A DATAFLOW GRAPH CODE GENERATOR

Ron Nieuwenhout

RUU-CS-86-13

September 1986

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands

# A DATAFLOW GRAPH CODE GENERATOR

Ron Nieuwenhout

Technical Report RUU-CS-86-13

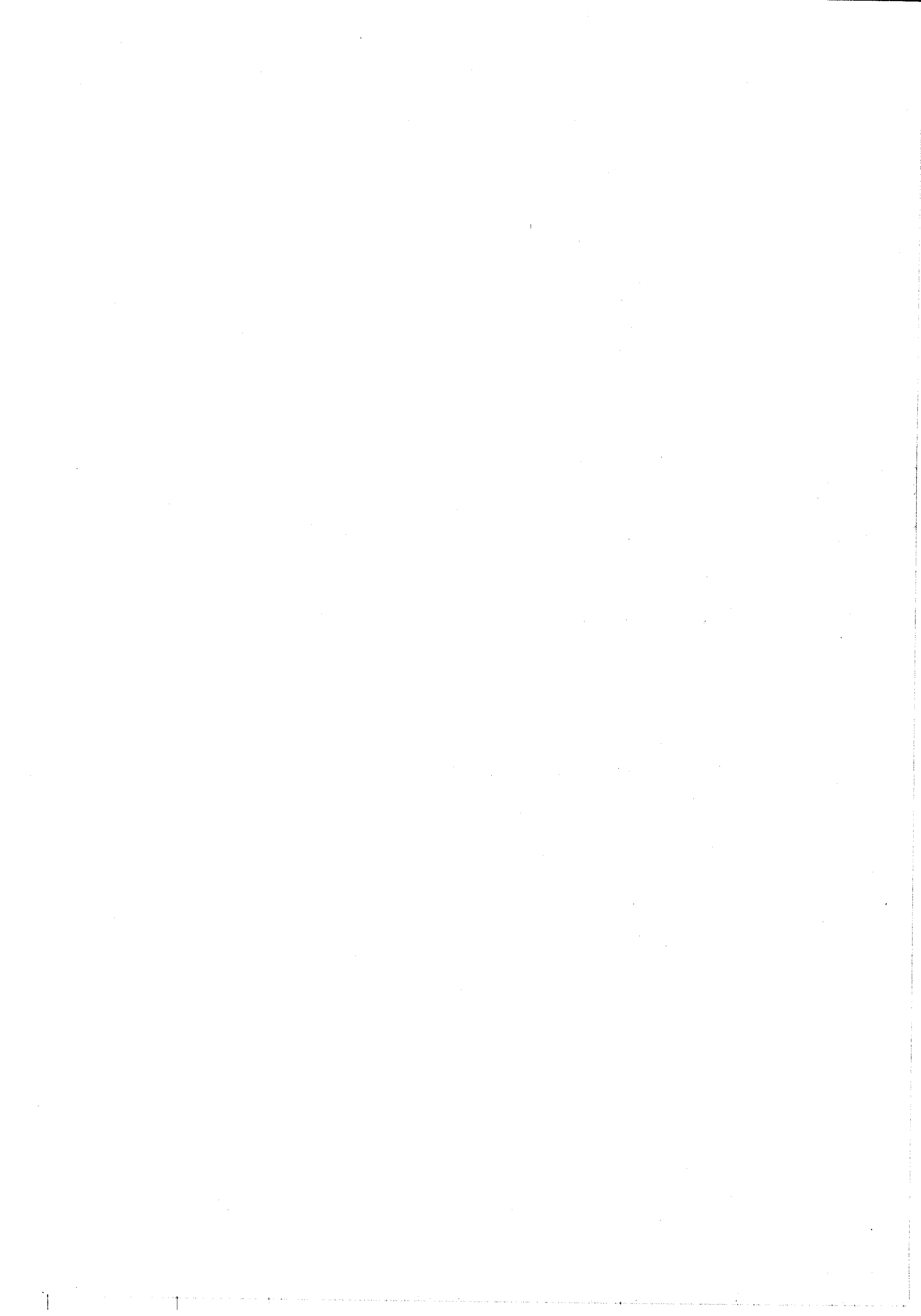
September 1986

Department of Computer Science

University of Utrecht

P.O.Box 80.012, 3508 TA Utrecht

The Netherlands



# ABSTRACT

This report presents the design of a code generator. The code generator is part of a compiler, which consists of a machine independent front-end, a machine and language independent global optimizer, and a language independent back-end. The intermediate representation, which is the output of the front-end and the input of the back-end, is a construction of dataflow graphs. The topics of this report are the intermediate representation and the back-end. Of the front-end, only those parts are discussed, which are related to the intermediate representation or to the back-end. A global optimizer, which optimizes the intermediate representation, is discussed. The back-end is a table-driven, optimizing, retargetable code generator, called the Dataflow Graph Code Generator. Also a brief introduction to the theory of dataflow graphs is presented.

# CONTENTS


| | | |
|---|---|---|
| **CHAPTER 1** | **INTRODUCTION** | **1** |
| 1.1. | Motivation | 2 |
| 1.2. | The Dataflow Graph Code Generator | 2 |
| 1.3. | Background | 3 |
| 1.4. | Outline of this report | 4 |
| **CHAPTER 2** | **DATAFLOW GRAPHS** | **7** |
| 2.1. | Graph terminology | 7 |
| 2.2. | Representation | 7 |
| 2.3. | The set of operators | 8 |
| 2.4. | Constructions | 12 |
| 2.4.1. | Conditional constructions | 12 |
| 2.4.2. | Loop constructions | 12 |
| 2.4.3. | Procedures | 12 |
| 2.5. | Dataflow machines | 18 |
| **CHAPTER 3** | **THE FRONT END** | **19** |
| 3.1. | The translation | 19 |
| 3.2. | Restrictions on the source language | 19 |
| 3.3. | Flow analysis | 20 |
| 3.4. | Type indication | 20 |
| 3.4.1. | Data-types of operands | 22 |
| 3.4.2. | Data-types of operators | 22 |
| 3.4.3. | Conversions | 22 |
| 3.4.4. | Example | 23 |
| 3.4.5. | Exceptional languages | 26 |
| 3.5. | The optimizations | 26 |
| 3.5.1. | Conditional expression reordering | 26 |
| 3.5.2. | Value propagation | 27 |
| 3.5.3. | Dead statement elimination | 27 |
| 3.5.4. | Constant folding | 28 |
| 3.5.5. | Constant propagation | 28 |
| 3.5.6. | Copy propagation | 28 |
| 3.5.7. | Copy retreat | 30 |

| | | |
|---|---|---|
| **CHAPTER 4** | **THE INTERMEDIATE REPRESENTATION** | **33** |
| 4.1. | Representation aspects | 34 |
| 4.1.1. | Compound nodes | 34 |
| 4.1.2. | Multiple arcs | 34 |
| 4.1.3. | Data-driven versus demand-driven | 35 |
| 4.2. | The Dataflow Graph Representation | 35 |
| 4.2.1. | The block structure | 36 |
| 4.2.2. | The dataflow graph | 42 |
| 4.3. | The type information in the DGR | 43 |
| 4.4. | Datastructures | 44 |
| 4.4.1. | Array | 44 |
| 4.4.2. | Set | 45 |
| 4.4.3. | Record | 45 |
| 4.4.4. | File | 45 |
| 4.5. | Pointers | 48 |
| 4.6. | Procedures | 48 |
| 4.6.1. | User defined procedures | 48 |
| 4.6.2. | External procedures | 48 |
| 4.6.3. | Permanent procedures | 49 |
| 4.7. | The dataflow in the DGR | 50 |
| **CHAPTER 5** | **THE GLOBAL OPTIMIZER** | **53** |
| 5.1. | Constant evaluation | 53 |
| 5.2. | Algebraic simplification | 54 |
| 5.3. | Common subexpression elimination | 54 |
| 5.4. | Dead variable elimination | 55 |
| 5.5. | Dead statement elimination | 56 |
| 5.6. | Invariant expression elimination | 56 |
| 5.7. | Strength reduction | 57 |
| 5.8. | Test replacement | 57 |
| 5.9. | Unrolling | 58 |
| 5.10. | Loop fusion | 58 |
| 5.11. | Cross jumping | 59 |
| 5.12. | Inline substitution | 60 |
| 5.13. | Is optimization cost-effective? | 62 |
| **CHAPTER 6** | **THE BACK END** | **63** |
| 6.1. | General description | 63 |
| 6.1.1. | The modules | 63 |
| 6.1.2. | The tables | 63 |
| 6.1.3. | The Cost Function | 64 |
| 6.2. | The Graph Transformer | 65 |
| 6.2.1. | The Transformed Dataflow Graph Representation | 66 |
| 6.3. | The Pattern Matcher | 66 |
| 6.3.1. | The order of covering | 67 |
| 6.3.1.1. | The exhaustive search | 67 |
| 6.3.1.2. | The machine dependent method | 67 |
| 6.3.1.3. | The top-down method | 67 |
| 6.3.1.4. | The bottom-up method | 68 |
| 6.3.1.5. | The bootstrapping method | 69 |

| | | |
|---|---|---|
| 6.3.1.6. | Backtracking | 69 |
| 6.3.2. | The matching of a node | 70 |
| 6.3.3. | The Covered Dataflow Graph Representation | 71 |
| 6.4. | The Instruction Generator | 71 |
| 6.4.1. | Register assignment | 75 |
| 6.4.1.1. | Description of the register table | 75 |
| 6.4.1.2. | Register replacement policy | 75 |
| 6.4.1.3. | The validity of the register table | 76 |
| 6.4.1.4. | Other register assignment methods | 76 |
| 6.4.2. | Condition codes | 77 |
| 6.4.2.1. | Description of the condition code table | 77 |
| 6.4.2.2. | The validity of the condition code table | 78 |
| 6.4.3. | Code generation | 78 |
| 6.4.3.1. | Code generation and register assignment | 78 |
| 6.4.3.2. | Code generation and construction blocks | 79 |
| 6.4.3.3. | Code generation and condition codes | 80 |
| 6.4.3.4. | The information transformation | 80 |
| 6.4.3.5. | Three and two operand instructions | 81 |
| 6.4.3.6. | User-defined procedures | 81 |
| 6.4.4. | Storage allocation | 82 |
| 6.5. | The Peephole Optimizer | 82 |
| **CHAPTER 7** | **THE MACHINE DESCRIPTION** | **85** |
| 7.1. | The machine architecture | 85 |
| 7.2. | The addressing modes | 85 |
| 7.3. | The machine instruction set | 86 |
| 7.4. | The library of runtime routines | 86 |
| **CHAPTER 8** | **SUMMARY AND FUTURE RESEARCH** | **87** |
| 8.1. | Summary | 87 |
| 8.2. | Future research | 88 |
| 8.2.1. | The DGR-language | 88 |
| 8.2.2. | The Graph Transformer | 88 |
| 8.2.3. | Implementation of the tables | 88 |
| 8.2.3.1. | The sequential approach | 88 |
| 8.2.3.2. | The procedural approach | 89 |
| 8.2.3.3. | A fast implementation of the tables | 89 |
| APPENDIX A | THE SYNTAX FORMAT | 91 |
| APPENDIX B | DGR-LANGUAGE | 93 |
| APPENDIX C | SYNTAX OF THE TMD | 99 |
| APPENDIX D | THE MACHINE ARCHITECTURE | 103 |
| APPENDIX E | ADDRESSING MODES OF THE VAX11 | 105 |
| APPENDIX F | MACHINE INSTRUCTIONS OF THE VAX11 | 113 |
| APPENDIX G | EXAMPLE | 115 |
| REFERENCES | | 125 |

*Note:* This report forms the author's Master Thesis at the Department of Computer Science, University of Utrecht.

## Acknowledgement

The author wishes to thank Professor S. D. Swierstra for his motivation and encouragement during the writing of this report.

# CHAPTER 1
# INTRODUCTION

A compiler is a program in the computersystem that takes as input source programs, written in a specific programming language (e.g. Pascal), and translates them into machine code programs. Those machine code programs can be executed on a specified machine.

Each compiler is related to just one programming language and to just one object machine. The proliferation of programming languages and machine architectures has created the need for compilers which can easily be adapted to new programming languages or to new machines. Nowadays most compilers consist of a front-end and a back-end.

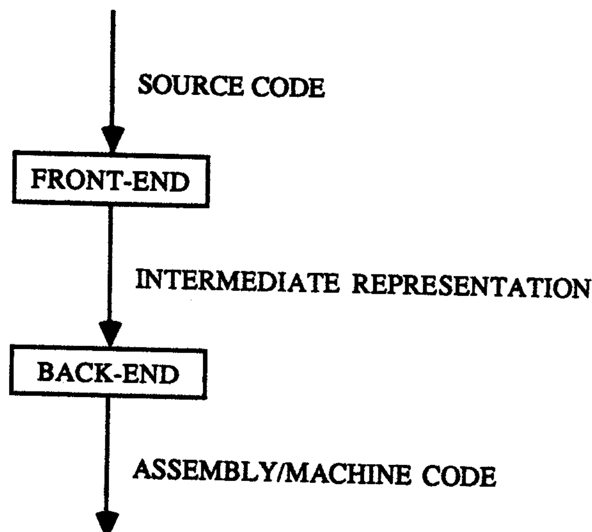



Figure 1.1 The modules of a general compiler

The front-end translates source code, written in a specific language, into an intermediate representation. The back-end translates the intermediate representation into assembly language or machine code. The idea is that the front-end is language dependent but machine independent, and that the back-end is machine dependent but language independent. The intermediate representation should be both language and machine independent. This approach makes it possible to produce a compiler for a different target machine simply by writing a new back-end. By writing a new front-end, a compiler for an other programming language can be produced. Suppose there are M different machines and P different programming languages available. Instead of writing $M \times P$ different compilers, it is possible to write P front-ends and M back-ends and then combine these together into $M \times P$ compilers. Because the back-end generates the object code, some people speak of code generator instead of back-end. We will use both phrases in this report.

## 1.1. Motivation

Most intermediate representations have been designed to enable a compiler to be bootstrapped quickly onto a new machine, either by interpreting the intermediate code or by using a macro generator to expand it into machine code, as for example in [Brow74]. Considerations of portability and machine-independence have caused the problems of optimization to be overlooked. Therefore, the compilers using these intermediate representations produce rather poor object code. There is need for an intermediate representation, which supports both retargetability and optimizations. With such an intermediate representation, an optimizing code generator, which is easily retargetable to different machines, can be produced. The emitted code should be of a high quality, having undergone several optimizations.

The proliferation of programming languages and machine architectures has not only created the need for compilers which can easily be adapted, but has also created the need for automating the code generation phase in compilers. A classification of automated retargetable code generation techniques in general, and a survey of the work on these techniques are presented in [Gana82]. A significant breakthrough in the automating of code generators was made by Glanville and Graham (see section 1.3). Their approach is not a complete solution, although it is a big step toward automating the code generator phase. In the Dataflow Graph Code Generator a method, related to the method of Glanville and Graham, is used. Retargetable code generation is achieved by separating the code generation algorithm from the target machine data, that drive the algorithm. Such a system is also a good experimental environment for architecture development. The architecture designer can make modifications to the machine description and observe the effects on the quality of the generated code, the compilation time etc.

## 1.2. The Dataflow Graph Code Generator

The compiler system has been designed to accept programming languages of the ALGOL variety, like Pascal, and to generate code for target machines with von Neumann architectures. More exotic programming languages and machine architectures have not been investigated. They are left for future research. The main emphasis, in the system, is placed on retargetable code generation and code optimization.

The compiler system consists of a machine independent front-end, a language and machine independent global optimizer, and a language independent back-end, called the Dataflow Graph Code Generator.

The intermediate representation is a construction of dataflow graphs, called the Dataflow Graph Representation (DGR). In contrast to the sequential control and memory cells semantics of the von Neumann model, the dataflow model of computation is based on *asynchronicity* and *functionality*. Asynchronicity means that all the operations execute when and only when all the required operands are available. Functionality means that all the operations are functions, that is, there are no side-effects. This implies that any two enabled operations can be executed in either order or concurrently. However, only code generators for sequential machines are discussed in this report.

The global optimizer transforms the DGR-code to better DGR-code. In optimizations there is sometimes a speed/size conflict. The user has to specify his wishes, for example

speed : size = 60 : 40

which gives the speed-optimization a higher priority than size-optimization.

The table-driven back-end translates the intermediate representation into assembly code. Assembly code was chosen instead of the more obvious machine code, for the following reasons:

- assembly code is easier to read, which is important when checking the correctness of the object code or when comparing the quality of the object code.
- we think of assembly as some intermediate representation, therefore the assembler should be a separate module.
- it is probably somewhat easier to construct a peephole optimizer for assembly code than it is for

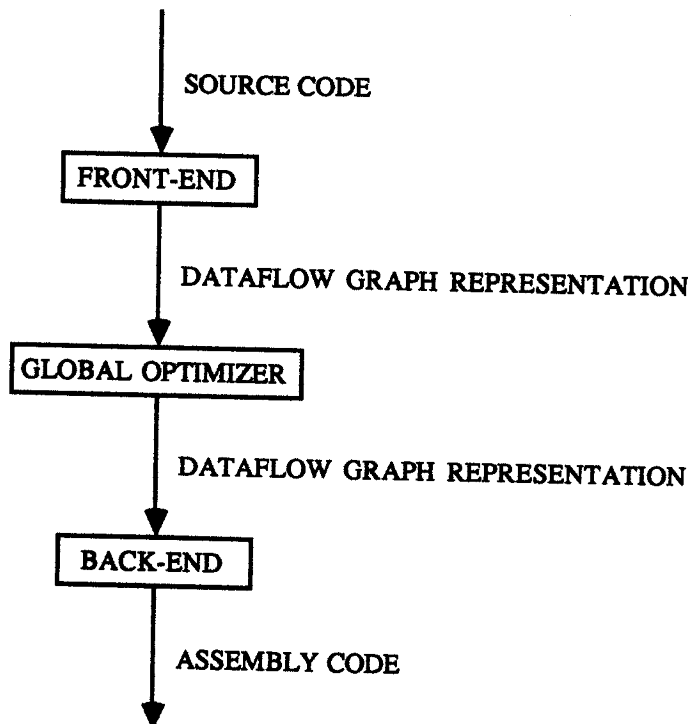



**Figure 1.2** The modules of the compiler, presented in this report

machine code.
- assemblers can easily be constructed out of assembler generator systems, like GEN in [John77].

The back-end consists of four modules all of them using a target machine description and a pattern matcher, which makes the system easier to use and understand. The four modules are:
- the Graph Transformer, which performs a transformation on the dataflow graphs;
- the Pattern Matcher, which covers the intermediate representation with subgraphs representing addressing modes and machine instructions;
- the Instruction Generator, which generates an assembly code sequence, representing the intermediate representation;
- the Peephole Optimizer, which performs peephole optimizations on the assembly code.

As an example, a back-end for the VAX-architecture[1] is discussed.

## 1.3. Background

The front-end process and the back-end process are both translations, so similar techniques ought to be applicable. The use of grammars is central in the front-end and so it should be in the back-end. Robert Glanville was the first one to use this approach. This approach can take advantage of the research into analysis and parsing (i.e. correctness proof of the parser, ease to use, etc.). In his thesis *A machine independent algorithm for code generation and its use in retargetable compilers* (1977), Glanville describes his method, that nowadays is called *The Graham-Glanville Method*.

The Graham-Glanville method is a table-driven pattern-matching approach to code generation. It is used in the construction of retargetable compilers, which consist of a front-end, an intermediate representation and a back-end. Retargetable code generation is achieved by separating the code generation algorithm from the target machine data that drive the algorithm. Glanville chooses a very

[1] VAX is a trademark of the Digital Equipment Corporation.

low level intermediate representation in the form of Polish prefix expressions. It is assumed that storage allocation and binding are done in the front-end. (So, the intermediate representation contains assumptions about the addressing structure of the target machine and therefore the code generator is not completely portable.)

The back-end, or the code generator, is automatically constructed from a target machine description grammar, using a modified LR(1) parsergenerator. The instructions of the target machine are described as grammar productions. Such a production consists of a left-hand side, specifying the result of an operation, and a right-hand side, specifying the operation in prefix form. Each production is accompanied by an assembler instruction, which computes the right-hand side. For example, the production

r.0 → + r.0 k=1 "INCR r0"

specifies that an addition of the value one to a register, with the sum going to that same register, can be obtained by the INCR instruction. Sometimes semantic restrictions, such as a constant required to have value one in the example above, may not be satisfied by any production in the set of possible reductions in a particular state. In such cases, default instructions are needed to prevent the code generator from blocking for a valid input.

The intermediate representation code is parsed according to the context-free grammar and the appropriate assembler instructions are emitted. The grammar is almost always ambiguous, because multiple matches will produce shift-reduce and reduce-reduce conflicts. These are resolved in favour of the production with the longer right-hand side.

For each triple <opcode, data-type, addressing mode> there must be a separate production. Therefore the grammar can become very large: a grammar for the VAX11 consists of over 8 million productions (see [Lune83]).

The quality of the generated code is rather poor, because the method uses limited context and does not perform machine dependent optimizations. Because of the limited context information, the quality of the code is dependent on the form of the intermediate representation generated by the front-end. For example, if in an addressing context, explicit addition is performed in the intermediate representation, then there will be an explicit addition in the generated code instead of using for example indexing ([Gana82]). Glanville ignored all machine dependent optimization issues, there is no redundant load/store elimination, no use of autoincrement instructions, etc.

More on the Graham-Glanville method can be found in, for example [Glan78] and [Grah80]. Glanville's work was extended in 1980 by Mahadevan Ganapathi, who included attributes in the scheme. For a brief overview of Ganapathi's extension see [Lune83], it also contains an overview of Glanville's method. Previous research in this area can be found in [Grah80], [Bird82], [Craw82], [Gana82], [Land82], [Grah84] and [Gana85].

## 1.4. Outline of this report

Chapter 2 contains an introduction to the theory of dataflow graphs. We present this theory by giving some examples and discussing some problems. At the end of the chapter a brief description of a dataflow machine is presented.

Chapter 3 presents a discussion on the tasks of the front-end as far as they are related to the intermediate representation or to the back-end. The source language is not specified.

In chapter 4 the intermediate representation is presented. We describe a representation of dataflow graphs, as it should be produced by the front-end.

Chapter 5 presents the Global Optimizer, which optimizes the intermediate representation, so it comes between the front-end and the back-end.

In chapter 6 the back-end is presented. The back-end consists of four modules: the Graph Transformer which performs transformations on the intermediate representation; the Pattern Matcher which covers the intermediate representation with subgraphs representing addressing modes and machine instructions; the Instruction Generator which generates an assembly code sequence; and the

Peephole Optimizer which optimizes the assembly code sequence.

Chapter 7 presents the target machine description which specifies the types and accessibility properties of data on the machine and the properties of the machine instructions.

Chapter 8 presents a summary of this report and a list of areas which are left for future research.

In the appendices the syntax format used in this report, the DGR-language, the target machine description and an example are presented.

# CHAPTER 2
# DATAFLOW GRAPHS

In this chapter we discuss briefly the theory of dataflow graphs, give some examples, discuss some problems and give a brief description of a dataflow machine. Dataflow graphs are interesting because they are the best way to represent the data-dependencies in a computation. Not everything presented in this chapter is essential for understanding the Dataflow Graph Code Generator. This chapter is inserted particulary for the reader, unfamiliar with dataflow graphs. We start this chapter with some definitions.

## 2.1. Graph terminology

A *directed graph* is a pair (N,A), where N is a finite nonempty set of nodes and A is a relation on N. A pair (x,y) from A is called an *arc* from node x to node y. We call node x a *predecessor* of node y and node y a *successor* of node x. PRED(y) is the set of predecessors of y and SUCC(x) is the set of successors of x. A node without a predecessor is called a *source* and a node without a successor is called a *sink*. All other nodes are called *interior*. A *path* is a finite sequence of two or more nodes x1,x2,...,xk (k>1), such that there is an arc (xi,xi+1) in A for all $1 \leq i \leq k-1$. If there is a path from x to y, we say x is an *ancestor* of y and y is a *descendant* of x. A *cycle* is a path x1,...,xk (k>1) in which x1 is equal to xk. A graph is called *acyclic* if it has no cycles.

