an instruction to be executed.

There is obviously an overhead associated with tagging each token, with updating the tags, and with the matching function required to distinguish between different sets of tokens during iteration, recursion and array processing. However, if tags are not used, it is important to

ensure that tokens cannot overtake one another on an arc; thus a strict ordering must be enforced on all arcs so that tokens arriving at a node's input points are correctly matched. This can be achieved by using a first-come-first-served queue discipline on the set of tokens travelling on every arc in the dataflow graph. The obvious restriction here is that some degree of parallel execution may be lost by enforcing this sequential processing of tokens compared with what happens in the tagged token model [Gurd, Watson & Glauert, 1980]. Recent work [Abramson & Egan, 1988] has been carried out at the Royal Melbourne Institute of Technology on a hybrid dataflow machine which combines the dynamic and static models.

To initiate the execution of a dataflow graph, priming tokens are required to trigger the firing of the topmost initial nodes in the graph. For example, the dataflow graph representing a function module in the programming language VAL[1] [Ackerman & Dennis, 1979] may be initiated by supplying priming tokens corresponding to each of the function's arguments and directing these tokens to each node in the graph that requires a copy of the data value corresponding to the argument. Once triggered, the nodes in the graph continue to fire without further interference until all input tokens are used and an output token (or tokens) is produced.

1.1.2 Dataflow languages

Dataflow computers have been designed with a view to exploiting the parallelism inherent in most programs. The use of variables in conventional languages intended for Von Neumann machines to represent stored data, and assignment statements to model the storage of

[1] A function module in VAL will be defined in Chapter 3.

data, makes such languages fundamentally foreign to the dataflow mode of computation in which there is no notion of data storage – although it has been shown that a conventional language can be implemented on a dataflow machine [Allan & Oldehoeft, 1980]. Alternatively, as some dataflow models do not allow recursion, a purely functional language is also inappropriate because it is desirable to have loops in the language to express repetition.

It is characteristic of dataflow languages that identifiers represent values rather than storage locations, and it turns out that so-called single assignment languages are particularly appropriate; in a single assignment language each identifier appears on the left hand side of an assignment-like statement only once and the statement serves to bind a name to a data value. In addition, freedom from side effects and locality of effect are necessary to ensure that data dependencies determine the flow of execution. Each expression is applicative and function parameters are passed by value. Thus, in this sense, dataflow languages are functional, with each function having access only to its arguments and locally defined identifiers.

There have been a number of research studies involved in designing suitable dataflow languages [Ackerman 1982], some of the first attempts including ID [Arvind, Gostelow & Plouffe, 1978], Lapse [Glauert, 1978; Gurd, Glauert & Kirkham, 1981] and VAL [Ackerman & Dennis, 1979]. An important and more recently designed language, SISAL [McGraw et al, 1985], is based on the earlier language VAL. VAL was selected for use in our project in 1984, because it exhibited the properties described above, information about the language was at hand, and a preliminary experiment had been carried out to implement a subset of VAL on the University of Manchester dataflow machine [Jones, 1984]. A description

and analysis of this language is given by McGraw [1982]. Its choice over other languages is not significant for the purposes of this thesis. It should be noted that it was not within the scope of this project to improve upon the design of the original language VAL, but rather to implement a sufficient subset of VAL to allow effective analysis of the optimizations considered in Chapter 6.

1.2 Overview of general aims

Research into dataflow computing has continued for the last two decades with several dataflow machines being built, including the University of Manchester machine [Gurd, Kirkham & Watson, 1985] which is discussed in this thesis. The dataflow idea is designed to take advantage of the ultimate inherent parallelism in programs by executing operations as soon as the operands are available. Inevitably any speedup (relative to sequential one-operation-at-a-time execution) due to the fine grained concurrency is at the expense of many additional duplicating, switching and tagging instructions. In fact unoptimized dataflow code contains a very high proportion of such unproductive operations. Scepticism [Gajski, Padua, Kuck & Kuhn, 1982] about the practicality of the dataflow model will remain until convincing performance is obtained from real dataflow machines.

Note that this project has been carried out using a *fine-grained* dataflow model in which dataflow nodes are defined in terms of machine-level executable instructions. Alternatively, *coarse-grained* dataflow models can be constructed which define data dependencies only between segments of code, such as, for example, a function module; coarse-grained dataflow will not be considered further in this thesis.

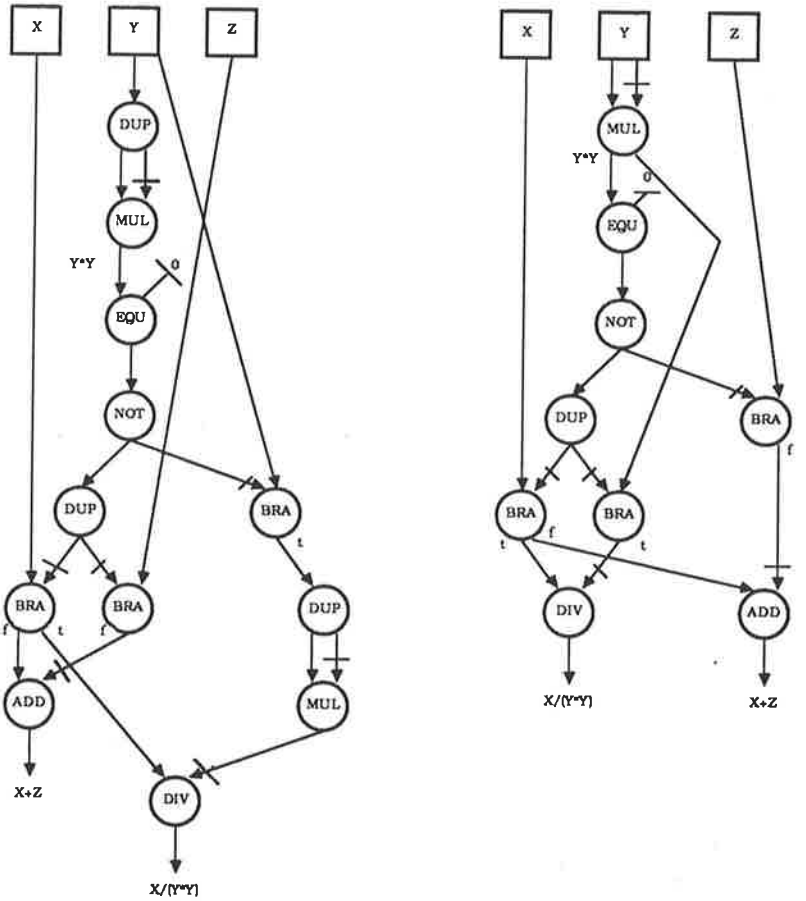
One way of improving the performance of dataflow programs is to

optimize the target code produced by the compiler; at the time this project was commenced in 1984, in fact very little work had been published on dataflow code optimization. A preliminary study explored several mechanisms for code optimization in expressions [Jones, Kidman & Morello, 1985]. The aim of this thesis is to examine comprehensively the use of common subexpression (CSE) detection in the compilation of a dataflow language.

Intuitively, the effect of utilizing CSE detection in the dataflow environment is not altogether clear. As all instances of the CSE are evaluated when their common operands become available, their separate evaluation can take place in parallel (aside from the complication of duplication operations). So at first sight, a single evaluation will not reduce execution time, at least when no switching is involved. However, lesser computational effort is involved in that the total number of operations executed and the total number of tokens generated will be reduced. In addition, the preliminary studies [Jones & Kidman, 1986] indicated that in many cases execution time was also reduced, incidentally.

In the work undertaken for this thesis, care has been taken to ensure that the implementation scheme is "clean", that is no unnecessary operations are performed and all unused tokens are killed. The significance of these requirements is illustrated in the examples of dataflow graphs corresponding to the following expression:

if y*y <> 0 **then** x/(y*y) **else** x+z **endif**

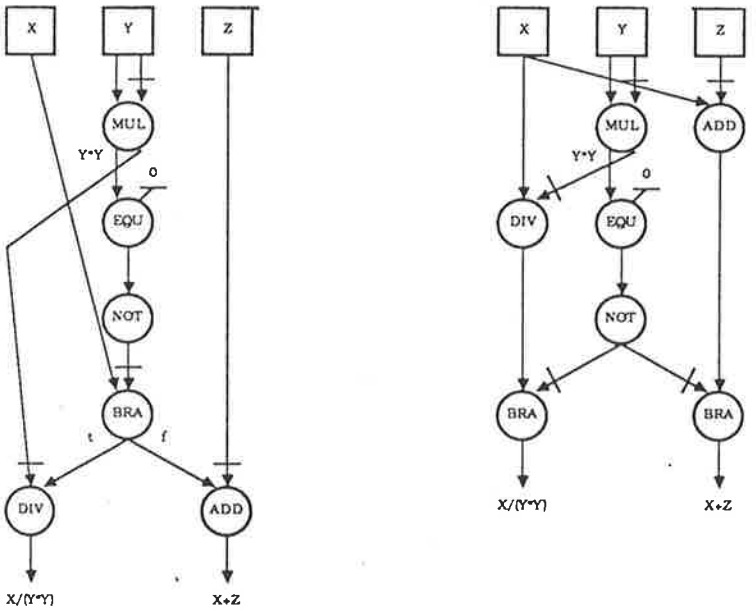


(a) Without CSE detection **(b) Evaluating CSE y*y once**

Figure 1.1 Correct, clean dataflow graphs.

Figure 1.1(a) shows a correct implementation, without optimization, in which the productive operations in the subexpression of the two arms of the conditional are performed only when necessary to produce the result; in addition, execution of the graph produces only one result token, all intermediate data tokens being absorbed. Three switching or branch nodes (BRA nodes are defined in Chapter 2) are required, one for each identifier in the arms of the conditional. Figure 1.1(b) is also a clean and correct implementation in which the CSE, $y*y$, is computed once only, the second use within the conditional arm being obtained by switching a copy of the subexpression which is computed unconditionally as part of the

boolean condition. The same number of branch nodes are required in this case, but the graph executes in less steps.



(a) Tokens retained as garbage **(b) Unnecessary operations**

Figure 1.2 Unacceptable dataflow graphs

Figure 1.2(a) shows an alternate correct implementation of the above graph in which the tokens $y*y$ and z are passed directly to their respective operations without switching. Nevertheless, the operations in the arms are executed only under correct conditions because their other input is properly switched[2]. The defect in this scheme is that one or other of the tokens $y*y$ or z will remain in the system as garbage after the completion of execution of the conditional expression. Figure 1.2(b) shows yet another attempted implementation in which the two arms of the conditional are first computed unconditionally and then switched. Thus the operations in one arm or the other are performed unnecessarily. While execution time will not increase in a highly parallel machine, non-terminating computations or errors (as here) may arise. Hence such unnecessary computations, in general, are unacceptable. Note however, that VAL

[2]The semantics of node execution will be explained in Chapter 2.

allows the program to compute a value in a **let** construct which is subsequently only used conditionally. Therefore, the unused token must be killed if that condition is not met.

At the start of the project there was no compiler available to experiment with optimizations, so the design and implementation of a compiler became the first objective. As explained above, the language VAL was considered appropriate as the source language for the compiler. However, the design and implementation of the compiler, apart from the parser, and the structure of the intermediate form was intended to be flexible enough to allow the use of other languages suitable for dataflow machines.

The choice of the initial target language was that for the University of Manchester dataflow machine, because a simulator for this machine had already been developed at the University of Adelaide [Morello, 1982]; it was intended that the target language would not determine or significantly influence the design and implementation of the compiler and intermediate form; it was hoped that the implementation scheme would be sufficiently general to generate other dataflow target code.

A dataflow target instruction (for the University of Manchester dataflow machine) contains the address of the instructions which use the result of the operation; hence a target program constitutes a network of interconnected nodes with links from operands to users, but with no links from operations to operands; this structure does not allow for direct comparison between operation-operand groups, as required for optimizations such as common subexpression detection. Thus the design and generation of an intermediate code form directly linking operations to their operands is crucial in the implementation. This intermediate form

is called the *triple graph* and is described in detail in Chapter 4.

In considering the potential efficiency of execution of a dataflow program with a given number of processors, both execution time and token traffic are significant. The term *token traffic* is used in a general way to refer to the density of circulation of data tokens in the system. To assess the impact of the optimization techniques described in Chapter 6, various performance measurements were obtained from the simulated execution of several real-life dataflow programs.

1.3 Structure of the thesis

The following chapters describe and explain the methods used and results obtained in attempting to fulfil the above aims. Chapter 2 describes how the fundamental concepts of dataflow are implemented on the University of Manchester dataflow machine. Chapter 3 describes the dataflow programming language VAL and defines a subset and variant of the language which is used in this project. Chapter 4 describes the intermediate form generated by the compiler, Chapter 5 explains how dataflow code is produced from this intermediate form and Chapter 6 deals with optimizations which are performed. A brief description of the test programs is presented in Chapter 7, and in Chapter 8, the results from their execution are analysed and discussed.

2. SIMULATION OF THE MANCHESTER DATAFLOW MACHINE

2.1 The Manchester Dataflow Machine

An operational dataflow machine [Kirkham, 1984; Gurd, Kirkham & Watson, 1985] has been running at the University of Manchester since 1981; it is based upon the dynamic tagged token dataflow model defined in Gurd, Watson & Glauert [1980]. The model is a refinement of earlier work [Dennis, 1974] on dataflow models of computation and subsequent work [Arvind & Gostelow, 1977] on tagged token systems.

As explained in Chapter 1, a dataflow graph consists of a set of nodes interconnected by directed arcs. The nodes (or instructions) in the Manchester model represent machine-level instructions with a maximum of two input and two output points to which the arcs are connected. At execution time, each token travelling along one of these arcs contains a label which is made up of three fields or tags: an iteration level, an activation name and an index number. The iteration level is used to distinguish values from different loop iterations, the activation name is unique for any function activation and the index number identifies array elements.

A block diagram of the prototype University of Manchester dataflow machine (MDFM) is given in Figure 2.1 [Gurd, Kirkham & Watson, 1985]. This simplified diagram shows the basic ring structure which is at the heart of the design of the Manchester machine architecture. It consists of four basic modules connected to a host system via an Input/Output switch. The modules operate independently with token packets circulating around the ring in a pipelined fashion. Tokens used by an instruction with more than one input are paired together in the matching store. These pairs, together with tokens used by one-operand instructions,

are passed to the instruction store which contains the machine executable program code. The input token or tokens and the appropriate instruction constitute an executable packet which is then passed to the processing unit where a processor is allocated to execute the instruction. The resulting computation creates an output token (or tokens) which is circulated back to the token queue to enable subsequent executions of instructions. The token queue is a simple first-in-first-out buffer used to smooth out the uneven rate of generation and consumption of token packets.

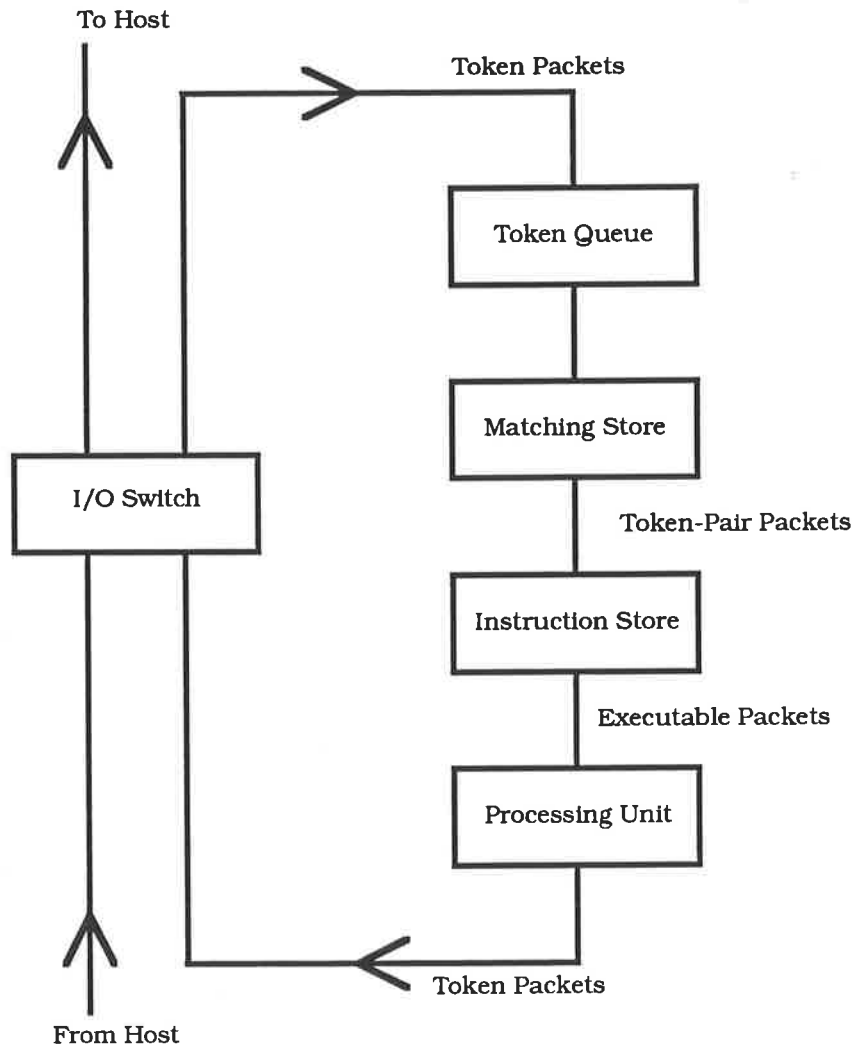


Figure 2.1 Block diagram of prototype MDFM

2.2 Subsequent development of the MDFM

The original design of the University of Manchester machine was extended with an overflow for the matching store [Gurd, Kirkham & Watson, 1985] and more recently with the inclusion of a structure store between the processing unit and the I/O switch [Sargeant, 1981; Bohm & Sargeant, 1985]; this is used to store large data structures, including arrays. Consequently much of the processing information associated with token packets can be compacted and unnecessary duplication of data structures is eliminated. Another enhancement which has been suggested, is the integration of multiple ring structures via a network, allowing results to be passed between different rings [Gurd, Kirkham & Watson, 1985]. These modifications and others, though important enhancements to the basic hardware, are not considered further as the simulation described in the next section deals with the simple model illustrated in Figure 2.1.

2.3 The Morello Simulation of the MDFM

The simulation [Morello, 1982; Kidman & Morello, 1984] of an idealised machine [Gurd, Watson & Glauert, 1980] modelled upon the MDFM was written in Pascal and run on a VAX 11/780 computer. This so-called perfect machine model ignores transmission costs and the time associated with the execution of different instructions, hence program execution proceeds in a sequence of equal execution steps. This kind of simulation has proved useful [Gurd & Watson, 1983] for analysing the efficiency and correctness of dataflow programs and was used to generate the test results for this thesis. However, it should be noted that the simulator used to produce the results shown in Appendix D was not designed to model the exact execution behaviour of the MDFM nor check

its performance. Due to the differences between the simulated execution on such an idealised machine and execution on the actual dataflow machine, the results should be interpreted broadly as giving a general indication of program performance on a real machine.

Within the MDFM simulator, the function of each unit, namely the matching store, instruction store, processing unit and I/O switch, is performed by a separate Pascal procedure. The different types of token packets passed around the system and stored in these units are represented using Pascal record structures. The token queue, matching store and instruction store are implemented as arrays of the appropriate record structures used to represent token packets and instructions. The token queue is implemented using a first-in-first-out queue discipline and the array index values used in the instruction store correspond to the addresses of the instructions being represented. The passing of token packets around the ring structure is modelled by the passing of parameters, between the procedures which emulate the functions of the units within the system [Morello, 1982].

Input to the simulator consists of priming tokens representing input data and a set of instructions representing the dataflow graph corresponding to the original high-level program (in this case a program written in VAL–S). These are contained in a flat file, each line in the file starting with a 'P' to represent a priming token or 'I' for an instruction. This file is read during the initialization phase of the simulation, the priming tokens being placed into the token queue and the instruction set into the instruction store. The execution phase of the simulation is then performed.

During the execution phase, a token is removed from the token queue and

passed to the matching store procedure. If the destination address of the token is an instruction with two inputs, an attempt is made to find the second input. If it is not found, the token must wait in the matching store for the second input to arrive. If the second token is found or if only one input was required by the instruction, the token (or tokens) is passed to the instruction store procedure. The corresponding instruction is selected and passed, with its inputs, to the processing procedure for execution. This procedure selects a processing unit to process the instruction, or waits for one to become available, and passes on the executable instruction for processing. The instruction is executed and the result token (or tokens) is normally passed to the token queue for further processing. However, if the address of a result token is negative, the token is sent to the I/O switch for removal from further computation and displayed as output from the program.

The simulator has been set up so that programs can be run with different numbers of processors. Output statistics produced include the number of priming tokens and instructions initially read into the token queue and instruction store, the latter being equal to the total number of nodes in the corresponding dataflow graph. Run time statistics include the number of execution steps needed to execute the program, this being dependent upon the number of processors used, and the maximum, total and average numbers of tokens passing through the token queue and matching store. Certain traces may also be turned on to produce further statistics including some that are useful when debugging programs.

2.3.1 The instruction set

The instruction set implemented in the Morello simulator is given in Appendix A. Most of the instructions correspond precisely to those

defined in the Manchester model [Kirkham, 1984] although some additions and changes were made for efficiency and simplicity. The instructions which execute arithmetic, relational and boolean operations require no explanation. The rest of this chapter describes some of the special dataflow instructions referred to in describing code generation in Chapter 5.

Selected from among the various switching instructions defined in the Manchester model [Kirkham, 1984], the BRA (for branch) instruction was implemented in the simulator (shown in Figure 2.2). This instruction accepts a data value as its left operand and a boolean value as its right operand. If the boolean value is true, the data value is passed to the right output destination and if false, to the left output destination. There is no situation during execution in which a data value may be passed to both the left and right outputs of a branch instruction. If one output destination is not specified and the boolean input value indicates that the data value is to be passed to that output destination, then the data value is killed or absorbed and no output is produced by the instruction. This node is shown in Figure 2.2 both for the case when a result is produced from one or other output ports, and when a data value is absorbed by the false output port.

The GEN (for generate) instruction is used to create a literal value required in a program. The literal value is built into the instruction, but a left input token from some other node within the graph is required to trigger the execution of the node. The node appears most often when identifiers are initialized to a literal value. Again the structure of the node is shown in Figure 2.2.

The DUP (for duplicate) instruction, again shown in Figure 2.2, performs

the function of copying or duplicating its input data token to provide two output tokens identical to that of the input token. This is required when the value of an identifier or an expression is used several times in a program.

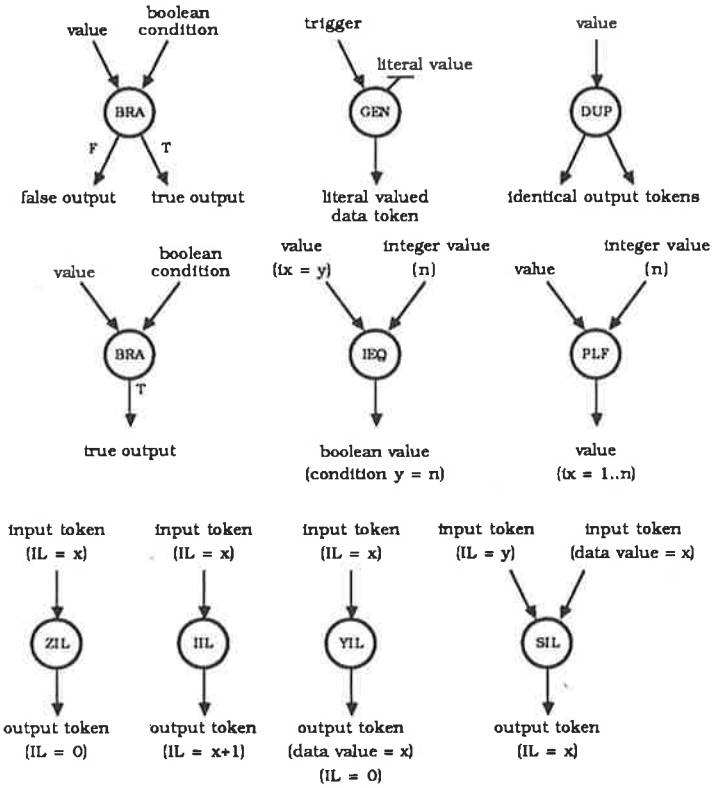


Figure 2.2 Some MDFM instructions represented by dataflow nodes

A further set of instructions, also illustrated by nodes in Figure 2.2, is used in processing the iteration tag field of the label of data tokens. The Zero Iteration Level (ZIL) node simply sets the iteration tag of its input token label to zero, the Increase Iteration Level (IIL) node increments this iteration tag by one, and the Yield Iteration Label (YIL) node extracts the iteration tag from its input token label, the output token having this tag value as its data value; the Set Iteration Level (SIL) node sets the iteration tag field of its left input token to the data value of its right input token. A similar set of instructions is defined for the maintenance of the index tag, used for array processing. In addition there is an Index EQual (IEQ) instruction for comparing the index tag of a data token with an

integer value or a literal; it returns a boolean value as its result. Another instruction designed mainly for array processing is the ProLiFerate (PLF) instruction which, at run time, creates as many copies of its left operand as specified by its integral right operand. The copies are distinguished by their index tag, the integral tag values being created sequentially starting from one. These last two instructions are also shown in Figure 2.2.

The instructions available for processing the activation name are very similar, except for the Generate Activation Name (GAN) node which, when triggered, creates a new, unique activation name. The Yield Activation Name (YAN) node is similar to the YIL node described above, extracting the activation name of the input token label and setting the data value of its output token to this activation name. The Set Activation Name (SAN) node is similar to the SIL node as it sets the activation name of its left input token label to the data value defined by its right input token. This set of instructions is primarily associated with function invocations but is also used in iterative expressions, as described in Chapter 5.

As mentioned earlier, the MDFM design has subsequently been enhanced by the inclusion of a structure store which alleviates some of the overheads incurred in array processing. This has not been incorporated into the Morello simulator, but as a compromise, two new instructions have been introduced (similar to instructions in Kirkham [1984]) to allow more efficient array processing. The EXT (for extract) instruction accepts an array as its left operand and an integer value, proliferated over the same range as the array, as its right operand. The function of this instruction is to extract the array element whose index value equals the integer value specified by its right operand. This array element is sent as

a single value to the right output destination of the instruction, the rest of the array being sent to the left output destination. This operation is equivalent to the boxed part of the subgraph shown in Figure 2.3, composed from more primitive operations. The instruction is used to produce a new array with element *i* modified without explicit duplication of the input array.

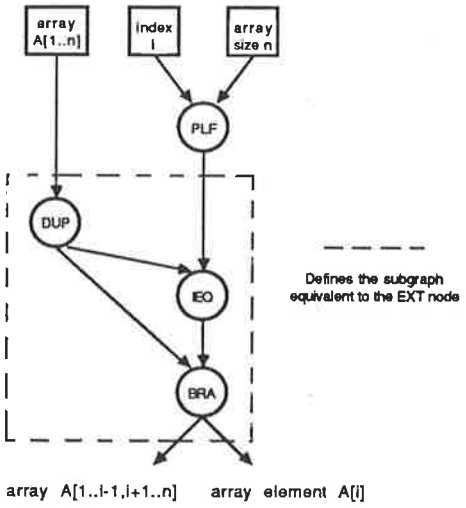


Figure 2.3 Primitive subgraph illustrating the EXT instruction

A similar second new operation, the SEL (for select) instruction, is used to select array elements from existing arrays. The instruction accepts the same input as the EXT instruction but sends the whole array, including the element whose index value equals the integral right operand value, to its left output destination. The right output is used to obtain the selected array element, but the advantage here is that the whole array remains available for further operations from the left output, so no array elements are discarded. If several selections from an array are required, only one copy of the array is needed as it can be passed from one selection to the next. This has the effect of sequencing array selections but reduces the number of tokens produced by a large factor. The subgraph of primitive instructions representing the SEL instruction is similar to the subgraph

for the EXT instruction shown in Figure 2.3, except that the array element produced by the right output requires duplicating to provide a copy for the left output.

The final instruction to be introduced in this chapter is also associated with array processing. This instruction, the SEQ (for sequence) instruction, is used to output arrays in array index order; it requires only one operand, that being an array. The index tags of the input token labels are used to order the tokens into an ascending sequence enabling the recognition of array elements returned by a dataflow program, which would otherwise be generated and output in an undetermined order.

3. THE DATAFLOW PROGRAMMING LANGUAGE VAL–S

3.1 Description of VAL–S

VAL (Value-oriented Algorithmic Language) was designed at the Massachusetts Institute of Technology [Ackerman & Dennis, 1979; McGraw, 1982] specifically for implementing computations on data driven machines. The compiler described in Chapter 4 compiles programs written in a subset of VAL. This subset, which is termed VAL–S, is described informally below; the syntax is given in full in Appendix B. The major features of VAL excluded from the subset relate to restrictions on the data types, the only data values permitted being integer, real and boolean values and one-dimensional arrays. VAL–S includes most constructs of VAL, the main exceptions being those specifically associated with *record*, *union* and *error* data types. Recursive function calls have been allowed in VAL–S, whereas VAL must achieve repetition using only iterative constructs.

The range of constructs in VAL–S is similar to those of older conventional languages although it should be noted that each VAL–S expression returns a tuple of values, the number of which determines the arity of the expression.