## 2.2. Representation

In this section, an informal introduction to the theory of dataflow graphs is presented. A *dataflow graph* is a directed graph, with nodes representing operators and arcs representing the flow of the data. In a dataflow graph, the dataflow is made explicit. There is no explicit control flow in a dataflow graph, the control flow depends only on the available data. Because of the lack of explicit control flow, there is no explicit evaluation order, which makes a dataflow graph also useful for describing parallel evaluations. There are no variables present in a dataflow graph, so there is no explicit mentioning of memory locations, at least at an abstract level.

Figure 2.1(a) pictures an example of a dataflow graph. In this example there are two nodes, one representing the addition(+) operator and the other representing the multiply(*) operator. The small circles on the arcs indicate the tokens, which are associated with an unique identifier (i.e. a letter). A token represents a data object. The point where an arc enters a node is called an *input port*. An *output port* is the point where an arc leaves a node. We say that an input port contains a token, when the port has an arc containing a token. Each of the input ports of the + node, in the example, contains a token, so the addition operator can be executed. The execution of an operator is called the *firing* of a node. When a node fires it removes at most one token from each input port and places at most one token on each of its output ports. The output produced by a node, depends only on its input and not on some global information. This property is called the *functionality* of the dataflow graph. Figure 2.1(b) shows the situation, of the example, after the firing of the + node. The tokens a and b are removed from the input ports of the + node and a new token c is placed on its output port.

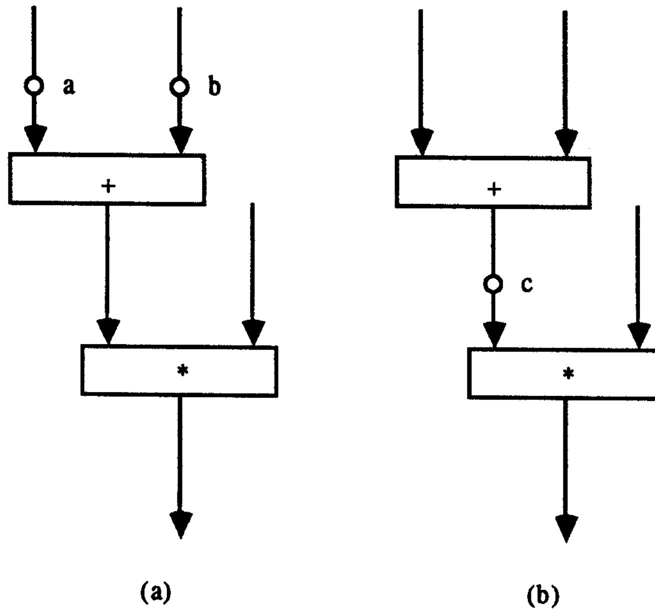



**Figure 2.1** Example of a dataflow graph

The firing of a node, as shown above, can only occur when the node is *enabled.* A node is enabled when the operation, it is representing, can be executed, that is when all the necessary tokens are available. This property is called the *asynchronicity* of the dataflow graph. The order in which the nodes fire, depends only on the flow of the data. For this reason, the execution of a dataflow graph is called *data-driven* execution.

We now give two well-known implementations of dataflow graphs. The first implementation is *the feedback interpreter,* in which all the input ports are associated with a buffer. When all the buffers of a node are filled, the node is enabled. After firing, the node clears its buffers and places a message "buffer free" on the arcs of its input ports. A node that has placed a token on one of its output ports may not use this output port until the message "buffer free" has been received. The advantage of this method is that there is always a place to store the token in and it simplifies the implementation of loop constructions, as will be shown later on. The disadvantage is that it reduces the possibility of parallel computing and all the arcs have to be half-duplex.

The second implementation is *the queued architecture,* in which the input ports of the nodes are queues. Tokens that arrive over an arc at an input port, are placed in the queue. So, more than one token may be placed on an output port. When the node fires, it takes a token out of the queue in a FIFO manner. The advantage of this method is that a node can place a token on its output port anytime, i.e. it can fire whenever its input is available.

## 2.3. The set of operators

In this section, some operators and their abstract representation are presented. Operators are instructions that implement arithmetical, logical and relational operations, and possible combinations of them.

The *duplicate* operator consumes a token from its input port and places a copy of the token on both of its output ports. Figure 2.2 shows the representation of this operator.

The *split* operator handles conditional branching. It consumes a token from its input port and places a copy of the token on the true or false output port, depending on the value of the token absorbed from the control port. Figure 2.3 shows the representation of this operator. Although in this figure, the control port is pictured on the right side of the node, it may also be pictured on the left side.

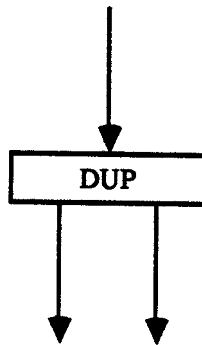



**Figure 2.2** The duplicate operator

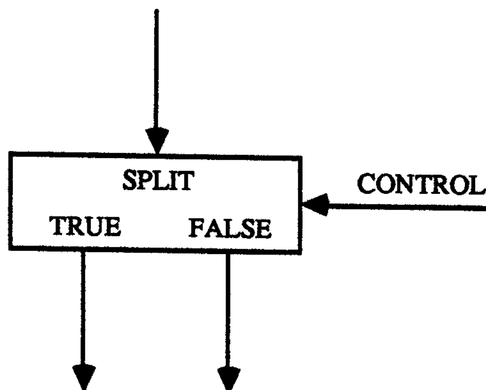



**Figure 2.3** The split operator

The *merge* operator is the opposite of the split operator. It consumes a token from its true or false input port depending on the value of the token absorbed from the control port. A copy of this token is placed on the output port. Note that the merge operator does not have a strict enabling rule, that is not all its input ports have to contain a token before the node can fire. Figure 2.4 shows the representation of the merge operator. As in the case of the split operator, the control port may also be pictured on the left side of the node.

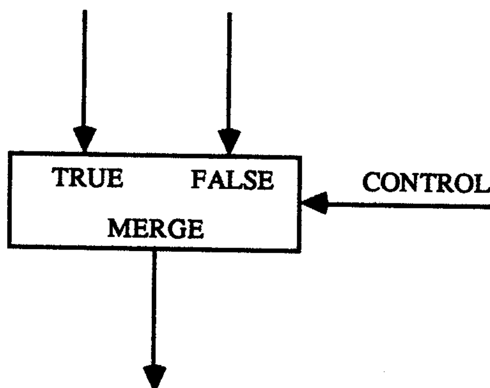



**Figure 2.4** The merge operator

The *join* operator is a non-deterministic merge operator, that is a merge operator without a control input port. The node is enabled as soon as one of its input ports contains a token. This token is consumed and a copy is placed on the output port. If both the input ports are containing a token,

then the operation is not defined. Figure 2.5 shows the representation of the join operator.

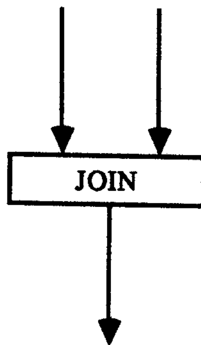



**Figure 2.5** The join operator

The *dereference* operator consumes a value, representing a pointer, and a datastructure, representing a set of values to which the pointer can refer, from its input ports and produces the value pointed to by that pointer. When the pointer is actually an address, then the datastructure is a memory section (heap in ALGOL68 terminology), the address is consumed and the value stored at that address is produced. Figure 2.6 shows the representation of dereference operator. Notice that the datastructure pointed to is explicitely passed to the dereference operator.

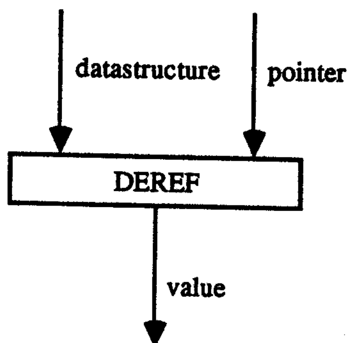



**Figure 2.6** The dereference operator

The *update* operator consumes a pointer, a datastructure and a value from its input ports and produces a datastructure which is a copy of the consumed datastructure in which the consumed value is added at a place pointed to by the consumed pointer. When the pointer is actually an address, then the datastructure is a memory section and the value is stored at that address in the memory section. Figure 2.7 shows the representation of update operator.

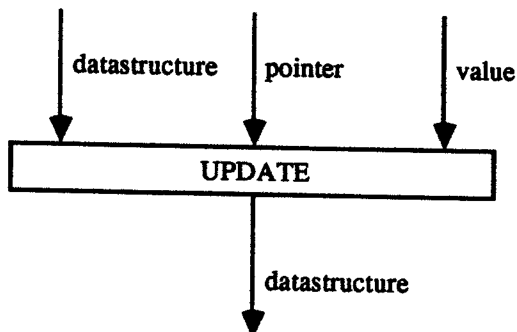



**Figure 2.7** The update operator

The *operation* operator is a standard operator, like +, *, >, sin, and, or etc. As an example, the *and* operator is described. The *and* operator consumes a token from both its input ports. These tokens are representing boolean values. The result of the logical AND operation on these tokens is placed on the output port. Figure 2.8 shows the representation of this operator.

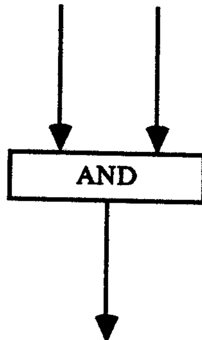



**Figure 2.8** The and operator

The *constant* operator produces a constant value. Although the constant operator is always enabled (because it has no input ports), it fires only once. Figure 2.9 shows the representation of this operator.

**Figure 2.9** The constant operator

The *halt* operator consumes all the tokens from its input port, but does not produce anything. The halt operator can be used to terminate the execution of a dataflow graph. Figure 2.10 shows the representation of the halt operator.

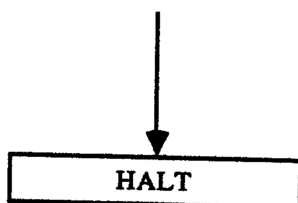



**Figure 2.10** The halt operator

It was noticed that the join operator is non-deterministic. However a deterministic behavior of the dataflow graph is prefered, even in the presence of non-deterministic operators. If a deterministic behavior is ensured, the graph is called *safe*.

## 2.4. Constructions

This section shows how some common constructions, which can be found in most programming languages, are represented in dataflow graphs. The constructions are conditional constructions, loop constructions and procedures.

### 2.4.1. Conditional constructions

Figure 2.11 shows the implementation of a conditional construction. The figure represents the dataflow graph of the conditional expression

IF y > x THEN y := y+1 ELSE x := x+1 FI

It is assumed that a token representing the value of variable y, will be placed on the arc with label y1, and a token representing the value of variable x, on the arc with label x1. After the dataflow graph is executed, the arc with label y2 contains a token representing the (new) value of variable y, and the arc with label x2 contains a token representing the (new) value of variable x. Notice that although the dataflow graph contains two non-deterministic operators, the graph is safe.

### 2.4.2. Loop constructions

When loop constructions are implemented, cycles are introduced in the dataflow graph. These cycles introduce two problems. First a constant node can only fire once, so this action can never be repeated. Secondly, different reentrances in the loop can interfere with each other. Figure 2.12 shows the representation of a loop construction in which both problems occur. The figure represents the dataflow graph of the loop construction

WHILE x < 10 DO x := x+1; y := y+x OD;

The first problem occurs in the constant-10-node and constant-1-node. It can be solved by introducing a preserve operator. The *preserve* operator consumes a token from his input port and places a copy on its output port. A second copy is placed on its own input port. The second problem can occur in the + node with label p. A new token, representing the value of variable x, may arrive at the input port of the + node before the previous one is absorbed. So this is an unsafe way to implement a loop. We will discuss briefly three methods for solving the second problem. These methods are: the lock method, the feedback interpreter system and the tagged architecture.

*The lock method:* A compound split and join node are introduced, which ensure that there will be no interference. The method is a simple and safe way to implement a loop. However it reduces the possibilities of parallel evaluation. Figure 2.13 shows this method for the example.

*The feedback interpreter system* (see also section 2.2): A way to implement this method is to add extra acknowledge arcs from the consuming node to the producing node. These acknowledge arcs ensure that no arc will ever contain more than one token and therefore the graph is safe. More on this method can be found in [Tane81].

*Tagged architecture:* A tag (or color or label) is attached to each token. The tag identifies the instance of the reentrant subgraph. A node is enabled if its input ports contain tokens with identical tags. This method is mostly used in dataflow machines.

### 2.4.3. Procedures

There are two problems concerning the implementation of procedures: we need a way for invocation and a way to return to the calling site. To solve these problems three new operators: the

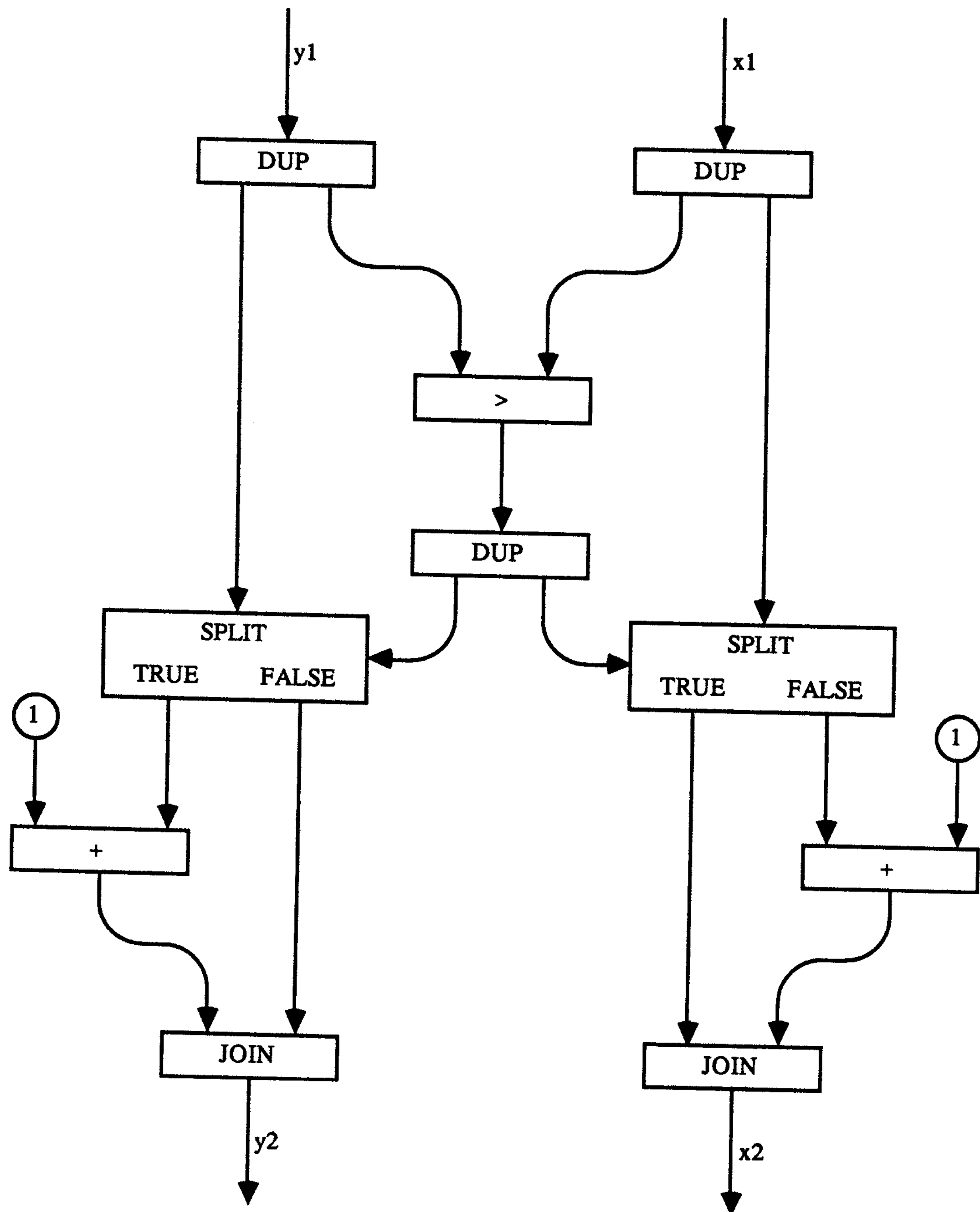



**Figure 2.11** Example of a conditional construction

call, the bag and the dyn operator, are introduced.

The *call* operator consumes a token from its input port and places a copy of this token on its output port. The call node is always a predecessor of one bag node. The call operator has an extra output port to place a token on representing an address or a label. This token is send to a dyn operator or absorbed by a halt operator. Figure 2.14 shows the representation of the call operator.

The *bag* operator consumes a token from its input port and places a copy on its output port. Different from the operators so far, this operator has one input port which may contain several arcs.

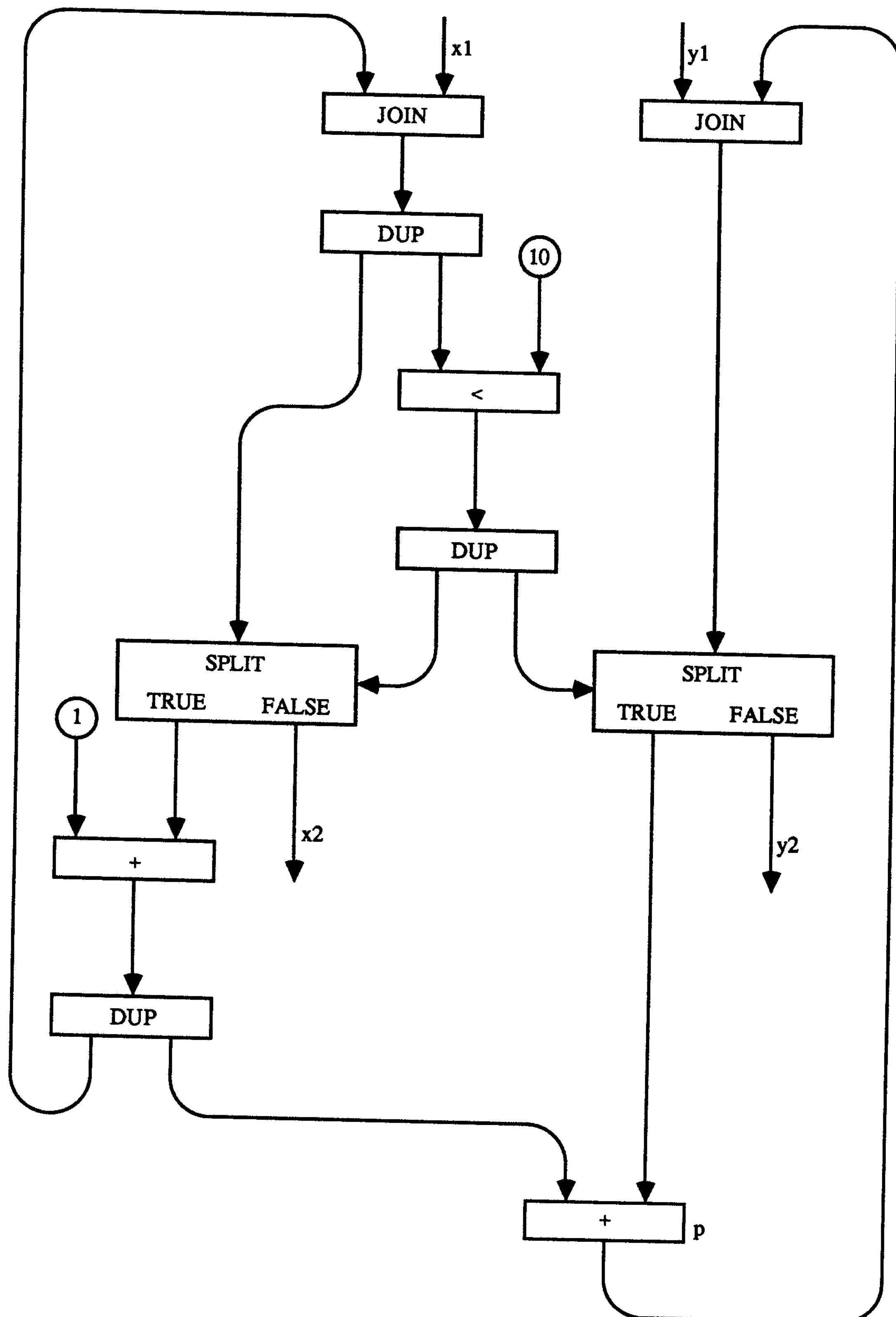



Figure 2.12 Example of a loop construction

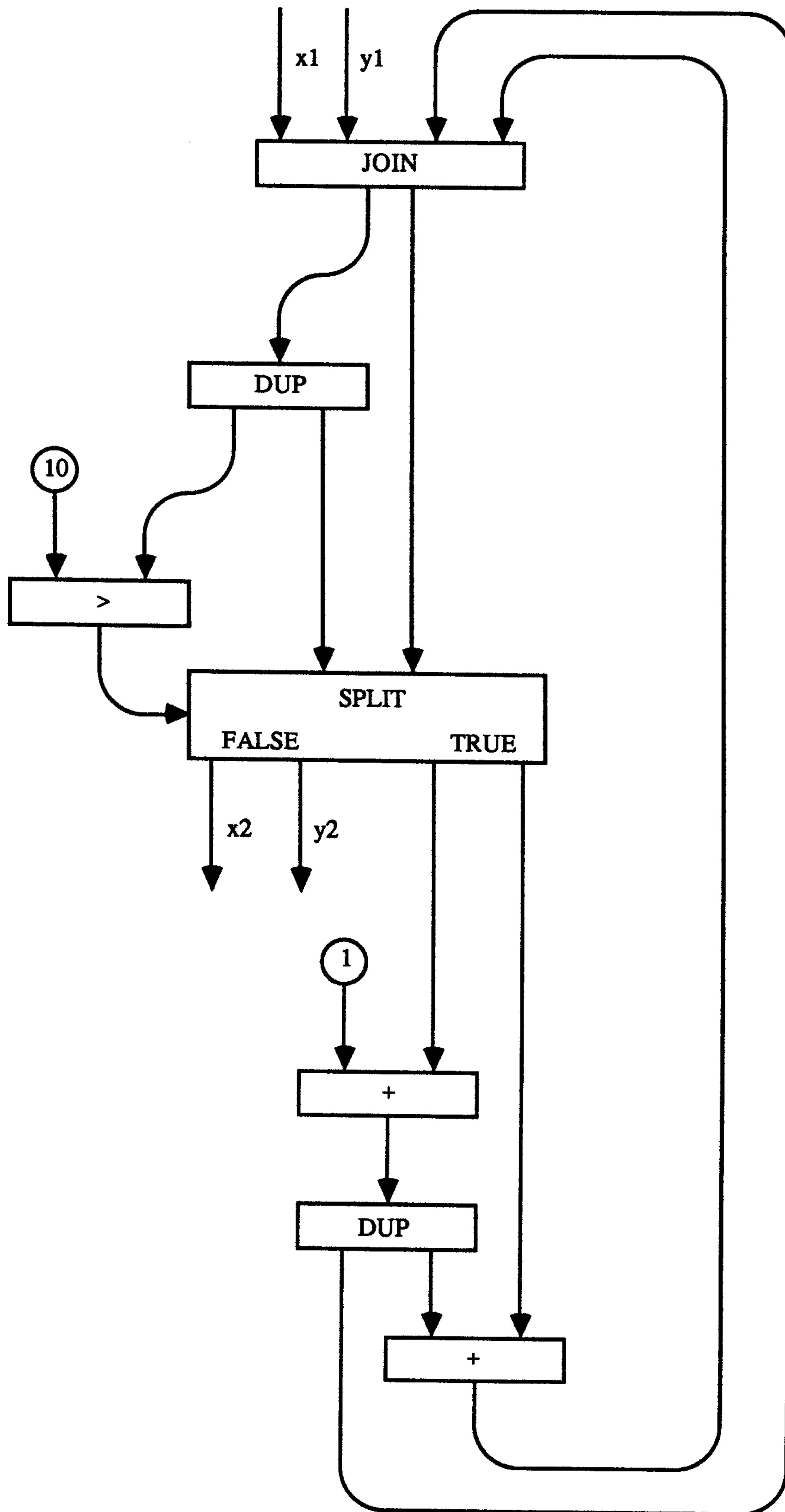



Figure 2.13 Example of the lock method

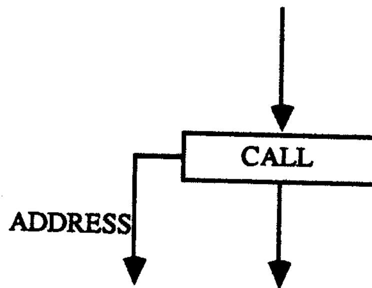



**Figure 2.14** The call operator

The bag node is always a successor of one or more call nodes. Figure 2.15 shows the representation of this operator.