Input to and output from a VAL–S program is achieved by defining a global function module with one or more arguments (parameters) which trigger the execution of the program. Input values to the program are passed using these parameters of the global module, and output from the program is returned as the result of the module. Thus to execute the program, the global module must be called with appropriate values for its parameters. As a result of the computations within the program, many data values may be returned as output. The order in which this output is

returned is unpredictable, so to ensure correct interpretation when more than one scalar result is produced, the implementation assigns an index to each output value which is the result of the global module. These elements are then ordered as they are created and returned in index order. There are some obvious limitations to this mechanism in the current implementation.

Programs are written using the ASCII character set to make up the program elements, which consist of operation and punctuation symbols, reserved words and identifiers. The separation characters, space, tabulate and formfeed, are used to separate program elements but cannot be used within elements themselves. There are distinctions between upper and lower case characters in reserved words and identifiers, so consistent capitalization should be used to avoid confusion. A comment may be placed anywhere within a program, starting with a percent symbol and terminating at the end of the line.

An identifier consists of a sequence of letters, digits and underscores, the first of which must be a letter; also it must not appear in the list of reserved words. The length of a name is arbitrary in VAL but for implementation purposes up to twelve characters only are significant in VAL–S. An identifier may refer to a function definition or call, a loop variable, an index value, or a constant expression defined in **let** and **forall** expressions. A value name is an identifier defined as a loop variable, a constant in a **let** expression, or a formal parameter in a function header. Each value name denotes a single computed data value of a specifed type, the scope of which includes the entire construct in which it is defined less any inner constructs that reintroduce the same name. Note that value names are local to the enclosing function.

3.1.1 Function definitions

A VAL–S program consists of a collection of separate function modules, each module corresponding to a pure mathematical function. The general form of a function definition is:

function <function header>
    <internal function definitions>
    <function body>
endfun

The scope of each module includes any immediate outer module within which it is defined, any other modules declared below it at the same level, and, as recursion is allowed, the module itself. To allow for mutual recursion, a forward function declaration has been included in VAL–S, which extends the scope of a module to include all modules at the same level after the forward declaration.

Each function module computes one or more data values as a function of one or more arguments. A module only has access to its argument values as there are no global variables; it retains no state information from one invocation to the next, making each function call strictly independent and the values returned dependent only on the argument values at the time of the call. A function call has the conventional form of a function name followed by one or more arguments in a list enclosed between parentheses. Each of the actual parameters or arguments may be any expression including other function calls and each must conform to the data type and arity of the corresponding formal parameter declared in the called function definition's header. The number of actual parameters must equal that of the formal parameters. As VAL–S is a dataflow language, actual parameter *values* are transmitted to the function body.

The function body is an expression which may be a conditional, let, iterative or arithmetic expression.

3.1.2 Expressions

Expressions in VAL–S consist of conventional operand-operator expressions. Operands are values represented by literals, identifiers or function calls. Operators associated with the basic arithmetic and boolean operations are shown below in Table 3.1. Operations defined in VAL have the property that if the types and arities of all the operands to a particular operation are known then the type and arity of the result yielded by that operation can be determined. The functions and operations which operate on array values are described in Section 3.1.7.

Operators	Precedence Level
+ – (unary operators)	8
* / (multiplying operators)	7
+ – (adding operators)	6
|| (concatenation operator)	5
= < > <= >= ~= (relational operators)	4
~ (boolean NOT)	3
& (boolean AND)	2
| (boolean OR)	1

Table 3.1 Operators in VAL–S.

3.1.3 Conditional expressions

The conditional expression in VAL selects one of several branches to be evaluated depending upon the boolean values of the conditional test expressions. The general format is:

if <boolean expression> **then** <expression>
{ **elseif** <boolean expression> **then** <expression> }
else <expression>
endif

The **elseif** branch is optional and may be repeated as many times as required. The boolean test expressions are evaluated in order until one evaluates to *true*, the corresponding arm then being executed, otherwise the **else** arm is executed. The data types and arities of all the arms must conform. Version B of Example 1 in Appendix C (see also Table 4.3), contains a simple conditional expression. It consists of one boolean test expression which returns the value of *i* if it is *true*, otherwise the function is recursively called and the result returned is the value of this recursive call multiplied by *i*.

3.1.4 Let expressions

The **let** expression in VAL–S, superficially like assignment, is used to associate an identifier with one particular expression value. However, within one scope, an identifier can be given a value only once. It can then be referenced in the body of the **let** expression. The general form of the **let** expression is:

let <decldef part>
in <expression>
endlet

Any number of identifiers may be introduced within the declaration part of a single **let** expression so long as each one is declared and defined, or given a value, exactly once. These declarations and definitions may be combined in several ways, the only restriction being that a declaration

must precede a definition. For example:

```
A : integer;                          % declaration
A := 5;                               % definition
B : integer := 10;                    % declaration and definition
C : integer, D : real := B+1, 2.5     % multiple declaration
                                      %  and definition
```

Once defined, the identifiers may be used in further definitions as in the last line above. An identifier is undefined until its definition is complete. Hence an identifier may *not* be used in its own definition. The only apparent exception is described in the next section.

3.1.5 The for-iter expression and iterative expressions

Sequential iteration in VAL is performed using the **for-iter** expression. This construct, like the **let** statement, includes a declaration part which introduces special identifiers called loop variables and assigns them a data type and an initial value. The general format of the **for-iter** expression is:

```
for <decldef part>
do  <iteration body>
endfor
```

The iteration body controls the iterative cycle, each cycle possibly being dependent upon the results of the previous cycle, and may contain function calls, conditional, **let**, iteration or arithmetic expressions as well as further **for-iter** expressions. As shown in the example below, an iteration expression enclosed between two keywords serves to redefine only loop variables.

iter <decldef part> **enditer** (for loop variables only)

Iteration only continues while an iteration expression is executed during each loop cycle. If an iteration expression is not executed, the iteration is terminated returning the value of a non-iterative expression. Thus to achieve effective iteration, the iteration body must contain a conditional expression which has at least one arm containing an iteration expression and one arm that does not. Otherwise, the whole expression will produce only one iteration (equivalent to a **let** expression) or infinite iterations and will never terminate.

The single assignment rule is still in effect here as a loop variable may only be redefined in one statement inside an iteration expression. At the end of this statement the old value is updated. For the purpose of that statement, the occurrences of the loop variable on the right hand side can be thought of as the old value of the variable and the occurrence on the left as the new value to be used in any subsequent iterations. Note that each loop variable is already initially defined inside the loop body, so it may be referenced on the right hand side of its own redefinition and those of other loop variables. Also, as the new value is assigned to the loop variable at the end of the individual definition in which it is redefined, the order of redefinitions is significant. Any loop variables not redefined in an iteration expression retain their current values on subsequent cycles.

```
for  i, j : integer := n, 1;        % initial values of i and j
   if i<=1 then
     j                              % final result of iteration
   else
     iter j := j*i;                 % new values of i and j
          i:= i–1
     enditer
   endif
endfor
```

Thus a **for-iter** expression in VAL–S consists of declarations and initializations of loop variables, a test to determine whether the loop should be terminated, an expression giving the value to be returned upon termination, and redefinitions of loop variables while iteration continues. The above example demonstrates the usage of the **for-iter** expression in calculating the factorial of *n*.

The loop variables *i* and *j* are introduced in this example and initialized to *n* and *1* respectively. The loop body is executed with *i* being compared with *1*. If *n* is greater than *1* the **else** part of the conditional expression is executed with *i* and *j* being redefined. The loop body is then executed from the beginning on the second iteration cycle and the test *i<=1* is executed again. This continues until the value tested becomes *true* and the iteration terminates, returning the current value of *j*. Notice that the order of redefinitions is important here and that if the order were reversed, *j* would be multiplied by the new value of *i* rather than the old value and the factorial of *n–1* would be returned.

3.1.6 Forall expressions

The **forall** expression is used to create sets of values and either returns them as arrays or combines them using an arithmetic operation.

Essentially it is designed to allow parallel processing of array elements. To create arrays, the **construct** form of the **forall** expression is used, whereas to combine the elements of each set, the **eval** form is used. The general format of the **forall** expression follows:

forall <index name> **in** [<expression>]
[<decldef part>]
construct <expression>
| **eval** <expression>
endfor

As only one-dimensional arrays are included in VAL–S, **forall** expressions can introduce only one index name before the word **in**. Following this word must appear an expression of arity two which returns integral values to specify the range over which the index name takes its values. In addition, but optionally, temporary identifiers may be introduced in the declaration part, which conforms to the syntax rules as specified for the identifiers defined in **let** expressions. The body of the **forall** expression is defined and may contain either a **construct** or an **eval** statement. The arity and type of the entire **forall** expression depends upon the arity and type of the expressions defined in its **contruct** or **eval** statement. The following examples illustrate further details of both formats:

```
forall i in [1, n]                forall i in [1, n]
    construct n + i                   eval times i
endfor                            endfor
```

In these examples *i* assumes each of the integral numbers from *1* to *n* and the **construct** and **eval** constructs are executed for these values of *i*. Thus the values returned are firstly, from the **construct** statement, an

array with index values from *1* to *n* and corresponding data values from *n+1* up to *n+n*, and in the second example, the factorial of *n* is calculated in the **eval** statement, because all values that *i* assumes are multiplied together to return a single result.

3.1.7 Array–valued expressions

Arrays may be created in several other ways besides **forall** expressions. Arrays, like other values in VAL–S, may only be defined and given a value once. This implies that when one array element is changed an entirely new array value is created. Although VAL allows the definition of sparse arrays, where some elements in the array are undefined, for simplicity, in implementing arrays in VAL–S, it is assumed that all array elements are defined in order for index values from a specified low bound to the low bound plus the size of the array.

An array definition may create an empty array using the *array_empty* expression which defines the type of the array without specifying any array elements. The *array_fill* expression is also useful in defining and initializing an array by specifying the bounds of the array index (low and high for one-dimensional arrays) and the same data value for each element of the array within those bounds. The final method descibed below creates the array by elements, specifying the lower bound of the array followed by an expression of arbitrary arity to specify the values (and consequently the number) of the array elements. For example,

```
A : array[integer] := [ 2 : 34, 35 ]
```

creates an integer array of two elements whose data values are 34 and 35 with the corresponding array index values being 2 and 3 respectively.

Arrays may also be defined from existing array values, for example, a

new array equal to an existing array. Alternatively, a new array could be defined using an existing array with some or all of its elements modified. The append operation performs this function and has the following format:

A[i : v]

This would define a new array equal to A but with the ith element replaced by the value v; in this case VAL–S assumes that i is within the defined bounds of the array A. To define an array by extending the index bounds of an existing array, the *array_add* operation is used. This operation has two forms to extend either the high or the low bounds of an array. For example,

array_addh (A, v)

returns an array value equal to A with the high bound extended by one and the new element taking the value v.

Another operation that has been implemented is the *array_setl* operation which returns an array with the array bounds shifted and has the form:

array_setl (A, j)

This returns an array with values the same as A but with the low index bound equal to j and all indices of the elements of the new array shifted as required.

The final operation in VAL–S which can be used to create an array is concatenation. If two arrays are concatenated in the form $A \mid\mid B$, then the size of the new array returned equals the sum of the sizes of A and B with the low index value equal to that of A. The elements copied from A retain their original index values with those copied from B having their

index values shifted as necessary.

Once an array has been defined, any element in the array may be selected. For example, the *i*th element of an array *A* is selected using the usual notation *A[i]*. The programmer is required to ensure that array elements selected are within the defined bounds of the array, otherwise unpredictable problems will be encountered without warning during the execution of the program as error data types have not been implemented.

Three other operations associated with arrays have been implemented each of which returns an integral value. The *array_liml* operation is used to find and return the low index of an array, and the *array_limh* operation returns the high index. The third operation is the *array_size* operation which simply returns the size of an array.

Further qualification of the type and format of operations and expressions defined in VAL–S may be found in the syntax charts provided in Appendix B or in the original VAL language reference manual [Ackerman & Dennis, 1979].

4. PARSING AND GENERATION OF THE INTERMEDIATE FORM

4.1 Overview of the compilation process

The processing of a VAL–S program is composed of three basic phases. An overview of this processing is shown in Figure 4.1. The first phase, described in this chapter, is to parse the program and generate the intermediate form structure, termed the triple graph (TG). The second phase, to generate the corresponding dataflow graph from this intermediate form, will be described in the next chapter. The third phase is to run the dataflow graph through the simulator of the MDFM, described in Chapter 2, to produce the program's results.

The compiler described in this thesis and illustrated diagrammatically in Figure 4.1 performs the first two phases mentioned above. Each of these phases is broken down into a number of smaller steps. The first step in phase one is to generate the TG during parsing, with links from triples to their operand nodes. The second step is performed when parsing is complete; it involves a pass over the TG to insert BRANCH triples and to create bidirectional links between triples by entering links from triples to their users. The last step of phase one involves a second pass over the TG and includes removing redundant triples and writing out the TG to a file. These processes will be further explained in the following sections of this chapter.

The first step of phase two again involves a pass over the TG to create the corresponding dataflow nodes, but in general, does not link the nodes together. The linking of the dataflow nodes is done in the second step by again passing over the TG to create the arcs between the dataflow nodes from the user links defined in the TG; at the same time duplicate nodes are created where necessary. This phase will be fully described in Chapter 5.

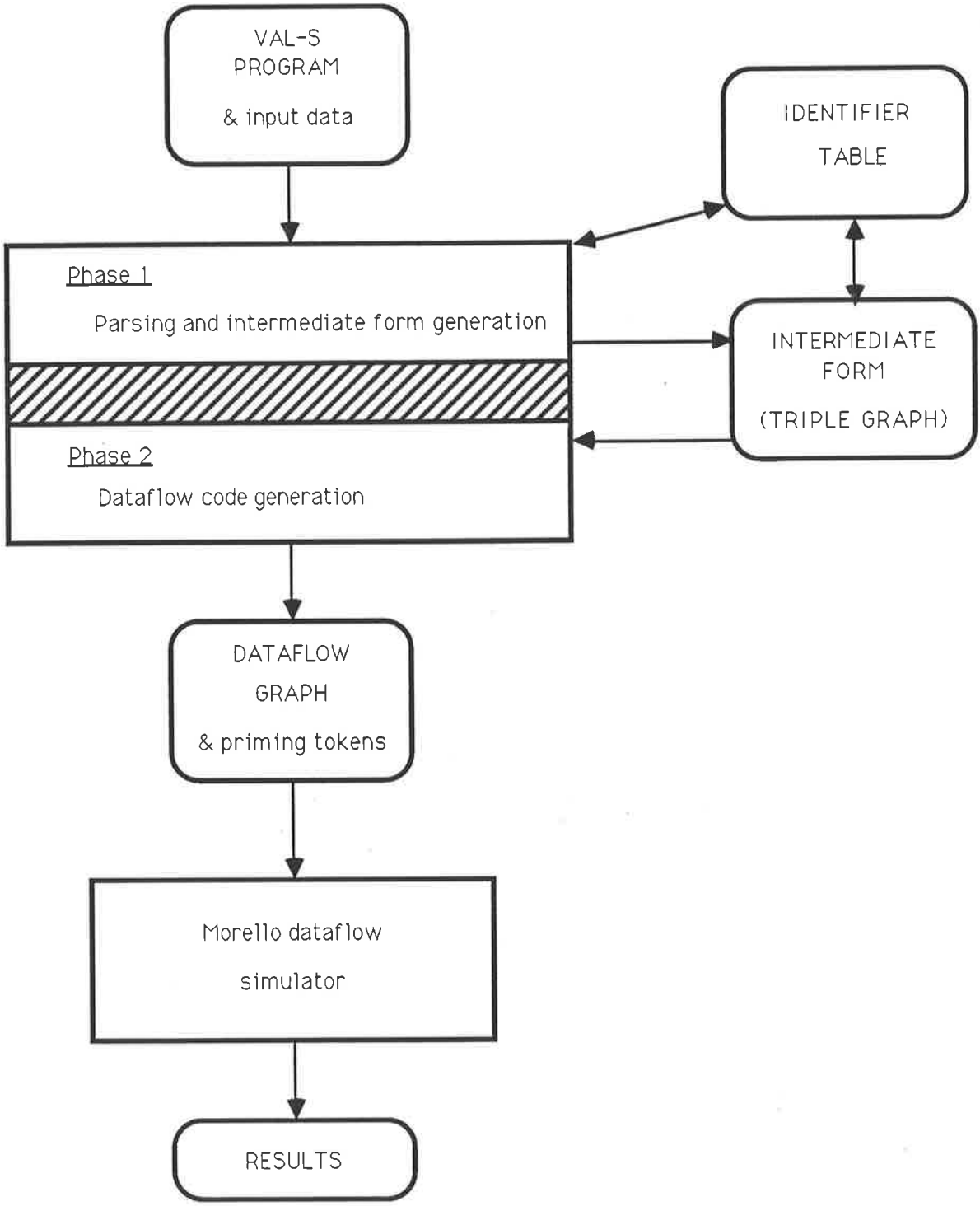


Figure 4.1 System overview

4.2 Parsing

Parsing of VAL–S programs is done using a combination of two methods, recursive descent down to the level of an expression, and then switching to operator precedence. The source language program is scanned one line at a time, each line being read into a character array for processing. Recursive descent parsing [Wirth, 1976] of programs is based upon the syntax of VAL–S given in the syntax charts in Appendix B; expressions are parsed by operator precedence [Gries, 1971] using the precedence levels for operators given in Table 2.1. During parsing, the intermediate form of the program, the TG, is generated; this is related both to conventional intermediate forms and in its final state, to the target dataflow graph. The implementation uses an operator stack, a table of operator priorities, an operand stack and an identifier table. An operand may be a literal value, an identifier (including a function application), or the result of a previous subexpression.

4.3 The triple graph

The term *triple* has been used in the literature [Gries, 1971] to describe the components of intermediate forms during compilation, such components basically being composed of an operation and two operands as in a simple arithmetic operation such as addition. Analogously, the nodes of our intermediate structure have been termed triples, even though in some cases multiple operands are involved, together with more complex associations. Thus each triple node of the TG consists of a triple conceptually made up of basically two operands and an operator. Each operand of a triple must either be a literal value or a reference to another triple, identifiers being represented by triples. A literal operand contains both its type and data value; a subexpression used as an operand simply references the triple in the corresponding subgraph which produces the

result of the subexpression. As described in Chapter 2, each identifier in VAL–S must be defined exactly once when it is declared. The expression defining the runtime value of an identifier is translated into a subgraph of triples similar to an expression tree. The root triple in this subgraph represents the value to be used wherever the identifier is referenced in subsequent expressions. So where an identifier or subexpression is an operand of a particular operation, this operand is represented in the triple for the operation by the root triple generating the value of the identifier or subexpression. Thus triples in the TG are linked together directly by their operands. Modifications to the basic triple structure are required when the number of operands differs from two. Representation of arrays, conditional expressions, iterative loops, function calls and function definitions, requires more complex triples which are described in later sections.

When complete at the end of phase one, each triple also includes a linked list (termed the user list) of references to those triples which use the result of the operation represented by the triple. This means that from any node in the TG, links (pointers) between triples can be traced in two directions, both to the operands of the triple and to the users of the triple. This is in contrast to the target language dataflow graph representing the program, where the arcs are only from nodes to users, in the direction of data flow. The ability to traverse the TG through the operands of triples is an important property of the intermediate form, required for the implementation of the optimizations described in Chapter 6. Figure 4.2 shows the triple subgraph for the expression $a*b+c*d$, the corresponding dataflow graph having only the links shown with dashed lines.

A conventional triple consists of an operation and two operands, which are shown in the Pascal record structure representing a triple node in Table 4.1. However, this structure has been expanded to incorporate other

information required in the TG. The operand list which is used in some special triples, as described in later sections, can have a variable number of operands. The purpose of the other fields shown will be explained later in this chapter, although it should be noted that, as with the operand list, not all fields are required for every triple created.

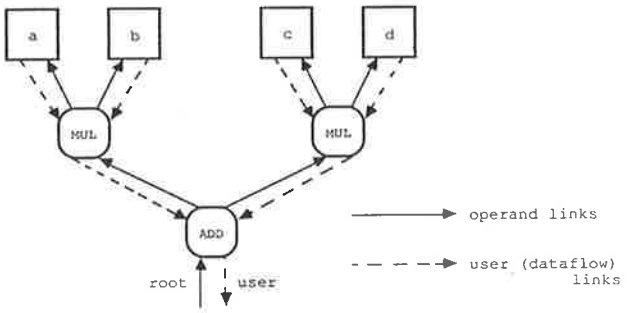


Figure 4.2 Triple subgraph for a*b+c*d

```
triple_rec = record
        opd1,opd2   : operands;      {operands of triple}
        opdlist     : ^operands;     {operand list – optional}
        operation   : nodeops;       {operation performed}
        EC          : ^trip_index;   {execution condition}
        dfnode      : ^node_ref;     {address of corr. df node}
        loop_level  : level_index;   {nesting of iterations}
        new_triple  : ^trip_index;   {replacement triple}
        sim_triple  : ^triple_ref;   {list of similar triples}
        no_users    : user_index;    {number of user triples}
        user_list   : ^user_triples; {list of user addresses}
        end.
```

Table 4.1 Pascal record structure of a triple node

4.3.1 Parsing expressions

Expressions are traversed by the parser from left to right; during this processing operands and operators are stored on their respective stacks. When an operator is encountered with a precedence less than or equal to

that of the top-most operator on the operator stack, that operator is popped from the stack. The operands associated with the operator are then popped from the operand stack, the corresponding triple is created and a reference to it placed back on the operand stack. This process continues until the operator stack is empty, or until the top-most operator has a precedence lower than that of the newly found operator. This new operator is then placed onto the operator stack and parsing continues. The Pascal record structure of an operand on the operand stack is given in Table 4.2. Each operand consists of a data type (integer, real, array, etc), the kind of the operand (literal or triple) and either a literal value for a literal operand, or in the case of a triple operand, the triple address, execution condition (described in Section 4.3.3) and a flag indicating whether the operand triple represents a common subexpression or was created (explained in Chapter 6).

```
operand = record
            datafield : data_record;                {data type}
            case opd_kind of
              triple  : (trip_addr : 1..tripmax;    {triple address}
                         EC        : integer;       {exec cond of operand}
                         created   : boolean);      {indicates if CSE}
              literal : (lit_val : literals)        {literal value}
            end
```

Table 4.2 Pascal record structure of elements on the operand stack

4.3.2 The identifier table

Each identifier declared within a program is entered in the identifier table during parsing. An identifier record contains the name of the identifier, the block level at which the identifier is declared, the data type and kind of the identifier, a reference to the value associated with the identifier and

other special fields in the case of loop variables, function names and arrays. VAL–S is a block structured language, so the same name may be used for different identifiers in different blocks. To find a particular identifier record in the identifier table, a hash function is used. Each identifier has a hash value which is calculated using its name, and identifiers with the same hash value are stored together in a linked list. When a new identifier is declared, the identifier table is searched for an identifier with the same name at the same block level to ensure that the identifier has not already been declared at this block level. If one is not found, a new identifier record is created containing the information described above. As each identifier declared in VAL–S must be defined by an expression, the root triple reference (or address) for that expression can be inserted at this time. Whenever the identifier is referenced in expressions encountered subsequently, this triple reference is used. The linked list of identifiers with the same hash value is searched (stored as a last-in-first-out list) so that the first identifier found with the correct name is the most recently declared. This ensures that the correct identifier record is found in the identifier table when multiple definitions for the same name occur.

The type and kind of identifier being defined is also recorded in the identifier table. An identifier can be a function module name or application, a parameter, a loop variable, a constant, or an index value name. Function names and loop variables, along with array definitions, require extra fields associated with their identifier table entries. The number of parameters and the ~~return~~ RESULT address (link created dynamically to point of call - explained in Sections 4.3.7 + 5.2.7) of a function module are stored in the identifier record of a function name, the size and offset of an array is recorded in an array identifier table entry and the addresses of two special triples, which are described in Section 4.3.5, are recorded with loop variable identifier records.

At the end of parsing a block, all new identifiers introduced in that block are removed from the identifier table. This ensures that only identifiers in scope remain in the identifier table.

4.3.3 The Execution Condition

Consider the VAL–S conditional expression from example 1, version B in Appendix C, also shown in Table 4.3. The value of the boolean test expression *i–1=0* determines which of the arms of the conditional

```
if i–1 = 0 then
        i
else
        fact2 ( i–1 ) * i
endif
```

Table 4.3 Example VAL–S conditional expression

expression is executed. This information is stored as the execution condition (EC) for the triples representing the arms; for example, *i–1 = 0* is the EC of the expression *i* occuring in the true arm, and its negation is the EC of the expression in the false arm. The EC of a triple specifies the conditions under which the corresponding operation will be executed at run time; on a dataflow machine it specifies through which boolean test expressions the triple's operands must be switched before they can be used by the operator in that triple. As conditionals in VAL–S can be nested, the EC of a triple is a conjunction of boolean expressions or their negations, each conjunct corresponding to the test condition (or its negation) following the word **if** in a VAL–S conditional expression (see example in Table 4.4). However, for a dataflow machine, the order of the booleans in the EC is significant as it determines the order in which the operands of the triple are to be switched to reflect the nesting of the boolean expressions in the

original source language program. For example, the EC of E2 in Figure 4.4 is ~B1 ^ B2; its operands cannot be switched through B2 first because if B1 is true, B2 should not be computed (its computation when B1 evaluates to true could lead to an infinite computation or an error). Thus the normal commutative and associative rules of conjunctions cannot be utilized when dealing with ECs.

The notion of an EC is extended to all expressions and specifies under what conditions the expression will be evaluated. Table 4.4 gives the ECs for the subexpressions in the conditional expression shown.

Expression	EC	Representation
if B1 **then**	true	nil
E1	B1	[B1]
elseif B2 **then**	~B1	– [B1]
E2	~B1 ^ B2	[B2]
else E3	~B1 ^ ~B2	– [B2]
endif		

Table 4.4 ECs for a nested conditional expression

The EC of a triple is represented by an indirect reference to the root triple of the most recent boolean expression specified in the EC. These triple references actually refer to COND triples described in the next section; a COND triple refers to the associated boolean expression. In the implementation, only the latest COND triple reference needs to be stored with each triple as the complete EC is the conjunction of the boolean condition (or its negation) and the EC of the COND triple. As each triple stores this reference, all other boolean expressions through which the triple's operands must be switched can be found in reverse order by linking backwards through the triple references of each boolean expression in the EC until an unconditional EC of *true* (or a nil reference) is found. In the

representation of each EC in Table 4.4, an expression is enclosed within square brackets to indicate a reference to the triple giving the result of the last boolean expression in the EC, this triple always being a COND triple; a minus sign is added when the corresponding boolean expression is false in the EC. For example, the EC of E2 is recovered from the chain starting with a reference [B2]. This refers to the COND triple associated with the expression B2 which has its own reference – [B1]. The reference associated with B1 is *true* indicating that the complete EC of E2 has been found, namely ~B1 ^ B2.

4.3.4 Special triples associated with conditional expressions

Four special triples are used to represent a simple conditional expression in the TG. They are used to identify and link together the different parts of a conditional expression. The first is the COND triple which is used to identify the boolean expression that determines which arm of the conditional will be evaluated. The COND triple has a single operand, namely the root triple giving the result of this boolean expression. The COND triple has the same EC as the boolean expression. It was explained above that it is the COND triple which is used as the representation of the EC in the TG. In Chapter 6 it will be explained how the triple representing a boolean expression can be shared, but not the COND triple.

The second triple is the THEN triple which has a single operand, the root triple giving the result of the true arm of the conditional expression. The EC of this triple is the same as the conjunction of the EC of the COND triple with the COND triple itself. Similarly the ELSE triple has as its only operand the root triple giving the result of the false arm of the conditional; it has the same EC as the THEN triple but with the negation of the COND triple conjoined with the EC of the COND triple. Finally the IFF triple is

used to reference the whole conditional as a single expression; the THEN triple is its left operand with an EC equal to that of the COND triple, and the ELSE triple its right operand with an EC which is the complement of that of the THEN triple. Note that the COND triple is only linked to the triple subgraph of the conditional expression through the EC fields of the triples for the arms.

This set of four triples generated for simple conditional expressions can be extended to include nested conditionals and those containing **elseif** branches. The method for doing this becomes apparent from Figure 4.3 which shows the basic triple subgraph for the conditional expression in Table 4.4; user links have been omitted and the dashed links show the EC chains.

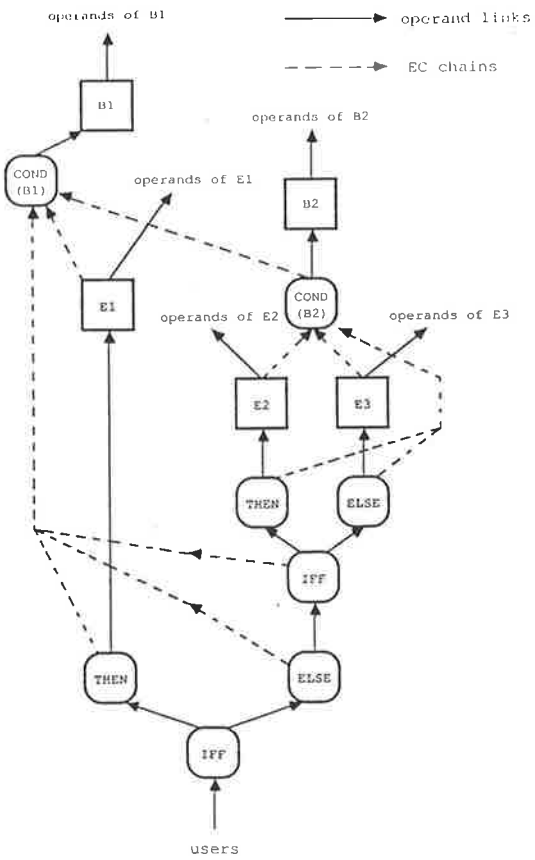


Figure 4.3 Triple graph of a conditional expression

From this example it is apparent that the basic group of four triples (COND, THEN, ELSE and IFF) needs to be repeated for each **elseif** branch and for

each nested conditional expression to obtain a correct representation of more complex conditional expressions. This representation allows common conditional expressions to be recognized as described in Chapter 6.