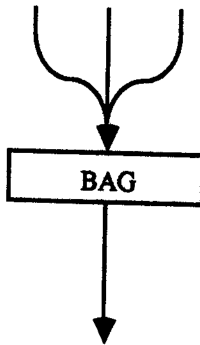



**Figure 2.15** The bag operator

The *dyn* operator consumes a token from its input port and sends a copy of the token to the address absorbed from its address port. So the arc of its output port is not static, but dynamic. Figure 2.16 shows the representation of this operator. The address port is pictured on the right side of the node, but may also be pictured on the left side.

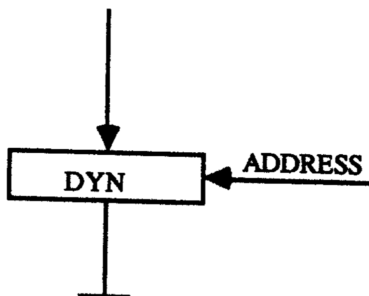
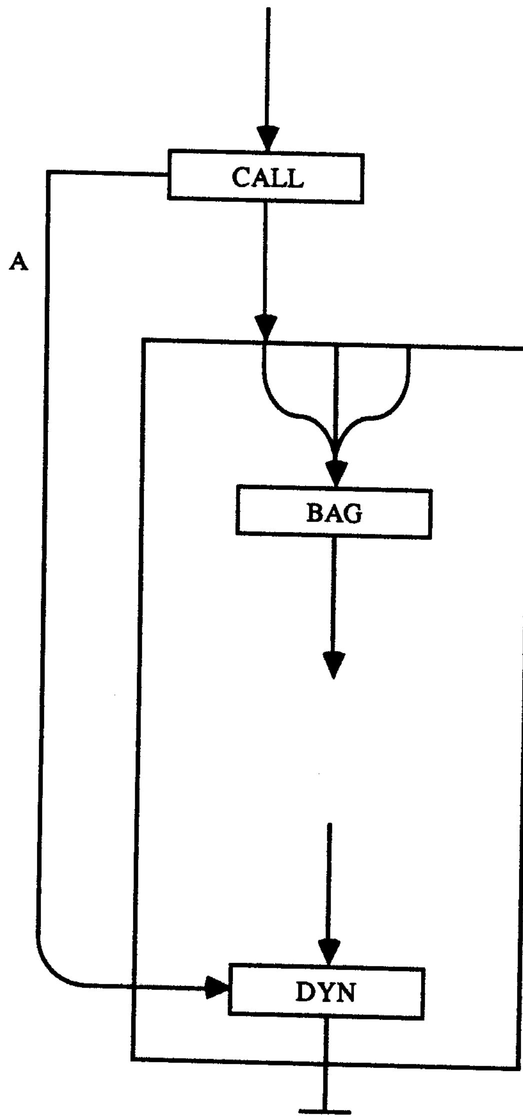



**Figure 2.16** The dyn operator

Figure 2.17 shows the implementation of a procedure invocation. The call node sends a token representing a parameter, and an extra token representing an address or label, into the procedure body. The extra token is used by the dyn operator of the procedure body to direct an output token to the proper place.

When there are no recursive or formal procedures (i.e. procedures with have another procedure as parameter), there is a much simpler method called *in line expansion*. The call node is replaced by a copy of the procedure body. However this method increases the size of the graph and the detection of no-recursion can be hard to do.



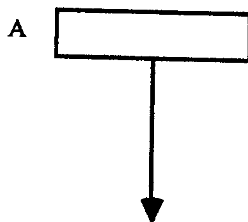




**Figure 2.17** Procedure invocations

## 2.5. Dataflow machines

In this section, a brief description of a model of a dataflow machine is presented. A dataflow machine has two major parts: the enabling unit and the functional unit.

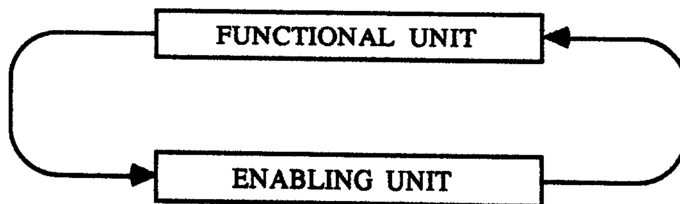



**Figure 2.18** An abstract dataflow machine

The enabling unit accepts tokens from the left and stores them in their destination templates. A template contains the operand code, a list of destination addresses and space for the input tokens. If a node is enabled, an executable packet is sent to the functional unit. After processing the packets, the functional unit sends the output tokens back to the enabling unit.

Dataflow machines are described in [Wats82] and [Denn83]. The performance of dataflow machines is investigated in [Gost80] and [Gurd83]. Several high level dataflow languages have been proposed, some of them in conjunction with the design of a dataflow machine. Examples are DFL ([Patn84]), LUCID ([Wadg85]) and FCL ([Maur83]). A dataflow interpreter is described in [Arvi82].

# CHAPTER 3
# THE FRONT END

In this chapter, we discuss briefly some parts of the front-end of the compiler. Although the front-end is not the topic of this report, we have to discuss the tasks of the front-end, which are related to the intermediate representation or to the back-end. The front-end is not discussed for a specific source language, merely a general design is presented.

## 3.1. The translation

The front-end translates the source code into an intermediate representation. In this report the intermediate representation is a dataflow graph, or to be more precise a construction of dataflow graphs (see chapter 4). It has to be sure that it is actually possible to translate a program, written in a programming language like Pascal, into a dataflow graph. Arthur Veen shows in his thesis [Veen85] that "Dataflow graphs can be generated for all kind of programs including those written in more conventional, so called imperative, languages". The techniques used to translate source language programs into dataflow graphs are similar to the methods used in conventional optimizing compilers to analyze the paths of data-dependency in source programs ([Patn84]).

There are some problems in this translation, for example:

- side-effects on global objects (a procedure[1] may change global objects)
- aliasing (one memory location can be addressed and modified through different access paths)
- multiple assignment (a variable can appear as the target of several assignments)
- exceptions (exceptions can disturb the control flow)

For the sake of compactness, the front-end translates all the FOR-loops into WHILE-loops. This has no impact whatsoever on the code to be generated.

More on the translation of source code into dataflow graphs can be found in [Maur83] and [Veen85].

## 3.2. Restrictions on the source language

The source language has to obey some restrictions. Firstly, the language definition must state that the order of evaluation of operands and the order of performance of operations, within an expression are not specified. Otherwise, a program can not be translated into a construction of dataflow graphs, because in a dataflow graph there is no explicit ordering. The programmer has to be aware of this. Suppose he wants to evaluate the expression a-b+c-d with a=10^12, b=10^12, c=0.1, d=0.1, using floating point arithmetic on a machine with less than 12 digits accuracy. When the machine evaluates the expression a-b+c-d from left to right, the result is 0.0. However, when the machine evaluates the expression as a+c-b-d from left to right, the result is -0.1.

Secondly, the source language should ban programs, which contain jumps into a loop from the

[1] Throughout this report referring to *procedures* also includes *functions*.

outside. The syntax of the source language should make these jumps impossible, otherwise the front-end has the task of enforcing this restriction.

## 3.3. Flow analysis

The front-end performs flow analysis, which is needed to translate the source code into the construction of dataflow graphs. As a matter of fact, much of the flow analysis is inherent to the translation. Flow analysis is also used to improve the efficiency of program execution, i.e. to perform optimizations (see chapter 5). Flow analysis consists of control flow analysis and data flow analysis.

Hecht says in [Hech77] that : "Control flow analysis is the encoding of pertinent, possible program control flow structure or flow of control for an ensuing data flow analysis". Control flow analysis can be hard to do, if the user deceives the control flow. A way to do this is using exception handling routines (Ada[1] , PL/I, Modpas). Another way to deceive the control flow is, for example, the use of the C library routines *setjmp* and *longjmp*, which may be used to jump out of a procedure or to create an extra entry point in a loop ([Bal85]). More on control flow analysis can be found in [Hech77].

Hecht also states in [Hech77] that : "Data flow analysis is the pre-execution process of ascertaining and collecting information about the modification, preservation, and use of quantities in a computer program". Typically, the values of variables are selected as the quantities. More on data flow analysis can be found in [Hech75], [Alle76] and [Hech77]. A mathematical view on data flow analysis can be found in [Scha73].

## 3.4. Type indication

One of the items, to be translated from the source code into the intermediate representation, is the data-type of an operator or operand. Before discussing this translation, the type indication as present in the intermediate representation is introduced. A more complete presentation is found in chapter 4.

Each operator in the intermediate representation will contain a semantic field *type* , which contains the type of the operator. Examples are +numeric, +set, =string and =numeric. Each operator also has two attributes, called *class* and *data-type*.

An operator deals with one or more classes of data-types. Those classes are found in the operators attribute *class*. For example, an operator with type +numeric, can have integer as its class of data-types, which means that its operands and its result must have data-types from the class integer. An operator with type conversion, can have, for example, integer and real as its classes of data-types, which means that the operand has a data-type from the class integer, and the result has a data-type from the class real. Other classes of data-types are presented in chapter 4.

The attribute *data-type* contains the data-types, the operator deals with. For example, if the attribute *class* of an operator contains integer, then the attribute *data-type* of that same operator must contain a data-type from the class integer, such as normal_integer, long_integer and long_long_integer. When the data-type of an operator is unknown, the attribute *data-type* simply contains the indication "unknown". Other data-types are presented in chapter 4.

Each operand[2] in the intermediate representation contains, similar to the operators, a semantic field *type*. Examples of the type of an operand are numeric, character, record, etc. Also the two attributes *class* and *data-type* are included. The attributes can contain the same items as in the case with the operators. However instead of more than one item, as in the case of the operators, the

[1] Ada is a registered trademark of the US Government (Ada Joint Program Office).

[2] From this point on, the word operand is, in this section, intended to include the word result.

attributes contain precisely one item.

Clearly, the attribute *class* is superfluous if the data-type is known (i.e. if the attribute *data-type* does not contain "unknown"). However, for the sake of consistence both attributes are present, as well in the case of an operator as in the case of an operand. In a later implementation, the two attributes should be mixed into one in order to save space.

Summarized, each operator and each operand has a specified type. Each type is associated with a set of classes of data-types. Each class of data-types contains a set of data-types. The types, their associated classes of data-types and the data-types themselves are presented in detail in chapter 4.

Figure 3.1 shows, as an example, the intermediate representation of the statement c := a+b The operator, with type +numeric, has integer as its class of data-types and long_integer as its data-type (i.e. its attribute *class* contains integer and its attribute *data-type* contains long_integer). The operands a, b and c are all of type numeric, have integer as their class of data-types and have long_integer as their data-type.

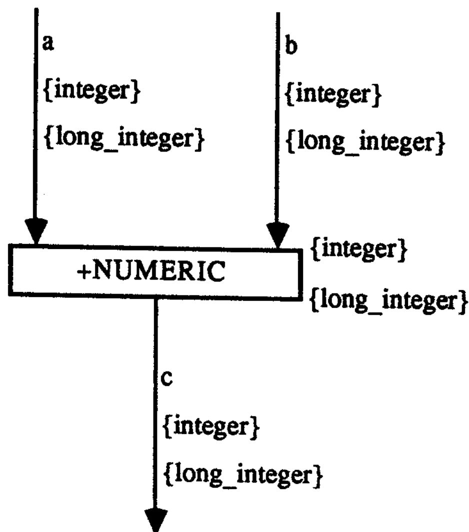



**Figure 3.1** Type indication example

The intermediate representation of the compiler has a property, called *well-defined,* which means that the following rules have to be satisfied.

1) each operator-type deals with fixed operand-types.
2) each operator-type deals with fixed classes of data-types.
3) each operator-type deals with fixed data-types.

Rule 1 means that the type of the operator has to correspond with the types of its operands. For example, the +numeric operator deals only with numeric operands.

Rule 2 means that the classes of data-types of the operator has to correspond with the classes of data-types of the operands. For example, a +numeric operator with integer as its class of data-types, deals only with operands which have also integer as their class of data-types.

Rule 3 means that the data-types of the operator has to correspond with the data-types of the operands. For example, a +numeric operator with long_integer as its data-type, deals only with operands which have also long_integer as their data-type.

Notice that rule 3 implies rule 2, and that rule 1 and the way in which each operator- and operand type is associated with a set of classes of data-types (see chapter 4), implies rule 1.

When a dataflow graph is well-defined, the attributes *class* and *data-type* of an operator are superfluous in the intermediate representation. The information stored in these attributes can also be found in the attributes of the operands of the operator.

It is a task of the front-end to perform transformations and conversions in order to produce a well-defined intermediate representation. These transformations and conversions are discussed in this section. An example is added as an illustration.

### 3.4.1. Data-types of operands

The operands in the source code have language dependent data-types, called *source data-types*, which have to be transformed to data-types of the intermediate representation, called *IR data-types*. To perform this transformation, the front-end contains a *Data-Type Transformation table* (DTT-table) which contains all the source data-types and their corresponding IR data-types. A similar transformation is performed on behalf of the classes of data-types. The classes of IR data-types and the classes of source data-types are also contained in the DTT-table. The table is made only once during the compiler construction.

In some languages, like SNOBOL4, APL/360 and SETL, variables (operands) can be *overloaded*, which means the class of data-types can vary during the same program. For example, one moment variable x is an integer and the next moment it is a character. The presence of overloaded variables increases the work (i.e. dataflow analysis) to be done in order to transform the data-types.

The intermediate representation, normally, contains operands that are not visible in the source code. For example, figure 3.5 shows the dataflow graph corresponding to the source code expression

f := e+(a+b)*(c+d)

The operands t1, t2 and t3 are called *intermediate operands*. The source data-types of intermediate operands are usually not explicitely known, but may easily be deduced.

### 3.4.2. Data-types of operators

Each operator-type in the intermediate representation, deals with fixed classes of data-types. Suppose, an operator in the source code has a class of source data-types, then this class is transformed to a class of IR data-types. The DTT-table is used to perform this transformation. When not only the class of data-types is known, but also the data-type itself, then this is inserted in the attribute *data-type*.

An *overloaded* operator is an operator that deals with a set of operand-types, instead of fixed operand-types, or with a set of classes of data-types, instead of fixed classes of data-types. Formally there are no overloaded operators, just overloaded operator-identifiers. An overloaded operator-identifier refers to a set of operators instead of just one. The front-end has to transform overloaded operators to operators which do deal with fixed classes of data-types.

Most programming languages have overloaded operators, for example, the relational operators (e.g. =) are normally overloaded. When dealing with such an operator, it is necessary to look at the operands before it is known what operation has to be performed. When dealing with the "=" operator, the method for comparing two strings will be different from the way of comparing two integers. Because of this inconvenience, the front-end replaces overloaded operators by a set of more specific operators, for example =numeric (i.e. =numeric is inserted in the field *type* of the operand), and adds the proper classes of source data-types (respectively source data-types) to the overloaded operators. After that they are transformed to classes of IR data-types (respectively IR data-types).

Another example of an overloaded operator is the + operator which, in some languages, can indicate either numeric addition, set union or tuple concatenation. When for example an integer addition is needed, the front-end replaces the + operation by a numeric addition operator and gives the operator, integer as the class of data-types.

### 3.4.3. Conversions

Each operator-type in the intermediate representation deals with fixed classes of data-types (rule 2). If one or more of the operands have a class of data-types, which differs from the class the operator deals with, then conversions are used. This means that an extra operator is inserted, which converts the operand to another operand with the correct class of data-types. Which conversions are used, to achieve a well-defined intermediate representation, depends partly on the programming

language. Algol68, for example, is defined in such a way that an integer is converted to a real, whenever needed. For example, the integer in the expression

real := integer + real

is converted into a real. Figure 3.2 shows this conversion, pictured as dataflow graphs. The classes of data-types are pictured in the figure.

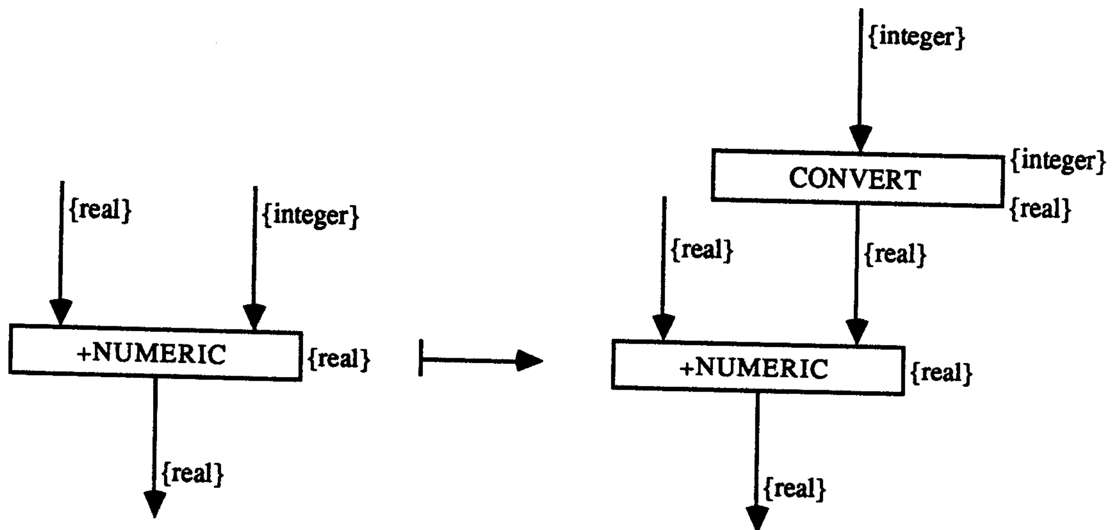



**Figure 3.2** Example of a conversion

Each operator in the intermediate representation will only receive operands of the correct data-type (rule 2). When this rule is not satisfied, conversions are used. For example, suppose an operator deals only with data-type long_long_integer and one of its operands has data-type long_integer. Then this operand is converted to an operand with data-type long_long_integer, just like the conversions discussed above.

## 3.4.4. Example

Suppose there is a (Pascal alike) programming language P, which contains a class of data-types integer, with the data-types integer and long_integer, and a class of data-types real, with the data-type real. In P, operators can be overloaded and the variables have all a fixed data-type (and thereby a fixed class of data-types). Figure 3.3 shows a part of the DDT-table belonging to P.

| source data-type | IR data-type |
|---|---|
| integer | normal_integer |
| long_integer | long_integer |
| real | long_real |

| source classes | IR classes |
|---|---|
| integer | integer |
| real | real |

**Figure 3.3** The Data-Type Transformation table for P

Figure 3.4 shows a program written in P. At this point we are only interested in the expression. Our goal is to translate the expression into a well-defined dataflow graph. The construction of the dataflow graph is, in this section, of minor importance, we are only interested in the data-type

transformations. For the sake of clarity, the expression is pictured in figure 3.5 as a dataflow graph.

```
VAR a, b, c : integer;
    d       : long_integer;
    e, f    : real
BEGIN
  read(a,b,c,d,e);
  f := e+(a+b)*(c+d)
END.
```

**Figure 3.4** Program example in P

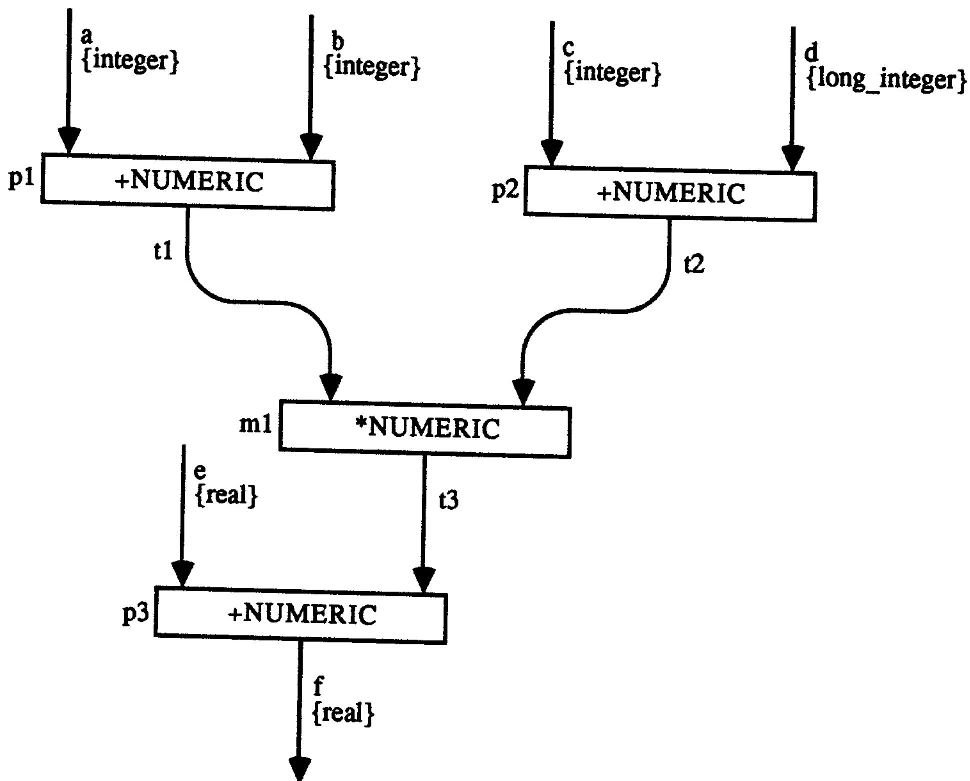



**Figure 3.5** The expression as a dataflow graph

The transformation consists of two passes: a pass to transform the classes of data-types, and a pass to transform the data-types.

The first pass is started, for example, with the operands a and b. The DTT-table gives the IR classes of operands a and b, which are both integer. As a consequence, operator p1 has IR class integer and so has its result operand t1. Obviously, the same goes for the operands c and d, the operator p2 and the result t2. The intermediate operands t1 and t2 have IR class integer. As a consequence, operator m1 has IR class integer and so has operand t3. Operands e and f have real as their source data-type, so their IR class is real. As a consequence, operator p3 has IR class real. A conversion is inserted to convert operand t3 with IR class integer to an operand t4 with IR class real.

The second pass is performed, because overflow during a computation has to be avoided. If the source data-type of an operand is known, it is easily replaced by an IR data-type, using the DTT-table. Intermediate operands do not have source data-types, they always get the largest possible IR data-type in order to avoid possible overflow. Operand f has source data-type real, the DDT-table gives the IR data-type long_real. As a consequence the IR data-type of operator p3 is also long_real.

Operand e has source data-type real, the DDT-table gives the IR data-type long_real. The intermediate operand t4 gets the largest possible IR data-type, i.e. long_real. As a consequence the IR data-types of the conversion operator c1 are long_integer and long_real. Operand t3 gets long_integer as its IR data-type and as a consequence so does the operator m1. The intermediate operands t1 and t2 get the largest possible IR data-type, i.e. long_integer. As a consequence the IR data-type of operators p1 and p2 is long_integer. Operands a, b and c have source data-type integer, which is transformed into IR data-type normal_integer, using the DTT-table. Conversions are inserted to convert the operands a, b and c to operands with an IR data-type long_integer. Operand d has source data-type long_integer, which is transformed to IR data-type long_integer. Figure 3.6 shows the intermediate representation of the expression.

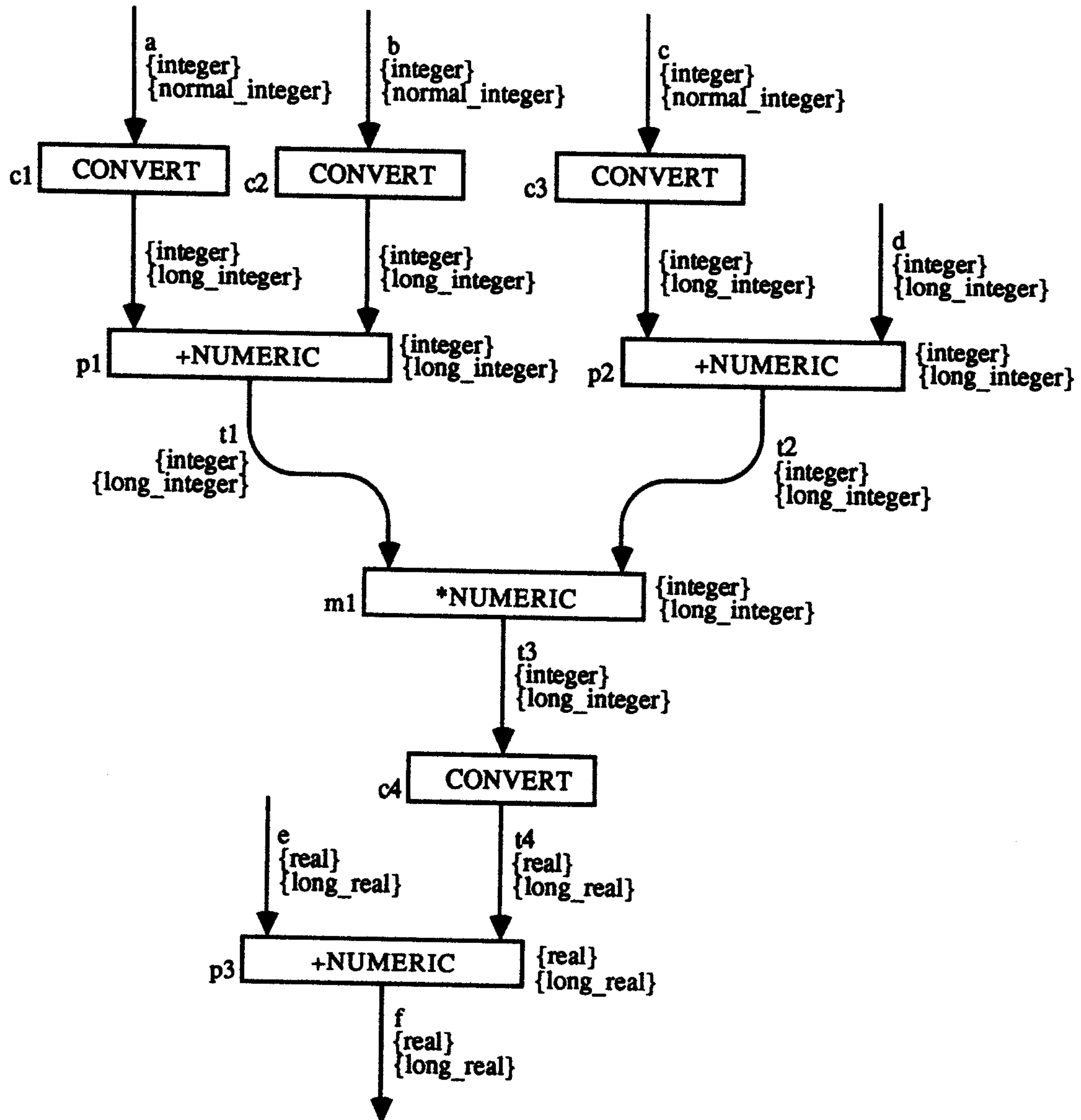



**Figure 3.6** The intermediate representation of the example

### 3.4.5. Exceptional languages

In our descriptions, we use qualified terms such as *most languages, in some languages* and so on. This is because there are languages that are rugged individualists. Such as LISP and SNOBOL4, in which we may dynamically create program text, so the program itself may not be frozen until run-time. Other exceptional languages are languages like RUSSELL and HOPE ([Harl84]), in which objects can be polymorphic. In such cases we do not have knowledge of the types of all objects, prior to run-time. Therefore, it may be necessary to determine types dynamically at run-time, slowing down execution. These exceptional languages are left for future research.

## 3.5. The optimizations

The front-end performs two types of optimizations. The first class of optimizations consists of optimization techniques, which are language dependent but machine independent. Just one optimization technique from this class is discussed, namely *conditional expression reordering*.

The second class of optimizations consists of optimization techniques, which are implicitely performed in the construction process of the intermediate representation. These optimization techniques can also be performed without the knowledge of the target machine. These implicitely performed optimization techniques are *value propagation, dead statement elimination, constant folding, constant propagation, copy propagation* and *copy retreat*. Most optimization techniques are explained via examples. These examples are Pascal programs, on which the optimization techniques are explicitely performed. For the sake of clarity, the optimized programs are also pictured as Pascal programs. This does not mean that the source programs are actually replaced by other source programs, it is merely an illustration of the optimization technique. Remember that the front-end performs these optimizations implicitely, which means that when an example program and its optimized program would both be translated into a construction of dataflow graphs, the same construction would result.

Other optimization techniques are performed in the Global Optimizer and in the back-end (peephole optimizations).

### 3.5.1. Conditional expression reordering

For the sake of clarity, this technique is explained via an example. Figure 3.7(a) shows a program fragment which can be converted to the fragment shown in figure 3.7(b). When booleanexpression1 is true, then there may be no real need to calculate booleanexpression2.

```
IF booleanexpression1            IF booleanexpression1
OR booleanexpression2            THEN  statement1
THEN  statement1                 ELSE  IF booleanexpression2
ELSE  statement2                       THEN  statement1
                                       ELSE  statement2

         (a)                              (b)
```

**Figure 3.7** Example of conditional expression reordering

Some languages however insist that both the booleanexpressions are evaluated, because booleanexpression2 can have a side-effect on global objects. When dealing with such a language, this optimization cannot be performed. In this example the optimization is performed on the source code,

so before the actual translation. It is also possible to perform the optimization on the intermediate representation, so after the translation.

The same optimization can be performed on the AND operation and combinations of the OR and AND operation.

## 3.5.2. Value propagation

A dataflow graph obviously uses value propagation, that is, it uses the value of a variable instead of the variable itself. The dataflow graph representing

y := x + 7

is pictured in figure 3.8(a). Without value propagation, the dataflow graph would be as pictured in figure 3.8(b). Value propagation can not always be used, for example in datastructures it is inconvenient, as is shown in chapter 4.

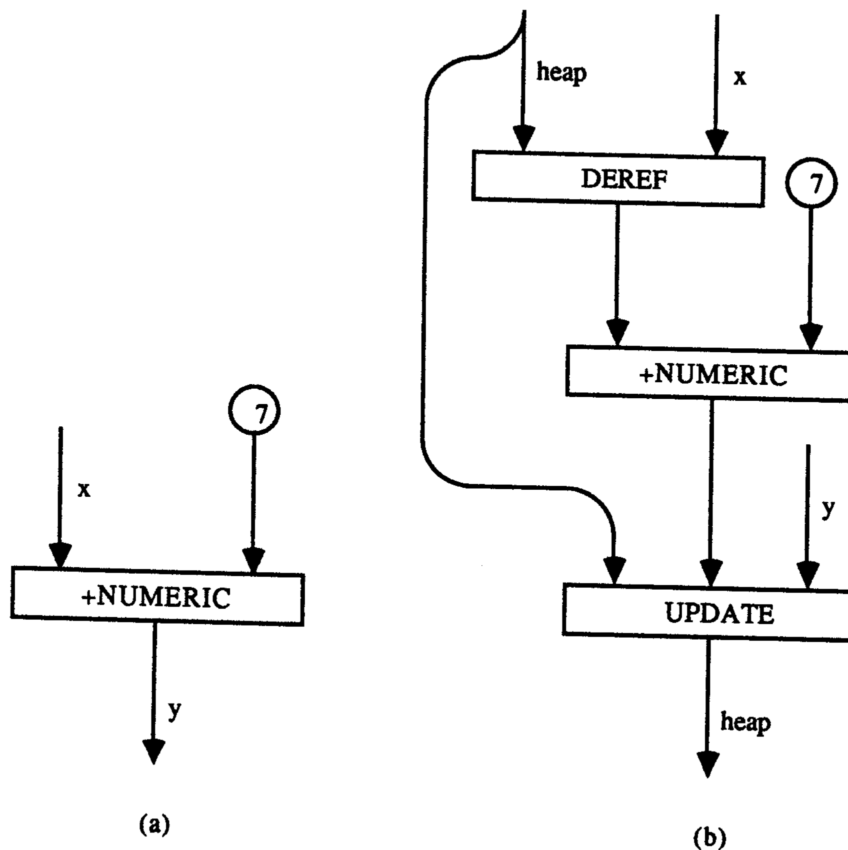



Figure 3.8 Example of value propagation

## 3.5.3. Dead statement elimination

In a structured language, like Pascal, there is no need for an optimization technique, removing unreachable and useless code, because there will (almost) never be that kind of code. In a less structured language, like FORTRAN, the programmer is forced to use lots of GOTO-statements, thereby

possibly introducing unreachable code. This optimization technique is then useful.

This optimization technique is mostly used in combination with other optimization techniques. The other optimizations can introduce unreachable and useless code (even when dealing with a structured language), which is then removed by the dead statement elimination technique. All of the following optimization techniques presented in this chapter are performed in combination with the dead statement elimination technique, as will be shown in the examples.

### 3.5.4. Constant folding

The value of a constant is known at compile-time, therefore each occurrence of a constant in the source program can be replaced by its value. This optimization technique is called constant folding. Figure 3.9 pictures a program on which this optimization is performed as an example. The constant *multiply* in line 16 is replaced by the value FALSE, and the constant *addition* in line 22 is replaced by the value TRUE. The dead statement elimination technique removes all unreachable code, that is line 17 to 21, and all useless code, that is line 2, 3, 16, 22, 23 and 27. Figure 3.10 shows the optimized program.

### 3.5.5. Constant propagation

The front-end performs, implicitely, the following form of constant propagation. Each simple variable, like an integer, real, boolean or character, which is just once assigned to with a constant value, is regarded as a constant and therefore treated like a constant in the constant folding optimization technique. Figure 3.10 pictures a program on which this optimization is performed as an example. The variable m1 in line 7 is replaced by value 1 and the variable m2 in line 12 is replaced by value 2. The useless code, that is line 6 and 11, is removed by the dead statement elimination technique. The variables m1 and m2 are removed from line 4. Figure 3.11 shows the optimized program.

### 3.5.6. Copy propagation

This optimization is best explained via the program fragment, shown in figure 3.12(a). We want to replace the x in line ii) by the y from line i), we hope that line i) becomes useless and is removed by the dead statement elimination technique. It has to be sure that the value of y and the value of x at line i) are the same as at line ii). If the variable x is used nowhere else, the assignment at line i), becomes useless and can be eliminated.

Notice that this optimization technique is only (implicitely) performed with simple assignments as at line i) in figure 3.12(a). The program fragment in figure 3.12(b) shows a more complex assignment at line i). The optimization is not performed in this case.

In case of an assignment through a pointer variable, it is in general impossible to see which variables may be affected by the assignment. In such a case, it is impossible to check whether a value has changed or not. Similar problems occur in the presence of procedure calls. In such cases this optimization is also not performed.

Figure 3.11 pictures a program on which this optimization is performed as an example. Notice that line 24 can not be replaced by

```
t := t+y;
```

because the value of t is changed in line 13. However, line 24 can be replaced by

```
t := x+t;
```

Line 15 is useless now, and is eliminated. The variable y is removed from line 4. Line 25 can be replaced by

```
1    PROGRAM mult_or_add (input,output);
2    CONST  multiply  =  FALSE;
3           addition  =  TRUE;
4    VAR m1, m2, t, x, y, z : integer;
5    BEGIN
6        m1 := 1;
7        writeln('integer',m1);
8        read(t);
9        writeln(t);
10       x := t;
11       m2 := 2;
12       writeln('integer',m2);
13       read(t);
14       writeln(t);
15       y := t;
16       IF multiply THEN
17       BEGIN
18           t := x*y;
19           z := t;
20           writeln('product',z)
21       END;
22       IF addition THEN
23       BEGIN
24           t := x+y;
25           z := t;
26           writeln('sum',z)
27       END
28   END.
```

**Figure 3.9** Optimization example

z := x+t;

and line 24 is then eliminated. Finally, line 26 is replaced by

writeln('sum',x+t)

The useless line 25 is eliminated and the variable z is removed from line 4. Figure 3.13 shows the optimized program.

```
1     PROGRAM mult_or_add (input,output);
4     VAR m1, m2, t, x, y, z : integer;
5     BEGIN
6         m1 := 1;
7         writeln('integer',m1);
8         read(t);
9         writeln(t);
10        x := t;
11        m2 := 2;
12        writeln('integer',m2);
13        read(t);
14        writeln(t);
15        y := t;
24        t := x+y;
25        z := t;
26        writeln('sum',z)
28    END.
```

**Figure 3.10** Optimization example after constant folding

### 3.5.7. Copy retreat

This optimization technique is best explained via an example. Figure 3.14 shows a fragment in which we want to replace the x in line i) by the y from line ii), we hope that line ii) becomes useless and can be removed by the dead statement elimination technique. It has to be sure that the value of x at line i) is the same as at line ii) and that x is not used after line ii). It has also to be sure that the variable y is not used between the lines i) and ii). Notice that all the x'es from line i) to ii) have to be replaced by y, and not only the x in line i).

In case of an assignment through a pointer variable or in the presence of procedure calls, the same problem occurs as in copy propagation. And as a consequence the optimization is not performed.

Figure 3.13 pictures a program on which this optimization is performed as an example. The variable t in the lines 8 and 9, is replaced by the variable x. The useless line 10 is eliminated. Figure 3.15 shows the optimized program.

```
1    PROGRAM mult_or_add (input,output);
4    VAR t, x, y, z : integer;
5    BEGIN
7        writeln('integer',1);
8        read(t);
9        writeln(t);
10       x := t;
12       writeln('integer',2);
13       read(t);
14       writeln(t);
15       y := t;
24       t := x+y;
25       z := t;
26       writeln('sum',z)
28   END.
```

Figure 3.11 Optimization example after constant propagation

```
          :                        :
 i)   x := y;            i)   x := a+b*c;
          :             ii)   y := x+1;
ii)   z := x+t;        iii)   z := x+1;
          :                        :

         (a)                      (b)
```

Figure 3.12 Copy propagation examples

```
 1   PROGRAM mult_or_add (input,output);
 4   VAR t, x : integer;
 5   BEGIN
 7       writeln('integer',1);
 8       read(t);
 9       writeln(t);
10       x := t;
12       writeln('integer',2);
13       read(t);
14       writeln(t);
26       writeln('sum',x+t)
28   END.
```

**Figure 3.13** Optimization example after copy propagation

```
       :
 i)  read(x);
       :
ii)  y := x;
       :
```

**Figure 3.14** Copy retreat example

```
 1   PROGRAM mult_or_add (input,output);
 4   VAR t, x : integer;
 5   BEGIN
 7       writeln('integer',1);
 8       read(x);
 9       writeln(x);
12       writeln('integer',2);
13       read(t);
14       writeln(t);
26       writeln('sum',x+t)
28   END.
```

**Figure 3.15** Optimization example after copy retreat

# CHAPTER 4
# THE INTERMEDIATE REPRESENTATION

An intermediate representation in a compiler provides an interface between the machine independent front-end and the language independent back-end. The idea of a standard intermediate representation suggests a way of saving effort in the production of compilers. Different compilers for the same machine could use the same code generator, and porting a compiler to another machine is easy, the development of the code generator is all that remains.

When considering the design of an intermediate representation, it is useful to visualize the representation as an abstract machine. This helps understanding the intermediate representation. There are two opposing designs; a high-level and a low-level intermediate representation. A high-level representation offers many more implementation freedom than a low-level one. When using a high-level representation however, it is harder to implement an existing compiler on a new machine, than it is when using a low-level representation.

The most widely used intermediate representations are low-level. They are comparable to assembly codes for simple abstract machines. The simplicity of these abstract machines, makes it easy to move an existing compiler to a new machine. Most of the low-level representations are designed for one special source language, like P-code for Pascal. With these low-level representations, there is generally no attempt to make them also suitable for another source language, because this involves a lot of effort.

Designing a high-level intermediate representation offers much more flexibility than designing a low-level representation. This flexibility extends both to the range of source languages, and to the variety of target machines. A disadvantage of a high-level representation is the large amount of work involved when porting a compiler to a new machine (i.e. the development of a new code generator). A way to overcome this obstacle is to develop code generators automatically from machine specifications, as is discussed in for example [Catt82].

An intermediate representation should not only support retargetability but should also support optimizations. The use of dataflow graphs as an intermediate representation is more powerful than the use of trees, since optimized code generation using graphs is usually better than optimized code generation using trees. This is mostly true when dealing, for example, with common subexpressions. In this chapter a construction of dataflow graphs is described, which is used as the intermediate representation. This construction is called the Dataflow Graph Representation (DGR), which can be regarded as some kind of abstract dataflow machine. The DGR has been designed as an intermediate representation for compilers dealing with programming languages of the ALGOL variety, like Pascal.

The construction of dataflow graphs is translated into assembly code. The dataflow graphs are not executed, as is done by dataflow machines, so there is no need for tokens representing data. As a consequence, the problems described in chapter 2, when implementing loop constructions, do not occur. As a matter of fact, the DGR is structured in such a way, that the dataflow graphs used in the DGR are always acyclic. Although only sequential machines are investigated in this report, the DGR supports parallel evaluations.

## 4.1. Representation aspects

In this section the dataflow graph theory, as described in chapter 2, is extended with compound nodes, multiple arcs and demand arcs. The compound nodes and multiple arcs are introduced, because they give a more natural representation of the source code. The demand arcs are introduced for implementation reasons only.

### 4.1.1. Compound nodes

It is natural to represent an expression such as a+b+c, with a, b, c integers, as pictured in figure 4.1(a), instead of a representation as shown in figure 4.1(b).

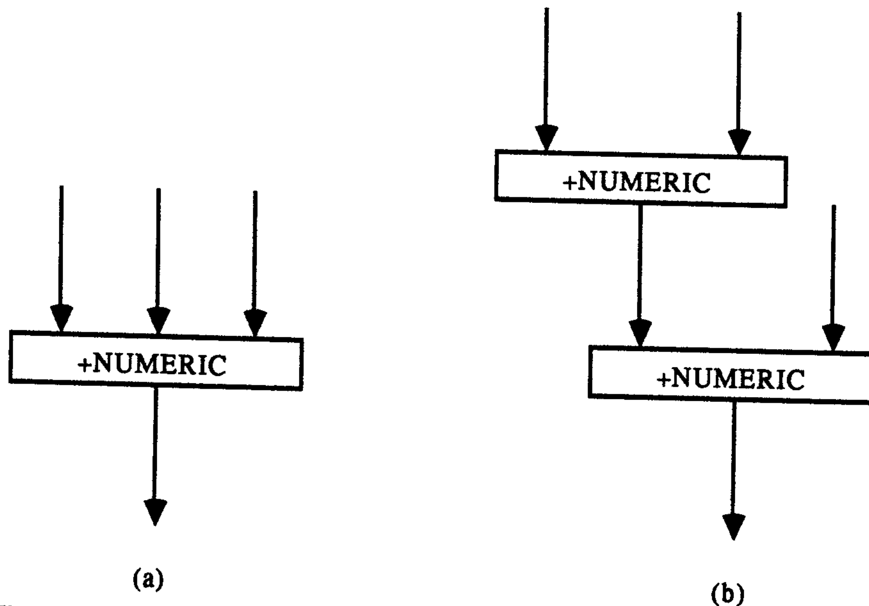



Figure 4.1 Example of a compound node

A node, as shown in figure 4.1(a), is called a *compound* node. A compound node is a node, representing an operator, with an arbitrary number of input ports. Notice that subtraction is some kind of addition, and can therefore be represented in the same compound addition node. Of course an unitary minus operator is needed. Another example of a compound node is the compound multiplication node, which represents multiplication and division operators. In this case an inverse operator is needed.

As a consequence, there is no need for subtraction and division operators in the DGR. It is assumed that, whenever possible, compound nodes are used by the front-end. So, constructions as shown in figure 4.1(b) do not occur in the DGR. However, the DGR may still contain subtraction and division operators.

Several operations which are sometimes difficult to perform on a strict dataflow graph are quite easy on a graph with compound nodes. For example, the optimization technique constant evaluation as described in chapter 5.

### 4.1.2. Multiple arcs

It is allowed that more than one arc enters an input port of a node and more than one arc leaves an output port of a node. This principle of *multiple arcs* was already used in chapter 2 concerning the BAG node. While multiple arcs are used in the DGR, the duplicate and join nodes are

superfluous. As a consequence, it is assumed that there are no DUP and JOIN nodes present in the DGR.

In the DGR construction blocks are used (see section 4.2), as a consequence of which never more than one arc will enter an input port.

### 4.1.3. Data-driven versus demand-driven

A dataflow graph, as presented this far, is *data-driven*. However it could also be *demand-driven*, the graph is then called a *demand graph*. A demand graph is structurally similar to a dataflow graph with all the arcs reversed. In this report a dataflow graph is used, which is both data-driven and demand-driven. This means that, if there is an arc between two nodes, then there is also the reversed arc between these nodes. The arcs of the data-driven dataflow graph are called just *arcs* and are pictured in the figures of this report. The arcs of the demand graph are called the *demand arcs* and are not pictured in the figures.

## 4.2. The Dataflow Graph Representation

In this section the structure of the Dataflow Graph Representation (DGR) is described. The DGR is a dataflow graph in which the nodes represents modules. These nodes are dataflow graphs themselves, and the nodes of these dataflow graphs are also dataflow graphs, and so on. For the sake of clarity, a block description is used to describe the DGR. The DGR is structured by different kind of blocks. Such a block is regarded as a function of some specified type. It consumes input data and produces output data. The function of the block is defined by the operations which are performed in the block. The operations are performed by operators which are blocks themselves. These blocks are called the *inner-blocks* of the block. The block itself is called the *outer-block* of the inner-blocks. For each block type a formal description is given. The block descriptions are part of a grammar describing the DGR-language. The complete grammar is found in appendix B. The syntax format used in this report is found in appendix A. All the block descriptions use the same formalism, which will also be used to describe strict dataflow graphs in this section, and addressing modes and machine instructions in the Target Machine Description as presented in the chapters 6 and 7.

```
<object> ::=
    <type>,<identifier>,{,<input_data>,→,<output_data>,}=
  ,[,<definition>,]
  .
```

**Figure 4.2** Description formalism

Figure 4.2 shows the description formalism. The input-data-part of the description describes all the data consumed by the block. The block consumes all the available data which is or can be used within the block. The inner-blocks of the block consume parts of the data consumed by the block, and if necessary parts of the data produced by other inner-blocks at the same level. As a consequence, the input data of a block is always a subset of the union of input data of its inner-blocks. The output-data-part of the description describes all the data produced by the block. The data produced by the block, is data which is altered in that block and which is used elsewhere, i.e. it is used by other blocks at the same level as input data, or it is used by its outer-block as output data. As a consequence, the output data of a block is always a subset of the union of output data of its inner-blocks. The description of both the input and output data uses global names to identify the data objects.