4.3.5 Special triples associated with for-iter expressions

Iteration is performed in VAL–S using the **for-iter** loop construct as described in Chapter 3. Termination of iteration is controlled by certain boolean conditions which are part of conditional expressions within the loop body. In VAL–S, identifiers which are referenced within the loop body can be split into two groups, namely loop variables, all of which are declared as such at the beginning of the loop, and any other externally defined identifiers which are referenced within the loop body. At execution time on the MDFM, each iteration cycle requires a set of input tokens with matching labels corresponding to these identifiers. In the compiler, loop variables are treated differently from other identifiers for two reasons. Firstly all loop variables are declared and defined at the start of the loop, and also, during execution, loop variable values can be redefined in iteration expressions to create new values for subsequent iterations. In the execution of these loops on the MDFM, if a loop variable is used but not redefined in an iteration expression, then a copy of the identifier's value must be circulated to the next iteration, with appropriate updating of the token label (see Figure 5.1 for example). Values corresponding to identifiers defined externally from the loop also require copying and circulating in this manner. Data tokens carrying the new values of loop variables and the copied values of other identifiers at execution time must have their labels updated on each loop iteration. The instructions required to achieve this are created during dataflow code generation. To model this in the TG, for each identifier, an IDENT0 and a NEWID triple are created, the addresses of these triples being stored in the identifier table.

The IDENT0 triple represents the initial value of a loop variable and has as its operand the triple defining that value. For other identifiers defined outside the loop, the IDENT0 triple represents the constant value of the identifier; its operand is again the root triple defining that value. The NEWID triple represents the new value of a loop variable as generated by a loop iteration. Its operands are the one or more iteration expressions generating new values of the loop variable in different arms of a conditional expression. For other identifiers defined outside the loop, the NEWID triple represents the copy of the identifier's value, required at execution time for the next iteration. Hence, in this case, the operand of the NEWID triple is the IDENT0 triple. This will also apply in the event that a loop variable is not redefined in an iteration expression. To explain this further, consider the loop variables in the VAL–S example in Table 4.5 and the corresponding triple subgraph in Figure 4.4.

```
for a, b := 1, n do
        if a<n then
                iter a:= a+1;
                        b:= b+a
                enditer
        else
                b
        endif
endfor
```

Table 4.5 Example of for-iter loop construct

In this example there are two loop variables a and b, initialized to the values 1 and n respectively. These initial values are the operands of the IDENT0 triples from which references are made to the loop variables when they are used within the loop body. In this particular example there is just

one iteration expression and each loop variable is redefined within it, so at this stage, the respective redefinitions are the operands of the NEWID triples for *a* and *b*. Note that subsequently, when user lists are created, the users of IDENT0 triples will be the same as those for the corresponding NEWID triples.

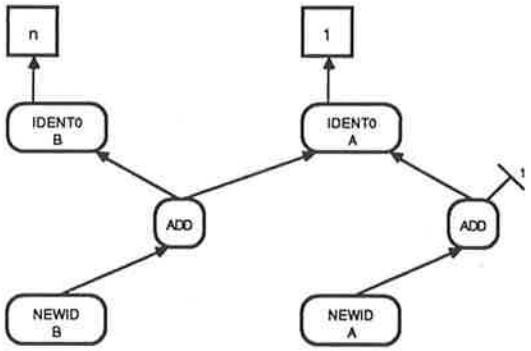


Figure 4.4 Basic triple graph of loop variables

Unlike loop variables which are defined at the head of the loop, it is not known which other identifiers are used inside a loop until they are encountered during parsing. To store the information required to model the circulation of copies of the values of these identifiers, a special FORITER triple is created at the beginning of a loop. This triple can be viewed as providing an interface between the identifiers defined outside of the loop and their usages within the loop. Instead of the usual two operand fields, this triple node requires a linked list of operands, each of these operands corresponding to an identifier, which is not a loop variable, referenced inside the loop. Identifiers are added to this operand list as they are encountered during parsing of the loop, and for each new identifier both an IDENT0 and a NEWID triple will be created as explained above.

Again the IDENT0 triple has one operand, the root triple generating the value of the identifier at execution time. This operand is also added to the operand list of the FORITER triple and the IDENT0 triple is then added to

the *user list* of the FORITER triple. This means that later, the operands of the FORITER triple can be matched to the corresponding IDENT0 triples in its user list. This is necessary in order to recognize multiple occurrences of an identifier that are encountered within a loop body, as each occurrence must optimally reference the same IDENT0 triple. Figure 4.5 illustrates the TG structure at this stage for the example of Table 4.5.

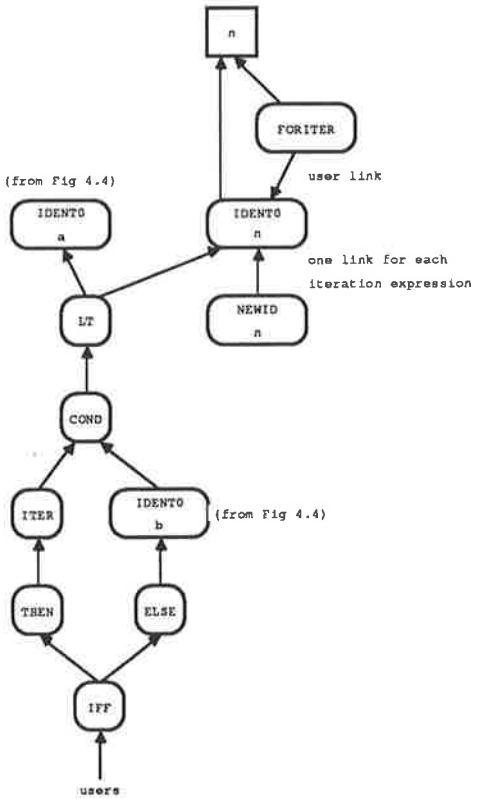


Figure 4.5 Rest of triple graph of for-iter expression from Table 4.5

Iteration within a **for-iter** loop continues only if an iteration expression is executed on each iteration; no results are generated from the loop cycle when an iteration expression is executed. Whenever an iteration expression is encountered during parsing, a special ITER triple is created. As an iteration expression can only contain redefinitions for loop variables, the ITER triple is used to represent the result of the iteration expression. In the translation of VAL–S programs, the ITER triple is always used as a dummy operand of either a THEN or an ELSE triple, the ITER triple itself

not having any operands. The main purpose for introducing the ITER triple is to store in the triple graph the EC specifying the conditions under which the iteration expression will be executed. At execution time, if these conditions are met, the iteration expression is executed and further iteration continues. On each such iteration cycle, a copy of the value of each identifier used within the loop must be passed on for use in the next iteration cycle; as explained earlier, this is modelled in the TG with the NEWID triple, where the value is represented by the IDENT0 triple associated with the identifier. There may be many iteration expressions inside a loop and a new copy of the value of the identifier is required at execution time for each iteration expression. In the TG, each usage is represented by a link between the IDENT0 and NEWID triples. Thus the NEWID triple may have many operands, so each operand is stored in the operand list associated with this triple.

Note that BRANCH triples required for switching are not inserted into the TG until a later stage of processing, so the EC of each operand must also be stored in the operand list of the NEWID triple in order to create the appropriate switching triples subsequently. Figure 4.5 shows the NEWID triple for identifier n has one operand, the IDENT0 triple, corresponding to the one iteration expression.

It is not until the end of a loop is reached during parsing that the operands of the NEWID triples are created for each identifier in the operand list of the FORITER triple. This is done by taking each triple in the user list of the FORITER triple and, for each iteration expression inside the loop, the IDENT0 triple is added to the operand list of the NEWID triple with an EC equal to that of the iteration expression. Note that IDENT0 and NEWID triples are created together so the address of the NEWID triple (in the TG) is the address of the IDENT0 triple plus one. The address of each ITER triple

is stored in a linked list with the FORITER triple, so each EC can be found directly. The address of the FORITER triple itself is stored separately in a globally defined loop level counter, which is used for the dual purpose of storing these addresses and also counting the level of nesting of loops. The counter is updated each time a loop is entered or exited.

The FORITER triple is used further when considering nested loops. Consider the case of two nested loops and an identifier used in the inner loop that is defined outside the outer loop. This identifier must pass through the outer loop to the inner loop and therefore must be circulated in the outer loop. All triple operands have an associated loop level field which is set to the value of the loop level counter when the operand is created; this is the address of the FORITER triple of the loop currently being parsed. When the end of a loop is encountered, any operand of the FORITER triple whose loop level field is less than that of the loop level of any outer loop, is added to the operand list of the FORITER triple of each such outer loop. This ensures that all identifiers are passed through consecutive nested loop constructs correctly.

4.3.6 Special triples associated with forall expressions

Several special triples are used to represent the information contained in a **forall** expression. The first of these is the INDVAL triple which is used to represent an INDex VALue (IV) name. An INDVAL triple is created for each IV name declared within a **forall** expression, its left operand being the low bound for the name and the right operand the high bound, as shown in the example in Figure 4.6. A reference to an IV name inside the **forall** expression uses this INDVAL triple as its operand. The number or range of index values can be determined from the bounds of the IV names; in Figure 4.6, *J* has B–A+1 index values. When one or more IV names are defined,

each has its own range; the range of the whole **forall** expression can be calculated by cross multiplying the individual ranges of each IV name.

As explained in Chapter 3, only one-dimensional arrays have been implemented in VAL–S. As each IV name in a **forall** expression corresponds to a dimension of an array created by the expression, only one IV name is allowed in each **forall** expression in VAL–S. However, the TG has been designed to allow for further dimensions by introducing a RANGE triple. This triple takes as its operands the INDVAL triples of each IV name, as shown in Figure 4.6. Whenever the whole range of a **forall** expression is required it is from this triple that the value can be calculated. Further complications involved with accessing IV names within **forall** expressions will be discussed in Chapter 5.

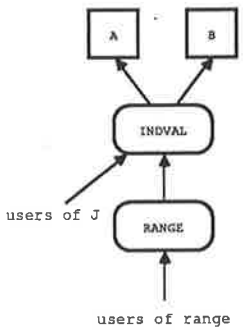


Figure 4.6 Triples created for IV name J in [A,B]

Forall expressions are used to create arrays with index values specified over a range set by the IV names. At execution time on the MDFM, any identifier values used to create an array must be duplicated over the entire range of the **forall** expression and the index tags set appropriately. This means that for a range of [A,B], B–A+1 copies of the value of the identifier must be duplicated. This is modelled in the TG by a ProLiFerate triple (PLF) which takes a data value as its left operand and duplicates it over the range specified by its integral right operand. The function of this triple

corresponds to the function associated with a MDFM node of the same name, as described in Chapter 2. Again note that here as with loops, at the start of the **forall** expression it is not known which identifiers are used inside the expression and hence which require proliferation. The example in Table 4.6 illustrates this point (*n* will require proliferating on the MDFM).

forall J **in** [1, n]	**forall** J **in** [1, n]
construct J*n	**eval times** J
endall	**endall**

Table 4.6 Examples of forall expressions

When a **forall** expression is encountered during parsing, a special FORALL triple, similar to the FORITER triple, is created. The operands of this triple are those root triples giving the values of identifiers, other than IV names, used inside the **forall** expression. When a new identifier is found, it is added to the operand list of the FORALL triple and a corresponding PLF triple is created; in Figure 4.7 this occurs for the identifier *n*. As with IDENT0 triples in **for-iter** loops, the PLF triple is used as an interface between the defined value of the identifier outside the **forall** expression and the uses of that identifier inside the expression, which are taken from this associated PLF triple.

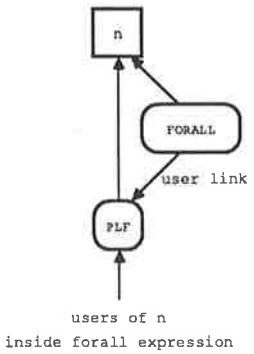


Figure 4.7 Array creation in forall expressions

Nested **forall** expressions and mixtures of **forall** expressions and **for-iter** loops introduce similar complications to those encountered with nested loops, in that identifiers used in inner constructs must first pass through the interfaces of the outer constructs. The same method is used here as with nested loops, namely using the loop level counter to record the construct currently being parsed, either a **for-iter** loop or a **forall** expression. When the end of the construct is encountered, if any outer constructs exist, the loop levels of all operands of the inner FORITER or FORALL triple are compared with the loop level of the outer construct. Any identifier whose loop level is less than that of the outer construct is added to the operand list of the FORITER or FORALL triple of the outer construct.

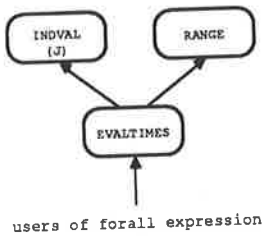


Figure 4.8 EVALTIMES triple associated with example in Table 4.6

The final group of triples to be dealt with in **forall** expressions are those associated with the **eval** statement. Figure 4.8 above illustrates one such EVAL triple (evaltimes) used for the *times* operation (from Table 4.6). There is a special EVAL triple for each operation defined in an **eval** statement. The left operand contains a reference to the expression, in this case *J*, and the right operand references the range over which the operation is to evaluate, which in this example is the range of values *1* to *n*, defined by the INDVAL triple associated with *J*.

4.3.7 Special triples for function definitions and function calls

A function definition is represented in the TG as an independent subgraph

with an input and output interface to the rest of the program; a call to a function can occur in VAL–S before the function body has actually been defined (in the case of forward declarations). When a function header is declared, either as the function definition or in a forward declaration, several special triples are created. A PARameter triple is created for each of the formal parameters in the header. As the actual parameters are not known at this stage, the triple is created without operands. PAR triples are used as an interface between the actual parameters of a function call and the formal parameters defined in the function header. Uses of the formal parameters inside the function definition have the PAR triples as operands. A RETurn triple is also created for each result returned by the function (the result may be a single value, an array, or a tuple of these). The operands of these result triples cannot be inserted until the function body is parsed. Figure 4.9 illustrates the triple graph for the following function header definition:

function example (a, b : integer **returns** integer, real)

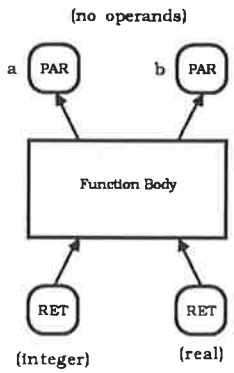


Figure 4.9 Triple Graph of function interface for example

When a function call is encountered, a CALL triple is created whose operands are the actual parameters. If the function is in scope there will be an entry for it in the identifier table from which compatibility between actual and formal parameters can be assessed. A reference to this

identifier table entry is stored with the CALL triple. As with the FORITER and NEWID triples, CALL triples may have many operands so each actual parameter is stored in the operand list of the CALL triple. Note that links between the actual parameters in the operand list of the CALL triple and the formal parameters represented by the PAR triples created from the function definition are not inserted until the next step described in Section 4.4.

The design of CALL triples allows multiple function calls to the same function but requires only one definition of the function within the TG. A similar design is required for values returned by a function. A CRET triple is created for each data value returned by a function each time the function is called. The CRET triple is referenced wherever values returned by the function call are used later in the program. The operand to the CRET triple is the RETURN triple created for the corresponding result during the function definition. However, note that the user link from the RET to the CRET triple is not a static link and will need to be generated dynamically in the corresponding dataflow graph (see Section 5.6). Thus a function call is translated into one CALL triple to link actual and formal parameters and a number of CRET triples corresponding to the number of results returned by the function.

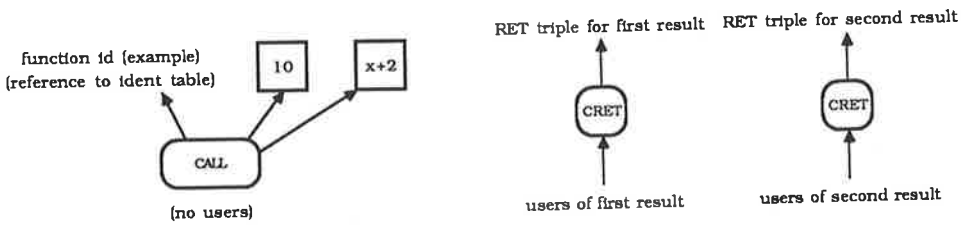


Figure 4.10 Triple Graph for function call example(10,x+2)

The CALL triple and CRET triples that would be created for a call to the

function defined in Figure 4.9 are shown above in Figure 4.10.

4.3.8 Array, select and append triples

Whenever an array is created in a VAL–S program, a corresponding ARRay triple is created. This triple holds all information associated with the array in an easily accessible form. The values of the array elements may be generated from many sources; the triples generating the values of these array elements are the operands of the ARR triple, accessed in index order by the operand list of the ARR triple.

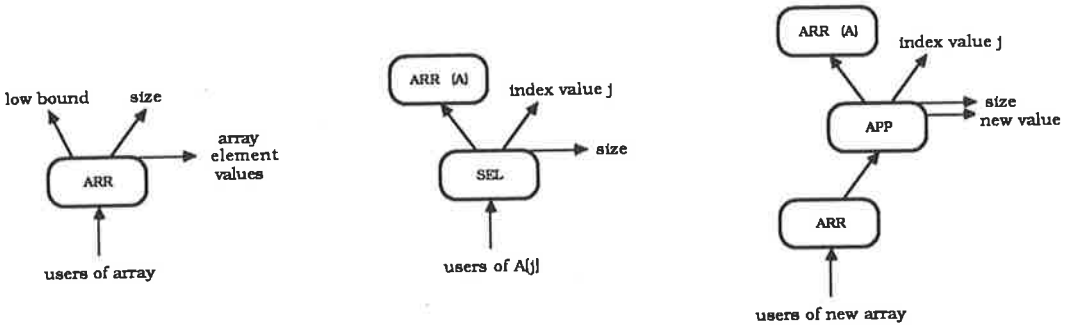


Figure 4.11 Triples associated with array operations

Two other pieces of information are kept with the ARR triple, the low bound minus one (the offset) of the array and the size of the array, both of which are required to perform array operations. In the TG, all arrays are maintained with index values starting at one to simplify matching of index values; so to get the *jth* element of an array whose bounds are *[A,B]*, the offset *(A–1)* is simply subtracted from *j*.

Each time an array element is selected, a SELect triple is created. To select an element from an array, the array, the index of the element to be selected and the size of the array are all required. The left operand of the SEL triple is an ARR triple representing the array, the right operand is the index of the array element to be selected, adjusted by subtracting the offset and the

size is stored in the operand list of the SEL triple[1]. Any uses of the array element are stored in the user list of the SEL triple. Figure 4.11 illustrates the structure of the SEL triple.

An APPend triple is created for every new element added to an array. The operands of an APP triple are the same as those of a SEL triple with the addition of the new element value in the operand list of the APP triple. At execution time, an append operation creates a complete new array, so a new ARR triple is required to represent this new array value. The new ARR triple takes the old offset and size from the array to which a new value was appended, and has the APPend triple as the source of the array values. It is this new ARR triple which is referenced whenever the new array is used, as shown in Figure 4.11.

4.3.9 Priming tokens

As mentioned in Chapter 1, priming tokens are used to trigger the execution of a dataflow graph by supplying input tokens to the topmost starting nodes in the graph. In the VAL–S source language, these priming tokens correspond to input values supplied for the formal parameters of the function module specifying the input/output requirements of the program. In the intermediate form, these input values are stored in special PTOK triples from which priming tokens are generated in phase two of the compilation. Each PTOK triple has one literal operand corresponding to the input value and a user list containing the triples which use this input. Note that each priming tokens can serve as input to one node only, so more than one entry in the user list of a PTOK triple will require the creation of one or more duplicate nodes, as described in the next chapter.

---

[1]The size of the array is required to proliferate the index value being selected, as described in Chapter 5.

4.4 Inserting user lists and generating BRANCH triples

In dataflow, all inputs to the arms of a conditional expression must be passed through switching nodes or gates (see Section 2.3.1). A switching node is represented by a BRA (for branch) instruction for the MDFM; the execution of this instruction was described in detail in Chapter 2. Consider the example conditional expression given in Table 4.3; the only identifier, apart from the function call, used in either arm is *i*. At execution time, the associated data value must pass through a switching node before it is used, either as the result of the conditional expression in the true case, or in the function call to *fact2* and the following multiplication, in the false case. The execution of switching nodes ensures that only one of the arms of the conditional expression will be executed on a dataflow machine, depending on the boolean value of the condition.

In the TG, a BRANCH triple corresponds to the BRA node, this triple being required in the TG to generate dataflow code correctly. However, to facilitate the optimizations discussed in Chapter 6, it is convenient to create the BRANCH triples after the rest of the TG has been built. This means that switching requirements associated with the operands of each triple must be able to be determined by comparing operands and the triples which use them. For this reason the Execution Condition (termed Data Path Condition in Jones, Kidman & Morello, [1985]) or EC, as explained in Section 4.3.3, is defined and recorded for each triple in the TG.

After parsing is complete, BRANCH triples are created and user lists inserted into the TG. One pass over the TG is required in this step. However, triple graph nodes are processed one at a time, so that a node's address can be inserted into the user lists of its operands. In order to create BRANCH triples, the EC of a triple is compared with the EC of its operands.

Note that the EC of an operand must be at the root of the EC of the triple that uses it. If this were not true, the TG would be inconsistent. As each BRANCH triple is created, the user lists of its operand triples are inserted.

When comparing the EC of a triple with that of its operand, it may be found that several BRANCH triples are required to correctly switch the operand to its user triple. In order to perform the optimization on BRANCH triples described in Chapter 6, these BRANCH triples must be created in the order in which the corresponding boolean expressions appear in the EC, with the first BRANCH triple created being linked to the operand and the last one created linked to the user triple. The generation of these BRANCH triples is achieved firstly by reducing a copy of the EC of the user triple until it is the same as that of its operand, but saving the original EC. Then each BRANCH triple is created in turn and linked to the corresponding boolean condition specified in the original EC of the user triple, until the end of this EC is reached.

As an example of how this is done, consider a triple created for the expression E2 in Table 4.4, with EC ~B1 ^ B2, and suppose that an operand of this triple has an EC of true. In this case, the operand will require switching through two BRANCH triples linked to B1 and B2 respectively. The following steps summarize this procedure:

(i) Search a copy of the EC (list) of E2 until the reduced EC equals that of its operand (true in this case). In this process, B2 is encountered first, then ~B1 and finally nil or true. Note that as an EC is represented by a reference, a single comparison serves to compare two ECs.

(ii) The first BRANCH triple is created with the original operand as its left operand and the boolean expression B1 as its right operand.

(iii) The address of this first BRANCH triple is inserted into the user lists of the original operand and the root triple representing B1.

(iv) The second BRANCH triple is created with the false output of the first BRANCH triple as its left operand, and the boolean expression B2 as its right operand.

(v) The address of the second BRANCH triple is inserted into the user lists of the first BRANCH triple and the root triple representing B2. Note that the usage in the first BRANCH triple must be marked as false.

(vi) The end of the original EC is reached, so the address of the original E2 user triple is inserted into the user list of the second BRANCH triple.

This completes the generation of the required chain of BRANCH triples and insertion of user lists for the example of Table 4.4.

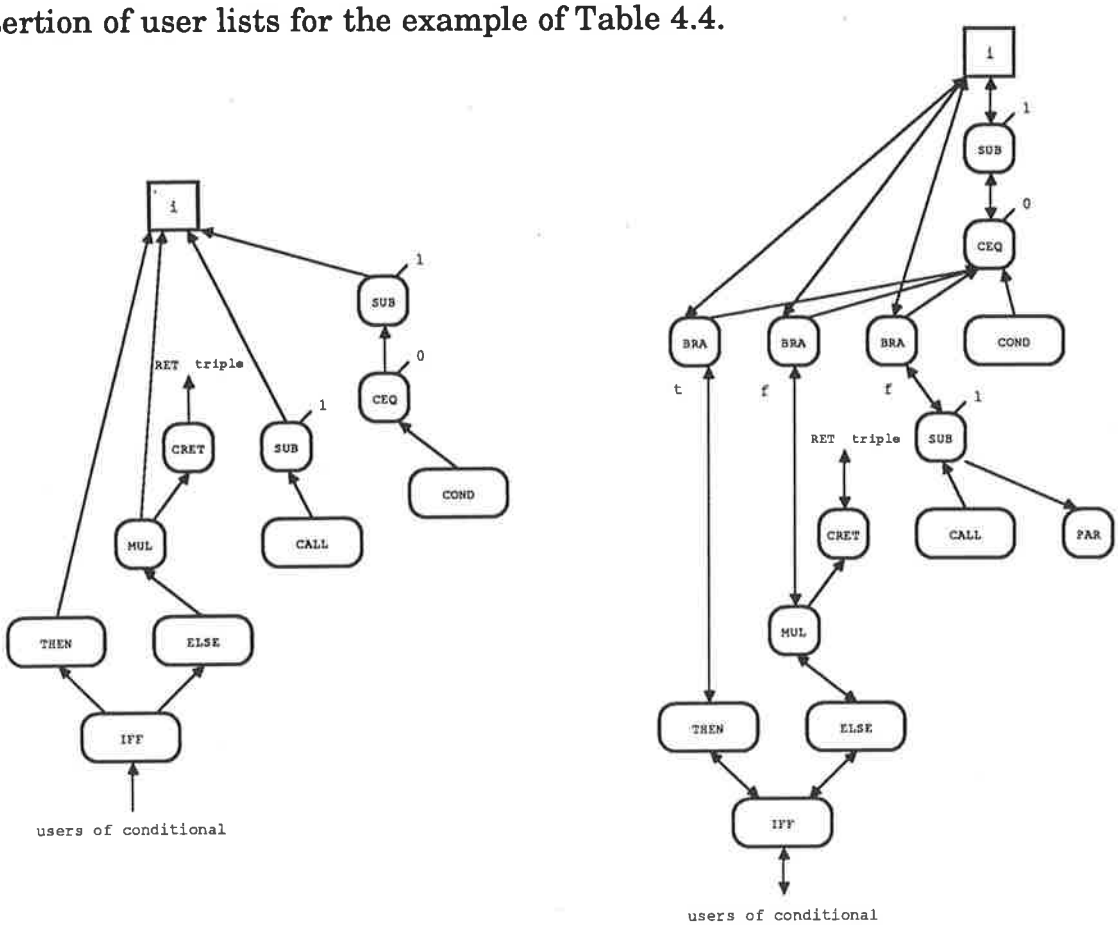


Figure 4.12 Triple Graphs for example conditional expression in Table 4.3

The complete TG for the example conditional expression of Table 4.3 is

shown in Figure 4.12 with BRANCH triples and user lists inserted. The smaller graph on the left is the incomplete TG prior to insertion of BRANCH triples and user lists.

In the case that no BRANCH triples are required (when the EC of a triple is the same as that of its operand), only the equivalent of part (vi) of the above process is required. Thus the address of the user triple is inserted directly into the user list of its operand.

This process becomes more complicated when all possible situations are considered, in particular when dealing with arrays where each element of the array requires switching. Most triples have either one or two operands, but in the case of some special triples such as the ARR, CALL and NEWID triples, a list of operands must be processed using this algorithm. Other special triples including ITER, FORITER, FORALL and PAR triples have no switching requirements. A further two triples must be considered as special cases, namely IFF and CRET triples, because their special design results in their ECs being inconsistent with those of their operands. However, this does not matter as their operands have already had appropriate switching and all that remains is to insert their user lists.

4.5 Elimination of unused triples

A source of unused triples can arise when an identifier in VAL is defined in a **let** expression but is not referenced in the body of the expression. This leads to a situation where a triple is created with no users. In this case, its user list is empty so there will be no need to create dataflow nodes corresponding to the triple, as the result of the corresponding operation is not used by any other node. Moreover, the token or tokens generated in the unnecessary **let** statement would be garbage, so in the implementation, the triple subgraph corresponding to the unnecessary **let** statement is removed

from the TG and the compiler issues an informative message. However, before such an unnecessary operation can be deleted from the TG, references to this unused triple in its operands' user lists must be removed. Once an entry has been deleted from an operand triple's user list, the operand triple itself may not have any users, and so on. The elimination of unused triples is performed in a second pass over the TG as the last step in the intermediate form generation phase and, for most triples, simply consists of a check that the triple has at least one entry in its user list. If not, a recursive algorithm removes this triple and any others consequently without any users, effectively deleting unnecessary triples in the TG before the corresponding dataflow code is generated. As this process is being performed, each triple is written to a file providing a listing of the TG (for checking purposes), which is otherwise lost on the termination of the compilation.

5. DATAFLOW CODE GENERATION

5.1 Introduction

A dataflow program is represented by a dataflow graph whose nodes are operations and whose arcs represent data flow between operations in the program. In this project, the target machine for the dataflow graph is the University of Manchester prototype dataflow machine (MDFM); programs are executed on the simulator described in Chapter 2. Each instruction for this machine consists essentially of an operation code and one or two destination instruction addresses. Figure 4.2 shows the basic relationship between the intermediate code of the triple graph and the dataflow graph for a single arithmetic expression. However, in many cases, branch nodes corresponding to the BRANCH triples will be required to switch data tokens between alternative paths and to absorb or kill unused tokens. Also duplicate nodes will be required in the dataflow program to create copies of data tokens; the TG allows multiple links between triple nodes. The dataflow graph is created as a sequential list of dataflow instructions by processing the TG as described in the following sections. Basically each operation recorded as a triple becomes a dataflow node or instruction, and the user list of the triple is translated to dataflow links represented by destination addresses in the dataflow instruction corresponding to the operation.