## 4.2.1. The block structure

A DGR, representing a source program, consists of one program block. The program block contains zero, one or more module blocks. Figure 4.3 shows the formal description of a program block. The type of a program block is program. Its input and output data are normally files. Its definition consists of three parts: a first-part, a body-part and a flow-part. The body-part, called *body*, is defined as the description of the function of the block. This body-part consists of the inner-blocks of the program block. The first-part, called *first*, is defined as a set of pointers to the inner-blocks which should be handled first (i.e. the inner-blocks for which the IG has to generate code before it generates code for the other inner-blocks). It thereby describes a part of the control flow graph of the module blocks. The input data of the inner-blocks, described in the first-part, is always a subset of the input data of the outer-block, because those first inner-blocks can not consume data produced by other inner-blocks at the same level. The flow-part, called *flow*, is defined as a set of pointers to the blocks which should be handled next by the IG, i.e. after all the inner-blocks are handled. The flow-part of a program block is always nil, because a DGR consists of one and only one program block.

```
<program_block> ::=
    program,<program_identifier>,{,<input_data>,→,<output_data>,} =
   ,[,first =
     ,<module_identifier> sequence option
     ,body =
     ,<module_block> sequence option
     ,flow =
     ,nil
   ,]
   .
```

Figure 4.3 Formal description of a program block

A module block normally represents a module, as in Modular Pascal, a package construction, as in Ada, or just one program, as in Pascal. Each module has its own global objects. A module block contains zero, one or more procedure blocks and/or construction blocks. Figure 4.4 shows the formal description of a module block. The type of a module block is module. Its input and output data are normally files and external objects (i.e. objects, declared in another module, which may be used in other modules). As with the program block, the module blocks definition consists of a first-part, a body-part and a flow-part. The inner-blocks of a module block are procedure blocks and construction blocks. The flow-part points to the module block(s) which should be handled after this one is handled. When the flow part is nil, then there are no more module blocks to handle.

For the sake of clarity, figure 4.5 shows, as an example, that part of a DGR which implements the control flow graph of the module blocks. The program, in this example, consists of four modules, none of them having any input or output data. Figure 4.6 pictures the control flow graph of the module blocks.

A procedure block represents a procedure, a function or a routine, with its possible parameters and global objects. Each procedure can have its own local objects. A procedure block contains zero, one or more construction blocks and, in the case of nesting, one or more procedure blocks. Figure 4.7 shows the formal description of a procedure block. The type of a procedure block is procedure. Its input and output data are normally files, external objects and global objects, including possible parameters and results. The definition of a procedure block is equal to the definition of a module block as far as the first- and body-part are concerned. The flow-part points to the procedure block(s) and/or construction block(s) which should be handled after this procedure block is handled.

A construction block represents some kind of construction, however it has nothing whatsoever to do with the block structure of the DGR. It is just the name of a set of blocks which can be present at the same level in the DGR. A construction block has to deal with the implicit jumps from the source program, like the implicit jump at the end of a loop. The construction blocks are while-do block, do-

```
<module_block> ::=
    module,<module_identifier>,{,<input_data>,→,<output_data>,} =
   ,[,first =
     ,(<procedure_identifier>
      ;<construction_identifier>
      ) sequence option
     ,body =
     ,(<procedure_block>
      ;<construction_block>
      ) sequence option
     ,flow =
     ,<module_identifier> sequence option
   ,]
   .
```

**Figure 4.4** Formal description of a module block

```
program p1 { → } =
  [first = m1
   body =
     module m1 { → } =
     [first =
      body =
      flow = m2 m3]
     module m2 { → } =
     [first =
      body =
      flow = m4]
     module m3 { → } =
     [first =
      body =
      flow = m4]
     module m4 { → } =
     [first =
      body =
      flow = nil]
  flow = nil]
```

**Figure 4.5** An example of a DGR implementing a control flow graph

while block, case block, if-then-else block, call block and basic block. Figure 4.8 shows the set of construction blocks in a production rule.

The first construction block going to be discussed, is the while-do block. A while-do block represents a loop construction

WHILE conditional expression DO statement

It also represents the FOR-DO loop constructions, which are transformed to WHILE-DO loops in the front-end. A while-do block contains one control block and zero, one or more construction blocks. Figure 4.9 shows the formal description of a while-do block. The type of a while-do block is while-do. The definition consists of a control-part, a first-part, a body-part and a flow-part. The control-part will be discussed later on. The inner-blocks of a while-do block are construction blocks. When regarding the DGR as an abstract dataflow machine, then the type of this block, while-do, implies that the control

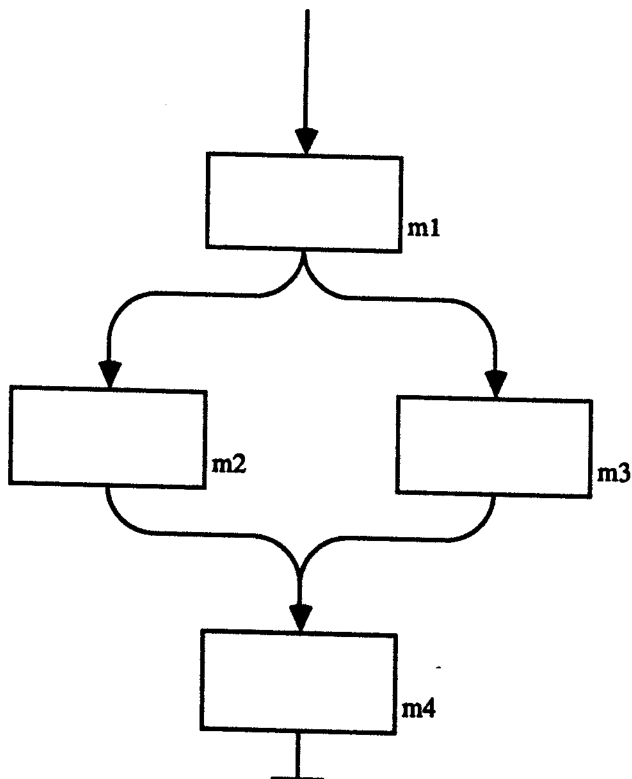



**Figure 4.6** The control flow graph of the example

```
<procedure_block> ::=
    procedure,<procedure_identifier>,{,<input_data>,→,<output_data>,} =
   ,[,first =
     ,(<procedure_identifier>
      ;<construction_identifier>
      ) sequence option
     ,body =
     ,(<procedure_block>
      ;<construction_block>
      ) sequence option
     ,flow =
     ,(<procedure_identifier>
      ;<construction_identifier>
      ) sequence option
   ,]
   .
```

**Figure 4.7** Formal description of a procedure block

block should be evaluated before the inner-blocks.

Closely related to the while-do block is the do-while block, which represents the loop construction

DO statement WHILE conditional expression

which is also known as the REPEAT-UNTIL loop. The type of a do-while block is do-while. The definition of the do-while block is the same as the while-do blocks definition. When regarding the

```
<construction_block> ::=
    (<while-do_block>
    ;<do-while_block>
    ;<case_block>
    ;<if-then-else_block>
    ;<call_block>
    ;<basic_block>
    )
    .
```

**Figure 4.8** The set of construction blocks

```
<while-do_block> ::=
    while-do,<construction_identifier>,{,<input_data>,→,<output_data>,} =
   ,[,<control_block>
    ,first =
    ,<construction_identifier> sequence option
    ,body =
    ,<construction_block> sequence option
    ,flow =
    ,<construction_identifier> sequence option
   ,]
   .
```

**Figure 4.9** Formal description of a while-do block

DGR as an abstract dataflow machine, then the type of this block, do-while, implies that the control block should be evaluated after the inner-blocks are. Figure 4.10 shows the formal description of a do-while block.

```
<do-while_block> ::=
    do-while,<construction_identifier>,{,<input_data>,→,<output_data>,} =
   ,[,<control_block>
    ,first =
    ,<construction_identifier> sequence option
    ,body =
    ,<construction_block> sequence option
    ,flow =
    ,<construction_identifier> sequence option
   ,]
   .
```

**Figure 4.10** Formal description of a do-while block

A case block represents the conditional construction

```
CASE conditional expression OF
label-list : statement;
     :
     :
label-list : statement
END
```

A case block contains one control block which produces an integer, and zero, one or more entries. Each entry contains zero, one or more construction blocks. Figure 4.11 shows the formal description of a case block. The type of the case block is case. The definition consists of: a control-part, which will be discussed later on; zero, one or more entries, each of them containing a first- and a body-part; and one flow-part.

```
<case_block> ::=
    case,<construction_identifier>,{,<input_data>,→,<output_data>,} =
    ,[,<control_block>
      ,(<entry_identifier>
       ,first =
       ,<construction_identifier> sequence option
       ,body =
       ,<construction_block> sequence option
       )sequence option
      ,flow =
      ,<construction_identifier> sequence option
    ,]
    .
```

Figure 4.11 Formal description of a case block

Related to the case block is the if-then-else block, which has two entries instead of an arbitrary number. An if-then-else block represents the conditional construction

IF conditional expression THEN statement ELSE statement

An if-then-else block contains one control block and two entries, which can contain zero, one or more construction blocks. Figure 4.12 shows the formal description of a if-then-else block. The type of the if-then-else block is if-then-else. There are two entries, called true and false. Like the entries of the case block, each entry contains a first- and a body-part.

```
<if-then-else_block> ::=
    if-then-else,<construction_identifier>,{,<input_data>,→,<output_data>,} =
    ,[,<control_block>
      ,true
       ,first =
       ,<construction_identifier> sequence option
       ,body =
       ,<construction_block> sequence option
      ,false
       ,first =
       ,<construction_identifier> sequence option
       ,body =
       ,<construction_block> sequence option
      ,flow =
      ,<construction_identifier> sequence option
    ,]
    .
```

Figure 4.12 Formal description of a if_then_else block

A control block produces a boolean or, in the case of the case block, an integer. A control block contains one or more construction blocks. Figure 4.13 shows the formal description of a control block. The type of the control block is control. Notice that where other block descriptions allow

empty bodies, the control block always contains at least one construction block. Notice also that the flow-part always has to be nil.

```
<control_block> ::=
    control,{,<input_data>,→,<output_data>,} =
   ,[,first =
    ,<construction_identifier> sequence
    ,body =
    ,<construction_block> sequence
    ,flow =
    ,nil
   ,]
   .
```

**Figure 4.13** Formal description of a control block

A call block represents a procedure-call or a call to a runtime-routine. it contains no inner-blocks. Figure 4.14 shows the formal description of a call block. The type of the call block is call. Its input data consist of a possible set of parameters and global objects. The first-part of its definition identifies the procedure or the runtime-routine. The body-part is always nil. Procedure-calls are also discussed in section 4.6.

```
<call_block> ::=
    call,<construction_identifier>,{,<input_data>,→,<output_data>,} =
   ,[,first =
    ,(<procedure_identifier>
     ;<runtime-routine_identifier>
     )
    ,body =
    ,nil
    ,flow =
    ,<construction_identifier> sequence option
   ,]
   .
```

**Figure 4.14** Formal description of a call block

A basic block is a collection of operations, which is represented by a strict dataflow graph. A basic block does not contain any intermediate entry points or exit points, nor procedure calls. Figure 4.15 shows the formal description of a basic block. The type of the basic block is basic. It has no inner-blocks, but contains a dataflow graph.

```
<basic_block> ::=
    basic,<construction_identifier>,{,<input_data>,→,<output_data>,} =
   ,[,first =
    ,<graph_identifier>
    ,body =
    ,<graph>
    ,flow =
    ,<construction_identifier> sequence option
   ,]
   .
```

**Figure 4.15** Formal description of a basic block

As a consequence of the block structure of the DGR, there is no need for the operators split, merge, bag and dyn (all described in chapter 2), as is easily verified.

Both the first-part and the flow-part of a block description are included for the convenience of the user. In effect, both are superfluous because the control flow they describe, is already implemented by the data.

### 4.2.2. The dataflow graph

At this point, the formal description of (strict) dataflow graphs is introduced. As noticed earlier, this description uses the same description formalism as in the descriptions of the blocks of the DGR. Figure 4.16 shows the formal description of a graph.

```
<graph> ::=
    graph,<graph_identifier>,{,<input>,->,<output>,} =
    ,[,<description>,]
    .
```

**Figure 4.16** Formal description of a graph

A graph is a collection of operations, representing some computations. In this set of computations, there are no intermediate entry points or exit points, nor does it contains procedure- or runtime-routine-calls. A graph obviously represents arithmetic expressions, like a := b+c, but it also represents boolean expressions, like x := a<b, and relational expressions, like a<b, as present in the control block of construction blocks. The boolean and relational expressions are treated in the same way as arithmetic expressions.

The graph description is best explained via an example. Figure 4.17 shows the graph representing the expression c := 3*(a+b). The input and output consist of object sequences. Each object contains an identifier, the type, the attribute *class* and the attribute *data-type* (see section 4.3 for an explanation of the type and the attributes). The graph description consists of a sequence of definitions. A definition is an object_identifier followed by a "=", followed by a function which is also described as a graph. For example

t1 = f(x1,x2)

means that the object t1 is defined as the result of the function f, when objects x1 and x2 are used as input data for function f. Suppose, function f produces two results instead of just one, then

t1,t2 = f(x1,x2)

means that object t1 is defined as the first result of the function f and object t2 is defined as the second result. A function is also described as a graph, however its description only states the word *primitive_node* and the type of the operator. Notice that the input and output data of the graph describing a function use local object identifiers and not the global object identifiers presented in graph G. The reason for this is obvious, a defined function can now be used more than once. The data presented in a function-call is placed in an order corresponding to the order of the data in the function description. A detailed formal description of the graph is found in appendix B.

```
graph G
    {(a numeric integer normal_integer),
     (b numeric integer normal_integer) →
     (c numeric integer normal_integer)} =
    [t1 = add(a,b)
     t2 = const()
     c= times(t1,t2)]

graph add
    {(v1 numeric integer normal_integer),
     (v2 numeric integer normal_integer) →
     (v3 numeric integer normal_integer)} =
    [primitive_node
     type = +numeric]

graph const
    { → (v1 numeric integer_constant small_integer 3)} =
    [primitive_node
     type = constant]

graph times
    {(v1 numeric integer normal_integer),
     (v2 numeric integer normal_integer) →
     (v3 numeric integer normal_integer)} =
    [primitive_node
     type = *numeric]
```

**Figure 4.17** Graph example

## 4.3. The type information in the DGR

Each operand in the DGR contains a semantic field *type*, which contains the type of the operand. The DGR supports the following operand types: numeric, boolean, numeric_array, boolean_array, numeric_set, boolean_set, numeric_pointer, boolean_pointer, record, file and user_defined. Strings and characters are (yet) not supported by the DGR.

Each operand type is associated with a set of classes of data-types. All the numeric types (i.e. numeric, numeric_array, numeric_set and numeric_pointer) are associated with the classes: integer (i.e. integer_scalar), real (i.e. real_scalar), integer_subrange, real_subrange, integer_constant and real_constant. The boolean types are associated with the classes: boolean (i.e. boolean_scalar) and boolean_constant. The record type is associated with the classes of data_types, its fields are associated with. The file type can be associated with every class of data-types, the context defines which class or classes. The user_defined type is associated with classes classes of data_types, specified by the user, possibly the class user_defined.

Each class of data-types is associated with a set of data-types. All the integer classes (i.e. integer, integer_subrange and integer_constant) are associated with the integer set of data-types, which contains: small_integer, normal_integer, long_integer, long_long_integer and long_long_long_integer. All the real classes (i.e. real, real_subrange and real_constant) are associated with the real set of data-types, which contains: normal_real, long_real and long_long_real. All the boolean classes (i.e. boolean and boolean_constant) are associated with the boolean set of data-types, which contains only boolean.

Each operand has the attributes: *class* and *data-type*. The attribute *class* contains the class of data-types and the attribute *data-type* contains the data-type of the operand. Sometimes an operand has more than two attributes. For example, the graph const in figure 4.17 has an output operand v1 with three attributes. The third attribute, called *constant_value* is the constant value of the operand. When

dealing with an array object, extra attributes are *lowerbound, upperbound, base* and *size*. Other extra attributes may be added, when implementing the compiler system.

Each attribute has itself an attribute called *fixed*. This attribute of an attribute denotes that the value of the attribute may or may not be changed during the compilation process. An attribute that may not be changed is called a fixed attribute. For example, the attribute *constant-value* is always fixed, while the attribute *lowerbound* can be fixed or can be not fixed.

Each operator in the DGR contains, similar to the operands, a semantic field *type* , which contains the type of the operator. The DGR supports all the arithmetical and relational operator types, like: +numeric, -numeric, *numeric, /numeric, =numeric, ≠numeric, >numeric, <numeric, ≥numeric, ≤numeric, etc. The DGR also supports a set of special operator types: dereference, update, constant, halt (all described in chapter 2), conversion (described in chapter 3) and a serie of set-operators.

Similar to the operands, an operator has two attributes: *class* and *data-type*. The operators attribute *class* contains one or more classes of data-types, and its attribute *data-type* contains one or more data-types. The classes and the data-types are the same as with the operands. Each operator attribute has an attribute *fixed*, just like the operand attributes. The difference between the operand and operator attributes, is that the operand attributes always contain one item, whereas the operator attributes may contain several.

The description part of a graph, describing a primitive node, contains the type of the operator but does not contain the operators attributes *class* and *data-type*. The classes of data-types and the data-types themselves are defined in the input and output parts of the graph. Figure 4.17, for example, shows such a graph.

The DGR is *well-defined* (see also chapter 3), which means that the following rules have to be satisfied:

- each operator-type deals with fixed operand-types.
- each operator-type deals with fixed classes of data-types.
- each operator-type deals with fixed data-types.

As a consequence, the attributes *class* and *data-type* of an operator are superfluous in the DGR; the information is already specified in the attributes of the operands. In the DGR description however, the type information is included in the descriptions of the operators (i.e. in the input-data-part and output-data-part) for the sake of clarity. Notice that the input-data-part and output-data-part of the graphs, which do not represent a primitive node, are superfluous; the information is already specified in the input-data-part and output-data-part of the primitive nodes. The data-parts are included however for the sake of clarity.

## 4.4. Datastructures

A datastructure is represented and treated as a single data object. This method is called *the copy method* , although there will be no explicit copying of datastructures in the assembly code. In this section the following datastructures are discussed: array, set, record and file.

### 4.4.1. Array

To fetch the value of an array element a[i], the dereference operator is used with the array as the datastructure and the address a[i] as the pointer. To store a value in an array, the update operator is used with once again the array as the datastructure and the address a[i] as the pointer. Figure 4.18 shows a Pascal fragment as an example, and figure 4.19 pictures the abstract dataflow graph as an implementation of the example. The constant operators s produce the value size(a) and the constant operators b produce the value base(a)-lowerbound(a)*size(a).

Notice that the code of a[j] := x has obviously to be executed before z := a[j]. Even before y := a[k], because it is possible that k = j. Notice that although aliasing is possible, this can not introduce any problems because of the control flow implemented by the dereference and update operators. Although the code of w := a[i] has to be executed before the code of a[j] := x, because it is possible

```
w    :=  a[i];
a[j] :=  x;
y    :=  a[k];
z    :=  a[j];
```

**Figure 4.18** Array example

that i = j, there is no explicit control flow in the graph which implements this order. Still the graph is a correct datafow graph, while the code of w := a[i] uses the 'old' datastructure a and not the datastructure t9 which is produced by the update operator. Figure 4.20 shows the graph of the example, described as a part of a DGR. For the sake of compactness the subgraphs constant_s, constant_b, multiply, addition, dereference and update are not described. However, the identifiers are chosen equal to their types, which should make the intention clear. The constant_s function produce the constant value size(a), and the constant_b function produce the constant value base(a)-lwb(a)*size(a). Although not specified in the example, all the objects are assumed to have data-type normal_integer.

### 4.4.2. Set

A set is treated, like all the other datastructures, as a single data object. The difference is, that there are special set operators like union, intersection and difference. Because of these special set operators, there is no need for dereference or update operators to handle a set. Notice that strict value propagation is still used.

### 4.4.3. Record

When the datastructure is a record, then the dereference operator is used to obtain the value of a record-field, and the update operator is used to store a value in a record-field. Figure 4.21(a) shows the fetch of a value of a record-field. and figure 4.21(b) shows the store of a value in a record-field. The record is used as the datastructure and the address of the record-field is used as the pointer.

Pascal contains the with clause

WITH record_identifier DO statement

Within the statement, the components (i.e. the fields) of the record specified can be denoted by their field identifier only, i.e. without preceding them with the denotation of the entire record identifier. The with clause opens the scope containing the field identifiers of the specified record. The with clause can be implemented in the DGR as a construction block. When doing so, it should also be used when the programmer had the opportunity to use it, but did not. The with clause is left for future research. In this report the with clause is omitted as it was never used by the programmer.

### 4.4.4. File

When the datastructure is a file, then the dereference operator is used to obtain (read) a value of the file and the update node is used to store (write) a value on the file. The pointer of the update and dereference node is the read/write head of the type-drive. It is assumed in this report that files are handled by runtime-routines and need no special attention.