The dataflow code generation phase of the compiler, as shown in Figure 4.1, requires two further passes over the completed TG, generated in the first phase. The first pass in the second phase creates the dataflow nodes which either directly relate to operations represented by triples or are subgraphs of nodes translated from the special triples described in Chapter 4, such as function CALL triples and array SEL and APP triples.

The second pass over the TG is used to insert the destination addresses of the dataflow nodes and, where necessary, create duplicate nodes.

5.2 Dataflow node generation

5.2.1 Creating dataflow nodes for arithmetic operations

Simple triples representing arithmetic operations in the TG translate directly to single dataflow nodes or instructions. The node will contain the kind of the arithmetic operation, the number of inputs to and outputs from the node, the type and value of any literals used as operands, and the destination addresses and ports of nodes that receive the result of the operation. This information is stored directly in the corresponding triple with the exception of the destination addresses, which must be addresses of nodes in the dataflow graph, rather than the triple addresses stored in user list of the triple. These dataflow node destination addresses must be inserted at a later stage as the destination nodes may not have yet been created when the dataflow node is generated. To accomplish this, as each dataflow node is created, a pointer from the triple to its corresponding dataflow node is created and stored with the triple. This pointer is used when final dataflow destination node addresses are inserted during the second step described in Section 5.3.

More complex code generation techniques are required when generating the dataflow nodes corresponding to the special triples used to represent conditional expressions, iterative expressions, function calls and definitions, and array operations. These triples do not have a one to one correspondence with dataflow nodes, some have no corresponding node and others expand into a subgraph of nodes. However, complications arise when a triple's operands are required by several different nodes or when the result of a triple is produced by several different nodes in the

expanded dataflow subgraph. To handle this situation, pointers to dataflow nodes are associated with each operand of the triple and with the result. For most triples, these pointers reference the same node address, indicating a one to one correspondence between triples and dataflow nodes, but for complex triples like the CALL, SEL and PAR triples, they may be different. Later sections describe these code generation techniques in detail.

5.2.2 Eliminating special triples and generating triggers

Some of the special triples created in the TG to facilitate optimizations do not have any corresponding nodes in the dataflow graph. These triples can be ignored during both passes of the dataflow code generation stage unless they appear in the user lists of other triples. As each user list entry in the TG is translated to an arc in the dataflow graph, these triples will need to be removed from the user lists in which they occur in the second pass of dataflow code generation, described in Section 5.3. Triples which can be ignored include the FORITER and ITER triples associated with VAL–S for-iter expressions and FORALL triples associated with forall expressions. The main purpose of these triples is to hold information required in the generation of the TG, in particular the EC required to create the BRANCH triples. Once the TG is complete, these triples are no longer required and can be ignored.

Triggers are simply tokens which are sent to nodes in order to activate their execution. Every node in a dataflow graph must have at least one input token as it is not executed until its input token (or tokens) is passed to it. Some dataflow nodes, such as GEN and GAN nodes (see Chapter 2) actually contain the information to perform the required operation without receiving any further input data, but nevertheless, still need an

input token in order to execute. Thus a trigger, which can be any token, is sent to the node enabling it to be executed.

In this implementation triggers are always taken from the last formal parameter in the function definition in which they are required. This imposes the constraint that any node requiring a trigger in order to execute, must wait for the parameter from which the trigger is taken to become available before it can execute. Another approach [Skedzielewski & Glauert, 1984], is to use a special trigger token. There would be an additional overhead associated with passing this trigger into loops and across functions but the constraints of the above method are removed. Experimental comparisons between the two different kinds of trigger have not been reported.

5.2.3 Code generation for conditional expressions

Along with the triples mentioned in the last section, the special triples associated with conditional expressions, COND, THEN, ELSE and IFF triples, must also be removed from user lists, as these triples have no corresponding dataflow nodes. This is achieved by replacing each occurrence of one of these triples in other triples' user lists by the user list of the special triple itself. For example, in a conditional expression, the triples producing the results of the true and false arms have in their user lists THEN and ELSE triples associated with the conditional expression. The THEN and ELSE triples have the IFF triple in their user lists and the IFF triple has the addresses of other triples requiring a result from the conditional expression. In effect these last users are placed in the user lists of the triples producing the results of the true and false arms and the THEN, ELSE and IFF triples are subsequently ignored. A similar method is used to remove references to ARR triples from user lists.

5.2.4 Code generation for iterative loops

The IDENT0 and NEWID interface triples created for each identifier used within a loop provide the basis for creating the dataflow nodes required to manage execution of loops. As described in Chapter 2, on the MDFM, part of the label associated with a data token contains an iteration counter which is set during execution to distinguish between tokens from different loop cycles. The IDENT0 triple translates directly to the ZIL (Zero Iteration Level) node required to initialize the iteration counter of every data token used in the loop to zero at the start of the loop. The NEWID triple translates directly to the IIL (Increase Iteration Level) node required to increment the iteration level of each token on every successive iteration cycle; in effect, every token used inside the loop must be *circulated*. Figure 5.1 below shows the dataflow subgraph created for the identifier *n* of the example given in Figure 4.5. It should be noted that at this stage, node destination addresses are not inserted, this being done in step 2 of phase 2 described in Section 5.3.

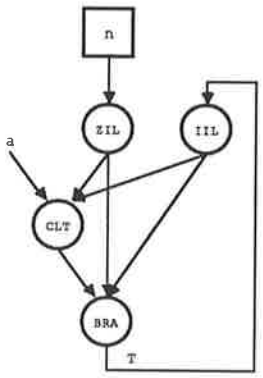


Figure 5.1 Dataflow graph corresponding to TG of Figure 4.5

A further complication arises in the case of nested loops. As there is only one iteration counter, values returned by an inner loop which are used in an outer loop must have their iteration levels reset to the value of the outer iteration level before re-entering the outer loop. This is done by saving a

copy of the outer iteration level before entering the inner loop and then resetting the iteration level field of all values returned by the inner loop to this saved iteration level. The resetting of the iteration levels is done by using the Set Iteration Level (SIL) node. This node takes a data valued token as its left operand and sets the iteration level of this operand to the value of the right operand. However, to do this requires both operands to have the same iteration level, so both must have their iteration levels set to zero. The outer iteration level is obtained using the Yield Iteration Level (YIL) node.

Figure 5.2 shows the dataflow graph required to reset these iteration levels. Links or arcs emanating from nodes such as the YIL node in Figure 5.2, which do not have a corresponding triple in the TG, are inserted as destination addresses when the node is created. All other destination addresses and any linking to nodes outside of this subgraph are inserted in the second step of phase two described in Section 5.3.

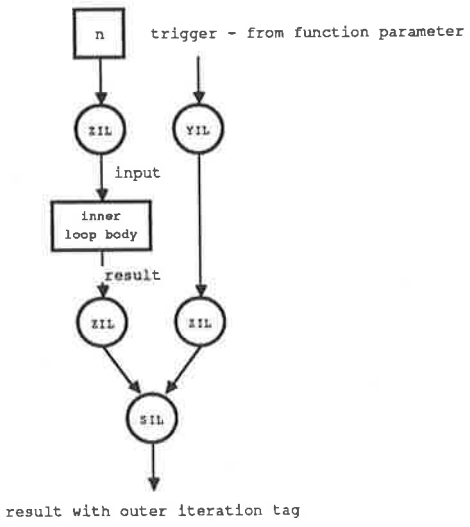


Figure 5.2 Dataflow nodes required to reset iteration levels (Phase 1)

This technique keeps data tokens of different cycles distinct in the outer

loop; however, consider the situation where the inner loop is nested inside an iteration expression of the outer loop. There is a difficulty here in that different iteration cycles of the outer loop may be executing the inner loop at the same time. In this case more than one set of data values would trigger the inner loop and distinction between them would be lost when their iteration levels were zeroed at the start of the inner loop. The solution to this problem involves using another field of the label of each data token. Before entering the inner loop a new unique activation name (used to distinguish between activations of the same function on the MDFM – see Chapter 2) is generated and all tokens entering the inner loop have their activation names set to this new value. This will ensure that different iteration cycles of the outer loop will remain distinct inside the inner loop.

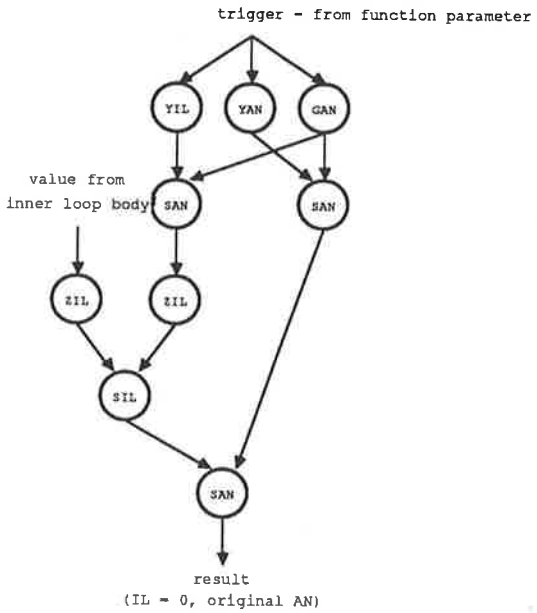


Figure 5.3 Setting activation names of inner loop tokens (Phase 1)

As with the iteration levels, the activation names of all resultant values of the inner loop must be reset to their original value before entering the inner loop. The difference between the activation name and the iteration level is that each activation name is distinct throughout the entire

execution life of the program. Figure 5.3 illustrates the dataflow subgraph required to reset the labels of data tokens leaving an inner loop, making use of the activation name to ensure uniqueness between data tokens in the inner loop. This ensures that all activations of inner loops can be executed concurrently. Again, only those nodes which do not have a corresponding triple in the TG have their destination addresses inserted at this time.

5.2.5 Code generation in forall expressions

Proliferation of all identifiers used in **forall** expressions is required to obtain a copy of their data values over the entire range of index value names; this is achieved by the use of proliferate nodes. These nodes can be generated from the triples of the same name which are used as an interface between identifiers defined outside the scope of a **forall** expression and their uses inside the expression, as explained in Chapter 4.

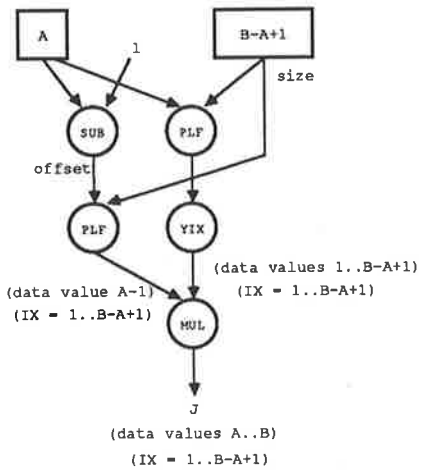


Figure 5.4 Dataflow subgraph for referencing Index Value name J in [A,B]

When an index value (IV) name in VAL–S is referenced inside the body of a forall expression, the values associated with the IV name must be

created from the corresponding INDVAL triple, with the correct index label values. This is done by proliferating one of the IV bounds over the range defined in the forall expression, to yield the index label values; then the offset, also proliferated, is added as shown in Figure 5.4. The corresponding source code is given in Table 4.6 and its triple graph in Figure 4.8. Note that the users of the IV name are stored in the user list of the corresponding INDVAL triple. Only the four destination addresses linking nodes in the subgraph are inserted at this time.

Each EVAL triple created from a **forall** expression also needs further processing. The triple takes an array as input and combines the elements together using a simple arithmetic operation to produce a single value. The dataflow subgraph for the example given in Figure 4.8 is shown in Figure 5.5.

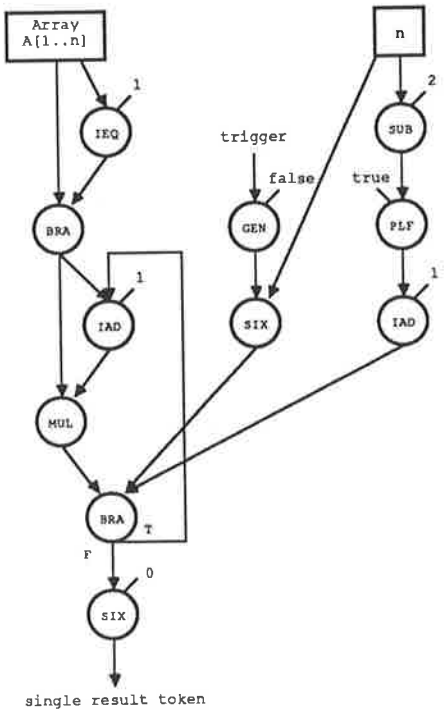


Figure 5.5 Dataflow subgraph corresponding to an EVAL triple

The graph shows an array A[1,n] whose elements are multiplied together

sequentially by the MUL instruction. The first element A[1] is extracted from the array using the initial BRA instruction and has its index tag incremented by one in the IAD instruction. This makes its index tag equal to that of A[2] so the elements can be multiplied together. The right hand side of the graph generates the boolean values of true for index values *2..n–1* and false for index value *n*. These boolean values are used in the second BRA node which circulates the array elements back to the IAD instruction until all have been combined in the MUL instruction, whence the result has its index tag set to 0. Again, only nodes without a corresponding triple in the TG have their destination addresses inserted at this time.

5.2.6 Code generation for function calls and definitions

The dataflow code required to interface a function definition with the corresponding calls is generated from the PAR and RET triples associated with the function definition, and the CALL and CRET triples associated with the function calls, which were described in Chapter 4. For each function call a new activation name (see Chapter 2) must be generated at execution time and used to tag actual parameter tokens and the result tokens of the function call to distinguish activations of the same function and allow their concurrent execution. Figure 5.6 shows the dataflow operations required to tag each actual parameter before it can be used in the function body. The link from the GAN node to the SAN node is inserted here.

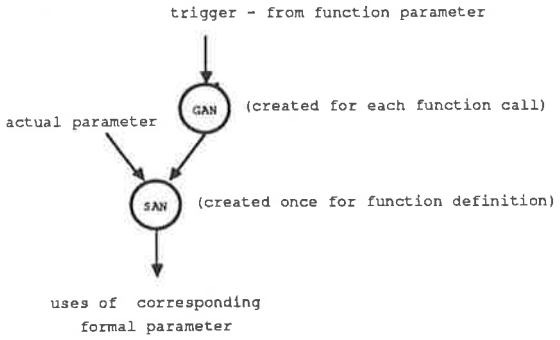


Figure 5.6 Parameter correspondence and tagging of actual parameters

The activation tag is created by a Generate Activation Name (GAN) node which requires an input trigger to initiate the execution of the instruction. In this implementation, the last actual parameter in the function call is used for this purpose. Each actual parameter stored in the operand list of the CALL triple must have its old activation name reset to this new value using a Set Activation Name (SAN) node. Note that there is one SAN node for each formal parameter; the unique activation name associated with any one function call enables the one SAN node to be used for each parameter, for all calls to that function. In the function definition, to set the new activation names, a SAN node is created for each PAR triple and linked to the GAN node as shown in Figure 5.6. The actual parameter nodes are found from the corresponding triples in the operand list of the CALL triple. Each actual parameter is linked to the corresponding SAN node as shown again in Figure 5.6. The destination addresses of output from the SAN node are taken from the PAR triple users, which represent the uses of the formal parameter inside the function body.

If one of the parameters is an array then the activation name needs to be set for each element of the array. This is done by creating a PLF node to proliferate the value of the new activation name over the range of the

array to provide a new activation name with an index value to match that of each array element.

Figure 5.7 gives the dataflow code required to return each result of a function call. As a particular call to one function must only send results back to that function activation and not to all calls of the same function, the return link must be dynamically created. This is achieved on the MDFM by means of two special dataflow nodes, the SLK and RET nodes. The Set LinK (SLK) node strips the entire label field from its left input and combines it with a destination address specified by its right input to form a so-called *context* [Kirkham, 1984].

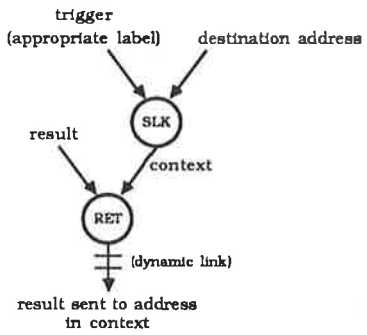


Figure 5.7 Returning results of function calls

In generating the SLK node, the destination address (of the RET node) is obtained from the user list of the corresponding CRET triple created for each result returned by the function. The context is sent to the RET node corresponding to the RET triple created for the function definition; the RET node receives the result of the function as its left operand and sends the result token to the destination address specified in the context of its right operand, with the label set to the specification in the context. Note that the only destination address inserted into the subgraph at this time is the link between the SLK and RET nodes. Again if an array is produced as the result of a function, the context must be proliferated over the range

of the array to provide a context for each array element. If a tuple of results is returned by the function, each result requires its own SLK–RET node pair.

5.2.7 Code generation for array operations

The dataflow code generated for array select and append operations using the more complex instructions introduced in Chapter 2 is shown in Figure 5.8. SELect and APPend triples contain the necessary information to generate the dataflow subgraphs corresponding to each of these VAL–S operations. Either the SEL or EXT dataflow instruction could be used for an array selection by using the right output, but for array appends, the left output of the EXT instruction is used when an existing array element is replaced by a new value, and the SEL instruction is used to add a value with a new index. The corresponding subgraph for the EXT dataflow instruction using primitive MDFM instructions is shown in Figure 2.3. Dataflow code is not generated for ARR triples, as these are special triples introduced for reasons explained in Chapter 4; they are removed from triple user lists in the second step of the dataflow code generation phase.

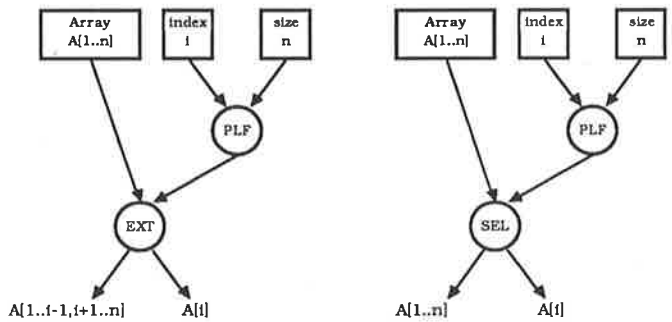


Figure 5.8 Dataflow subgraphs for array selections and appends

Notice that the required index value *i* still requires proliferation over the range of the array, the range being stored in the operand list of the SEL or

EXT triple. The only destination addresses inserted at this time are those of the PLF triples.

5.3 Generation of dataflow links and duplication nodes

After the generation of dataflow nodes has been completed, the dataflow graph requires the insertion of destination addresses or links in the dataflow instructions for all nodes which have a corresponding triple in the triple graph and have not yet been inserted. At the same time it is necessary to create duplicate nodes to produce copies of data tokens which are required at execution time. At this stage all relevant triples will contain a pointer to the address of the corresponding dataflow node to enable the triple addresses in user lists to be converted to dataflow node addresses in the destination fields of the dataflow instructions.

DUP nodes are created after all other nodes in the dataflow graph. Their creation is solely determined by the number of users of each of the other nodes. As described in Chapter 2, instructions for the MDFM produce either one or two output tokens, whereas it is common for more than two output uses to be made of the result of an operation. Thus to generate several copies of the same token, DUPlicate nodes, as described in Chapter 2, are created. For example, if an identifier is used four times, two duplicate nodes would be required, each duplicating one of the maximum two output tokens from the subgraph generating the value for the identifier. In general, the number of duplicate nodes required to copy the outputs from any one node equals the number of users minus two. The exceptions to this are true and false outputs of BRA nodes and outputs from SLK nodes which produce one output token only, so for these cases, the number of duplicate nodes required equals the number of uses minus one. This also applies to the program inputs, the priming tokens,

which are generated from the corresponding PTOK triples.

Note that DUP nodes cannot be shared between users of different nodes and are therefore not considered in the optimizations described in Chapter 6. For this reason no corresponding triples are created, rather these DUP nodes are created directly from user lists of triples in the TG.

Triples corresponding to dataflow nodes are scanned in sequence; for a particular triple with no duplication requirements, the destination addresses in the corresponding dataflow instruction are inserted by copying across the dataflow node addresses corresponding to the triples in the user list of the triple under consideration.

When duplication is required, a subgraph of one or more duplicate nodes is generated. This subgraph could take several different forms – a balanced binary tree was selected because it has the effect of minimizing the number of steps in the subgraph of duplicate nodes. However, it is quite possible to give simple counter examples when this pattern is not optimal, because the depth of the nodes receiving the duplicated data will vary. Some simple trials comparing a left-hand chain of duplicate nodes with the binary tree were carried out in a preliminary investigation [Jones, Kidman & Morello, 1985]. In this second method, all right outputs are connected to users in sequence and duplicate nodes are attached to the left outputs. Thus users at the top of the chain of duplicate nodes would receive their inputs ahead of users at the bottom, so if users at the top were involved in more complex operations than those at the bottom, this would enhance performance in a particular graph. These trials on rather small test programs showed that the binary tree was not always better than the chain. However, it is expected that on average the binary tree pattern would be preferable, especially when many copies of one data

value are required. While it may be possible to rearrange the allocation of output by hand in order to improve the performance of a small individual graph, as was mentioned also by Gurd, Glauert & Kirkham [1981], it seems unlikely that this could ever be done automatically in a compiler. Therefore, in our implementation, duplicate nodes are generated as a binary tree and the outputs allocated to users sequentially in list order.

6. COMMON SUBEXPRESSION DETECTION

6.1 Introduction

The nature of single assignment functional languages, such as VAL–S, which are free from side effects, appears ideal for the application of common subexpression detection (CSD) because, in the one scope, all occurrences of the same expression represent the same value. Thus it should be possible to eliminate multiple evaluations of different occurrences of the same subexpressions, each being evaluated once only, and its value passed to other usages in the program.

Although common subexpressions (CSEs) always yield the same values in different parts of a VAL–S program, when they occur inside the arms of a conditional expression their evaluation is dependent upon the boolean test expressions governing the execution of the arms of the conditional. The conditional execution of an expression is modelled during parsing by the execution condition (EC) of triples, defined in Section 4.3.3. In general, two identical subexpressions with different ECs cannot be implemented in one evaluation, as they require evaluation under different conditions.

One way to deal with this on a dataflow machine, might be to compute the shared data value regardless of the conditions that prevail and to pass the result through appropriate switches to all uses in the program. The problem with this is that if none of the conditions of execution are met, the result of the expression is not required at all and the computation would have been performed unnecessarily; in the worst case this could lead to a non-terminating computation or to the introduction of execution time errors that were not inherent in the original source program (see example in Chapter 1).

There are, however, occurrences of CSEs in different arms of conditional expressions which can share the one computed data value on a dataflow machine without introducing the above mentioned potential errors; an obvious example is when one occurrence of a CSE involves unconditional evaluation, for this value can then be switched for any other occurrences. The method described in the following sections involves a thorough program analysis which enables the sharing of certain CSEs without introducing unnecessary computation or the generation of unused tokens. The ECs of such shareable CSEs are said to be **compatible**.

To enable the detection of CSEs, expressions must be stored in a representation which allows occurrences of the same expression to be recognised. The TG described in Chapter 4 was designed for this purpose, because the dataflow graph (for the MDFM) itself is not appropriate (see Section 1.1.3). The optimizations discussed in this chapter are performed on the fly, during parsing and TG generation in the first phase of the compilation (see Figure 4.1). Occurrences of the same subexpression which can be shared are represented by one triple in the TG.

6.2 Detection and sharing of CSEs

A subexpression is termed a CSE if there is more than one textual occurrence of the same subexpression. During parsing, before a new triple is entered, a search is made of the TG to determine whether that triple is already a node in the graph. If so, this would indicate that the corresponding subexpression has been encountered previously and so a CSE has been found. A new triple may not be required, in which case all references can be made to the existing triple.

The process of determining whether a new triple must be entered involves

three steps. The first is a search of the TG to find a triple with the same operation. If found, the next step is to compare the operands. If these are the same, a CSE has been found and the final step determines whether the two occurrences of the subexpression in the program have compatible ECs and so can share the one triple corresponding to the one **computed** data value. These three steps apply to all subexpressions including function calls and array selections; only conditional subexpressions constitute a special case and are considered separately in Section 6.4. No attempt has been made to find common loop subexpressions.

A subexpression inside a loop is not shared with one outside; the possibility of moving the evaluation of loop invariant subexpressions outside of the loop is discussed in Section 8.4. However, CSEs inside a loop may be shared.

Triples in the TG are assigned an EC when they are created. For each expression a Current Execution Condition (CEC) is maintained during parsing; this is a conjunction of the conditions under which the current expression will be evaluated. When the same subexpression (or CSE) is found in the TG, its EC in the TG must be compatible with the CEC for the CSE to be shared, in which case a new triple is not created. Note that the CEC associated with the latter use will be preserved in the EC of its user triple.

In the event that the two subexpressions have ECs which disallow sharing, then an identical triple representing the new subexpression must be created with an EC equal to the CEC. Links between equivalent triples with incompatible ECs are set up in the TG; this not only simplifies the searching algorithm defined in Section 6.2.2, but is also useful for other purposes. Note that it is not strictly necessary to duplicate

the incompatible subexpressions providing different users are distinguished – this was a convenient implementation technique.

The next section specifies the criteria for sharing CSEs by defining a set of selection rules based on comparing their ECs. In order to explain these rules, some special terminology is now introduced.

An EC is a conjunction such as A ^ ~B ^ C, where the right most conjunct is the latest encountered, or innermost condition in a VAL–S nested conditional construct. C is referred to as the **top** of the EC, and any left part of the EC such as A or A ^ ~B, is referred to as a **root**; ~B ^ C is referred to as the **top part**. One EC is **at the root** of another when it occurs at the beginning of the second, for example A ^ ~B is at the root of A ^ ~B ^ C; such ECs are compatible because a CSE with the root EC can be evaluated once and the result switched through the top part for the second occurrence. Two CSEs with ECs which have a **common root** with different **top parts**, for example, B1 ^ B2 and B1 ^ B3, cannot be shared by evaluating the CSE once under the condition B1 with further switching of the result, because if the condition B1 ^ ~B2 ^ ~B3 is satisfied, then the value of the CSE is not needed and should not be evaluated in a clean implementation[1]. Note that ECs with different roots but the same **top part** cannot occur in the implementation.

Under certain conditions the EC of a CSE in the TG must be **reduced** to enable the value of the CSE to be shared within a program. Reducing the EC of a triple means that one or more conjuncts are removed from the **top part** of the EC, the ECs of its operands also being reduced as necessary.

[1]Refer to Chapter 1 for the definition of a clean implementation.

6.3 Selection rules defining compatible ECs

As defined above, two ECs are termed compatible if the CSEs with which they are associated may share the one evaluation. Such sharing is implemented in the TG by the use of a common triple for the different occurrences of a CSE.

All selection rules discussed in this section are based upon the assumptions that in execution on the MDFM, no unnecessary computations are performed and that no redundant tokens remain after the completion of a computation. In each rule, two ECs are compared, that of the existing triple (TEC), and that of the current expression (CEC); the representation of the TG allows this to be done by comparing two pointers.

The first three rules state when a newly found subexpression is able to share the value of a previously encountered occurrence of the same subexpression, by setting up a reference to the existing triple without modifying it. Rules (4) – (6) allow sharing of existing triples provided their ECs are reduced to that of the new expression. The first six rules apply when two subexpressions are in the same conditional expression or when one or both are unconditional; rule (7) relates to subexpressions in two different conditionals. Rule (8) given in the next section, applies specifically to entire conditional subexpressions.

(1) TEC = CEC

This states that if the EC of the triple representing a CSE is the same as the CEC, then this existing triple may be shared with the current subexpression. This situation arises when both CSEs occur in the same arm of a conditional expression or when both subexpressions occur

outside of loops and conditionals.

(2) TEC = True

If an existing triple has an EC of true, that is, the corresponding expression does not appear inside a conditional expression and its evaluation is unconditional, it can be shared by any other occurrence of the same subexpression, regardless of its switching requirements specified in the CEC.

(3) TEC is at the root of **CEC**

A triple whose TEC is at the root of the CEC can be shared by the new subexpression with further switching. For example, a triple with CEC B1 ^ B2 can share an earlier occurrence of the same subexpression with TEC B1 by switching through B2.

(4) CEC = True

If the current subexpression is not inside a conditional expression, the TEC of an existing triple corresponding to the same subexpression but inside a conditional, can be reduced to true so that it can be used directly as the current subexpression; uses of the first occurrence of the subexpression would then require switching in accordance with the original TEC.

(5) CEC is at the root of **TEC**

The CEC is at the root of the TEC of the existing triple, so the TEC is reduced until it is equal to the CEC; earlier uses of the CSE will then require switching through the top part of the original TEC.

(6) top(**CEC**) = ~top(**TEC**)

This rule identifies two CSEs with ECs which differ only in that the top part of one EC is the complement of the top part of the other, which means

that the CSEs occur in both the then and the else parts of the same conditional expression. Hence the last condition has no effect in determining whether the subexpression is evaluated, so it can be evaluated with this last condition removed from the EC, that is the TEC is reduced. Each of the two usages of the subexpression must then have the result of the subexpression evaluation switched through either this last condition or its complement.