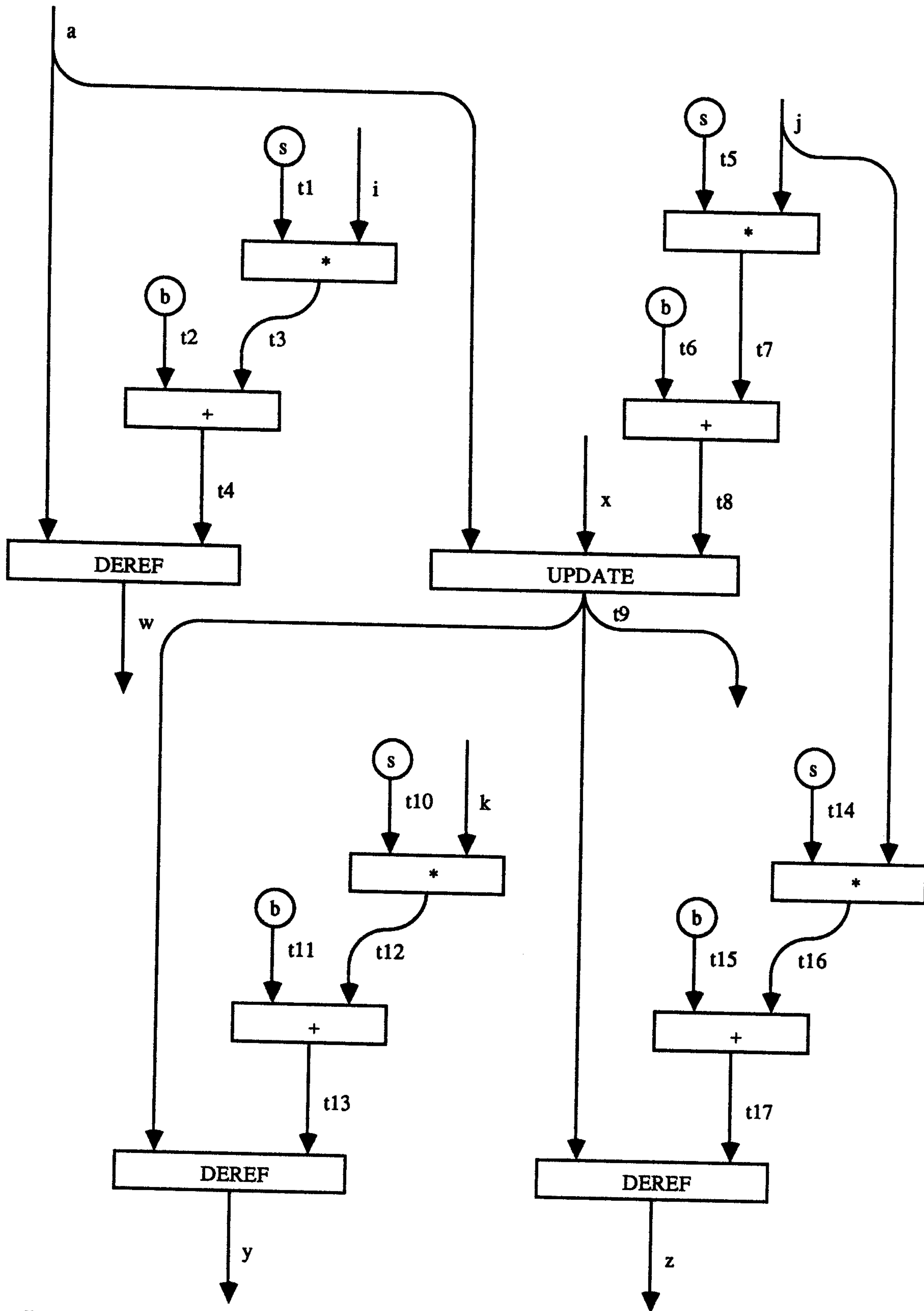



Figure 4.19 The dataflow graph of the array example

```
graph G
    {(a numeric_array integer normal_integer),
     (i numeric integer normal_integer),
     (j numeric integer normal_integer),
     (x numeric integer normal_integer),
     (k numeric integer normal_integer) →
     (t9 numeric_array integer normal_integer),
     (w numeric integer normal_integer),
     (y numeric integer normal_integer),
     (z numeric integer normal_integer),} =
    [t1 = constant_s()
     t2 = constant_b()
     t3 = multiply(t1,i)
     t4 = addition(t2,t3)
     w = dereference(a,t4)
     t5 = constant_s()
     t6 = constant_b()
     t7 = multiply(t5,j)
     t8 = addition(t6,t7)
     t9 = update(a,t8,x)
     t10 = constant_s()
     t11 = constant_b()
     t12 = multiply(t10,k)
     t13 = addition(t11,t12)
     y = dereference(t9,t13)
     t14 = constant_s()
     t15 = constant_b()
     t16 = multiply(t14,j)
     t17 = addition(t15,t16)
     z = dereference(t9,t17)]
```

**Figure 4.20** The DGR of the array example

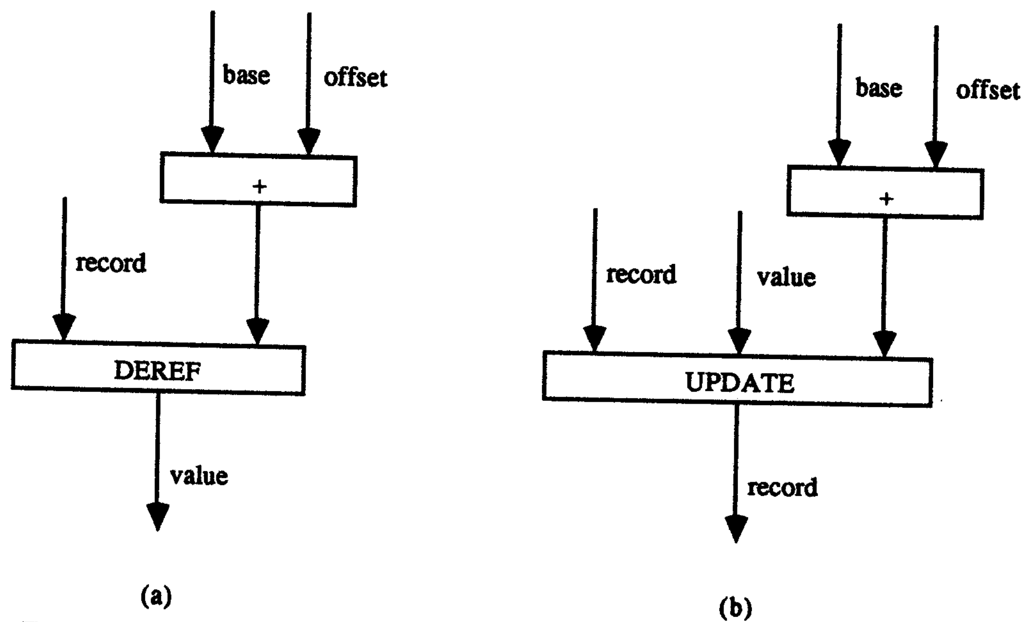



**Figure 4.21** Fetch and store a value of a record-field

## 4.5. Pointers

When dealing with pointers, the same method is used as with most of the datastructures. An assignment is represented by using the update node with the off-stack storage area, called the heap in ALGOL68 terminology, as a datastructure. To obtain the value of a pointer, the dereference node is used. Instead of using just one heap for all the pointers, each pointer type has its own special heap. Notice that although aliasing is possible, there are no problems introduced because of the control flow implementation by the dereference and update operators, just as in the case of arrays.

Value propagation is used in the DGR, so the value of a pointer is used instead of the pointer itself. Notice, the value of a pointer is, still, an address.

Also higher order pointers, such as pointers to pointers or to structures containing pointers, can be implemented using dereference and update operators. Figure 4.22 shows a Pascal program as an example, and figure 4.23 pictures the dataflow graph of the basic block of that example. The constant_V operator produces the offset for the field *value* of the record *node* and the constant_L operator produces the offset for the field *link* of the record *refnode*.

## 4.6. Procedures

The DGR handles three types of procedure-calls; the call of user defined procedures, the call of external procedures and the call of permanent procedures.

### 4.6.1. User defined procedures

An invocation of an user defined procedure can have side-effects on global objects. It has to be known which global objects can be affected by a procedure, so they can be passed to the call block. Although it is generally not possible to determine exactly all effects a procedure-call has or can have, we may determine a kind of upperbound for it. All global objects that may be changed by a procedure P, although they need not be changed at every invocation of P, are computed. This set can be determined by just looking at all assignment instructions in the body of the procedure. A procedure may of course call another procedure. To determine the effects of a procedure-call to P, also the effects of the procedures called by P have to be determined. This is done by computing the transitive closure of the effects. To do this, a conceptual graph, called the *call graph* , is constructed. In the call graph the nodes are the procedures and there is an edge from node P to node Q if procedure P can call procedure Q.

The presence of pointers in a procedure body can introduce more complex problems. The procedure body can contain, for example, an assignment through a pointer variable. In general, it is not possible to determine which variable is affected by such an assignment. The front-end must determine which variables can possibly be accessed by a pointer variable, and which variables can never be accessed that way.

Formal procedures (i.e. procedures which have procedures as parameters) are not investigated in this report, they are left for future research.

### 4.6.2. External procedures

Some languages (like Modular Pascal) provide means for compiling modules of procedures which can be accessed from other modules. Also, some languages (like FORTRAN) provide extensive libraries of procedures which users can access. In some languages external procedures written in a programming language different from the source language can be accessed. In the last case, the parameter passing mechanisms may be different. In order to cope with this problem, the call block of

```
PROGRAM higher_order_pointers (input,output);
TYPE    refint = ^node;
      refrefint = ^refnode;
          node = RECORD
                   value : integer
                 END;
       refnode = RECORD
                   link : refint
                 END;
VAR     i : integer;
       pi : refint;
      pi2 : refint;
      ppi : refrefint;
BEGIN
    i := 1;
    new(pi);
    pi^.value := i;
    new(ppi);
    ppi^.link := pi;
    new(pi2);
    pi2 := ppi^.link;
    i := pi2^.value
END.
```

Figure 4.22 An example of higher order pointers

the DGR must be sufficiently flexible to allow all the variations to be handled.

External procedures never affect global objects in the module in which they are called, but they can in the module in which the procedure body is present.

## 4.6.3. Permanent procedures

All high-level languages define a set of procedures, which is available on any implementation, such as the procedures *read* and *write* in Pascal. Such a procedure is called a *permanent procedure*. It is common for intermediate representations to provide specific code items to invoke permanent procedures. In the DGR, however, permanent procedures are treated in the same way as external procedures, because there is as far the DGR is concerned no difference between them.

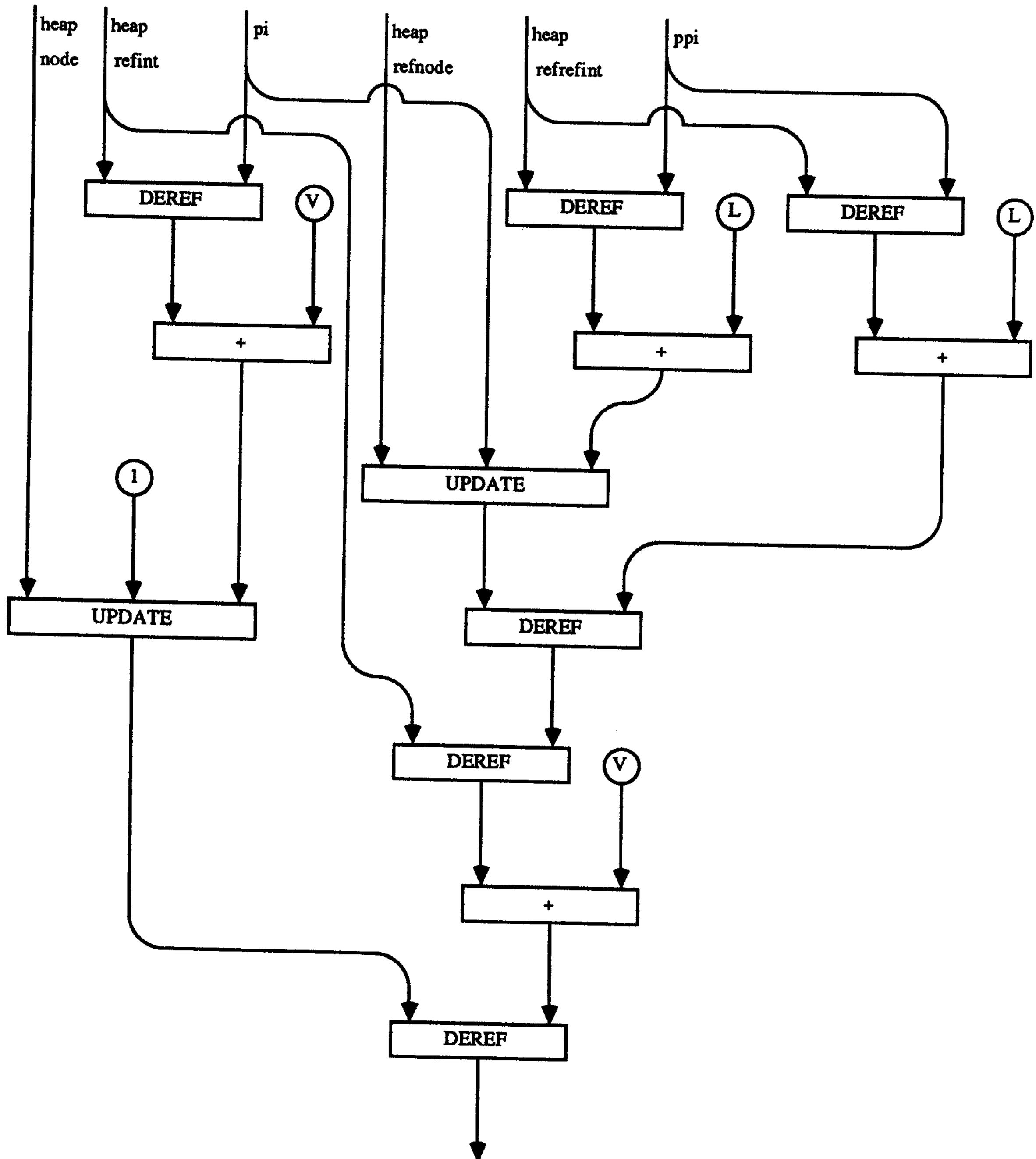



Figure 4.23 The dataflow graph implementing higher order pointers

## 4.7. The dataflow in the DGR

The dataflow as present in a dataflow graph, is rather obvious. When an object is defined, it is chained to the points where it is used, i.e. arcs from the output port of the operator which defines the object, to the input ports of the operators which use the object as an operand, are inserted.

The dataflow between the blocks of the DGR needs some special attention. When an object is defined in some block and is used outside of the block, the output data of the block contains the object. If the object is also used outside of the outer-block, the output data of the outer-block also contains the object, and so on. When an object is defined outside some block but is used in that block, then the

input data of that block contains the object. The flow arcs, as present in the block structure, ensure that the block containing the definition is handled before the blocks containing an use.

When conditional construction blocks are involved, it is less straightforward which objects have to be placed in the output data of a block. When dealing with a conditional expression like

IF condition THEN x := 1 ELSE x := y;

it is not clear which object should be placed in the output data of the if-then-else block. There are two possible definitions. Both definitions are concerned with the definition of the numeric object x, however in one definition the class of data-types is integer_constant and in the other it can be, for example, integer_subrange. This problem is solved by evaluating the intersection of the attributes, in this case integer_scalar. So, the numeric object x with the class integer_scalar is placed in the output data of the if-then-else block.

When dealing with a conditional expression like

IF condition THEN x := 1 ELSE y := 1;

a similar problem occurs. Also in this situation it is not clear which object should be placed in the output data of the if-then-else block. Object x is defined in the THEN-branch of the expression but not in the ELSE-branch. A subsequent use of x (i.e. a use of x after the conditional expression), could be 'connected' to the definition of x in the THEN-branch and also to the definition of x previous to the conditional expression. However, it is unwishful for an object to have possibly more than one definition. This problem is solved as follows. The previously defined x is inserted in the input data of the if-then-else block. In the ELSE-branch the instruction x := x; is inserted, and the rest of the problem is solved as above. Obviously, both the THEN-part and the ELSE-part of the block, should use the same global identifier to identify object x. The same is done on behalf of the object y. Notice that the simulated definition/use of x in the ELSE-branch and the simulated definition/use of y in the THEN-branch are only of theoretical importance. Important in practice is only that the input data and output data of a block cover all the possible objects in the block. This means that if an object can possibly be defined in a block, then it must be defined in that block (possibly equal to its former value).

Notice that the last problem discussed also occurs in if-then expressions. The problem is solved in the same way, the objects defined in the THEN-branch get a simulated definition/use in a simulated ELSE-branch. The treatment of case expressions is a generalization of the treatment of the if-then-else expressions. The treatment of while-do expressions is almost equal to the treatment of if-then expressions. The objects defined in the body of the loop get a simulated definition/use in a simulated FALSE-body which should be executed when the real body is not executed once. Notice that the problems discussed above do not occur in a do-while expression.

In the previous discussion, it was apparently assumed that there is no aliasing present in conditional construction-blocks. However, it is easily verified that this aliasing, called *conditional aliasing*, introduces no extra problems, as long as the simulated definition/use statements are inserted at the front of a block.

# CHAPTER 5
# THE GLOBAL OPTIMIZER

In this chapter we describe the global optimizer, called the Global Optimizer or GO for short. The Global Optimizer systematically transforms the DGR-code into better DGR-code. The DGR and the Global Optimizer are both language and machine independent, so the Global Optimizer can be used with any combination of source languages and target machines.

The Global Optimizer consists of the following optimization techniques, which are discussed in the next sections : *constant evaluation, algebraic simplification, common subexpression elimination, dead variable elimination, dead statement elimination, invariant expression elimination, strength reduction, test replacement, unrolling, loop fusion, cross jumping* and *inline substitution*. We give some Pascal program fragments to explain the techniques. Effectively however, the Global Optimizer optimizes the DGR-code. When, in this chapter, a source program is replaced by its optimized program, this means that the DGR representing the source program is optimized into a DGR which is a possible DGR of the optimized program.

Some of the optimization techniques try to optimize the execution time, while others try to optimize the size of the object code or both. Sometimes the optimization of the execution time implies an increase of the size of the object code, or the other way around, the optimization of the size of the object code sometimes implies an increase of the execution time. The table in figure 5.1 gives the effects of the different optimization techniques. These effects occur in general, not as a rule as is shown in some examples in the next sections.

The different optimization techniques may be run in any order, and an optimization technique may be run more than once. The ordering of the optimizations has a significant impact on the quality of the produced code. Experimental implementations should provide the best order. The ordering is not discussed in this report, we merely describe the different optimization techniques.

The interaction between different optimization techniques, is an issue that may not be overlooked, when designing a global optimizer. It can be advantageous to combine several optimization techniques into one algorithm, that takes into account all interactions between them. These interactions are also not discussed in this report, it is assumed that the Global Optimizer uses a separate algorithm for each optimization technique.

## 5.1. Constant evaluation

The constant evaluation optimization technique eliminates constant expressions and constant subexpressions. The technique is illustrated in figure 5.2 by some examples and their replacements. Notice that in the first and second example both the size of the code and the execution time are optimized, which is not obvious in the third example.

| optimization techniques | time | size |
|---|---|---|
| constant evaluation | - | - |
| algebraic simplification | - | - |
| common subexpression elimination | - | - |
| dead variable elimination | - | - |
| dead statement elimination | o | - |
| invariant expression elimination | - | + |
| strength reduction | - | + |
| test replacement | - | - |
| unrolling | - | + |
| loop fusion | - | o |
| cross jumping | o | - |
| inline substitution | - | + |

\- means decrease of time/size

\+ means increase of time/size

o means almost no change

**Figure 5.1** Optimization effects

```
a := 2+5       becomes  a := 7
a := 2+b+5     becomes  a := b+7
a := (2+b)*5   becomes  a := 10+b*5
```

**Figure 5.2** Examples of constant evaluation

## 5.2. Algebraic simplification

The algebraic simplification optimization technique replaces expressions by simpler, equivalent expressions. Figure 5.3 shows some examples and their replacements. Situations in which this optimization can be performed, are normally created by the constant evaluation optimization technique.

## 5.3. Common subexpression elimination

The common subexpression elimination optimization technique uses the fact that the results of a previously calculated expression may be used, if its various components have not been altered, to replace the same expression elsewhere. Figure 5.4 shows three program fragments with their common subexpressions. Example i) is an example of a general common subexpression. Example ii) shows a fragment in which it is obviously not an optimization to eliminate the common subexpression, however later on in the assembly code, it can be an optimization. In example iii), the fact is used, that an array is a 4-tuple <base,lowerbound,upperbound,size>.

```
a := a+0   becomes  nil
a := b+0   becomes  a := b
a := b*0   becomes  a := 0
a := a*1   becomes  nil
a := b*1   becomes  a := b
a := a/1   becomes  nil
a := b/1   becomes  a := b
a := b/b   becomes  a := 1
```

**Figure 5.3** Examples of algebraic simplification

```
i)   z := x+y*t;       common subexpression is y*t
     v := y*t;
ii)  z := x+7;         common subexpression is 7
     t := w*7;
iii) a[i+1] := a[i];   common subexpression is base-lowerbound+i*size
```

**Figure 5.4** Examples of common subexpression elimination

## 5.4. Dead variable elimination

The dead variable elimination optimization technique removes all the assignments to variables, which are not used later on. Figure 5.5(a) shows a program on which this optimization is performed. Figure 5.5(b) shows the optimized program.

```
PROGRAM deadvar;          PROGRAM deadvar;
VAR a, b : integer;       VAR a, b : integer;
BEGIN                     BEGIN
    a := 1;                   a := 1;
    b := 2+a;                 b := 2+a;
    a := b+7;                 write(b)
    write(b)              END.
END.

        (a)                       (b)
```

**Figure 5.5** Example of dead variable elimination

## 5.5. Dead statement elimination

This optimization technique removes unreachable and useless code. Although it can be implemented as a stand-alone optimization technique, like all the other techniques discussed in this chapter, it is only used in combination with other techniques which introduce opportunities to remove unreachable or useless code.

## 5.6. Invariant expression elimination

This loop optimization technique removes the invariant expressions from a loop, and places them before the loop. Invariant expressions are expressions whose evaluation is not affected by occuring either before or inside the loop. The result of the evaluation of an invariant expression in a loop, does not change on any iteration of that loop. This optimization may only be performed when it is sure that the body of the loop is at least executed once, unless the variables concerning the invariant expressions are not used anymore after the end of the loop. In this last case, the execution time can increase. There are two types of loops in the DGR:
- the WHILE-DO-loop with the test at the top
- the DO-WHILE-loop with the test at the bottom

```
i := 1;
WHILE i<10 DO
BEGIN
    base(a)-(lowerbound(a)*size(a))+(i*size(a))  :=  x+y;
    (* a[i] := x+y *)
    base(b)-(lowerbound(b)*size(b))+(i*size(b))  :=  z*i;
    (* b[i] := z *i *)
    c :=  x*y;
    i  :=  i+1
END
```

**Figure 5.6** Loop example

Consider the informal Pascal fragment in figure 5.6, as an illustration of this technique. The invariant expressions are

```
x+y
base(a)-(lowerbound(a)*size(a))
base(b)-(lowerbound(b)*size(b))
c := x*y
```

The last is actually an invariant statement and therefore removed as a whole from the loopbody. Figure 5.7 shows the optimized program fragment.

```
temp1 := x+y;
temp2 := base(a)-lowerbound(a)*size(a);
temp3 := base(b)-lowerbound(b)*size(b);
c := x*y;
i := 1;
WHILE i<10 DO
BEGIN
    temp2+(i*size(a)) := temp1;
    temp3+(i*size(b)) := z*i;
    i := i+1
END
```

Figure 5.7 Loop example after invariant expression elimination

## 5.7. Strength reduction

This optimization technique tries to replace expensive operators by cheaper ones, thereby reducing the execution time of the program. A classical example is the replacement of a multiplication by 2, by an addition. In this report strength reduction is regarded as a loop optimization. We are mainly interested in replacing a multiplication like i*x, where i is the loop variable, by an addition. Consider the example shown in figure 5.7 as an illustration of the strength reduction technique. It is assumed that size(a), size(b) and z are integers. While temp2, temp3, size(a) and size(b) are not altered elsewhere within the loop, the strength of the operations i*size(a), i*size(b) and z*i is reduced. Figure 5.8 shows the optimized program.

It is assumed that size(a), size(b) and z are integers, because when, for example, z should be a real, it is not possible to perform this optimization. Due to the roundoff error involved when using floating point numbers, the multiplication z*i is, in general, much more accurate than the addition z+z+...+z (i times).

Strength reduction does not always make a program more efficient. Consider the program fragment in figure 5.9(a) and in figure 5.9(b) the program fragment after the performance of the strength reduction optimization technique. The first program fragment uses 10 multiplications, whereas the fragment after strength reduction uses 100 additions.

## 5.8. Test replacement

This loop optimization technique tries to replace the test of a loop by another one, in the hope that the loop variable can be eliminated. If the loop variable is only used in the test and in its increment, and another recursively defined variable[1] exists, which arose through strength reduction on the loop variable, then the test can be transformed into a test on the other recursive variable, and the increment of the loop variable can be eliminated. For example consider the program fragment in figure 5.8, and in figure 5.10 the program fragment after the performance of the test replacement method.

[1] A recursively defined variable is a variable which is incremented with a fixed (invariant) value each time the loop body is executed.

```
temp1 := x+y;
temp2 := base(a)-lowerbound(a)*size(a);
temp3 := base(b)-lowerbound(b)*size(b);
c := x*y;
temp4 := size(a);
temp5 := size(b);
temp6 := z;
i := 1;
WHILE i<10 DO
BEGIN
    temp2+temp4  := temp1;
    temp3+temp6  := temp6;
    i := i+1;
    temp4  := temp4+size(a);
    temp5  := temp5+size(b);
    temp6  := temp6+z
END
```

**Figure 5.8** Loop example after strength reduction

Notice that the program fragment in figure 5.10 can be further optimized by previously discussed optimization techniques. For example, the invariant expression elimination technique can replace the expression 10*z in the while-condition by a temporary. These optimizations are left to the reader.

## 5.9. Unrolling

The unrolling optimization technique is a time optimization method for loops. Consider the loop in figure 5.11(a) as an example. The index i must be incremented and tested three times in this example. If the loop were unrolled, as shown in figure 5.11(b), several savings would result. No increments and tests are needed, because the addresses a[1], a[2] and a[3] are known at compile-time. Notice that this time optimization technique, in general, increases the size of the object code.

## 5.10. Loop fusion

The loop fusion optimization technique tries to replace two or more loops by one big loop. Consider the two loops in figure 5.12(a). These two loops can be replaced by the loop shown in figure 5.12(b).

Certain requirements have to be satisfied in order to use this optimization technique.