As only one of the CSEs would ever have been evaluated originally, this rule has the effect of reducing the size of the static dataflow graph, rather than reducing the number of times a subexpression will be evaluated. This rule also covers the case where a subexpression appears in each arm of a string of nested conditionals, or equivalently, in each arm of a conditional with many elseif arms. For example, a subexpression appearing four times in many elseif arms would have occurrences with ECs of A, ~A ^ B, ~A ^ ~B ^ C, ~A ^ ~B ^ ~C. This rule would be applied directly to the last two occurrences, reducing the EC to ~A ^ ~B. The rule can then be applied again with the second occurrence, further reducing the EC. This process is repeated until the EC is reduced to true and all occurrences of the subexpression can then share the same triple.

(7) (a) EC (TEC) = EC (CEC)

and

(b) Operand (top(**TEC**)) = Operand (top(**CEC**))

This rule is designed to enable sharing of CSEs appearing within two different conditional expressions which share a boolean condition. To distinguish between different conditional expressions, COND triples sharing the same boolean expressions are kept distinct. Thus CSEs appearing within different conditional expressions, where the boolean conditions specified in the conditionals are the same, would not be shared

by the previous rules, although the boolean expressions themselves are shared. Rule (7) enables sharing by detecting (a) that two conditionals have equal ECs, and (b) that the respective COND triples share the same boolean expression; note that (a) cannot be extended to the condition that TEC and CEC are compatible.

6.4 Detection and sharing of common conditional subexpressions

Although common conditional subexpressions are not likely to occur often in practice, the analysis has been designed to include them. A conditional subexpression is represented in the TG by the IFF triple, whose operands are THEN and ELSE triples, and whose EC is a pointer to a COND triple, the latter three triples not being shareable. For two conditional subexpressions to be recognised as identical, the operation and operand equality tests described in Section 6.2 are clearly inapplicable. Each conditional expression consists of three subexpressions; for two identical conditional expressions to be shareable, each of the corresponding component subexpressions must be shareable. Consider Table 6.1 showing two common conditional subexpressions beginning with the boolean condition B2:

```
if B1 then                    if B2 then
  if B2 then                    E1
     E1                       else
  else                          E2
     E2                       endif
  endif
else
  E3
endif
```

Table 6.1 Common conditional subexpressions

Rule (5) allows the one triple for the boolean expression *B2* to be shared by the two COND triples for the conditional expressions. However, the two occurrences of the subexpression *E1* (and those for *E2*) cannot be shared by the above rules. Thus during parsing, two copies of the triples representing the expressions *E1* and *E2* are created. However, it is clear that the whole conditional subexpression with the boolean condition B2 requires only one computed value, the first subexpression requiring this value to be switched through the condition *B1*. This sharing of conditional expressions can only be detected after the entire conditional expression has been parsed and the triples representing its arms have been created. When two IFF triples are recognised as shareable, it will be necessary to remove the latest COND, THEN and ELSE triples and their operands from the TG. In this situation where the CEC refers to the new IFF triple which is not to be created and the TEC to an IFF triple in the TG, the following rule specifies the criteria for sharing the corresponding conditional subexpressions:

(8) **(a)** **TEC** and **CEC** (of the two IFF triples) are compatible

(b) Operands (top (EC (**THEN** triples))) are equal

(c) Operands (**THEN** triples) are CSEs (Similarly for **ELSE** triples.)

(a) states that the ECs of the two IFF triples are shareable by rules (1) to (6). (b) states that both conditionals share the same boolean condition. Note that this implies that the ECs of the two COND and of the two IFF triples are compatible by the earlier selection rules. (c) states that the two THEN triples have operands which are occurrences of the same subexpression, but do not necessarily have to reference the same occurrence.

6.5 Searching the triple graph for CSEs

The implementation described in Chapters 4 and 5 enters triples in the TG in the order of their creation. All searches of the TG for CSEs are made in such a way to ensure that triples are encountered in this order. The following techniques have been employed which are designed to minimize the overheads of searching.

(1) Each VAL–S function only has access to its arguments and locally declared identifiers, as there are no globally defined identifiers which may be imported from any enclosing function. Thus the search for CSEs in the TG can be restricted to the triples generated from the function currently being parsed.

(2) When creation of a new triple is under consideration, searching the TG for a similar triple is only required when all of the new triple's operands are shared CSEs (except in the case of conditional subexpressions).

(3) Multiple occurrences of the same triple are linked, these multiple occurrences existing because of incompatible ECs, as defined by the selection rules (1)–(7). This is implemented by storing a linked list of pointers to similar triples from the first occurrence of each triple in the TG. When searching the TG serially for a CSE, this will be the first triple representing the CSE to be found; hence all similar triples will be accessible directly from this linked list.

After determining whether a CSE may exist from (2), and if so, the area of the TG to be searched, a simple search is carried out comparing operators and operands of triples with that of the current subexpression, using the selection rules to determine the compatibility of ECs (for

conditionals a modified procedure is used). This process may produce four outcomes:

(a) The subexpression is not a CSE in which case a new triple is created.

(b) The subexpression is a CSE with a compatible EC so the existing triple may be shared directly (rules (1), (2), (3), (7)).

(c) The subexpression is a CSE but its EC is not compatible with those of existing CSEs, so a new occurrence of the corresponding triple must be created. This is linked to the first occurrence of the triple as described above. Notice that in determining that the EC is incompatible, it must be compared with the EC of each existing triple corresponding to the CSE in the TG.

(d) The last outcome is the most complicated and requires the most processing. This occurs when a CSE is detected but the corresponding triple requires its EC to be reduced as a result of selection rules (4) to (6). Comparisons with other instances of the triple may find ECs that were previously incompatible and are now compatible, thus enabling further sharing of the value computed by the one CSE. If indeed an existing similar triple now has an EC compatible with that of the triple whose EC has been reduced, this existing triple is made redundant by changing its operation to DEL (for delete) and inserting a pointer to the address of the triple replacing it. Later when users lists are inserted, deleted triples can be removed and any operands which reference a deleted triple are changed to the triple replacing it, as specified by this pointer. Further comparisons between similar triples are required if, in the process of combining and deleting existing triples, an EC is further reduced. Extensive use of recursion was made in implementing these procedures.

6.6 Sharing switching nodes

Common SWitching node Detection (CSWD) is a similar optimization to CSD except that it is applied to BRANCH triples only. All uses of an identifier or an expression in the arms of the one conditional expression can be switched through the same BRANCH triple to both the true and the false arms. Thus in the BRANCH triple generation step described in Section 4.3, CSWD is carried out to minimize the number of BRANCH triples created.

Although all BRANCH triples are created and stored sequentially in the TG, which limits the search space for common BRANCH triples anyway, a more direct search method is used. Before a new BRANCH triple is created, a search is made, as in CSD, to determine whether the triple already exists which switches the identifier or expression through the appropriate condition. Rather than searching the TG, it is more efficient to search the user list of the COND triple generating the boolean valued condition operand of the BRANCH triple. This user list stores the triple addresses of all BRANCH triples which reference the COND triple as an operand. In the event that the associated boolean expression is shared by other COND triples, their user lists can be searched also. Compared with searching a segment of the TG directly for common switching nodes, in the worst case, this list is the same size as the search space of the TG but in practice is considerably smaller.

6.7 Compiler structure

The compiler has been structured in such a way that both CSD and CSWD are separate optimizations which can be turned on and off by flag settings in the compiler. This has allowed the separate analysis of results from these optimizations as shown in Appendix D. Some measurements were also made of compilation times; the results are reported in the next chapter.

7. TEST PROGRAMS AND RESULTS

7.1 Testing methodology

Appendix C contains VAL–S programs which have been compiled into an intermediate form and then translated into dataflow machine code using the methods described in Chapters 4, 5 and 6. This dataflow code was run on a simulator [Morello, 1982; Kidman & Morello, 1984] for the University of Manchester dataflow machine to produce the results shown in Appendix D. The simulator has been described in Chapter 2. The optimizations tested here are Common SWitching (or branch) node Detection (CSWD) and shareable Common Subexpression Detection (CSD), the latter excluding the former. Programs were compiled using different combinations of these optimizations. The first combination was with all optimizations switched off in order to provide a basis for comparison of the effectiveness of the subsequent optimizations. The second combination applied only CSD and was performed only for the larger examples. The third combination applied only CSWD while the fourth applied both CSWD and CSD. The code generated from each compilation was run several times through the simulator with different numbers of processors. In some cases, to provide further results, runs were carried out where only the input data was changed.

Appendix D gives the results of the runs in detail. The size of the generated static dataflow graph is indicated by giving the total number of nodes, with a breakdown giving the number of switching and duplicate nodes. Execution speed is measured by the number of execution steps. Figures relating to the token queue show the total number of tokens generated and the average number per step, together with the maximum token queue size. For the matching store, the maximum and total

number of tokens stored are given, followed by the proportion of executed instructions which receive only one input token. A blank space in the table indicates a result equal to that in the row above.

Finally, the maximum and average parallelism are given in terms of the number of instructions executed in parallel. The maximum parallelism gives the maximum number of instructions executed during the same time step; the average parallelism gives the average number of instructions executed per step, obtained by dividing the total number of executed instructions by the number of time steps with "infinite" processors, that is the minimum possible number of time steps. Note that the average parallelism of a dataflow program defined in this way [Gurd, Kirkham & Watson, 1985] for an idealised simulated machine is a property of the program and is not dependent upon the number of processors. To obtain these measurements it is sufficient to execute the program both with one processor and with a number of processors greater than the maximum parallelism.

7.2 Examples

The first four example programs while trivial, are included to highlight some of the basic effects in an environment which allows simple analysis. The other examples show the effects of the optimizations in programs of increasing complexity.

Examples 1–3

The first example shows three different versions of a program to calculate integer factorials, the different versions using iterative, recursive and doubly recursive algorithms respectively. Figure 7.1 shows the dataflow graph of Example 1B without and with optimization.

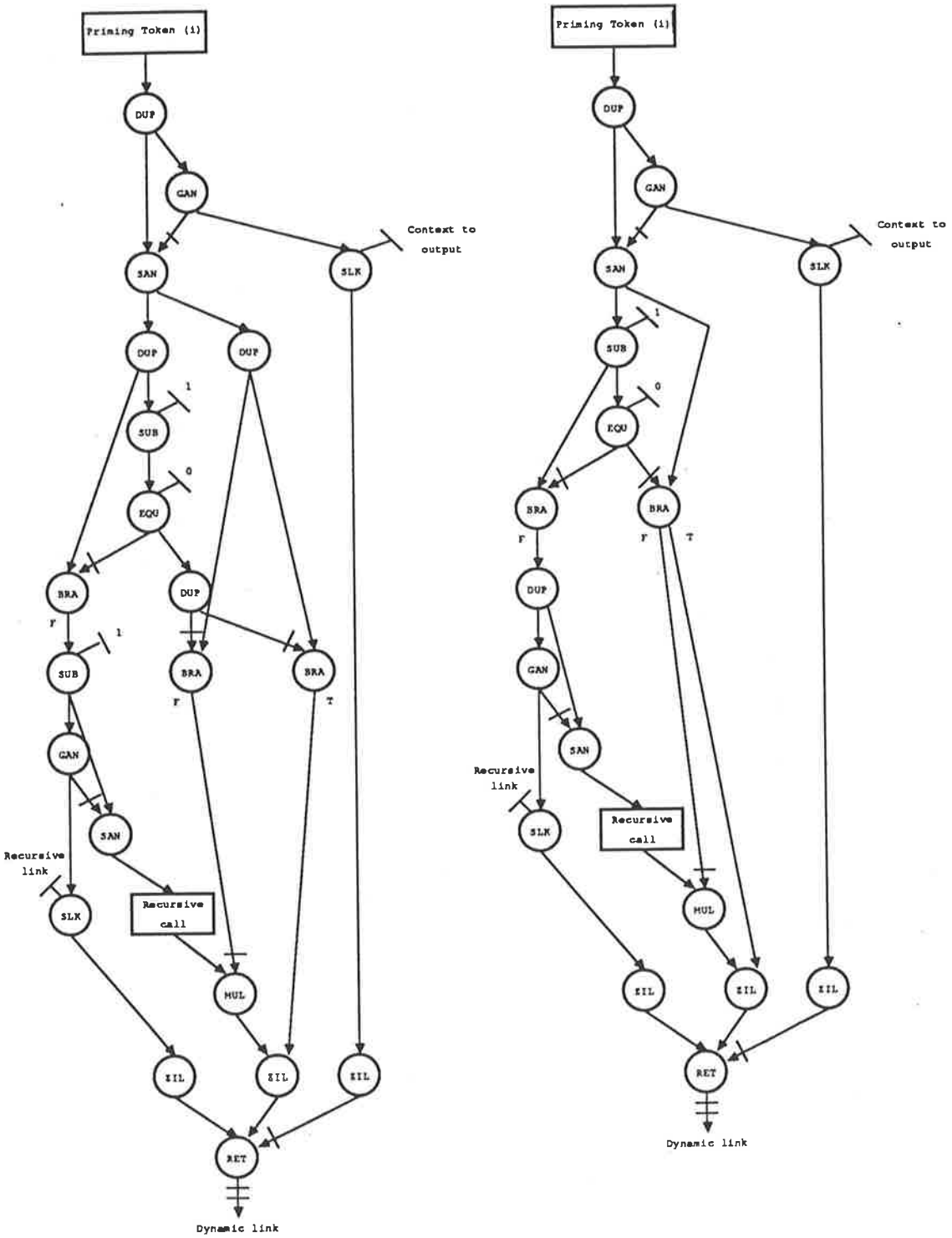


Figure 7.1 Dataflow graph corresponding to Example 1B

Example 2 uses iteration to return the result of a real value raised to an integral power.

Example 3 calculates the greatest common divisor of two integers, again using an iterative method, which defines two loop variables a and b and assigns them to the input parameters x and y. On each iteration, loop variables are compared, with the larger of the two being reduced by their difference, until they become equal. The algorithm has been deliberately coded so that the expression $a-b$ occurs four times, to demonstrate that CSD will eliminate three computations of this expression, and each occurrence of $a-b$ in the program will share the value from the one calculation.

Example 4

This program accepts an integer parameter n (greater than one) and calculates the nth number in the Fibonacci sequence. Another iterative algorithm is used here to find the Fibonacci number by computing each number in the sequence starting from the second.

Example 5

A simple algorithmic function is given in this example which sums the elements of an array. An array *A* is defined and initialized using a forall expression. The function *sum* is then called passing this array and its array bounds as parameters. The CSE *(high–low+1) / 2* within the function *sum* can be removed to save three arithmetic operations within the static dataflow graph.

Example 6

This example is included to show the effect of optimizing a simple program containing a common conditional expression. The example

given is contrived to contain a common conditional expression which calculates the absolute value of $x-y$.

Example 7

This next example calculates the positive root of a quadratic equation, if it exists. Two internal functions are defined to calculate the absolute value and the square root of a real parameter. Three versions of the example are given. The first uses nested conditional expressions which allows the expression $b*b-4*a*c$ to be recognised as a CSE, the second version defines $b*b-4*a*c$ in a let expression and the third re-orders the arms of the conditional expression in the main function body. The aim here is to again observe whether the optimizations used were effective, but also to observe any change in results due to the different but semantically equivalent coding.

Example 8

The well known quicksort algorithm [Wirth, 1976] has been translated into VAL-S in this example; it involves a large amount of array manipulation. An array of length n is defined in the let expression in the body of the main function and this is then sorted by calling the inner function *quick*. Three empty arrays are defined inside *quick* and the initial array is split into these in such a way that *middle* contains all the elements equal to *B[low]*, *left* contains all the elements less than *B[low]*, and *right* contains all elements greater than *B[low]*. *Quick* is then called recursively with the arrays *left* and *right* and the results are concatenated together to give the initial array sorted in ascending order.

The program was run three times using an array of size ten as a test example. First the elements of the array were provided in ascending order so that no sorting was required, then the elements were provided in

descending order to maximize the amount of sorting required and finally they were given in random order.

Example 9

The next example is another array sorting program based upon the insertion sort algorithm as defined in [Wirth, 1976]. The algorithm here uses an iterative process rather than recursive function calls to sort the array. Starting at the beginning of the array, each element is considered in array index order and inserted into its correct position amongst the elements already considered so that the array segment is in ascending order. When completed, this new array is then returned as the result of sorting the original array. Again three versions of the program were created using a given test array of size ten in ascending, descending and random orders.

Example 10

This next example [Ackerman, 1984] is a program which solves a tridiagonal system of equations where the system is represented by an *NxN* matrix where *N* must be *2**k–1*, for *k* greater than *2*. Four vectors are defined; *D[1..N]* contains the diagonal elements of the matrix, *A[1..N]* contains the elements above the diagonal, *B[1..N]* contains the elements directly below the diagonal, and *R[1..N]* contains the right hand side of the system. Note that all vectors must be defined from *1..N* to avoid problems at the end points in the algorithm; also each vector is considered to be cyclic or wrap around. Three versions of the program are presented, the difference being that the first uses several let statements whereas the second and third are rearranged to demonstrate that CSD is equally efficient. The second version removes all let statements, causing restructuring of the program in some instances; the third version only removes let statements when restructuring is not required.

Example 11

The periodic cyclic reduction algorithm coded in this example was again obtained from another paper [Ackerman, 1984]. The program solves certain second order differential equations in one dimension with periodic boundary conditions. Note that the input data used was created to ensure that the equations were solvable and the results easily verified.

Example 12

The sequencing problem [Gillett, 1979] addressed in this example considers n jobs waiting to be processed on two machines so that the total elapsed time from the start of the first job on the first machine to the completion of the nth job on the second machine is minimized. The processing times of each job on each machine are known and each job is processed on machine one before it is processed on machine two. Jobs remain in strict order during processing so that when a job finishes on machine one it must wait until the previous job has completed on machine two before it may commence.

The first part of this example accepts ten jobs and determines the sequence in which they must be processed through the two machines in order to determine the shortest total elapsed time. The process of determining this sequence consists of placing all the jobs with the shortest processing time on machine one at the beginning of the sequence and those with the shortest processing time on machine two at the end of the sequence. By applying this algorithm the sequence of processing for the ten jobs given in the example is determined.

Example 13

An extension to example 12 is given here which uses the scheduling sequence of jobs calculated in example 12 to determine the total elapsed

time and the idle times for each machine. The same ten jobs are again defined here, along with the order calculated to minimize the total elapsed time.

Example 14

This example deals with a queueing problem in which customers are waiting for a service [Gillett, 1979]. The program is based on an infinite-queue-infinite-source, multiple-server model and assumes that service is provided on a first-come, first-served (FIFO) basis, that customers arrive at random but at a constant average rate, and that the queueing system is in a steady state. Parameters for the example include *s,* the number of servers, *mu*, the rate at which a server provides service, and *lambda*, the average arrival rate of customers. It is assumed that lambda is less than *mu*s*.

The results produced by the example are *l*, the average number of customers in the system, *lq*, the average number of customers in the queue, *w*, the average time a customer spends in the system, *wq*, the average time a customer spends in the queue, *pn*, the probability of *n* customers being in the system at any point in time, *pzero*, the probability that no customers are in the system, and *ps*, the probability of at least *s* customers in the system.

The value of *pzero* is used in the calculation of each of the other results and as such appears in many places. The values of *l* and *lq* are used by both *w* and *wq*.

The mathematical calculations involved in computing these results are probably sufficiently complex for the programmer to calculate each of these results in its own internal function or in a let statement. Again three versions of this example are presented, the first simply calculates

the results disregarding the relationships between them, the second is an elegant modular program which functionalizes each computed result, and the third version packages the calculation of each of the results into a let statement, which in effect means that the programmer has factored out the CSEs.

7.3 Measurements of compilation times

In order to determine the overheads in performing these optimizations, the internal clock on the VAX 11/780 was used to measure the execution time of both phases of the compilation process with the optimizations turned on and then with them turned off. Table 7.1 gives the figures recorded for the compilation process of some of the test programs given in Appendix C.

Although the compilation times given in Table 7.1 depend somewhat upon other usage of the VAX machine, from the figures produced, it is apparent that the time involved in the creation and linking of extra triples and dataflow nodes required in the unoptimized code far exceeds the search time required in CSWD and CSD which reduces the number of triples and nodes required. Multi-pass compilation, as used in this compiler, is generally more expensive than single pass compilation but it enables more efficient target code to be generated.

Example Program	Optimization Setting	Compilation Time (sec)
8A	None	22.67
	CSWD	9.87
	CSWD & CSD	9.57
9A	None	5.77
	CSWD	3.94
	CSWD & CSD	4.06
10A	None	26.56
	CSWD	20.39
	CSWD & CSD	20.29
10B	None	29.03
	CSWD	21.09
	CSWD & CSD	20.21
11	None	13.15
	CSWD	11.93
	CSWD & CSD	11.91
12	None	20.82
	CSWD	10.41
	CSWD & CSD	10.30
13	None	18.29
	CSWD	9.85
	CSWD & CSD	6.90
14A	None	16.66
	CSWD	11.71
	CSWD & CSD	7.50

Table 7.1 VAL–S program compilation time (sec)

8. DISCUSSION AND CONCLUSIONS

8.1 Introduction

In this chapter, the significance of the results obtained for the example programs described in the last chapter is discussed critically. Appendix D gives the detailed results obtained from executing the compiled test programs given in Appendix C on the simulator described in Chapter 2. In what follows, these results are first analysed in some depth by examining the effects of the optimizations described in Chapter 6; after this there is a short general description of some further preliminary investigations (on loop invariant removal) which were initiated but not completed for reasons explained. Secondly, the results are compared with other similar recently published work, and finally consideration is given to the conclusions which can be drawn from the investigations described in this thesis. In the sections immediately below, some limitations of the implementation and of the results from the simulated performance figures are considered.

8.1.1 Limitations of the implementation and testing

VAL–S, the subset of VAL defined in Chapter 3 and implemented as part of this project, is quite limited in the complexity of the data structures which it includes. Record and union data types from the original specification of VAL have been omitted, and only one-dimensional arrays have been implemented. One effect of this is to limit the choice of available test programs to those which can be written using the restricted data types defined in VAL–S; this has also made it difficult to define large test programs. Apart from the effects on multi-dimensional arrays, the effects of the optimizations on stream types, as defined in SISAL, would enhance the results of this thesis. Thus the inclusion of more complex

data structures in the language implemented is an obvious extension.

It should also be noted that to obtain efficient performance on the MDFM, an average parallelism (number of executed instructions / number of execution steps) of about 35 is required [Gurd, Kirkham, Watson, 1985]; in the execution of many of the test programs given in Appendix C this figure is not achieved.

A less significant limitation of the compiler concerns the representation of the triple graph. As shown in Table 4.1 the information associated with each triple in the TG is stored in a Pascal record. It was convenient to represent the TG of the entire program as an array of these records; this limits the total size of the TG, as the array bounds must be specified in the Pascal code constituting the compiler. In all examples considered in this thesis, the total size of the TG was well within the limits set in the compiler, but in larger examples, these limits may be reached. One way to overcome this problem might be to represent the TG as a dynamically created data structure such as a Pascal linked list. A preferable approach would be to take advantage of the modularity inherent in languages such as VAL. As this language is free from side effects, each function module could be compiled and translated into the target machine code independently, providing the interface between functions could be handled. In the case of the TG, storage of the PAR and RET triples used to provide the interface between a function definition and its activations would provide all that was needed to perform independent compilations of separate function modules. This would greatly reduce the storage requirements of the array representing the TG, as all that is stored are function interfaces and the subgraph for the function module being compiled. Similar considerations apply to the array used as temporary storage for the dataflow graph. An alternative approach, further

discussed in Section 8.4, would be to design and generate an intermediate language rather than the triple graph structure.

8.1.2 Validity of simulated performance

A further important point of discussion involves the use of the Morello simulator to generate the results given in Appendix D from the execution of the test programs. As explained in Chapter 2, all results obtained are from the simulation of an idealised machine and thus cannot be regarded as an accurate reflection of the corresponding actual performance figures relating to the execution on a real machine. Rather these results are to be interpreted as giving a general indication of performance which does, however, provide some measures for comparison of the execution of different program implementations. This is perhaps analogous to the way theoretical analysis of a conventional program relates to performance on real computers.

Despite its limitations, it should be noted that this kind of simulation has been used a great deal at the University of Manchester [Sargeant, 1981; Gurd & Watson, 1983; Gurd, Kirkham & Watson, 1985] and it has been reported [Gurd, 1985] to relate reasonably well to performance on the real machine.

8.1.3 Generality of the intermediate form

At the end of parsing (the first step of phase one), the triple graph combined with the identifier table is a complete representation of the original source language program. In this form the program remains suitable for execution on any kind of target machine. It is not until step two of phase one, when BRANCH triples and user lists are inserted into the TG, that it becomes specialized for execution on a dataflow machine.

Furthermore, it is not until phase two, when the actual target machine language is generated, that the program is specialized for execution on the MDFM. Thus the front end of the compiler, including the optimizations described in Chapter 6, is largely target language independent. As the compiler uses recursive descent, it is inherently source language dependent, but the principles used in the source language analysis are applicable at least to any LL(1)[1] dataflow language.

8.1.4 Completeness of common subexpression detection and elimination

The only common subexpressions not detected by the techniques described in this thesis are complete loop subexpressions and CSEs where there are occurrences both inside and outside of a loop. The sharing of CSEs where the ECs are **compatible** is otherwise complete. The selection rules given in Chapter 6 define the compatibility of ECs of two CSEs under the requirements for a clean implementation as defined in Chapter 1.

Another optimization, discussed in Section 8.3, is that of loop invariant removal, which consists of removing constant subexpressions from within loops for execution once outside of the loop. This promotion of subexpressions outside of loops may also lead to further possibilities for sharing CSEs. This has not been implemented but is discussed in more detail in Section 8.3. Thus only CSEs all of which appear inside a loop or all of which appear outside of a loop are shared in the current implementation.

8.2 Discussion of results

The following discussion analyses the results shown in Appendix D in order to identify some general conclusions. The size of the static dataflow

[1]See Gries [1971] for the definition of this type of language.

graph is considered by looking at the numbers of dataflow nodes in the target program, and the number of execution steps required to execute a program on the simulator gives a measure of the execution time of a program. The token traffic through the system is measured by the figures given for the token queue and matching store showing the total tokens, maximum tokens and the average calculated as tokens per step. The bypass ratio is a measure of the tokens bypassing the matching store. As a measure of the parallelism of the program the average number of processes executing in one step is calculated. This is calculated by dividing the total number of operations by the number of execution steps with "infinite" processors, namely the minimum number of steps.

Of course, in a dataflow machine, the number of available processors will affect execution time. The results were produced using 1, 10 and 100 processors in most cases. The number of execution steps using one processor gives the total number of single instructions executed, which is used in the calculation of the bypass ratio and average parallelism. Ten processors is considered a realistic number for the current real machine, so these runs are considered in more detail than the others. One hundred processors were used to obtain results in which the maximum number of processors used in any one step did not exceed the number available. This produced the minimum number of steps in which the program could execute and this figure was used to calculate the average parallelism. If more than one hundred processors was required, a run with a maximum of five hundred was carried out.

8.2.1 Static dataflow graph size

The numbers in the column under "dataflow nodes" given in the tables in Appendix D, relate to the size of the dataflow graph generated as target

code by the compiler. The first of these columns gives the total number of nodes in the graph; the subsequent columns give the number of switching nodes and duplicate nodes respectively. Table 8.1 summarises the effects of CSD and CSWD on these figures for the larger test examples numbered from 8 to 14. The first, second and fourth columns of figures, respectively, show the percent *reductions* in the total number of nodes in the dataflow graphs with respect to the unoptimized form when CSD, CSWD and when both CSD and CSWD are applied. The third column shows the further percent reduction with respect to CSWD, when CSD is applied, that is, the percent reduction between columns 4 and 2. Note that as the different versions of examples 8 and 9 differ only in the input array used for sorting, the effects of optimizations on graph sizes in these examples are very similar and have been averaged.

Example	CSD	CSWD	CSD wrt CSWD	CSD & CSWD
8	21	68	13	72
9	27	45	10	51
10A	6	32	4	35
10B	18	35	13	43
10C	19	35	13	44
11	2	19	1	20
12	9	64	4	65
13	57	57	46	77
14A	67	41	62	78
14B	8	27	6	31
14C	26	39	20	51

Table 8.1 Percent decreases in number of dataflow nodes

In every example considered, CSD and CSWD each individually reduced the total number of dataflow nodes and further reductions were usually recorded when both were applied. One way of isolating the separate contribution of CSD to the total effect, is by relating the net effect of CSD

and CSWD to that of CSWD alone; it generally caused a further reduction in the total number of nodes required. However, in most examples, the number of switching nodes increased slightly when CSD was applied. To explain why this occurred, some very simple examples are now considered in more detail.

The three versions of example 1 all calculate the factorial of an integral parameter, but in different ways. The effect of CSWD and CSD on versions A and B was similar, with the same reductions in the number of BRA and DUP nodes and one less arithmetic operation. Although CSD decreased the number of arithmetic operations by one, it also increased the number of BRA nodes by one, thus maintaining the total number of dataflow nodes. This is explained as follows (see also Figure 7.1). The variable *i* in version B occurs three times in the arms of the conditional, so that without optimization three switching nodes are required. With CSWD, only one switching node for *i* is necessary as all three switched uses of *i* can share the one BRA node. However, when CSD is also applied, although the CSE *i–1* is calculated once only, an extra switching node is needed to switch *i–1*. Optimizations performed on version A have a similar effect on the CSE *b–1*.