- Neither loop may be exited prior to complete execution of the index range.
- Both the loops are executed, or neither is executed.

```
FOR i:=1 TO 10 DO
BEGIN
    CASE i OF
        1:  x := t1*i;
        2:  x := t2*i;
              :
        10: x := t10*i
    END
END
```

(a)

```
temp1  := t1;
temp2  := t2;
          :
temp10 := t10;
FOR i:=1 TO 10 DO
BEGIN
    CASE i OF
        1:  x := temp1;
        2:  x := temp2;
              :
        10: x := temp10
    END;
    temp1  := temp1+t1;
    temp2  := temp2+t2;
              :
    temp10 := temp10+t10
END
```

(b)

**Figure 5.9** Example of inefficient strength reduction

- The data in the second loop has to be independent of the computations in the first loop, and independent on the path from the first loop to the second.
- The data in the first loop has to be independent of the computations in the second loop.

## 5.11. Cross jumping

The cross jumping optimization technique is a space optimization method. It removes the common head and the common tail in CASE and IF-THEN-ELSE expressions, the common head is placed at the front of the construction and the common tail is placed at the back of the construction. Before the common head is replaced, two things have to be sure:

- the evaluation of the common head is not affected by the evaluation of the condition.
- when the common head is replaced, the evaluation of the condition is not affected by the evaluation of the common head.

The common tail can be replaced without special conditions. In general, the execution time will remain the same. Figure 5.13(a) shows a program fragment on which this optimization is performed as an example. Figure 5.13(b) shows the converted program fragment.

This technique might split the computation of an expression into two, by inserting a branch somewhere in the middle. This can result in rather poor generated code, because other optimizations can not be performed, for example, the use of addressing modes as a computation instruction.

```
temp1 := x+y;
temp2 := base(a)-lowerbound(a)*size(a);
temp3 := base(b)-lowerbound(b)*size(b);
c := x*y;
temp4 := size(a);
temp5 := size(b);
temp6 := z;
WHILE temp6<10*z DO
BEGIN
    temp2+temp4  := temp1;
    temp3+temp6  := temp6;
    temp4  := temp4+size(a);
    temp5  := temp5+size(b);
    temp6  := temp6+z
END
```

**Figure 5.10** Loop example after test replacement

```
FOR i:=1 TO 3 DO            a[1] := x;
BEGIN                       a[2] := x;
    a[i] := x               a[3] := x;
END


   (a)                         (b)
```

**Figure 5.11** Example of unrolling

Therefore this technique is not always performed.

## 5.12. Inline substitution

The inline substitution technique (or inline expansion, or procedure integration) tries to decrease the overhead associated with procedure invocations. During a procedure call, several actions must be undertaken to set up the right environment for the called procedure. Most of these effects must be undone when returning from the procedure. This entire process introduces significant costs in the execution time. The inline substitution technique replaces some of the calls by the modified bodies of the called procedures, hence eliminating the overhead. An other advantage of integrating a procedure is that this leads to extra opportunities for other optimization techniques. In [Ball79] a technique is presented for predicting the code improvement that can be expected when a procedure call involving

```
  :                          :
FOR i:=1 TO 5 DO           FOR i:=1 TO 5 DO
BEGIN                      BEGIN
    a[i] := x;                 a[i] := x;
END;                           b[2*i-1] := y;
  :                            b[2*i] :=y
FOR j:=1 TO 10 DO          END;
BEGIN                        :
    b[j] := y
END;
  :

  (a)                        (b)
```

**Figure 5.12** Example of loop fusion

```
IF condition                 statement1;
THEN  BEGIN                  IF condition
            statement1;      THEN  statement2
            statement2;      ELSE  statement4;
            statement3       statement3;
      END
ELSE  BEGIN
            statement1;
            statement4;
            statement3
      END;

        (a)                    (b)
```

**Figure 5.13** Example of cross jumping

constant parameters is integrated.

In this optimization technique there is a speed-size tradeoff. An inline substitution decreases the overhead, but increases the size of the code, unless the procedure body is very small or the procedure is called only once. Chapter 1 showed that the user can specify his wishes concerning the speed-size tradeoff. A model that takes this into consideration, is analyzed by Robert Scheifler in [Sche77]. In his model, it is allowed that the code grows by a certain amount which is specified in 'size-change'. Thus code size is sacrificed to speed up the execution time. If a small code size is prefered to fast execution then the size-change is set to zero, i.e. only very small procedures and

procedures that are called only once, are considered.

Scheifler gives the following conclusion concerning his model: "For structured programs with a low degree of recursion, the judicious use of inline substitution can eliminate almost all procedure calls with little or no increase in the size of compiled code. For recursive programs, a large number of calls can be eliminated, but at the expense of a rather substantial size increase. The execution time saved directly by inline substitution is small, even for fairly inefficient procedure call mechanisms; however, the enlarged context made available to other techniques may lead to much more optimization than would otherwise be possible."

## 5.13. Is optimization cost-effective?

Heavy optimization can easily double the compilation time of a program. For most programs such an optimization is not worthwhile, because the gain in execution time is outweighed by the extra costs of optimization. The compiler writer has to draw a line between useful and wasteful optimization techniques, this is difficult to do and mostly a matter of taste. We do not draw such a line in this report. It is possible to omit the Global Optimizer, a complete compiler will result, though the produced code will be of rather poor quality.

Wulf et al. argue, in [Wulf75], in favour of the optimization of system programs (like operating systems and compilers), written in a high-level programming language such as C. Those system programs are in almost constant use, and therefore it is worthwhile to optimize them.

Bornat states in [Born79] that there will always be programs just too slow to perform some particular function, which optimization can make into useful tools. "Some famous current examples are those programs which aid weather forecasting by simulating atmospheric circulation and which, rumour has it, can only just run faster than the real thing."

# CHAPTER 6
# THE BACK END

In this chapter, we discuss the back-end which translates the intermediate representation into assembly code. As explained in chapter 1, the code generation algorithm is separated from the target machine data that drive the algorithm. The target machine data, used by the back-end, are stored in different tables. The topics of this chapter are how a back-end can be implemented using these tables, and what information is needed in the tables. It is of no interest what the tables look like. The way the tables are used, is described in this chapter, their construction is described in chapter 7.

## 6.1. General description

In this section, an overview of the back-end, which consists of four modules, is presented. Also the items which have to do with the back-end in general, are discussed in this section. Items particular to one of the four modules, are discussed in the section concerning that particular module.

### 6.1.1. The modules

The back-end consists of four modules. Figure 6.1 pictures the different modules and the intermediate results.

The first module is the Graph Transformer (GT). The GT expands the compound nodes. The representation produced by the GT is called the Transformed Dataflow Graph Representation. The GT is discussed in section 6.2.

The second module is the Pattern Matcher (PM). The PM covers the TDGR with subgraphs, representing addressing modes and machine instructions. The representation produced by the PM is called the Covered Dataflow Graph Representation. The PM is discussed in section 6.3.

The third module is the Instruction Generator (IG). The IG generates an assembly code sequence out of the Covered Dataflow Graph Representation. A register manager is used to do the register assignment. The IG is discussed in section 6.4.

The last module is the Peephole Optimizer (PO). The PO performs peephole optimizations on the assembly code. The PO is discussed in section 6.5.

### 6.1.2. The tables

The PM, the IG and the PO are table-driven and use some kind of pattern matcher (do not confuse these pattern matchers with the module Pattern Matcher). Remember that in this report, the tables themselves are of no interest. Table descriptions, as the one in this section, are only presented for the sake of clarity. In reality the tables may be different.

The fundamental unit of each table is the rule, which consists of two parts: the source pattern and the target pattern. The pattern matcher tries to match a part of its input with a source pattern. If the match is unsuccessful then the rule fails and another rule is tried. If the match is successful then the source pattern is replaced by the target pattern. Rules may be grouped together in rule-blocks,

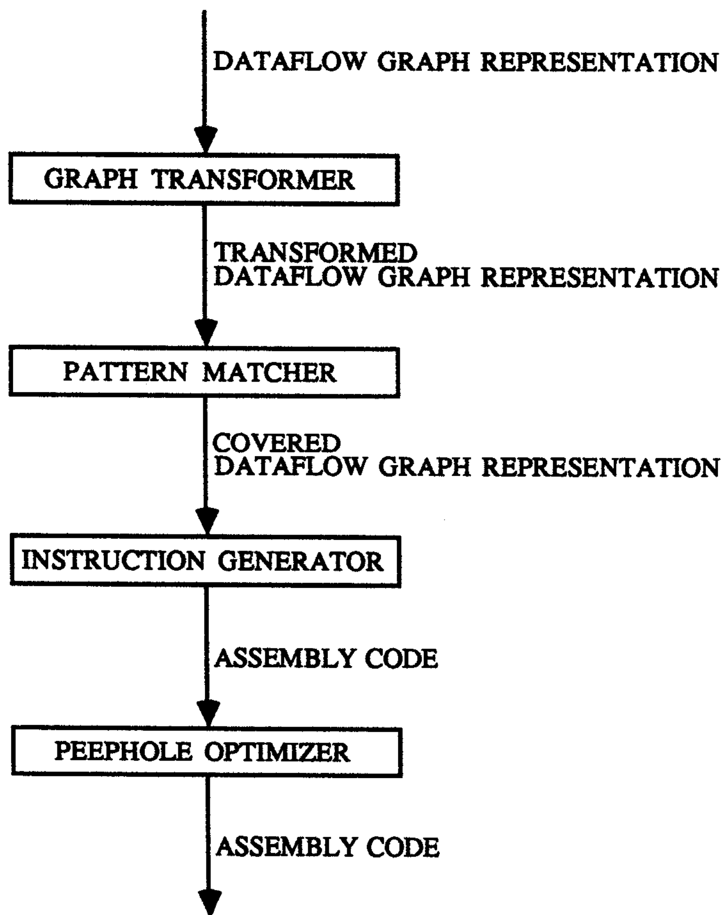



**Figure 6.1** The modules of the back-end

<table> ::= <rule> sequence.
<rule> ::= <source_pattern>,→,<target_pattern>.

**Figure 6.2** General table description

which are sequentially evaluated. Hashing techniques can improve the sequential scan by rapidly determining which rules may possibly match.

### 6.1.3. The Cost Function

The code generator should not only generate correct code but also efficient code. The costs of the instructions and addressing modes are the key to produce efficient code. The cost of each instruction including its addressing modes, in terms of time and space is specified as a set of integer values. The cost is defined as the triple <space,time,reference>. *Space* is the size (in bytes) of the object code, *time* is the number of the clock cycles, and *reference* is the number of memory references. Figure 6.3 shows the cost function of some MC68000 instructions.

When optimizing or generating code, there is a speed/size tradeoff. Normally, the code size is a more important consideration since the execution time of some portions of a program can not always be predicted. However, the user can specify which cost(s) he wants to use as the cost function. Suppose, the user wants the compiler to optimize for the fastest code sequence. Then the time costs

```
MOV.L    #7,D0            <6,13,0>
SUBX.L   -(A0),-(A1)      <2,32,3>
```

**Figure 6.3** The cost function of some MC68000 instructions

will be selected as the cost function. The time costs for each instruction within a code sequence are summed and the minimum cost sequence is chosen, independent of the space costs and the number of memory references. Identically, the user may want the compiler to optimize for the smallest code sequence, using only the space costs. For a fast and small code sequence, the user has to combine the time costs with the space costs. The number of memory references is probably also a good criterion, however this has to be verified experimentally.

When dealing with machines which use caches, pipelines (e.g. the VAX11) or multiple processors, the time part of the cost function is rather useless.

## 6.2. The Graph Transformer

In the Graph Transformer (GT), the compound nodes are expanded into subgraphs of single nodes, which can be handled by the PM. Figure 6.4 shows, as an example, the expansion of a compound node. In this expansion the future register use was taken into account, only one register will be used, as is easily verified.

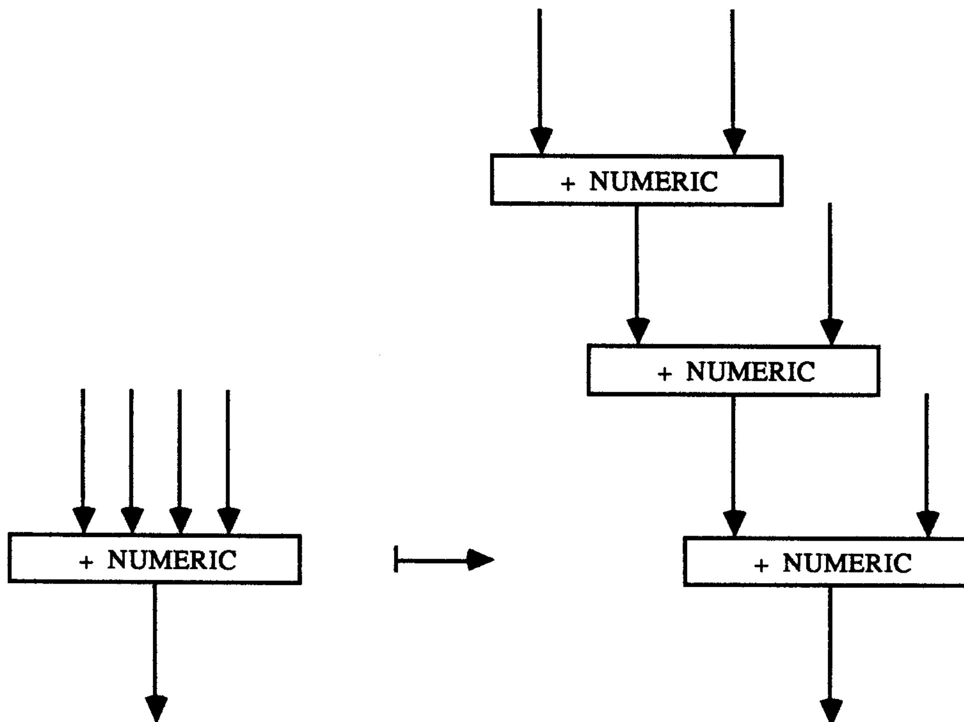



**Figure 6.4** Expansion of a compound node

Minimizing the number of registers required in order to evaluate an expression is an important issue in an optimizing compiler. Minimizing this use is a rather simple task when dealing with a single-processor machine. Machines, which have multiple arithmetic units, like the CDC 7600, are

most efficient when interlocks between the arithmetic units, as occurs in the expanded graph in figure 6.4, are avoided. Figure 6.5 shows the optimal subgraph for such a machine. Although this computation uses two registers, it will be faster executed than the computation discussed above. Machines with multiple arithmetic units are left for future research.

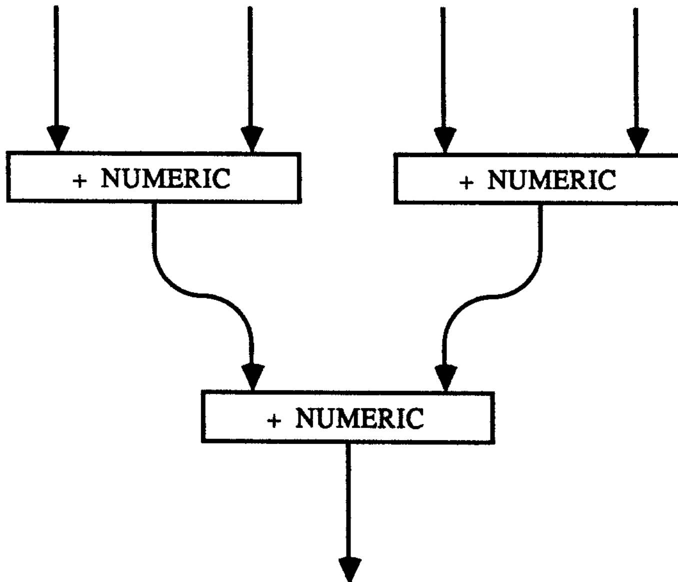



**Figure 6.5** A parallel computation

Operators present in the DGR but not on the target machine could also be expanded in this phase. However, it may often be preferable to implement them as 'primitive procedures' in the target machine description (see chapter 7).

### 6.2.1. The Transformed Dataflow Graph Representation

The GT produces a representation called the *Transformed Dataflow Graph Representation* (TDGR). The syntax of the TDGR-language is the same as the syntax of the DGR-language as described in appendix B. The fact that the compound operators are expanded in the TDGR, does not affect the description language, as is easily verified.

## 6.3. The Pattern Matcher

The goal of the Pattern Matcher (PM) is to find a covering of all the basic blocks of the TDGR with subgraphs, representing addressing modes, machine instructions and primitive procedures (all of them discussed in chapter 7). The PM is table-driven. All information the PM needs, i.e. the descriptions of the subgraphs and some additional information like conditions and evaluation rules, is found in the table of the PM. The PM has no sense of what some subgraph is representing, an addressing mode, a machine instruction or a primitive procedure. In general more than one covering is possible, the PM tries to select an optimal covering. The PM uses a heuristic to implement the order in which the nodes in the TDGR are covered. Before discussing the order of covering, the idea of covering is explained.

Let G be a graph and let V be a set of graphs. Graph G can be *covered* by elements of V, when V contains a multiset of graphs which can be connected into a larger graph equal to the graph G. It is assumed that each node in graph G, is part of one and only one subgraph which is covered by one element of set V, i.e. the covering of graph G contains no overlaps.

### 6.3.1. The order of covering

In this section some methods, implementing the order of covering, are discussed. There is no preference for one method, experimental implementations should provide the best method in general.

Notice that the basic blocks have strong borders, which means that the PM may not cover some nodes in one basic block and some in another, in one match.

#### 6.3.1.1. The exhaustive search

The straightforward implementation of the order of covering is just trying all the possibilities. Since the number of possible coverings is finite, this method will certainly produce an optimal covering. In general such an exhaustive search is hardly practical in a compiler. On the other hand, if the search could be restricted to a small subclass and still find an optimal covering, an exhaustive search might be practical. Another observation is that most of the basic blocks are rather small, which can also make an exhaustive search practical.

This method is the only one, discussed in this section, which really produces an optimal covering. All the other methods produce a covering which is expected to be good or almost optimal.

#### 6.3.1.2. The machine dependent method

In this method, the nodes in the dataflow graphs are covered in an order, which can be different for each target machine. Each node (operator) of the DGR-language has a class-identification-number (CIN) which is machine independent. For each target machine, there is a map

$$\text{CIN} \rightarrow \text{MCIN}$$

where MCIN is the machine dependent class-identification-number. The order of covering is implemented by using the MCIN's. First cover the nodes with the highest MCIN in an arbitrary order, then cover of the other nodes, the nodes with the highest MCIN, and so on. A node which can only be covered by one subgraph should have a high MCIN. In such a case there is no choice between possible subgraphs, and so the node has to be covered with this subgraph. A node which can be covered by a large subgraph, should also have a high MCIN. Using large subgraphs, as a cover, will normally generate less code.

This implementation does obviously not have to result in an optimal covering. When the PM selects a subgraph, it looks at the costs of that subgraph only, and not at the costs of the rest of the graph. It is possible that choosing a low cost covering for a particular subgraph can increase the costs of the covering of an other subgraph. In order to produce a better covering, the PM can consider the context of a node or group of nodes. However, this context has to be small, otherwise it will be very time consuming.

#### 6.3.1.3. The top-down method

In the top-down method, the nodes in the dataflow graphs are covered in an order in which the graphs can be executed by a dataflow machine. Figure 6.6 shows a dataflow graph and a boundary, which diverts the graph in the nodes already covered (above the boundary) and the nodes which are going to be covered (below the boundary). While the dataflow graph is covered, the boundary moves from the top to the bottom of the graph.

The implementation of this method is best explained via an example. Figure 6.7(a) pictures a node, its input and output arcs, and the boundary. The boundary intersects the input arcs of the node, which means that the subgraphs which produce the data for the input arcs are already covered. Figure 6.7(b) shows the situation after the node is covered.

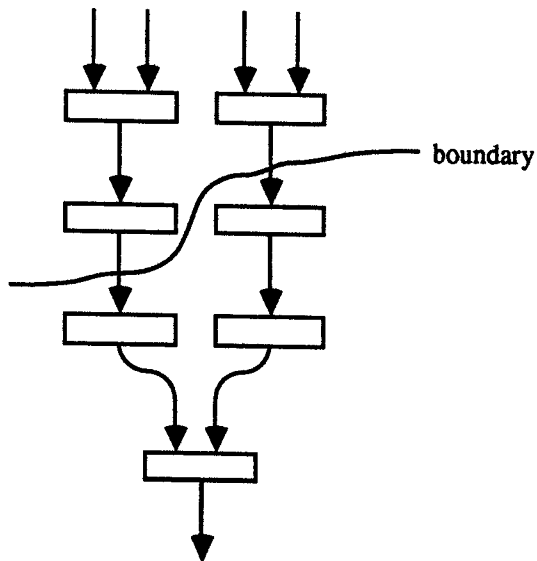



**Figure 6.6** The boundary in a basic block

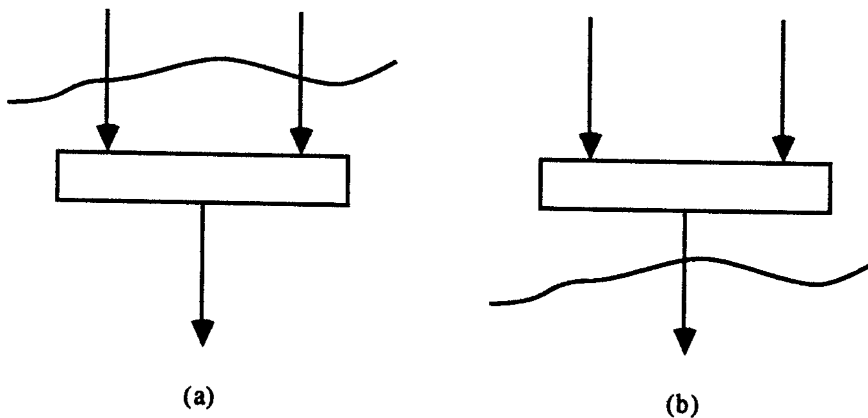



**Figure 6.7** The covering of a single node

A node may be covered when all its input arcs are intersected by the boundary. After the node is covered, the boundary intersects all its output arcs instead of the input arcs. To implement this heuristic, a collection of input arcs is needed to start the covering algorithm. The start collection consists of all the input arcs of all the source nodes, i.e. all the input data of the basic block. Figure 6.8 pictures a dataflow graph and the boundary at the start situation. The +node is covered as soon as the boundary intersects both its input arcs. Notice that, due to the constant node, this will never happen. To avoid this kind of starvation, all the constant nodes have a dummy input arc which is inserted into the start collection of input arcs. These dummy input arcs are used for implementation reasons only, therefore they are not pictured in the dataflow graphs.

### 6.3.1.4. The bottom-up method

In the bottom-up method, the nodes in a dataflow graph are covered by using the demand graph. This method starts with the collection of all output arcs of all sink nodes, i.e. all the output data of the basic block. Instead of moving from the top to the bottom, as in the top-down method, the

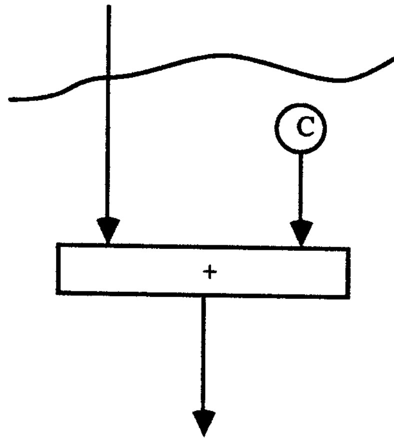



**Figure 6.8** An example of starvation

boundary moves from the bottom to the top.

A node may be covered when all its output arcs are intersected by the boundary. After the node is covered, the boundary intersects all its input arcs instead of the output arcs. Notice that starvation due to the constant nodes is avoided automatically in this method.