In this example, and in many others, the number of switching nodes increases when both optimizations are applied compared with CSWD only, because the promotion of the CSE outside of the conditional expression does not reduce the number of inputs to the arms of the conditional. However, in practice, in larger examples, the overall effect on execution time was not found to be counter productive.

The optimizations discussed usually reduced the number of duplicate nodes considerably; in only one small example (2), is the number of

duplicate nodes increased, again due to the input requirements of an extra branch node when CSD is applied. Example 2 is another simple program in which CSWD is effective in reducing the number of BRA nodes and consequently also the number of DUP nodes. The static dataflow graph size is almost halved by this optimization alone. The effect of CSD on the one CSE ($c/2$) is similar to the effects explained in the first two versions of example 1, where the promotion of a CSE outside of a conditional expression leads to an extra BRA node being required, in place of the arithmetic operation ($c/2$) removed in CSD. However, in this example, an extra DUP node is also required to supply the boolean value to this new BRA node, and the total number of nodes is actually increased by one when CSD and CSWD are applied together, compared with CSWD alone.

In summary, CSWD and CSD cause a substantial reduction in the size of the target program. In Sections 8.2.3 and 8.2.4, the effect on its execution characteristics is considered.

8.2.2 Inherent parallelism of target programs

The average parallelism of each test program, calculated as described in Section 8.2, is shown in the last column of results in Appendix D. The average parallelism is an inherent property of each program and the time to execute a program is dependent upon its inherent parallelism. However, in most programs considered, the average parallelism is decreased when CSD and CSWD are applied. Despite this, program execution time is decreased by the optimizations, as discussed in the next section. As already observed, CSD and CSWD lead to a dramatic decrease in graph size which would account for the lower average parallelism. However, in some examples, when CSD and CSWD are applied, the

parallelism increases with respect to that for CSWD alone. The reason for this is not clear but it may be related to an increase in the number of BRANCH nodes in the dataflow graph because of the promotion of CSEs from within conditional expressions as described in the next section.

8.2.3 Execution time

The number of execution steps in the simulated execution is taken as a measure for execution time. Table 8.2, in a similar format to Table 8.1, shows, for the larger examples, the percent reduction in the number of execution steps in the optimized forms compared with the unoptimized form. Note that all of these particular results are obtained for runs using a maximum of ten processors, except the figures in brackets which refer to execution with one processor. The final column shows the percentage reduction in execution steps from applying both CSD and CSWD. The large reduction in execution steps obtained for example 14A is artificial because the program was coded deliberately in an unreadable form without the use of **let** statements, to illustrate the effects of CSD in such a case. Ignoring this example, the average reduction in execution steps is about 20% for CSD alone, 50% for CSWD, and 60% when both CSWD and CSD are applied.

Looking at the results in more detail, it can be seen that CSWD alone produces reductions in the number of execution steps in all examples considered, ranging in the larger programs from 27% to 77%. The effects of CSD on the number of execution steps are much more variable from program to program with a maximum reduction of more than 50%; in a few of the smaller examples, CSD actually increases this value slightly. Again the smaller examples give insight into the reasons for this effect. In example 1, despite the extra switching nodes required, the overall

effect of the CSE promotion is to decrease the number of steps taken to execute the program. In example 2 however, with 2 processors the number of execution steps actually increases when CSD and CSWD are both applied compared with CSWD alone; this is the example discussed earlier in which there is an increase in the number of BRA and DUP nodes by one. Note that the low reduction recorded for CSD in Example 11 is due to very few shareable CSEs occurring in the source code.

The reduction from CSD alone in execution steps with one processor is generally slightly greater than with ten processors. This is because the parallelism of the programs is reduced by CSD, simply because the programs are so much smaller.

Example	CSD	CSWD	CSD wrt CSWD	CSD & CSWD
8A	28 (28)	74	11	77
8B	28 (29)	74	12	77
8C	28 (27)	77	12	80
9A	21 (24)	45	-1	44
9B	33 (35)	46	25	59
9C	31 (33)	45	20	56
10A	6 (6)	36	3	38
10B	21 (21)	40	13	48
10C	21 (21)	40	13	48
11	0.2 (0.5)	27	0	27
12	14 (15)	64	2	64
13	55 (58)	67	24	75
14A	70 (73)	61	56	83
14B	0.6 (0.6)	49	0	49
14C	23 (27)	52	8	56

Table 8.2 Percent decreases in number of execution steps with 10 processors (1 processor)

8.2.4 Token traffic

In a conventional Von Neumann computer, execution time and use of storage are two of the main machine resources used by an executing program. Often in a conventional program a reduction in the execution time can be achieved at the expense of increasing the storage requirements and vice versa. In a real dataflow machine, the number of tokens generated and flowing through the system may affect the execution time; in the MDFM, the matching store, for example, proved to be a bottleneck [Gurd, Kirkham & Watson, 1985], because tokens were held there awaiting matching. Thus in considering the number of simulated execution steps, it is important to consider the effect on token traffic of the optimizations introduced. Note that the term *token traffic* is used in a general way to indicate the generation and flow of tokens through the machine at execution time. Clearly a reduction in token traffic is desirable, other execution characteristics being equal.

As a crude measure, the throughput or token traffic passing through the token queue and matching store are examined. The maximum number of tokens in each of these units at any one time and the total number of tokens passing through them is given for examples executed by the simulator. Again, Table 8.3 gives a summarized form of the more detailed results from Appendix D, for the percentage reductions in the total number and the maximum number of tokens passing through the token queue. Although the absolute figures given in Appendix D are dependent upon the input data used in the simulation (as are the absolute figures for the number of execution steps discussed in the last section), the percent decreases between the different optimization settings remain constant for most programs when the input data is varied. Where the input data does affect the percentage changes, as in the array sorting

programs, results are shown for different sets of input data. Of course the total number of tokens generated during program execution is independent of the number of processing units used.

Example	CSD		CSWD		CSD wrt CSWD		CSD & CSWD	
	total	max	total	max	total	max	total	max
8A	29	34	81	86	14	2	84	87
8B	30	36	81	86	14	5	83	87
8C	28	26	81	87	12	14	84	89
9A	24	33	56	67	-4	-39	54	55
9B	34	32	55	54	21	0	65	54
9C	33	32	55	54	17	0	63	54
10A	6	17	40	33	3	16	41	44
10B	20	30	43	28	13	39	50	57
10C	21	28	43	20	13	43	51	54
11	0	5	31	43	0	0	31	43
12	15	-3	72	67	2	14	73	71
13	57	61	71	70	27	32	79	80
14A	74	74	64	53	67	67	88	85
14B	5	-2	52	47	-4	0	50	47
14C	30	33	61	50	16	17	68	58

Table 8.3 Percent decreases in total and maximum numbers of tokens in the token queue

Changes in total and maximum numbers of tokens through the matching store are similar to the changes in the token queue; this is because the bypass ratio (discussed in the next section) is relatively stable.

As would be expected from observations of graph size and execution steps, the application of CSWD and CSD decreases the total number of tokens generated by a considerable amount; the average reduction for all programs is approximately 63%. While CSWD alone always decreases the total number of tokens generated, when the effect of CSD is considered

with respect to CSWD, in a few cases, a small increase occurrs; 9A, for example, in which an initially sorted array is "sorted" using the quicksort algorithm, the total tokens increases. As explained previously, the promotion of CSEs outside of the boolean condition in which they occur textually can increase the number of BRANCH nodes required. It so happens in this example, that a section of code in which this happens is repeatedly executed for the case where the array is initially in ascending order – the sorting part of the code is never executed. Thus BRANCH nodes absorbed the unused tokens and as more BRANCH nodes are executed, more tokens are generated. In versions B and C, when the sorting part of the code is executed, the effects of eliminating CSEs by promoting them in this fashion, is fully utilized thus decreasing the number of tokens. If the results from Examples 1 and 2 are examined again (see Appendix D), it can be seen that the extra BRANCH nodes inadvertently introduced by CSD, results in extra token generation. The other examples exhibiting small decreases in numbers of tokens when CSD is applied contain very few CSEs.

Similar decreases are apparent in the maximum number of tokens in the token queue, the average reduction being 64% overall. However, the maximum figure for individual programs sometimes decreases by more (for example 8C) and sometimes by less than the total (for example 9C). In one example, 9A, CSD has a seemingly marked counter productive effect on the maximum number of tokens. This is again the example discussed above and the effect is not significant because the maximum figure is low, so a small change represents a relatively large percentage difference.

In some examples, the slight increase in maximum tokens in the token queue when CSD is applied seemed to be due to a chance rearrangement of the dataflow graph in programs where there are few common

subexpressions.

8.2.5 Bypass ratio

The bypass ratio [Gurd, Kirkham & Watson, 1985] is defined as the ratio of those tokens used by single operand instructions (which bypass the matching store in the MDFM), to those which must be paired up in the matching store for use by two operand instructions. As this value shows the proportion of all tokens which bypass the matching store, it gives an indication of the potential bottleneck which can occur at the matching store. Obviously the higher the bypass ratio, the less tokens will enter the matching store. In general, reducing the number of BRANCH nodes will increase the bypass ratio, but reducing the number of duplicate nodes will decrease it; in practice, CSWD tended to increase the bypass ratio and hence improve the program performance in this regard. The effect of CSD on the bypass ratio depends upon the operations involved in the CSEs eliminated. As extra BRANCH nodes are introduced as a result of CSD, the bypass ratio sometimes tended to increase, as highlighted by the results from the smaller examples.

Overall the results from applying the optimizations produced no marked trend in the bypass ratio, for in some examples it was increased, in others decreased. The measure was recorded as low as 0.37 and as high as 0.67, with percentage changes ranging from a decrease of 17% to an increase of 13%.

8.3 Loop Invariants

A related optimization technique considered in this project was that of loop invariant removal. This is the process of finding expressions within VAL–S for-iter loops which evaluate to constant values for each iteration

cycle of the loop. The basic idea is that each such expression is moved outside of the loop, evaluated once, and this value passed into the loop where it must be circulated. As described in Chapter 4, for each expression calculated inside a loop there is considerable overhead associated with circulating the values input to the expression for each iteration cycle. The idea behind loop invariant removal is to avoid the repeated computation of a constant expression and also to reduce the number of values being circulated.

Conditional expressions appearing inside loops can also be invariant. For a conditional expression to be invariant inside a loop, each part of the conditional must itself be invariant. The boolean test condition is handled in the same way as a simple expression. However, the THEN and ELSE arms can only be evaluated outside of the loop if the boolean test expression is also evaluated outside of the loop. This can be generalized to nested conditionals, so that *any* invariant expression can be evaluated outside of the loop, providing that the last condition in the execution condition (defined in Chapter 4) is calculated outside of the loop.

Function calls within loops can also be invariant. If the actual parameters are all invariant and the function call is only switched through invariant boolean test expressions then the call is invariant and may be evaluated outside of a loop. Similarly, array select and append operations may be performed outside of a loop if the array and the expression giving the index value are invariant, and for appends, the expression giving the value for the new array element is invariant. Again, the last condition in the execution condition must be calculated outside of the loop.

8.3.1 Experimental findings

Initial results from test programs showed that removing invariant expressions from inside loops caused the execution time to increase in many examples. The size of the dataflow graph often increased as well as the number of tokens in the token queue and matching store. This can be attributed to the relatively high overheads associated with executing loops on the MDFM. The tagged token model inherently requires the execution of many non-productive operations in order to process the tags used to distinguish tokens circulating around a loop. The process of removing invariant expressions from within loops often increased this overhead thus resulting in the deteriorated performance. The circumstances leading to this undesirable effect are explained in the next section.

8.3.2 Explanation

As described above, the process of identifying invariant expressions within loops and evaluating them outside of the loop often produced undesirable results. To explain why, consider the following example containing two identifiers *P* and *Q* which are defined outside of the for-iter loop construct but used within it:

```
P,Q : integer:= 1, 2
for i : integer:= 1
do
    if i <= n then
        iter i:= i+1 enditer
    else
        (P+Q) * i
    endif
endfor
```

The expression *P+Q* is invariant inside the loop so this expression may be

evaluated outside of the loop and its value passed into the loop. In this example it can be seen that the two aims, firstly to evaluate the invariant expression $P+Q$ once only, and secondly, to decrease the number of data values that need circulating, are both achieved. However, a problem arises when P or Q are involved in other expressions. If, for example, the expression $P-Q$, which is also invariant, were required, then the expressions $P+Q$ and $P-Q$ would both require circulating. Thus the number of data tokens that need circulating would remain the same. It can now be seen that the number of expressions in which P and Q are involved will determine how many data tokens need circulating. In fact, if P or Q are involved in expressions with loop variables, which are therefore not invariant, then the values of P or Q themselves will still require circulating. So from a possible two identifiers that must be circulated, if P or Q appear N times in expressions within a loop, there may be as many as N data tokens which require circulating.

This is not as bad as might first be thought, as circulation of all data values may be done in parallel, so the effective number of steps in the loop still decreases after the first iteration, as the invariant expressions do not need re-evaluation. Another point worth noting is that if P and Q are involved in many invariant expressions, several duplicate nodes will be required to provide copies of the data values represented by P and Q for each expression in which they occur. If loop invariant removal is employed, then this duplication can be done once outside of the loop instead of inside the loop during each iteration cycle. However, as indicated above, preliminary results were not promising and the implementation was not carried through to completion.

8.4 Comparison with other recent similar work

During the course of this project, related research has been carried out at other institutions [Ackerman, 1984; Bohm & Sargeant, 1985; Herath, 1985; Webb, Whiting & Pascoe, 1988], the work most closely related being done at Lawrence Livermore National Laboratory in California as reported recently by Skedzielewski and Welcome [1985]. This latter research involved translating dataflow programs written in SISAL [McGraw et al, 1984] (which was derived from VAL) into an intermediate code called IF1 [Skedzielewski & Glauert, 1984]. IF1 was designed as an intermediate language that could be targeted by different compilers for applicative languages. It is a hierachical graphical language, each graph consisting of nodes connected by edges. Nodes can either be simple or compound; simple nodes usually have a fixed number of inputs and outputs connected to edges, such as those representing arithmetic operations, while compound nodes represent more complex structures such as loops or conditional selections, which can themselves be broken down into subgraphs. Edges represent data values which are passed between nodes and specify data dependencies. They also contain source node and destination node information.

Thus a node combined with the edges which specify it as the destination node, corresponds to a triple of the triple graph (defined in Chapter 4) whose operation is the same as the node and whose operands contain similar information to the edges. Just as with the triple graph, IF1 specifies links from operations (nodes) to operands (edges). As pointed out in Chapter 4, dataflow code for the MDFM does not specify these links, (rather the links in dataflow code are from operations to users) which are required to efficiently perform the optimizations considered. In contrast to the optimizations described in Chapter 6, which are performed on the

fly when creating the triple graph from VAL source code, all optimizations performed by Skedzielewski and Welcome are done on the IF1 code after it has been created from the SISAL source code. The optimizations they considered included common subexpression detection and loop invariant removal.

The basic unit for CSE detection and sharing in the translation of conventional languages is a basic block, consisting of a sequence of statements not containing branches. In IF1, the basic unit of analysis consists of a graph; a graph extends further than a basic block as it may contain statements from the block, both before and after a branch. Shareable CSEs detected in such a graph correspond to those detected using only rules (1) and (6) from the selection rules in Chapter 6. It has been possible in our work to share more CSEs because the target machine is specifically dataflow.

Direct comparison of the results produced by Skedzielewski and Welcome with those in this thesis is difficult without using the same test programs. The static results given by Skedzielewski and Welcome refer to the reduction in the number of nodes at the innermost level of nesting; they are not, therefore, directly comparable with our figures. However, it seems appropriate to compare their reduction in execution time with our figures for CSD alone, even though CSD is more comprehensive than their common subexpression detection and sharing. From the three example programs for which dynamic results were given by Skedzielewski and Welcome, the overall reductions in nodes executed for "CSE + Loop" (which is equivalent to rules (1) and (6) from Chapter 6) were 24%, 30% and 21% on an IF1 interpreter. Note that our results given in Table 8.2 refer to execution on a dataflow simulator with ten processors, but the figures in brackets which are similar, are from

execution with one processor. The differences in implementation make it difficult to draw meaningful comparisons but it is clear that the IF1 results are in the same range as our results. As noted earlier, our CSD results were very variable from program to program.

It was also noted by Skedzielewski and Welcome that in a tagged token dataflow model, during the process of promoting invariant expressions outside of a loop, if the number of inputs to a loop increases, the overheads in circulating the extra data tokens may be more expensive than re-evaluating the invariant expressions on each iteration of the loop. This is precisely the conclusion drawn from preliminary experiments performed in our project.

8.5 Conclusions

Pure dataflow graphs of the kind involved in this project, automatically allow fine grained parallelism at the operation level to be exploited, at the expense of executing many non-productive operations. For example, the graph of Example 2 described in the last chapter, consists of 64 nodes, only 7 of which perform productive calculations. Therefore, it is apparent that it is particularly important to reduce the overheads incurred by the high proportion of non-productive operations in dataflow programs by optimizing the target code.

Programs represented by dataflow graphs have some unpredictable characteristics. An example of their unpredictable nature was indicated in Section 5.3; while a balanced binary tree of duplicate nodes would appear to be the optimal pattern for implementing multiple duplications, in practice, for small programs, at least, this was not always so [Jones, Kidman & Morello, 1985]. Likewise, as explained in Chapter 1, one might intuitively expect CSE elimination to have little effect upon dataflow

program execution time, whereas in practice, it has proved to be quite effective in reducing the execution time of most programs. It is, therefore, important to thoroughly test the effects of any optimizations applied to dataflow programs. Note that when this project was started, no studies at all of the kind undertaken had been reported.

As discussed above, CSD as implemented was almost complete; the removal and subsequent sharing of subexpressions from within loops was omitted from the implementation, because in practice it was often counter productive, as discussed in Section 8.3. The removal of a CSE from within a conditional expression for sharing with an occurrence outside of the conditional sometimes increased the number of branch nodes required; this occasionally resulted in a slightly counter productive effect on execution characteristics. The three versions of Example 9, differing only in the array data processed, highlight this point. CSD was highly effective for 9B and 9C but counter productive for 9A. Thus it would seem that the occasional counter productive effect of CSD is a peculiarity of dataflow programs, which must be tolerated.

The applicative nature of dataflow languages renders them ideal for common subexpression elimination, including the extension to sharing of certain subexpressions both from inside and outside of conditional expressions. Intuitively it may seem that elimination of CSEs would have little effect upon the execution time of a dataflow program as CSEs can be evaluated in parallel. However, in every example, CSD and CSWD caused remarkable decreases in static dataflow graph size, program execution time (measured by the number of exection steps), and token traffic (measured by total and maximum numbers of tokens in the token queue). The results produced demonstrate that CSWD is essential in any production compiler for dataflow languages; CSD alone also produced

significant improvements in performance in almost all programs. Note that the comprehensive implementation of CSD described in this thesis was managed by extending the analysis required for CSWD; it is more comprehensive than any other implementation reported in the literature.

While investigations in this project demonstrate the effectiveness of CSD in the dataflow environment, in principle, the sharing of CSEs as defined by CSD should also apply to any applicative language.

REFERENCES

Abramson, D. and Egan, G.K. "An Overview of the RMIT/CSIRO Parallel Systems Architecture Project", Australian Computer Journal, Vol. 20 No 3, p 113, 1988.

Ackerman, W.B. "Data flow languages", IEEE Computer, Vol. 15 No 2, p 15, 1982.

Ackerman, W.B. "Efficient implementation of applicative languages", Tech. Report TR-323, Laboratory for Computer Science, MIT, Cambridge, 1984.

Ackerman, W.B. and Dennis, J.B. "VAL - A Value-Oriented Algorithmic Language: Preliminary Reference Manual", Tech. Report TR-218, Laboratory for Computer Science, MIT, Cambridge, 1979.

Allan, S.J. and Oldehoeft, A.E. "A flow analysis procedure for the translation of high level languages to a dataflow language", IEEE Transactions on Computers, Vol. C-29 No 9, p 826, 1980.

Arvind and Gostelow, K.P. "A computer capable of exchanging processors for time", Information Processing 77, North Holland, p 849, 1977.

Arvind, Gostelow, K.P. and Plouffe, W. "An asynchronous programming language and computing machine", University of California at Irvine, Tech. Report TR114A, 1978.

Bohm, A.P.W. and Sargeant, J. "Efficient dataflow code generation for SISAL", UMCS-85-10-3, Dept. of Computer Science, University of Manchester, 1985.

Dennis, J.B. "First version of a data flow procedural language", Tech. Report MIT/LCS/TM-61, MIT, 1974.

Gajski, D.D., Padua, D.A., Kuck, D.J. and Kuhn, R.H. "A second opinion on data flow machines and languages", IEEE Computer,Vol. 15 No 2, p 58, 1982.

Gillett, B.E. "Introduction to Operations Research - a computer-oriented algorithmic approach", McGraw-Hill, 1979.

Gries, D. "Compiler construction for digital computers", Wiley, 1971.

Glauert, J.R.W. "A single assignment language for dataflow computing", M.Sc. Thesis, University of Manchester, 1978.

Gurd, J., Watson, I. and Glauert, J.R.W. "A multilayered dataflow computer architecture", Draft, Dept. of Computer Science, University of Manchester, 1980.

Gurd, J.R., Glauert, J.R.W. and Kirkham, C.C. "Generation of dataflow graphical object code for the Lapse programming language", Lecture Notes in Computer Science, Vol. 111, p 155, 1981.

Gurd, J. and Watson, I. "Preliminary evaluation of a prototype dataflow computer", In Proc. IFIP Conference, p 545, 1983.

Gurd, J., Kirkham, C.C. and Watson, I. "The Manchester Prototype Dataflow Computer", Communications of the ACM, Vol. 28, p 34, 1985.

Gurd, J. Personal Communication, 1985.

Herath, J., Saito, N., Toda, K., Yamaguchi, Y. and Yuba, T. "Not(operation) for high speed dataflow computing systems", In Proc. of First International Conf. on supercomputing systems, p 524, 1985.

Jones, P.E.C. "Generation of dataflow graphs from VAL programs", Honours Project Report, Dept. of Computer Science, University of Adelaide, 1984.

Jones, P.E.C., Kidman, B.P. and Morello, R. "Code generation from expressions in a dataflow language", In Proc. ACSC8, Australian Computer Science Comm., Vol. 7, p 6-1, 1985.

Jones, P.E.C. and Kidman, B.P. "Common subexpression detection in conditional expressions in a dataflow language", In Proc. ACSC9, Australian Computer Science Comm., Vol. 8, p 287, 1986.

Kidman, B.P. and Morello, R. "Inherent parallelism of dataflow programs", In Proc. ACSC7, Australian Computer Science Comm., Vol. 6, p 14-1, 1984.

Kirkham, C.C. "The Manchester prototype dataflow system", Basic programming manual, 5th edition, University of Manchester, 1984.

McGraw, J.R. "The VAL language: Description and analysis", ACM Transactions on Programming Languages and Systems, Vol. 4, p 44, 1982.

McGraw, J.R., Skedzielewski, S., Allan, S., Oldehoeft, J., Glauert, J., Kirkham, C., Noyce, W. and Thomas, R. "SISAL: Streams and iteration in a single assignment language", Draft language reference manual, Report M-146, Version 1.2, Lawrence Livermore National Laboratory, University of California, 1985.

Morello, R. "Dataflow architectures and the simulation of a dataflow machine", Honours Project Report, Dept. of Computer Science, University of Adelaide, 1982.

Sargeant, J. "Efficient stored data structures for dataflow computing", Ph.D. Thesis, Tech. Report UMCS-85-8-2, University of Manchester, 1981.

Skedzielewski, S. and Glauert, J.R.W. "IF1 An intermediate form for applicative languages", Reference manual, Draft 8, Lawrence Livermore National Laboratory, University of California, 1984.

Skedzielewski, S. and Welcome, M.L. "Dataflow graph optimization in IF1", Lawrence Livermore National Laboratory, University of California, 1985.

Treleaven, P.C., Brownbridge, D.P. and Hopkins, R.P. "Data driven and demand driven computer architecture", Computing Surveys, Vol. 14, p 93, 1982.

Webb, N.J., Whiting, P.G.and Pascoe, R.S.V. "Implementing a Functional Language on the RMIT Dataflow Architecture", Tech. Report TR 112 074R, RMIT, 1988.

Wendelborn, A.L. "Data flow implementations of a Lucid-like programming language", Ph.D. Thesis, University of Adelaide, 1985.

Wirth, N. "Algorithms + Data Structures = Programs", Prentice-Hall, 1976.

APPENDIX A. Dataflow instruction set

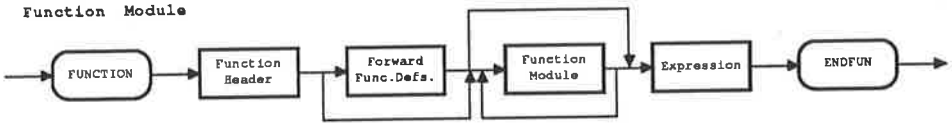
Key: IT = integer
RL = real
BL = boolean
LHV = left hand value

Code	Function	Left Operand	Right Operand	Result
ADI	integer addition	IT	IT	IT
SBI	integer subtraction	IT	IT	IT
MLI	integer multiplication	IT	IT	IT
DVI	integer division (real)	IT	IT	RL
DIV	(integer)	IT	IT	IT
MOD	integer modulo	IT	IT	IT
REM	integer remainder	IT	IT	IT
ADF	real addition	RL	RL	RL
SBF	real subtraction	RL	RL	RL
MLF	real multiplication	RL	RL	RL
DVF	real division	RL	RL	RL
NEG	integer/real negation	IT/RL	–	IT/RL
FIX	translate real to integer	RL	–	IT
FLO	translate integer to real	IT	–	RL
AND	logical AND	BL	BL	BL
IOR	logical inclusive OR	BL	BL	BL
XOR	logical exclusive OR	BL	BL	BL
NOT	logical NOT	BL	–	BL
CEQ	compare =	any	same	BL
CGE	compare >=	any	same	BL
CGT	compare >	any	same	BL
CLE	compare <=	any	same	BL
CLT	compare <	any	same	BL

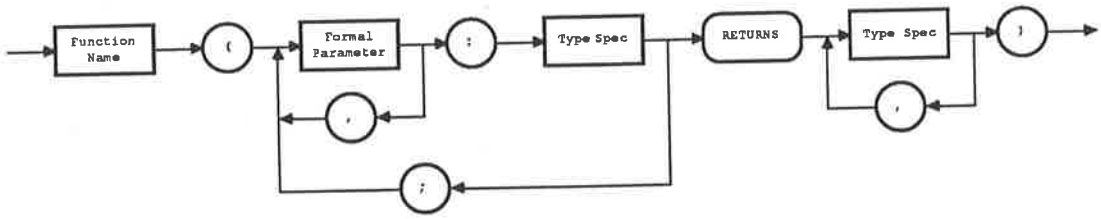
Code	Function	Left Operand	Right Operand	Result
BRA	branch or switch	any	BL	LHV
DUP	duplicate	any	–	LHV
GEN	generate literal	any	Trigger	LHV
KIL	kill token	any	–	none
PLF	proliferate	any	IT	LHV
SLK	set return link	IT	Address	Context
RET	return from function	any	Context	LHV
GAN	generate activation name	any	–	IT
YAN	yield activation name	any	–	IT
SAN	set activation name	any	IT	LHV
YIL	yield iteration level	any	–	IT
SIL	Set iteration level	any	IT	LHV
IIL	increase iteration level	any	–	LHV
ZIL	zero iteration level	any	–	LHV
YIX	yield index	any	–	IT
SIX	set index	any	IT	LHV
IAD	add to index	any	IT	LHV
ISB	subtract from index	any	IT	LHV
IEQ	compare index =	any	IT	BL
IGT	compare index >	any	IT	BL
IGE	compare index >=	any	IT	BL
ILT	compare index <	any	IT	BL
ILE	compare index <=	any	IT	BL
SEL	select array element	Array	IT	Array/ element
EXT	extract array element	Array	IT	Array/ element
SEQ	sequence array elmnts	Array	–	Array

APPENDIX B. Syntax diagrams for VAL-S

Function Module

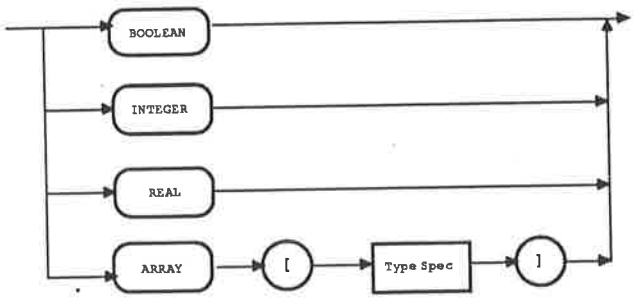


Function Header

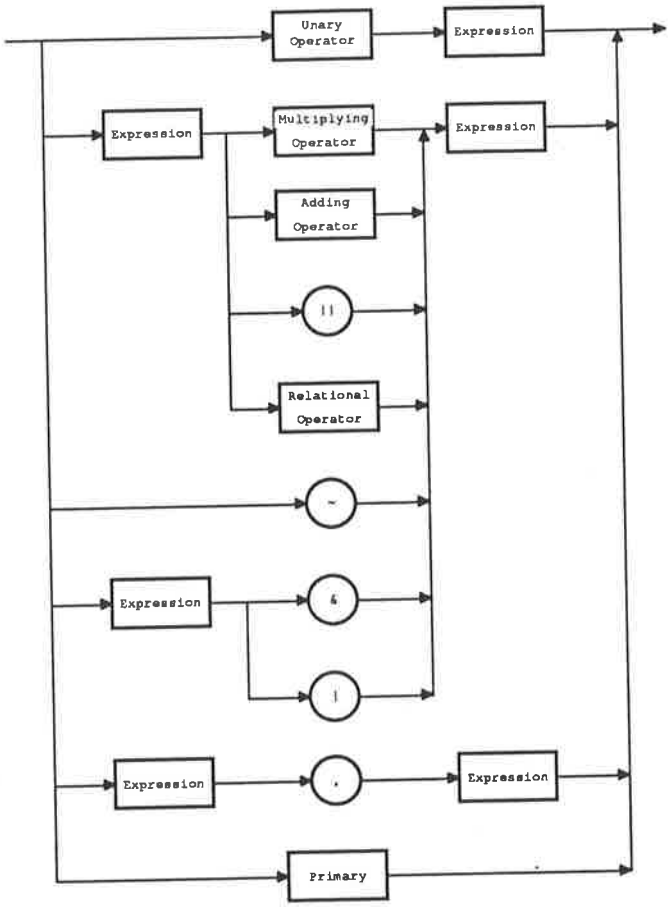


Forward Function Definition

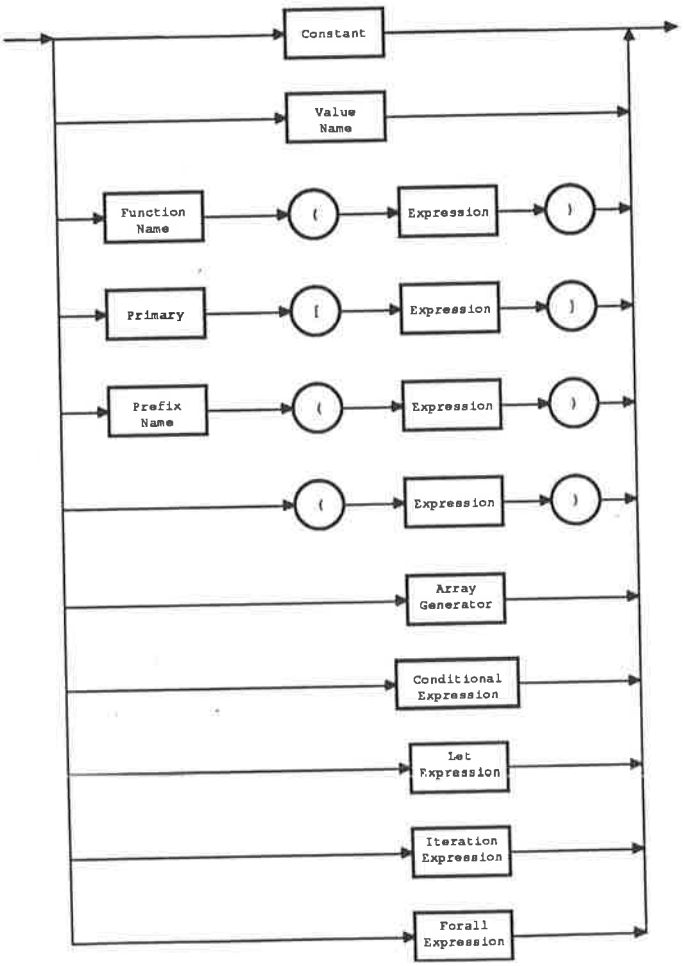
Type Spec:ification



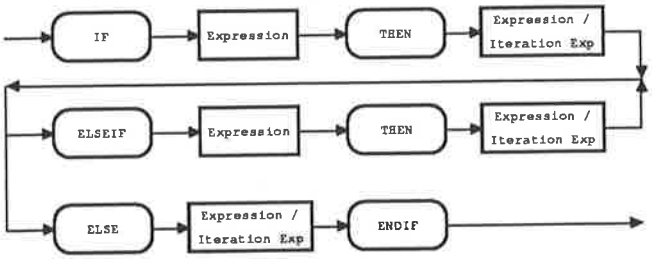
Expression (arity 1)



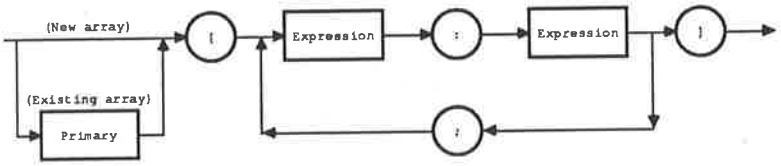
Primary



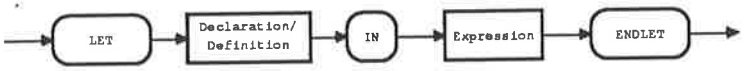
Conditional Expression



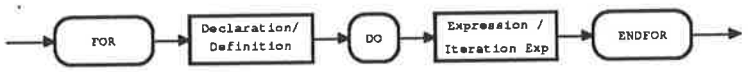
Array Generator



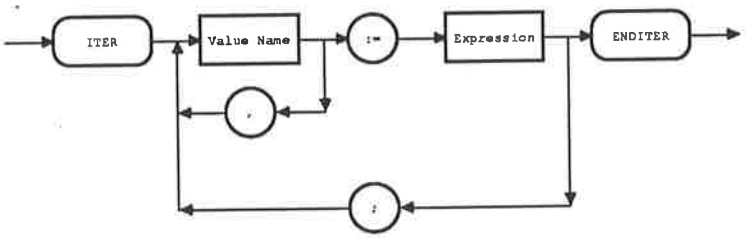
Let Expression



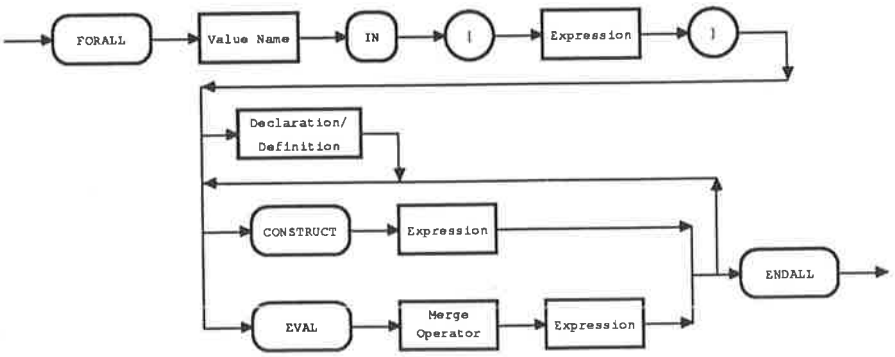
For-Iter Expression



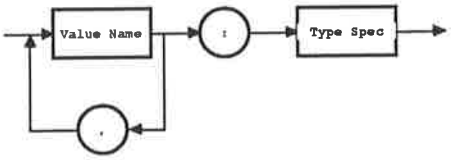
Iteration Expression



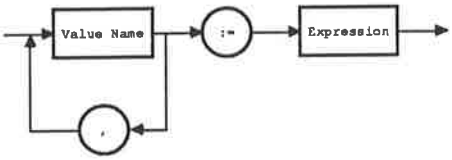
Forall Expression



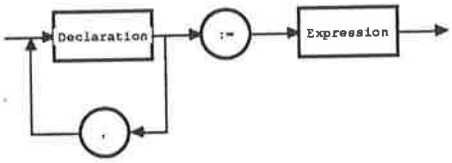
Declaration



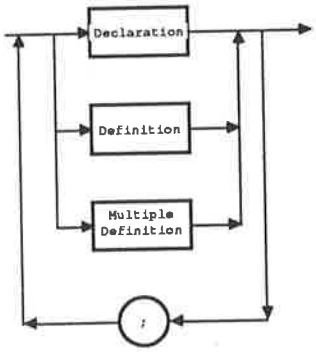
Definition



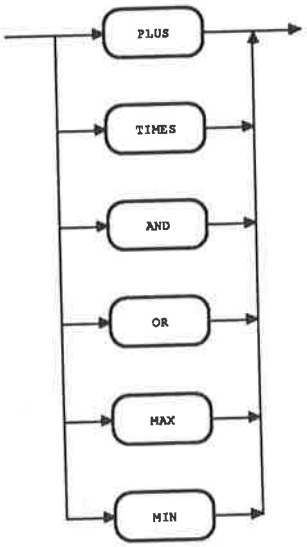
Multiple Definition



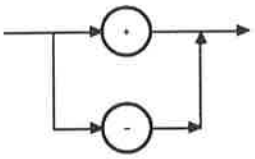
Declaration / Definition



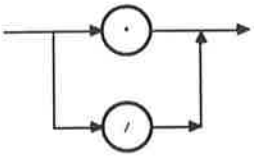
Merge Operator



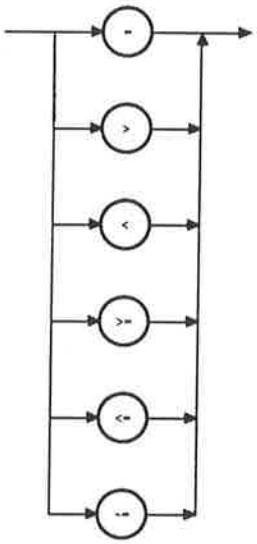
Unary Operator, Adding Operator



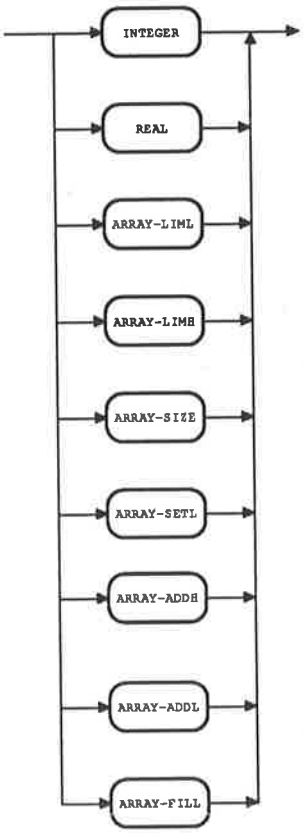
Multiplying Operator



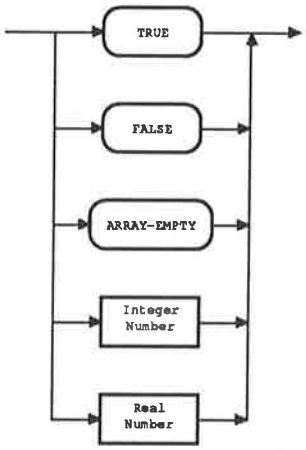
Relational Operator



Prefix Name



Constant



APPENDIX C. VAL–S TEST PROGRAMS

Example 1

Function: To calculate the factorial of an integer parameter.

Version A: Iterative version

```
% input data: i=12
% output data: 479001600

function fact1 ( i : integer returns integer )
    for a, b : integer := 1, i
    do
        if b-1 > 0 then
            iter a:= a*b;
                 b:= b-1;
            enditer
        else
            a
        endif
    endfor
endfun
```

Version B: Recursive version

```
% input data: i=12
% output data: 479001600

function fact2 ( i : integer returns integer )
    if i-1 = 0 then
        i
    else
        fact2 ( i-1 ) * i
    endif
endfun
```

Version C: Doubly recursive version

```
% input data: i=1, j=12
% output data: 479001600

function fact3 ( i, j : integer returns integer )
    if i=j then
        i
    else
        fact3 ( i, ( i+j ) / 2 ) * fact3 ( ( i+j ) / 2 + 1, j )
    endif
endfun
```

Example 2

Function: To calculate the value of x raised to the power y.

```
% input data: x=2.5, y=10
% output data: 9536.743

function expXY ( x : real; y : integer returns real )
    for  a, b : real := 1.0, x;
           c : integer := y
    do
        if  c > 0 then
            if c = c/2*2 then
                iter c:= c/2;
                     b:= b*b;
                enditer
            else
                iter a:= a*b;
                     c:= c–1;
                enditer
            endif
        else
            a
        endif
    endfor
endfun
```

Example 3

Function: To calculate the greatest common divisor of two integers.

```
% input data: x=433329, y=117
% output data: 39

function gcd ( x, y : integer returns integer )
    for  a, b : integer := x, y
    do
        if  a–b ~=0 then
            if a–b < 0 then
                iter  b:= – ( a–b );
                enditer
            else
                iter a:= a–b;
                enditer
            endif
        else
            a
        endif
    endfor
endfun
```

Example 4

Function: To calculate the nth number in the Fibonacci sequence.

```
% input data: n=25
% output data: 46368

function nfib ( n : integer returns integer )
    for f1, f2, count : integer := 0, 1, 1
    do
        if count < n then
            iter f1, f2 := f2, f1+f2;
                 count := count+1
            enditer
        else
            f1
        endif
    endfor
endfun
```

Example 5

Function: Returns the sum of the elements of an array.

```
% input data: n=10
% output data: 220

function test ( n : integer returns integer )

    function sum ( A : array[integer]; low,high : integer
                 returns integer )
        if low = high then
            A [ low ]
        else
            sum( A, low, ( high+low-1 ) / 2 )
            + sum ( A, ( high+low-1 ) / 2+1, high )
        endif
    endfun

    let A : array [ integer ] := forall i in [ 1, n ]
                                 construct i*2
                                endall
    in
        sum ( A, 1, n )
    endlet
endfun
```

Example 6

Function: Contrived example to contain a common conditional expression.

```
% input data:  x=10, y=11
% output data:  132

function root (x, y : integer returns integer)
    let XX, YY : integer:= if x>y then x–y else y–x endif + x,
                           if x>y then x–y else y–x endif + y
    in
        XX + YY
    endlet
endfun
```

Example 7

Function: To calculate the positive root of a quadratic equation.

```
% input data:  a=1.0, b=–5.0, c=6.0
% output data:  3.0

function root (a, b, c : real returns real)

    function absolute ( y : real returns real )
        if y >= 0.0 then
            y
        else
            –y
        endif
    endfun

    function sqrt ( x, eps : real returns real )
        for sroot : real := x / 2.0 do
            if absolute ( sroot*sroot–x ) > eps then
                iter
                    sroot := sroot / 2.0 + x / 2.0 / sroot
                enditer
            else
                sroot
            endif
        endfor
    endfun
```

[Cont.]

Version A

```
% root function body

if a ~= 0.0 then
    if b*b–4.0*a*c > 0.0 then
        ( –b+sqrt ( b*b–4.0*a*c, 0.0001 )) / ( 2.0*a )
    elseif b*b–4.0*a*c = 0.0 then
        –b / ( 2.0*a )
    else
        0.0
    endif
else
    –c/b
endif
endfun
```

Version B

```
% root function body

    let del : integer := b*b–4.0*a*c
    in
        if a ~= 0.0 then
            if del > 0.0 then
                ( –b+sqrt ( del, 0.0001 )) / ( 2.0*a )
            elseif del = 0.0 then
                –b / ( 2.0*a )
            else
                0.0
            endif
        else
            –c/b
        endif
    endlet
endfun
```

Version C

```
% root function body

    if a=0.0 then
        –c/b
    elseif b*b–4.0*a*c < 0.0 then
        0.0
    elseif b*b–4.0*a*c = 0.0 then
        –b / ( 2.0*a )
    else
        ( –b+sqrt ( b*b–4.0*a*c, 0.0001 )) / ( 2.0*a )
    endif
endfun
```

Example 8

Function: To sort an array using a quicksort algorithm.

```
% input data: n=10
% output data:    Version A:  [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
                  Version B:  [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
                  Version C:  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

function sort ( n : integer returns array [ integer ] )

    function quick ( B : array [ integer ]; low, high : integer
                     returns array [ integer ] )
        for left, right, middle : array [ integer ] :=
            array_empty [ integer ], array_empty [ integer ],
            array_empty [ integer ];
            i, j : integer := low, high
        do
            if low = high then
                array_addh ( middle, B [ low ] )
            elseif i>j then
                if array_size ( left ) = 0 then
                    middle || quick ( right, array_liml (right),
                                      array_limh (right) )
                elseif array_size ( right ) = 0 then
                    quick ( left, array_liml (left),
                            array_limh (left) ) || middle
                else
                    quick( left, array_liml (left),
                            array_limh (left) ) || middle ||
                    quick( right, array_liml (right),
                            array_limh (right) )
                endif
            elseif B [ i] = B [ low ] then
                iter
                    middle := array_addh ( middle, B [ i ] );
                    i := i+1
                enditer
            elseif B [ i ] < B [ low ] then
                iter
                    left := array_addh ( left, B [ i ] );
                    i := i+1
                enditer
            else
                iter
                    right := array_addh ( right, B [ i ] );
                    i := i+1
                enditer
            endif
        endfor
    endfun
```

[Cont.]

```
    let A : array [ integer ] :=

        % Version A: Ascending order

            forall i in [ 1, n ]
                construct i * 2
             endall;

        % Version B: Descending order

            forall i in [ 1, n ]
                construct ( n–i ) * 2
             endall;

        % Version C: Random order

            [ 1 : 6, 3, 8, 4, 1, 2, 9, 5, 0, 7 ]

    in
        quick ( A, 1, n )
    endlet
endfun
```

Example 9

Function: To sort an array using an insertion iteration.

```
% input data: n=10
% output data:    Version A:  [ 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 ],
                  Version B:  [ 0, 2, 4, 6, 8, 10, 12, 14, 16, 18 ]
                  Version C:  [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]

function insertionsort ( n : integer returns array [ integer ] )
    for A : array [ integer ] :=

        % Version A: Ascending order

            forall i in [ 1, n ]
                construct i * 2
             endall;

        % Version B: Descending order

            forall i in [ 1, n ]
                construct ( n–i ) * 2
             endall;

        % Version C: Random order

            [ 1 : 6, 3, 8, 4, 1, 2, 9, 5, 0, 7 ]

        i : integer := 1;
    do
        if i = n+1 then
            A
        else
            iter
                A := for B : array [ integer ] := A;
                          j :i nteger := i;
                     do
                          if j–1 = 0 then
                              B
                          elseif B [ j ] < B [ j–1 ] then
                              iter
                                  B := B [ j : B[ j–1 ]; j–1 : B[ j ] ];
                                  j := j–1
                              enditer
                          else
                              B
                          endif
                     endfor;
                i := i+1
            enditer
        endif
    endfor
endfun
```

Example 10

Function: To solve a tridiagonal system of equations.

The system of equations is represented by an NxN matrix where N must be (2**k)–1. The initial vectors are:

D[1..N] – the diagonal elements of the matrix
A[1..N] – the elements directly above the diagonal
B[1..N] – the elements directly below the diagonal
R[1..N] – the right hand side of the system

The function returns an array X[1..N–1] such that:

```
 0  D[1]  A[1]     0                        .... 0        X[1]          R[1]
    B[2]     D[2]     A[2]     0            .... 0        X[2]          R[2]
             B[3]     D[3]     A[3]         .... 0   *    X[3]     =    R[3]
                 ..............................
    0   .....      B[N–1]   D[N–1]   A[N–1]          X[N–1]          R[N–1]
    0   .....      0        B[N]     D[N]     0      X[N]            R[N]
```

% input data: n=7
% output data: [–1.0, 1.0, 2.0, 4.0, –2.0, 3.0,–12.0]

Version A

```
function tridiag ( n : integer returns array[real] )

% reduction part
    let  DR, RR, AR, BR : array[real] :=
         for  R, D, A, B : array[real] :=
                  [1 : 4.0, 12.0, 26.0, 20.0, 10.0, –1.0, –78.0],
                  forall  i in [1, N]
                       construct  i+1.0, N-i+0.0, i–1.0
                  endall;
              index : integer := 1
         do
              if  index = (N+1)/2 then
                  D, R, A, B
              else
                  let id : integer := index*2;
                       NEWD, NEWA, NEWB, NEWR : array[real] :=
                       for j : integer := id;
                            DL, AL, BL, RL : array[real] := D, A, B, R
                       do
                            if j = N+1 then
                                 DL, AL, BL, RL
                            else
                                 let  MU, LAM, RHO : integer :=
                                           D[ j–index ] * D[ j+index ],
                                           B[j] * D[ j+index ],
                                           A[j] * D[ j–index ]
                                 in
                                      iter  j, DL, AL, BL, RL := j + id,
                                        DL[j : LAM * A[j–index] + RHO
                                         * B[j+index] – MU * D[j]],
                                        AL[j : RHO * A[j+index]],
```

[Cont.]

```
                                        BL[j : LAM * B[j-index]],
                                        RL[j : LAM * R[j-index] + RHO
                                         * R[j+index] - MU * R[j]]
                                    enditer
                                endlet
                            endif
                        endfor
                    in
                        iter index, D, A, B, R :=
                            id, NEWD, NEWA, NEWB, NEWR
                        enditer
                    endlet
                endif
            endfor

% substitution part
        in
            for id : integer := N+1;
                X : array[real]:= RR
            do
                if id=1 then
                    X
                else
                    let index : integer := id / 2;
                        NX : array[real] :=
                            for j : integer := index;
                                NNX : array[real] := X
                            do
                                if j = N+1+index then
                                    NNX
                                else
                                    let ALPHA : real :=
                                            if j = index then
                                                0.0
                                            else
                                                BR[j] * X[j-index]
                                            endif;
                                        BETA  : real :=
                                            if j = N+1-index then
                                                0.0
                                            else
                                                AR[j] * X[j+index]
                                            endif;
                                    in
                                        iter j, NNX :=
                                            j+id, NNX[j: (X[j]-ALPHA
                                                    - BETA) / DR[j]]
                                        enditer
                                    endlet
                                endif
                            endfor
                    in
                        iter id,X := index, NX enditer
                    endlet
                endif
            endfor
        endlet
    endfun
```

Version B

```
function tridiag ( N : integer returns array[real] )

        % initialize test values
    for R, D, A, B : array[real] :=
            [1: 4.0, 12.0, 26.0, 20.0, 10.0, -1.0, -78.0],
            forall i in [1, N]
                construct i+1.0, N-i+0.0, i-1.0
            endall;
        index : integer := 1
    do
        if index = (N+1) / 2 then

    % substitution part

            for id : integer := N+1;
                X : array[real]:= R
            do
                if id=1 then
                    X
                else
                    iter id,X := id/2,
                        for j : integer := id/2;
                            NNX : array[real] := X
                        do
                            if j = N+1+id / 2 then
                                NNX
                            else
                                iter j,NNX := j+id,
                                    NNX[j: (X[j]
                                      - if j = id/2 then 0.0
                                        else B[j] * X[j - id / 2]
                                        endif
                                      - if j = N+1- id/2 then 0.0
                                        else A[j] * X[j+id / 2]
                                        endif)
                                     / D[j]]
                                enditer
                            endif
                        endfor
                    enditer
                endif
            endfor
        else
```

[Cont.]

```
% reduction part

            iter index,D,A,B,R := index*2,
                for j : integer := index*2;
                        DL, AL, BL, RL : array[real] := D, A, B, R
                do
                        if j = N+1 then
                            DL, AL, BL, RL
                        else
                            iter j, DL, AL, BL, RL := j+index*2,

                                DL[j:B[j] * D[j+index] * A[j–index] + A[j]
                                  * D[j–index] * B[j+index] – D[j–index]
                                  * D[j+index] * D[j]],

                                AL[j: A[j] * D[j-index] * A[j+index]],

                                BL[j: B[j] * D[j+index] * B[j–index]],

                                RL[j:B[j] * D[j+index] * R[j–index] + A[j]
                                  * D[j–index] * R[j+index] – D[j–index]
                                  * D[j+index] * R[j]]
                            enditer
                        endif
                endfor
            enditer
        endif
    endfor
endfun
```

Version C

```
function tridiag ( n : integer returns array[real] )

% reduction part

    let  DR, RR, AR, BR : array[real] :=
         for  R, D, A, B : array[real] :=
                   [1 : 4.0, 12.0, 26.0, 20.0, 10.0, –1.0, –78.0],
                   forall  i in [1, N]
                        construct  i+1.0, N-i+0.0, i–1.0
                   endall;
              index : integer := 1
         do
              if  index = (N+1)/2 then
                   D, R, A, B
              else
                iter index,D,A,B,R := index*2,
                  for j : integer := index*2;
                       DL, AL, BL, RL : array[real] := D, A, B, R
                  do
                       if j = N+1 then
                            DL, AL, BL, RL
                       else
                            iter j, DL, AL, BL, RL := j+index*2,

                                 DL[j:B[j] * D[j+index] * A[j–index] + A[j]
                                  * D[j–index] * B[j+index] – D[j–index]
                                  * D[j+index] * D[j]],

                                 AL[j: A[j] * D[j-index] * A[j+index]],

                                 BL[j: B[j] * D[j+index] * B[j–index]],

                                 RL[j:B[j] * D[j+index] * R[j–index] + A[j]
                                  * D[j–index] * R[j+index] – D[j–index]
                                  * D[j+index] * R[j]]
                            enditer
                       endif
                  endfor
                enditer
              endif
         endfor
```

[Cont.]

```
% substitution part

    in
        for  id : integer := N+1;
             X : array[real]:= RR
        do
             if id=1 then
                 X
             else
                 iter id,X := id/2,
                      for j : integer := id/2;
                          NNX : array[real] := X
                      do
                          if j = N+1+id / 2 then
                              NNX
                          else
                              iter j,NNX := j+id,
                                        NNX[j: (X[j]
                                    - if j = id/2 then 0.0
                                      else BR[j] * X[j – id / 2]
                                      endif
                                    - if j = N+1– id/2 then 0.0
                                      else AR[j] * X[j+id / 2]
                                      endif)
                                   / DR[j]]
                              enditer
                          endif
                      endfor
                 enditer
             endif
        endfor
    endlet
endfun
```

Example 11

Function: To solve certain second order differential equations in one dimension with periodic boundary conditions.

```
% The function produces an array X[2..size+1] where X must be non zero
% and size must be a power of 2 and at least 4 such that, for i=3..size:
%   X[i+1] + X[i-1] - 2*X[i]
%   ------------------------ + A * X[i] = -Q[i]
%         hx**2
% i.e. del-squared(X) + A*X = -Q in one dimension.

        % input data: c=4.0, size=4
        % output data: X = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0]

        function perred (c : real; size : integer returns array[real])

            function modd (i, j : integer returns integer)
                i - (i / j * j)
            endfun

            let fac : real:= 0.01;
                QA : array[real]:= [2: -7.04, -0.08, -0.12, -0.16, -0.2, -0.24,
                                    -0.28, 7.68];
        % reduction part
                logsize : integer, fina : real, finQ,finB : array[real]:=
                for ih, count : integer, a : real:= 1, 1, 2.0-c*fac;
                    Q, B : array[real]:= QA,array_empty[real]
                do
                    let newQ : array[real]:=
                        for k : integer:= 2*ih+1;
                            T : array[real]:= Q;
                            qk : real:= Q[ih+1]
                        do
                            if k > size+1 then
                                T
                            else
                                let nqk : real:= Q[modd(k, size) + ih]
                                in
                                    iter
                                        k, T, qk:= k+2*ih,
                                            T[k:qk + a*Q[k] + nqk], nqk
                                    enditer
                                endlet
                            endif
                        endfor;
                    newB : array[real]:= array_addh(B, a)
```

[Cont.]

```
                in
                    if ih = size / 2 then
                        count, a*a-4.0, newQ, newB
                    else
                        iter
                            ih, count, B, a, Q:= ih*2, count+1, newB,
                                                    a*a-2.0, newQ
                        enditer
                    endif
                endlet
            endfor;

        QB : array[real]:= finQ[size + 1 : finQ[size+1] / fina]

    % substitution part

        in
            for ih, logi : integer:= size / 2, logsize;
                Q : array[real]:= QB
            do
                let a : real := finB[logi];
                    newQ : array[real]:=
                        for k : integer, qk : real := ih+1, Q[size+1];
                            T : array[real]:= Q
                        do
                            if k > size+1 then
                                T
                            else
                                let nqk : real := Q[k + ih]
                                in
                                    iter
                                        k, T, qk := k+2*ih,
                                            T[k: (qk + Q[k] + nqk)/a], nqk
                                    enditer
                                endlet
                            endif
                        endfor
                in
                    if ih=1 then
                        newQ
                    else
                        iter
                            ih, logi, Q := ih/2, logi-1, newQ
                        enditer
                    endif
                endlet
            endfor
        endlet
    endfun
```

Example 12

Function: A job scheduling algorithm for 2 machines.
The problem is to determine the sequence in which ten jobs should be processed on two machines in order to reduce the total processing time through the machines. The ouput produced gives the optimal order in which the jobs should be run.

```
% input data: n=10
% output data: [10,9,8,7,6,5,4,3,2,1]

function job ( n : integer returns array [integer] )

    for T1,T2 : array [integer] := forall i in [ 1, n ]
                                       construct n-i, i
                                   endall;
        ind1, ind2 : integer := 0, 0;
        SEQ1, SEQ2 : array [integer] :=
            array_empty [integer], array_empty [integer]
    do
        if array_size ( SEQ1 ) + array_size ( SEQ2 ) = n then
            SEQ1 || SEQ2
        elseif ind1 > 0 then
            iter
                SEQ1 := array_addh ( SEQ1, ind1 );
                T1 := T1 [ ind1 : 1000000 ];
                T2 := T2 [ ind1 : 1000000 ];
                ind1 := 0
            enditer
        elseif ind2 > 0 then
            iter
                SEQ2 := array_addl ( SEQ2, ind2 );
                T1 := T1 [ ind2 : 1000000 ];
                T2 := T2 [ ind2 : 1000000 ];
                ind2 := 0
            enditer
```

[Cont.]

```
else
    iter
        ind1, ind2 :=
            for i, j, x, index1, index2 : integer :=
                    1, 1, T1 [ 1 ], 0, 0
            do
                if i>n then
                    if j>n then
                        index1, index2
                    elseif T2 [ j ] <= x then
                        iter
                            x:= T2 [ j ];
                            index1:= 0;
                            index2:= j;
                            j:= j+1
                        enditer
                    else
                        iter j:= j+1 enditer
                    endif
                elseif T1 [ i ] <= x then
                    iter
                        x:= T1 [ i ];
                        index1:= i;
                        index2:= 0;
                        i:= i+1
                    enditer
                else
                    iter i:= i+1 enditer
                endif
            endfor
    enditer
endif
endfor
endfun
```

Example 13

Function: To determine the total elapsed time and idle times of machines for a batch of jobs on two machines.

```
% input data: n=10
% output data:    Total elapsed time = 55
                  Idle time machine 1 = 10
                  Idle time machine 2 = 0

function job2 ( n : integer returns array [ integer ] )
    let T1, T2, Order : array [ integer ] :=
            forall i in [ 1, n ]
                construct n–i,i , n-i+1
            endall;
        A : array [ integer ] :=
            for Tim1, Tim2, Tom1, Tom2 : array [ integer ] :=
                    [1 : 0],
                    [1 : T1[ Order[1] ]], [1 : T1[ Order[1] ]],
                    [1 : T2[ Order[1] ] + T1[ Order[1] ]];
                i, itm2 : integer := 2, T1[ Order[1] ];
            do
                if i > n then
                    [1 : Tom2 [n] – Tom1 [n], itm2, Tom2 [n] ]
                elseif Tom2 [i–1] < Tom1 [i–1]
                                        + T1[ Order[i] ] then
                    iter
                        itm2 := itm2 + Tom1 [i–1]
                                + T1[ Order[i] ] – Tom2 [i–1];
                        Tim1 := array_addh (Tim1, Tom1[i–1]);
                        Tim2 := array_addh (Tim2, Tom1[i–1]
                                                + T1[ Order[i] ]);
                        Tom1 := array_addh (Tom1, Tom1[i–1]
                                                + T1[ Order[i] ]);
                        Tom2 := array_addh (Tom2, Tom2[i–1]
                                                + T2[ Order[i] ]);
                        i:= i+1
                    enditer
                else
                    iter
                        Tim1 := array_addh (Tim1, Tom1[i–1]);
                        Tim2 := array_addh (Tim2, Tom2[i–1]);
                        Tom1 := array_addh (Tom1, Tom1[i–1]
                                                + T1[ Order[i] ]);
                        Tom2 := array_addh (Tom2, Tom2[i–1]
                                                + T2[ Order[i] ]);
                        i:= i+1
                    enditer
                endif
            endfor
    in
        A
    endlet
endfun
```

Example 14

Function: To determine certain characteristics associated with queueing theory. The output data is returned as an array so that the data is ordered and each value can be identified from the others.

```
% input data:  lambda=15.0, mu=6.0, n=1, s=3
% output data:      l=6.0112;
                    lq=3.5112
                    w=0.4007
                    wq=0.2341
                    pn=0.1124
                    pzero=0.0449
                    ps=0.7022

function queue ( lambda, mu : real; n, s : integer returns array [real] )

    function fact ( x : integer returns integer )
        if x<=1 then
            1
        else
            fact (x-1)*x
        endif
    endfun

    function expo ( x:real; y:integer returns real )
        for  a, b : real := 1.0, x;
             c : integer := y
        do  if c > 0 then
                if  c = c / 2 * 2 then
                    iter c:= c*1 / 2;
                         b:= b*b;
                    enditer
                else iter a:= a*b;
                          c:= c-1;
                     enditer
                endif
            else
                a
            endif
        endfor
    endfun
```

[Cont.]

Version A

```
let A : array[real]:=
    for nk, sk : integer := 1, 0;
        sum : real := 0.0
    do
        if nk > s then
            [ 1 : lambda, mu, n, s,

% l:
1/ (sum + (1 / ( fact(s) * ( 1 – lambda / (mu * s)))) * expo (lambda / mu, s))
 * expo (lambda / mu, s+1) / (s * fact(s) * expo(1 – lambda / (mu * s), 2))
 + lambda / mu,

% lq:
1/ (sum + (1 / ( fact(s) * ( 1 – lambda / (mu * s)))) * expo (lambda / mu, s))
 * expo (lambda / mu, s+1) / (s * fact(s) * expo(1 – lambda / (mu * s), 2)),

% w:
1/ (sum + (1 / ( fact(s) * ( 1 – lambda / (mu * s)))) * expo (lambda / mu, s))
 * expo (lambda / mu, s+1) / (s * fact(s) * expo(1 – lambda / (mu * s), 2)),
 + lambda/mu) / lambda,

% wq:
1/ (sum + (1 / ( fact(s) * ( 1 – lambda / (mu * s)))) * expo (lambda / mu, s))
 * expo (lambda / mu, s+1) / (s * fact(s) * expo(1 – lambda / (mu * s), 2)),
 / lambda,

% pn:
if n >= s then
  1/ (sum + (1 / ( fact(s) * ( 1 – lambda / (mu * s)))) * expo (lambda / mu, s))
  * 1/ expo ( fact (s), n-s) * expo (lambda / mu, n)
else
  1/ (sum + (1 / ( fact(s) * ( 1 – lambda / (mu * s)))) * expo (lambda / mu, s))
  * 1/ fact(n) * expo (lambda / mu, n)
endif,

% pzero:
1/ (sum + (1 / ( fact(s) * ( 1 – lambda / (mu * s)))) * expo (lambda / mu, s))

% ps:
1/ (sum + (1 / ( fact(s) * ( 1 – lambda / (mu * s)))) * expo (lambda / mu, s))
 * expo (lambda / mu, s) / (fact(s) * (1 – lambda / (mu * s)))]


            else
                iter sum:= sum + 1.0 / fact(sk)
                             * expo (lambda / mu, sk);
                     sk:= nk;
                     nk := nk+1
                enditer
            endif
        endfor
    in
        A
    endlet
endfun
```

Version B

```
function pzero (sum, s : integer; lambda, mu : real returns real)
    1/ (sum + (1/ ( fact(s) * (1 – lambda / (mu * s))))
    * expo (lambda / mu, s))
endfun

function pn (sum, n, s : integer; lambda, mu : real returns real)
    if n >= s then
        1/ expo (fact(s), n–s) * expo (lambda / mu, n)
        * pzero (sum, s, lambda, mu)
    else
        1/ fact(n) * expo (lambda / mu, n)
        * pzero(sum, s, lambda, mu)
    endif
endfun

function ps (sum, s : integer; lambda, mu : real returns real)
    expo (lambda / mu, s) * pzero (sum, s, lambda, mu) / ( fact(s)
    * (1 – lambda / (mu * s)))
endfun

function lq (sum, s : integer; lambda, mu : real returns real)
    expo (lambda / mu, s+1) * pzero (sum, s, lambda, mu)
    / (s * fact(s) * expo(1 – lambda / (mu *s ), 2))
endfun

function l (sum, s : integer; lambda, mu : real returns real)
    lq (sum, s, lambda, mu) + lambda / mu
endfun

function w (sum, s : integer; lambda, mu : real returns real)
    l (sum, s, lambda, mu) / lambda
endfun

function wq (sum, s : integer; lambda, mu : real returns real)
    lq (sum, s, lambda, mu) / lambda
endfun

let A : array[real] :=
    for nk, sk : integer := 1, 0;
        sum : real := 0.0
    do   if nk > s then
            [1: lambda, mu, n, s, l(sum, s, lambda, mu),
             lq(sum, s, lambda, mu), w(sum, s, lambda, mu),
             wq(sum, s, lambda, mu), pn(sum, n, s, lambda, mu),
             pzero(sum, s, lambda, mu), ps(sum, s, lambda, mu)]
        else iter sum:= sum+1.0 / fact(sk) * expo(lambda / mu, sk);
                  sk:= nk;
                  nk := nk+1
             enditer
        endif
    endfor
in
    A
endlet
endfun
```

Version C

```
let  A : array[real]:=
     for nk, sk : integer := 1, 0;
          sum : real := 0.0
     do
          if  nk > s then
               let   pzero : real:=
                          1/ (sum+(1/ (fact(s) * (1 – lambda
                           / (mu * s)))) * expo (lambda / mu, s));
                     pn : real:=
                          if  n >= s then
                               1/expo (fact(s), n-s) * expo (lambda / mu,
                                n) * pzero
                          else
                               1/ fact(n) * expo (lambda / mu, n) * pzero
                          endif;
                     ps : real:=
                          expo (lambda / mu, s) * pzero
                          / (fact(s) * (1 – lambda / (mu * s)));
                     lq : real:=
                          expo (lambda / mu, s+1) * pzero
                          / (s * fact(s) * expo(1 – lambda / (mu * s), 2));
                     l : real:= lq + lambda / mu;
                     w : real:= l / lambda;
                     wq : real:= lq / lambda
               in
                     [1: lambda, mu, n, s, l, lq, w, wq, pn, pzero, ps]
               endlet
          else
               iter sum:= sum+1.0/ fact(sk) * expo (lambda / mu, sk);
                     sk:=  nk;
                     nk := nk+1
               enditer
          endif
     endfor
in
     A
endlet
endfun
```

APPENDIX D. RESULTS FROM SIMULATOR RUNS

Opt	No Procs	Dataflow nodes Tot	Swtch	Dup	Steps	Token Queue Max	Tot	Ave/Step	Matching/Store Max	Tot	Bypass Ratio	Parallelism Max	Ave
Example 1													
Version A													
None	1	23	4	4	162	5	223	1.4	5	61	0.62	1	2.1
	2				89			2.5				2	
	10				76			2.9				4	
CSWD	1	19	2	2	113	4	150	1.3	3	37	0.67	1	1.5
	2				75			2.0				2	
	10				75			2.0				3	
CSD and CSWD	1	19	3	2	115	4	164	1.4	4	49	0.57	1	2.1
	2				65			2.5				2	
	10				54			3.0				3	
Version B													
None	1	21	3	4	191	5	262	1.4	26	71	0.63	1	1.5
	2				131	4		2.0				2	
	10				130	5		2.0				3	
CSWD	1	17	1	2	142	3	189	1.3	24	47	0.67	1	1.2
	2				118			1.6				2	
CSD and CSWD	1	17	2	2	143	3	202	1.4	25	59	0.59	1	1.3
	2				107			1.9				2	

Opt	No Procs	Dataflow nodes Tot	Swtch	Dup	Steps	Token Queue Max	Tot	Ave/Step	Matching/Store Max	Tot	Bypass Ratio	Parallelism Max	Ave
Version C													
None	1	48	8	16	734	50	1043	1.4	77	309	0.58	1	11.7
	10				95	50		11.0				10	
	100				63	48		16.6				39	
CSWD	1	35	2	9	375	28	546	1.5	34	171	0.54	1	6.7
	10				61	27		9.0				10	
	100				56	25		9.8				15	
CSD	1	32	2	8	342	32	502	1.8	36	160	0.53	1	6.6
and	10				58	30		8.7				10	
CSWD	100				52	32		9.7				20	
Example 2													
None	1	64	19	23	285	11	404	1.4	12	119	0.58	1	3.7
	2				152			2.7				2	
	10				78			5.2				8	
CSWD	1	36	6	8	118	5	164	1.4	5	46	0.61	1	1.8
	2				66	5		2.5				2	
	10				64	4		2.6				3	
CSD	1	37	7	9	125	5	176	1.4	6	51	0.59	1	2.1
and	2				73			2.4				2	
CSWD	10				61			2.9				4	

Opt	No Procs	Dataflow nodes Tot	Swtch	Dup	Steps	Token Queue Max	Tot	Ave/Step	Matching/Store Max	Tot	Bypass Ratio	Parallelism Max	Ave
Example 3													
None	1	54	15	20	151943	9	218646	1.4	10	66703	0.56	1	2.9
	2				81533			2.7				2	
	10				51886			4.2				8	
CSWD	1	29	4	6	55592	5	81533	1.4	3	25941	0.53	1	1.4
	2				40768	4		2.0				2	
	10				40767	4		2.0				3	
CSD	1	27	6	5	55594	5	81536	1.4	4	25942	0.53	1	1.5
and	2				37064	4		2.2				2	
CSWD	10				37062	4		2.2				3	
Example 4													
None	1	33	6	7	435	8	636	1.5	8	201	0.54	1	2.8
	2				232	8		2.7				2	
	10				156	7		4.1				6	
CSWD	1	30	4	6	359	6	510	1.4	6	151	0.58	1	2.7
	2				182	6		2.8				2	
	10				131	5		3.9				4	
CSD	1	28	4	5	357	6	508	1.4	5	151	0.58	1	2.7
and	2				181	6		2.8				2	
CSWD	10				131	5		3.9				4	

Opt	No Procs	Dataflow nodes Tot	Swtch	Dup	Steps	Token Queue Max	Tot	Ave/Step	Matching/Store Max	Tot	Bypass Ratio	Parallelism Max	Ave
Example 5													
None	1	126	20	48	2322	179	3700	1.6	343	1378	0.41	1	30.0
	10				254	175		14.6				10	
	100				83	161		44.6				100	
	500				83	171						151	
CSWD	1	96	6	32	1240	109	2010	1.6	133	770	0.38	1	15.5
	10				148	95		13.6				10	
	100				80			25.1				69	
CSD and CSWD	1	91	6	31	1203	113	1964	1.6	129	761	0.37	1	15.6
	10				144	111		13.6				10	
	100				77	110		25.5				66	
Example 6													
None	1	40	8	14	38	14	56	1.5	12	18	0.53	1	2.7
	10				14	11		4.0				6	
CSWD	1	28	4	8	26	9	40	1.5	6	14	0.46	1	2.2
	10				12	7		3.3				4	
CSD and CSWD	1	19	2	4	18	7	28	1.6	5	10	0.44	1	1.6
	10				11	5		2.5				3	

Opt	No Procs	Dataflow nodes Tot	Swtch	Dup	Steps	Token Queue Max	Tot	Ave/Step	Matching/Store Max	Tot	Bypass Ratio	Parallelism Max	Ave
Example 7													
Version A													
None	1	168	45	59	282	20	404	1.4	21	122	0.57	1	2.5
	10				113	18		3.6				10	
	100				113	20		3.6				16	
CSWD	1	100	13	23	172	8	246	1.4	10	74	0.57	1	1.6
	10				105	7		2.3				4	
CSD and CSWD	1	86	14	16	166	7	238	1.4	10	72	0.57	1	1.6
	10				103			2.3				4	
Version B													
None	1	123	30	37	241	13	345	1.4	14	104	0.57	1	2.3
	10				107			3.2				8	
CSWD	1	87	15	16	167	8	240	1.4	10	73	0.56	1	1.7
	10				101	7		2.4				4	
CSD and CSWD	1	87	15	16	167	8	240	1.4	10	73	0.56	1	1.7
	10				101	7		2.4				4	

Opt	No Procs	Dataflow nodes Tot	Swtch	Dup	Steps	Token Queue Max	Tot	Ave/Step	Matching/Store Max	Tot	Bypass Ratio	Parallelism Max	Ave
Version C													
None	1	179	51	65	302	19	436	1.4	21	134	0.56	1	2.5
	10				119	18		3.7				10	
	100				119	18		3.7				16	
CSWD	1	99	13	23	184	8	264	1.4	10	80	0.57	1	1.6
	10				115	7		2.3				4	
CSD and CSWD	1	86	15	18	173	6	249	1.4	10	76	0.56	1	1.6
	10				110			2.3				4	

Opt	No Procs	Dataflow nodes Tot	Swtch	Dup	Steps	Token Queue Max	Tot	Ave/Step	Matching/Store Max	Tot	Bypass Ratio	Parallelism Max	Ave
Example 8													
Version A													
None	1	1573	616	753	92720	306	140436	1.5	353	47716	0.49	1	35.8
	10				9779	306		14.4				10	
	100				2625	282		53.5				100	
	500				2588	251		54.3				179	
CSD	1	1237	481	593	66543	203	99492	1.5	259	32949	0.50	1	29.1
	10				7046	203		14.1				10	
	100				2297	186		43.3				100	
	500				2289	203		43.5				128	
CSWD	1	501	103	194	17929	43	26935	1.5	67	9006	0.50	1	8.8
	10				2523	42		10.7				10	
	100				2044	39		13.2				31	
CSD	1	438	106	169	15327	41	23154	1.5	67	7827	0.49	1	8.3
and	10				2252			10.3				10	
CSWD	100				1853			12.5				26	

Opt	No Procs	Dataflow nodes Tot	Swtch	Dup	Steps	Token Queue Max	Tot	Ave/Step	Matching/Store Max	Tot	Bypass Ratio	Parallelism Max	Ave
Version B													
None	1	1576	614	756	93218	305	141233	1.5	353	48015	0.48	1	35.4
	10				9837	305		14.4				10	
	100				2702	291		52.3				100	
	500				2633	251		53.6				179	
CSD	1	1240	481	595	66591	194	99455	1.5	306	32864	0.51	1	28.9
	10				7040	194		14.1				10	
	100				2316	177		42.9				100	
	500				2308	194		43.1				128	
CSWD	1	504	102	196	18112	43	27219	1.5	75	9107	0.50	1	8.7
	10				2570	42		10.6				10	
	100				2089	39		13.0				31	
CSD	1	441	106	171	15510	40	23438	1.5	102	7928	0.49	1	8.3
and	10				2272	40		10.3				10	
CSWD	100				1872	39		12.5				26	

Opt	No Procs	Dataflow nodes Tot	Swtch	Dup	Steps	Token Queue Max	Tot	Ave/Step	Matching/Store Max	Tot	Bypass Ratio	Parallelism Max	Ave
Version C													
None	1	1594	615	761	48560	449	72597	1.5	608	24037	0.51	1	45.9
	10				5033	447		14.4				10	
	100				1130	422		64.2				100	
	500				1059	407		68.6				298	
CSD	1	1255	481	599	35234	332	52039	1.5	478	16805	0.52	1	37.8
	10				3642	332		14.3				10	
	100				956	325		54.4				100	
	500				933	301		55.8				256	
CSWD	1	522	103	201	9093	61	13518	1.5	82	4425	0.51	1	10.8
	10				1155	58		11.7				10	
	100				840	55		16.1				41	
CSD	1	456	106	175	7964	52	11885	1.5	89	3921	0.51	1	10.5
and	10				1018	50		11.7				10	
CSWD	100				758	47		15.7				37	

Opt	No Procs	Dataflow nodes Tot	Swtch	Dup	Steps	Token Queue Max	Tot	Ave/Step	Matching/Store Max	Tot	Bypass Ratio	Parallelism Max	Ave
Example 9													
Version A													
None	1	287	80	113	4717	96	7337	1.6	172	2620	0.44	1	14.4
	10				565	95		13.0				10	
	100				328	96		22.4				73	
CSD	1	209	54	76	3563	65	5591	1.6	134	2028	0.43	1	11.9
	10				447	64		12.5				10	
	100				299	59		18.7				45	
CSWD	1	155	16	45	2110	36	3222	1.5	69	1112	0.47	1	7.6
	10				313	31		10.3				10	
	100				277	31		11.6				16	
CSD	1	139	21	39	2183	44	3341	1.5	70	1158	0.47	1	7.9
and	10				317	43		10.5				10	
CSWD	100				277	42		12.1				22	

Opt	No Procs	Dataflow nodes Tot	Swtch	Dup	Steps	Token Queue Max	Tot	Ave/Step	Matching/Store Max	Tot	Bypass Ratio	Parallelism Max	Ave
Version B													
None	1	291	80	115	21713	93	34460	1.6	176	12747	0.41	1	17.8
	10				2446	93		14.1				10	
	100				1219	89		28.3				72	
CSD	1	213	54	78	14187	65	22634	1.6	138	8447	0.40	1	14.6
	10				1647	63		13.7				10	
	100				974	59		23.2				45	
CSWD	1	159	16	47	9989	44	15396	1.5	78	5407	0.46	1	9.6
	10				1330	43		11.6				10	
	100				1042	44		14.8				31	
CSD	1	143	21	41	7713	44	12185	1.6	74	4472	0.42	1	9.0
and	10				991	44		12.3				10	
CSWD	100				862	43		14.1				22	

Opt	No Procs	Dataflow nodes Tot	Swtch	Dup	Steps	Token Queue Max	Tot	Ave/Step	Matching/Store Max	Tot	Bypass Ratio	Parallelism Max	Ave
Version C													
None	1	309	80	121	13964	93	22121	1.6	176	8157	0.42	1	17.0
	10				1597	93		13.9				10	
	100				824	89		26.8				72	
CSD	1	229	54	83	9343	65	14880	1.6	138	5537	0.41	1	13.8
	10				1109	63		13.4				10	
	100				678	59		21.9				45	
CSWD	1	177	16	53	6428	44	9882	1.5	74	3454	0.46	1	9.1
	10				878	43		11.3				10	
	100				704	44		14.0				31	
CSD	1	159	21	46	5257	44	8241	1.6	73	2984	0.43	1	8.7
and	10				699	44		11.8				10	
CSWD	100				608	43		13.6				22	

Opt	No Procs	Dataflow nodes Tot	Swtch	Dup	Steps	Token Queue Max	Tot	Ave/Step	Matching/Store Max	Tot	Bypass Ratio	Parallelism Max	Ave
Example 10													
Version A													
None	1	1387	348	555	11398	369	17587	1.5	1275	6189	0.46	1	62.3
	10				1148	369		15.3				10	
	100				201	368		87.5				100	
	500				183	367		96.1				268	
CSD	1	1301	324	512	10719	308	16588	1.5	1196	5869	0.45	1	60.6
	10				1080	306		15.4				10	
	100				190	300		87.3				100	
	500				177	301		93.7				210	
CSWD	1	944	118	342	7190	254	10639	1.5	678	3449	0.52	1	39.3
	10				733	247		14.5				10	
	100				184	213		57.8				100	
	500				183	245		58.1				139	
CSD	1	908	119	324	6962	209	10317	1.5	640	3355	0.52	1	38.7
and	10				710	208		14.5				10	
CSWD	100				181	209		57.0				100	
	500				180	209		57.3				134	

Opt	No Procs	Dataflow nodes Tot	Swtch	Dup	Steps	Token Queue Max	Tot	Ave/Step	Matching/Store Max	Tot	Bypass Ratio	Parallelism Max	Ave
Version B													
None	1	1645	410	678	13617	496	20972	1.5	1103	7355	0.46	1	72.4
	10				1373	489		15.3				10	
	100				221	464		94.9				100	
	500				188	490		111.6				292	
CSD	1	1342	329	533	10786	340	16717	1.5	818	5931	0.45	1	59.9
	10				1091	340		15.3				10	
	100				193	326		87.0				100	
	500				180	332		92.9				206	
CSWD	1	1071	116	398	8084	371	11978	1.5	608	3894	0.52	1	42.8
	10				824	350		14.5				10	
	100				194	322		61.7				100	
	500				189	317		63.4				201	
CSD	1	936	119	337	6987	217	10396	1.5	433	3409	0.51	1	38.2
and	10				714	212		14.6				10	
CSWD	100				183	212		56.8				100	
	500				183	209		56.8				139	

Opt	No Procs	Dataflow nodes Tot	Swtch	Dup	Steps	Token Queue Max	Tot	Ave/Step	Matching/Store Max	Tot	Bypass Ratio	Parallelism Max	Ave
Version C													
None	1	1594	404	653	13512	443	20787	1.5	1450	7275	0.46	1	73.4
	10				1359	435		15.3				10	
	100				217	419		95.8				100	
	500				184	384		113.0				268	
CSD	1	1290	322	507	10676	312	16524	1.5	1192	5848	0.45	1	60.7
	10				1076	312		15.4				10	
	100				189	301		87.4				100	
	500				176	306		93.9				209	
CSWD	1	1033	116	380	8028	369	11860	1.5	791	3832	0.52	1	43.2
	10				816	348		14.5				10	
	100				192	319		61.8				100	
	500				186	315		63.8				203	
CSD	1	897	118	318	6926	200	10270	1.5	652	3344	0.52	1	38.5
and	10				706	200		14.5				10	
CSWD	100				181	200		56.7				100	
	500				180	195						132	

Opt	No Procs	Dataflow nodes Tot	Swtch	Dup	Steps	Token Queue Max	Tot	Ave/Step	Matching/Store Max	Tot	Bypass Ratio	Parallelism Max	Ave
Example 11													
None	1	581	105	190	4850	133	7430	1.5	470	2580	0.47	1	25.4
	10				494	133		15.0				10	
	100				191	109		38.9				73	
CSD	1	569	104	184	4827	133	7400	1.5	469	2573	0.47	1	25.3
	10				493	126		15.0				10	
	100				191	109		38.7				73	
CSWD	1	469	49	135	3469	78	5119	1.5	270	1650	0.52	1	17.7
	10				363	76		14.1				10	
	100				196	77		26.1				46	
CSD	1	462	49	132	3464	77	5113	1.5	269	1649	0.52	1	17.7
and	10				363	76		14.1				10	
CSWD	100				196	77		26.1				46	

Opt	No Procs	Dataflow nodes Tot	Swtch	Dup	Steps	Token Queue Max	Tot	Ave/Step	Matching/Store Max	Tot	Bypass Ratio	Parallelism Max	Ave
Example 12													
None	1	1302	505	622	157467	245	241861	1.5	417	84394	0.46	1	28.5
	10				16803	241		14.4				10	
	100				5538	224		43.7				100	
	500				5527	245		43.8				121	
CSD	1	1182	461	566	133544	257	205416	1.5	380	71872	0.46	1	24.6
	10				14505	248		14.2				10	
	100				5439	222		37.8				100	
	500				5439	257						138	
CSWD	1	471	113	183	43136	81	66727	1.5	93	23591	0.45	1	9.5
	10				6130	80		10.9				10	
	100				4548	79		14.7				50	
CSD	1	452	119	178	41994	71	65060	1.5	95	23066	0.45	1	9.4
and	10				5995	69		10.9				10	
CSWD	100				4475	69		14.5				41	

Opt	No Procs	Dataflow nodes Tot	Swtch	Dup	Steps	Token Queue Max	Tot	Ave/Step	Matching/Store Max	Tot	Bypass Ratio	Parallelism Max	Ave
Example 13													
None	1	1223	388	547	17823	548	27791	1.6	603	9968	0.44	1	58.3
	10				1852	544		15.0				10	
	100				362	528		76.8				100	
	500				306	478		90.8				360	
CSD	1	525	165	220	7502	211	11815	1.6	290	4313	0.43	1	28.1
	10				825	210		14.3				10	
	100				276	207		42.8				100	
	500				267	202		44.4				147	
CSWD	1	530	50	192	5350	164	7967	1.5	173	2617	0.51	1	19.7
	10				609	161		13.1				10	
	100				271	161		29.4				100	
	500				271	164		29.4				106	
CSD	1	286	55	89	3869	112	5830	1.5	139	1961	0.49	1	16.8
and	10				462	109		12.6				10	
CSWD	100				230	112		25.3				76	

Opt	No Procs	Dataflow nodes Tot	Swtch	Dup	Steps	Token Queue Max	Tot	Ave/Step	Matching/Store Max	Tot	Bypass Ratio	Parallelism Max	Ave
Example 14													
Version A													
None	1	1012	225	407	6061	224	8404	1.4	290	2343	0.61	1	46.3
	10				618	222		13.6				10	
	100				136	224		61.8				100	
	500				131	219		64.2				161	
CSD	1	337	73	120	1622	59	2239	1.4	79	617	0.62	1	14.4
	10				183	58		12.2				10	
	100				113	55		19.8				43	
CSWD	1	593	19	194	2217	105	3017	1.4	147	800	0.64	1	23.6
	10				240	104		12.6				10	
	100				94	104		32.1				76	
CSD	1	225	20	61	717	36	966	1.3	42	249	0.65	1	7.6
and	10				105	34		9.2				10	
CSWD	100				94	34		10.3				27	

Opt	No Procs	Dataflow nodes Tot	Swtch	Dup	Steps	Token Queue Max	Tot	Ave/Step	Matching/Store Max	Tot	Bypass Ratio	Parallelism Max	Ave
Version B													
None	1	625	100	232	4994	161	6953	1.4	270	1959	0.61	1	36.5
	10				517	159		13.4				10	
	100				137	153		50.8				100	
	500				137	153		50.8				108	
CSD	1	574	86	208	4964	164	6909	1.4	272	1945	0.61	1	36.2
	10				514	162		13.4				10	
	100				137	156		50.4				100	
CSWD	1	459	19	147	2447	88	3333	1.4	161	886	0.64	1	21.3
	10				266	85		12.5				10	
	100				115	86		29.0				59	
CSD	1	433	18	135	2448	86	3333	1.4	163	885	0.64	1	21.3
and	10				266	85		12.5				10	
CSWD	100				115	84		30.0				58	

Opt	No Procs	Dataflow nodes Tot	Swtch	Dup	Steps	Token Queue Max	Tot	Ave/Step	Matching/Store Max	Tot	Bypass Ratio	Parallelism Max	Ave
Version C													
None	1	461	105	176	2222	86	3066	1.4	97	844	0.62	1	19.0
	10				237	84		12.9				10	
	100				117	84		26.2				75	
CSD	1	339	73	121	1626	58	2243	1.4	76	617	0.62	1	14.4
	10				182	56		12.3				10	
	100				113	54		19.8				43	
CSWD	1	283	19	84	884	44	1190	1.3	52	306	0.65	1	9.7
	10				114	42		10.4				10	
	100				91	41		13.1				30	
CSD	1	226	20	61	719	35	968	1.3	41	249	0.65	1	7.6
and	10				105	35		9.2				10	
CSWD	100				94	34		10.3				28	