### 6.3.1.5. The bootstrapping method

This method is based on the observation that most of the addressing modes can be simulated by a sequence of simple addressing modes and simple instructions. In this method the dataflow graph is covered with small subgraphs, using one of the other methods. Then covered groups of subgraphs are covered themselves by larger subgraphs. The advantage is that the PM can make a fast first cover, and after that spent all its time optimizing the covering.

### 6.3.1.6. Backtracking

Apart of the exhaustive search, all the methods discussed above use *backtracking*. The PM first covers the graph using a method as discussed above, then it tries to find another covering with fewer costs. The process of backtracking has to be terminated, otherwise it will be very expensive. In order to terminate the process some limit, like a time limit is used. When the limit is reached, the process of backtracking will terminate.

The implementation of the backtracking method is as follows. The total costs, called C, of the covering of the graph are predicted. When the covering of the dataflow graph shows that the real costs are larger than C, then the backtracking process is started. When a covering with costs less or equal to the cost limit C is reached, the backtracking process is terminated. When the cost limit C is not reached after some amount of time T, then C is increased, to prevent domination of the backtracking process, and the process continues with a new amount of time T. This is continued until the process is terminated.

Backtracking as presented thus far, tries to find a better covering for an already covered graph. So, it is an extra phase after a graph is covered. It is also possible to use backtracking while the graph is being covered for the first time. While trying to cover a node, backtracking on the already covered nodes can be used in search for a globally less expensive covering.

## 6.3.2. The matching of a node

The matching of a node or group of nodes consists of three phases: the pure pattern match; the evaluation of the conditions; the evaluation of the attributes.

Using a method implementing the order of covering, the PM finds some node to be covered. The PM searches the table for patterns which match the node or which match a subgraph in the TDGR that contains the node. Each pattern in the table can have some conditions. These conditions are evaluated as a semantic test. If the condition is false, the pattern is rejected. If the condition is true, the costs of the match are calculated. If the pattern covers a subgraph containing already matched nodes, then the cost calculation can possibly use backtracking as discussed above. The pattern with the lowest costs is selected. When it is cheaper the PM may introduce common subexpressions which were earlier eliminated by the Global Optimizer, e.g. small constants. The source pattern in the TDGR is replaced by the target pattern and possible attributes are evaluated.

Notice that the source pattern in the TDGR is not really replaced by the target pattern, because this would make backtracking impossible. Instead of chancing the dataflow graph of the TDGR into covered graphs, a new graph is constructed.

Before trying to cover the dataflow graphs of the TDGR with subgraphs, it has to be sure that it is actually possible. The front-end and GT assure that the dataflow graphs of the TDGR are sentences of the TDGR-language. If each of the operators (including their operands) of the TDGR can be covered by subgraphs of the machine description (which is easily verified) then the dataflow graphs of the TDGR may be covered.

For the sake of clarity, two examples are presented. For each example, the dataflow graph and a covered graph are pictured in a figure.

The first example concerns the statement x := x+3. Figure 6.9 pictures the dataflow graph representing this statement. The dataflow graph can be covered in different ways. For example by a graph representing the autoincrement addressing mode, the displacement addressing mode, the increment machine instruction or the addition machine instruction with the literal/immediate addressing mode. All of the four possibilities have a graph which matches the dataflow graph of the example, as is easily verified. The PM evaluates the conditions of the target graphs. The increment machine instruction, for example, has the condition that the constant value has to be one. While in the example this condition evaluates false, the increment instruction is rejected as a match. Also the autoincrement addressing mode is rejected because in this case the constant value has to be 1, 2, 4, 8 or 16. The PM chooses the covering with the lowest costs. Figure 6.9 pictures a covered graph using the addition instruction with the literal addressing mode. The attribute of the addressing mode, i.e. the attribute *constant-value* is evaluated.

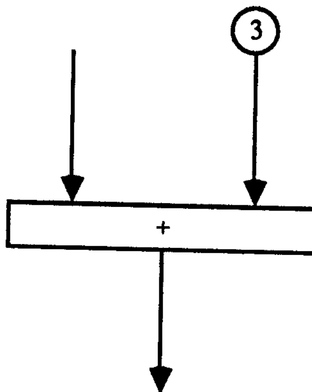



**Figure 6.9** The dataflow graph of example 1

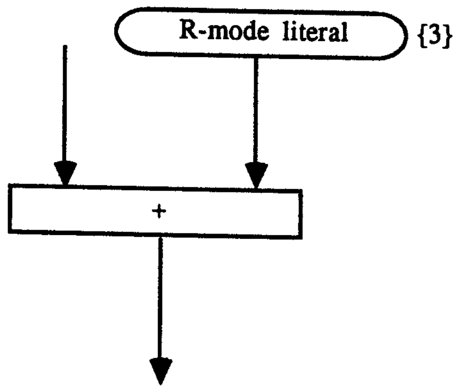



**Figure 6.10** A covered graph of example 1

The second example concerns the basic block representing the statements

```
w := a[i];
a[j] := x;
y := a[k]
```

Figure 6.11 pictures the dataflow graph and figure 6.12 pictures a possible covered graph. Notice that the Global Optimizer has eliminated the common subexpressions base(a) and size(a) in the DGR. Also notice that the arc, implementing the datastructure in the dataflow graph, is also present in the covered graph to ensure that the object code will be generated in a correct order. In the DGR only the update operator produce a data object, representing a datastructure, in the covered graph this is done by each subgraph which covered a DGR-graph containing an update operator. In the DGR only dereference operators consume data objects, representing datastructures, in the covered graph this is done by each subgraph which covered a DGR-graph containing a dereference operator.

### 6.3.3. The Covered Dataflow Graph Representation

The output of the PM is called the *Covered Dataflow Graph Representation* (CDGR). The CDGR-language is constructed in the same way as the DGR-language. It differs only in the description of the primitive types. Although the syntax of the CDGR-language is almost equal to the syntax of the DGR-language, it should be clear that the CDGR of a source program differs largely from its DGR. The DGR-language is machine independent, whereas the CDGR-language is machine dependent. As an example, figure 6.13 shows the CDGR description of the covered graph of the second example above.

## 6.4. The Instruction Generator

The Instruction Generator (IG) generates an assembly code sequence out of the CDGR. The order of the assembly instructions in the code sequence should be an order in which the DGR could have been executed by a dataflow machine. Normally, there is more than one possible sequence and the IG selects the one with the lowest costs.

Descriptions of specific assembly languages can be found in various books and manuals. In this report, the assembly language for the VAX11 is used in the examples. The VAX11 assembly language can be found in, for example, [Digi81] and [Lemo83]. The convention used in this report is that r0, r1, r2, and so on, denotes the registers, and symbolic names (e.g. result) denotes the memory locations.

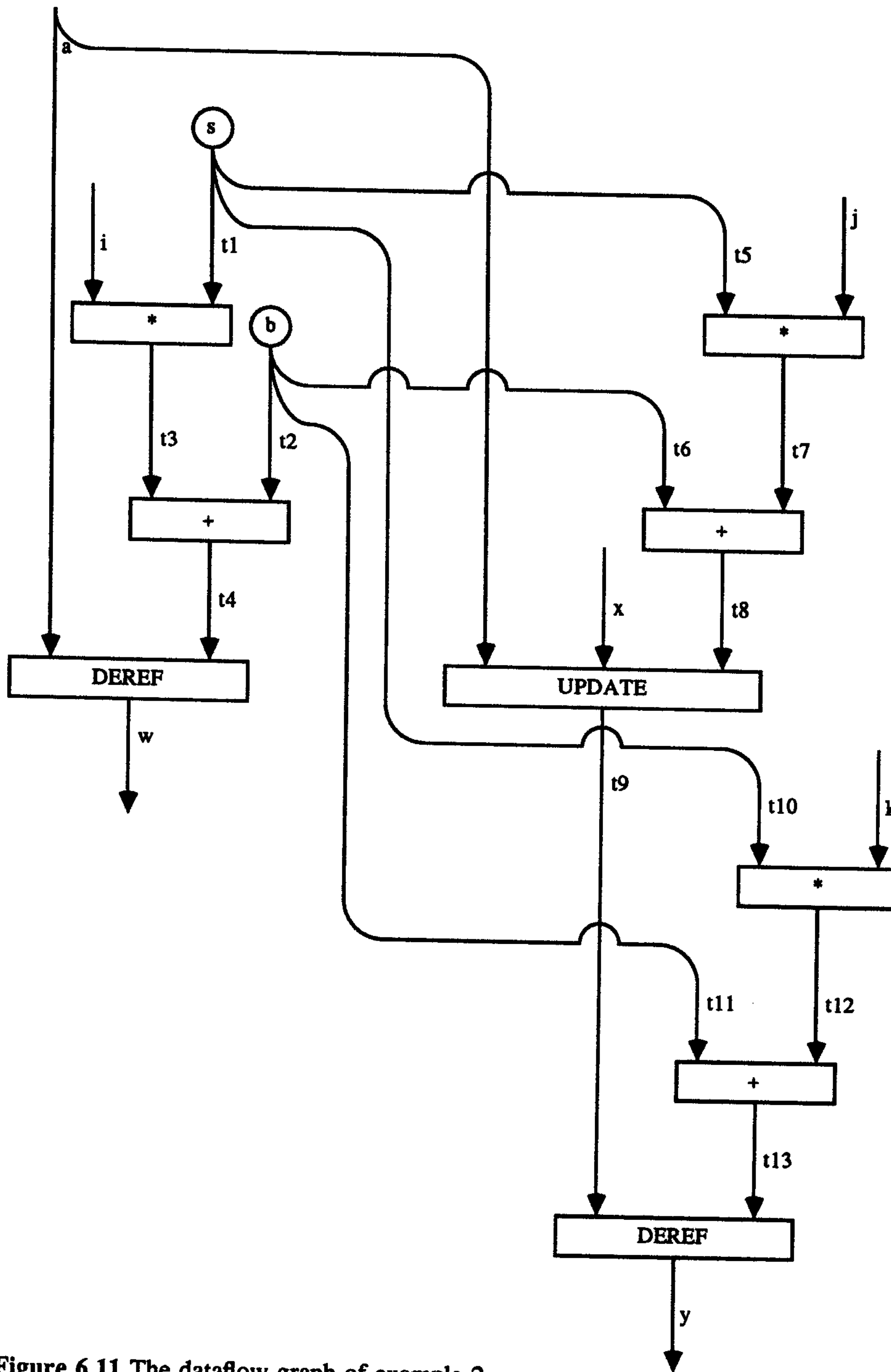



**Figure 6.11** The dataflow graph of example 2

As stated before, the order in which assembly instructions are generated, must be an order in which the DGR could have been executed by a dataflow machine. To ensure this order, the assembly code is generated in an order implemented by a heuristic like the top-down method discussed above. A boundary is used to divert the CDGR in a part for which the assembly code is already generated (above the boundary) and a part for which the assembly code is going to be generated (below the boundary). Assembly code may be generated for a node (or subgraph) when all its input arcs are intersected by the boundary. After the assembly code is generated for the node (or subgraph), the boundary intersects all

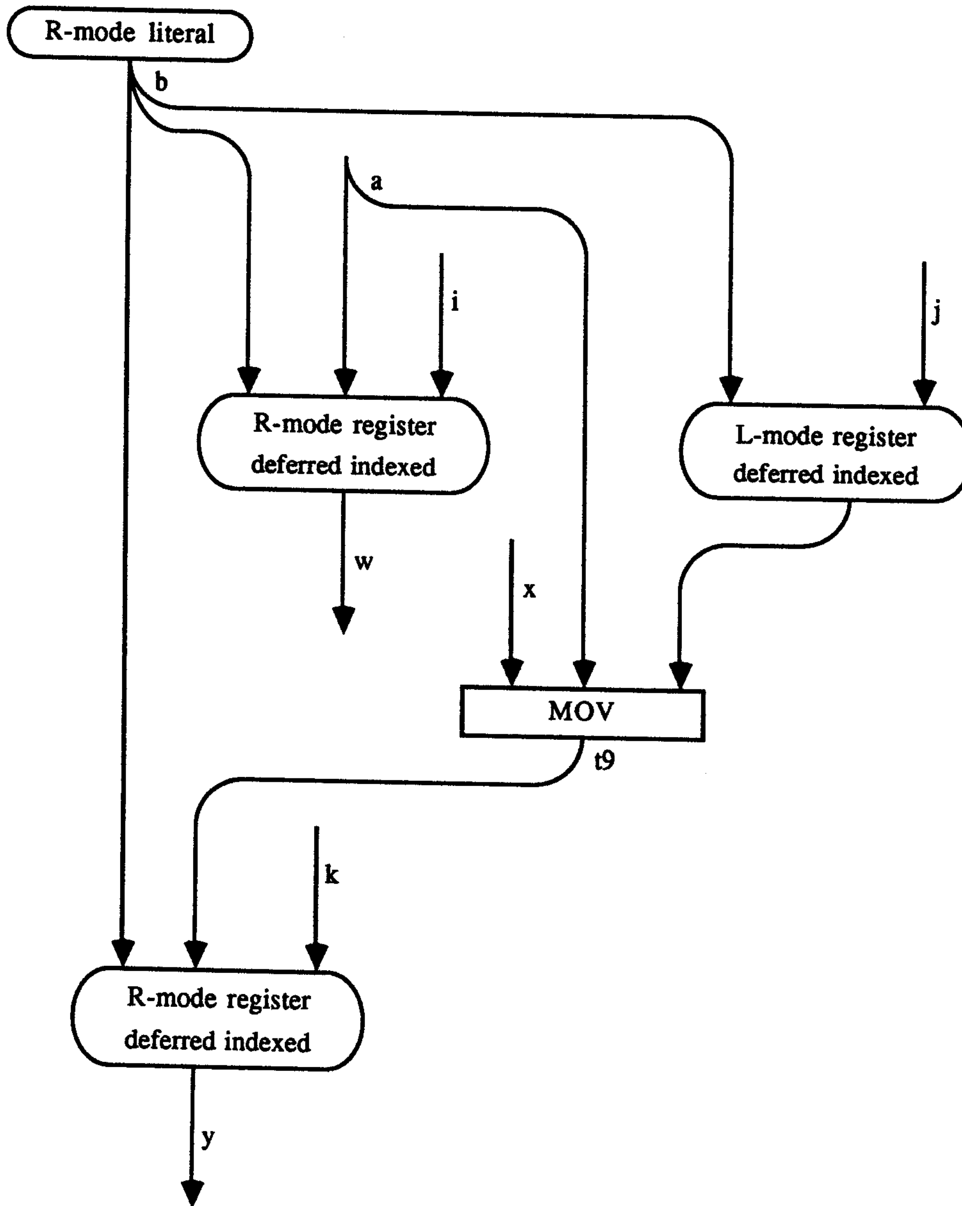



**Figure 6.12** A covered graph of example 2

its output arcs instead of the input arcs. When the code generator has to generate assembly code for some node (or subgraph), it scans the IG-table for a matching subgraph. The table provides the code generator with the associated assembly code.

Although code generation presented thus far is quite straightforward, there are some problems. The first problem is that not all the assembly code to be generated has an associated node (or subgraph) in the CDGR. For example, the register addressing mode is not represented in the CDGR. Also MOVE instructions may be necessary in the assembly code while they are not present in the CDGR. Further more, the PM only covers the basic blocks and no other construction blocks, although assembly code has to be generated for them also.

The second problem is the register assignment. Although the PM has no sense of registers, it uses for example addressing modes, which can only be performed using registers, as a cover. The IG has to provide the addressing modes with the necessary registers.

The third problem is that the code generator should generate (almost) optimal object code.

```
graph example_2
    {(a numeric_array integer normal_integer),
     (i numeric integer normal_integer),
     (j numeric integer normal_integer),
     (x numeric integer normal_integer),
     (k numeric integer normal_integer) →
     (w numeric integer normal_integer),
     (y numeric integer normal_integer),
     } =
    [t1 = constant()
     w = Rregdefind(t1,a,i)
     t2 = Lregdefind(t1,j)
     t3 = move(x,a,t2)
     y = Rregdefind(t1,t3,k)]

    graph constant
    { → (v1 numeric integer normal_integer b)} =
    [primitive_node
     type = R-mode_literal]

    graph Rregdefind
    {(v1 numeric integer normal_integer),
     (v2 numeric_array integer normal_integer),
     (v3 numeric integer normal_integer) →
     (v4 numeric integer normal_integer)} =
    [primitive_node
     type = R-mode_register_deferred_indexed]

    graph Lregdefind
    {(v1 numeric integer normal_integer),
     (v2 numeric integer normal_integer) →
     (v3 numeric integer normal_integer)} =
    [primitive_node
     type = L-mode_register_deferred_indexed]

    graph move
    {(v1 numeric integer normal_integer),
     (v2 numeric_array integer normal_integer),
     (v3 numeric integer normal_integer) →
     (v4 numeric_array integer normal_integer)} =
    [primitive_node
     type = MOV]
```

**Figure 6.13** The CDGR description of example 2

This involves, for example, the use of condition codes and the use of 2-operand instructions instead of 3-operand instructions.

Before code generation is discussed further, the register assignment and the use of condition codes are discussed.

## 6.4.1. Register assignment

The cost of the object code depends largely on the use of the registers. A *register manager* is used to achieve optimal register usage in the object code. The objective is to store almost no objects in memory locations, but to use the registers as much as possible. The register manager keeps track of the contents of all registers, and places information concerning the registers in a *register table*. The register table is a description of a runtime situation, made during code generation. The use of this runtime description aids in the generation of good local code. The register manager handles the requests for registers, using some *register replacement policy.*

### 6.4.1.1. Description of the register table

The register table contains for each register six fields namely: the register field, the in-use field, the contents field, the memory field, the ref-count field and the last-ref field.

The *register field* describes the identity (i.e. the number) of the register and its class (i.e. data or address register).

The *in-use field* describes the status of the register. If it contains 'no' then the register is free, otherwise it contains 'yes' and then the register is, in some way, in use.

The *contents field* describes the contents of the register. For example: constant 100; address a; value a; result function f; result expression evaluation; unknown.

If the contents of a register can also be found in memory then its *memory field* contains the address, otherwise the field contains 'no'.

The *ref-count field* contains the reference count, which is the number of usages of the contents of the register. In a basic block the reference count can easily be determined, just count the output arcs of the output port of the node. When dealing with conditional blocks, it can be impossible to determine the usages of a register. In such a case, the number of possible usages is used as the reference count.

The *last-ref field* contains a number related to the last reference made to the register. A high number means that the register has been recently used.

Notice that the in-use field is superfluous. The field in-use contains 'no' if and only if the field ref-count contains 0 and the field contents contains 'unknown'. Although the in-use field is superfluous, it is inserted in the table for the sake of clarity.

The description of the register table is also called the *register configuration.* The description of the register table as present at the start of a block is called the *initial register configuration* of that block, and the description at the end of a block is called the *final register configuration* of that block.

### 6.4.1.2. Register replacement policy

When a register is needed the register table is searched for a free register. If such a free register is available, then the register manager allocates it. If no free register is available, the table is searched for registers, that are not referenced by one or more operands (i.e. registers which field ref-count contains 0). Of these registers the least recently used register (i.e. the register with the minimal last-ref field), is released. This choice is based on the clustering tendency, which means that the more recently the contents of registers are used, the greater the probability they will be used again. If still no register is allocated, the table is searched for registers which contents can also be found in memory (i.e. registers which memory field contains an address). Of these registers, the least important register (i.e. the register with the minimal ref-count) is released. Should there be more than one of such registers, then the least recently used is released, just as above. If such registers are also not available, then a register is dumped. The table is searched, as above, for the least important and least recently used register. The IG generates code to place the contents of the register in memory and the register is

released. This dumping is also known as *register spilling.*

Notice that a register with ref-count 0 does not have to be free. If its contents field for example contains constant 1, then this knowledge may be used later on, i.e. a load of a constant 1 in a register may be avoided. However, when all the common subexpressions including small constants are eliminated by the Global Optimizer, then a register with ref-count 0 will have no more possible usages, i.e. it is free.

In the register replacement policy, the *clustering tendency* is used. This means that when register spilling is necessary, the least recently used register is spilled. The idea is that the use of a register tends to cluster, so a register which has not been used for some time, will probably not be used in the near future. The advantage of the clustering tendency is that it is rather simple to implement. Instead of this method it would be better to dump the register whose next use is farthest away in the list of assembly instructions to be generated. It needs proper flow analysis of the DGR to achieve this. This is left for further research.

In the policy, described above, it was assumed that some general register was needed. There was no assumption about the type of the register. When a special register is needed, such as a data-register or the stack pointer, then the register manager takes that into account when it is using the register replacement policy.

#### 6.4.1.3. The validity of the register table

The register table is a description of a runtime situation, made and used during code generation. Therefore, the table will become invalid at points where this runtime situation can (practically) not be determined during code generation. At such points the register table is cleared and code is generated to dump the contents of the registers into memory, if necessary.

One of those points is at the generation of a procedure-call of an user-defined procedure. It is rather foolish to leave values in registers, to call a procedure and to expect that the procedure invocation will not affect the values and the descriptions of those registers. Therefore, just before a procedure-call is generated, the registers which can be affected by the procedure invocation are dumped into memory and their descriptions in the table are cleared. The effects, with respect to the register table, of a procedure invocation are unknown until the code, belonging to the procedure body, is generated. Therefore, the IG must generate the object code for procedure blocks prior to the call blocks. At the generation of a runtime-routine-call and at the generation of an exception-call, the same problem can occur.

The validity of the register table can be affected at the generation of a procedure body. Procedures can be called from many different contexts, therefore it is unwise to make assumptions about the register table at the start of the generation of a procedure body. The same is obviously true at the generation of a new module.

The validity of the register table can also be affected at the code generation of conditional jumps, like in an if-then-else block. The runtime situation depends on the result of the condition block, which is normally unknown during code generation. Instead of clearing the table at the end of an if-then-else block, the table is replaced by the intersection of the register table of the true construction block and the table of the false construction block. The same can be done at other places where jumps occur.

Another obvious reason why the register table might become invalid is the assignment of values to memory locations by processes that are not controlled by the program currently compiled. For example an I/O process that stores data in memory. Such memory locations should not be used.

#### 6.4.1.4. Other register assignment methods

A more advanced register assignment technique could use the next idea. When a register is needed and there are no free registers available, the register manager releases one, using the register replacement policy as described above, or it tells the PM to make a new, more expensive, match that

uses less or no registers at all. For example, suppose a group of nodes is matched with subgraph B with costs K, and the register manager has to release a register at costs C. A new match with a subgraph B2 is made. Let K2 be the costs of this match, and C2 be the costs of the register manager. Of course, K2 will be greater than K, because the PM matches as optimal as possible. However, if K2 + C2 is less than K + C, then the group of nodes is rematched with this new match.

In this method there is an interference of the PM and the IG. It can only be performed when the PM (using the top-down method) and IG are performed in one single pass.

Another implementation of this method is, that the PM always tries to make two matches, one normal match and one extra match using fewer registers. After the PM is finished, the IG starts to generate object code using the normal covering. When a register is needed a request for that register contains the costs of the instruction, which is being generated, when using the register, and also the costs of the second possible instruction used when the request is not granted. The register manager can now take into account that dumping a register in use, can be more expensive than not to dump a register and to perform the instruction without the register.

Sometimes it is known in advance, that spilling will be inevitable. For example in constructions as function calls. A set of conversions in the GT could try to avoid register spilling. This means that the GT inserts extra nodes in the dataflow graph to store and retrieve intermediate values. However, this means that the DGR should have some sense of registers, which is undesirable. Another disadvantage is that the extra inserted operators may be difficult to remove, when optimizing the code sequence. Therefore there is no attempt to avoid spilling in advance. See for example [Grah82] or [Grah84].

In some code generators, a distinction is made between register allocation and register assignment. Crawford defines, in [Craw82], *register assignment* as "the process of assigning a specific register to hold an intermediate result, locking that register until the last use of that intermediate value, and remembering when registers hold useful quantities". *Register allocation* is defined by Crawford as "the process of allocating register sets for use in computing sub-expressions, and invoking a register spill mechanism when a sub-expression is too complicated to compute given the fixed number of registers in the target machine". An example of such a code generator can be found in [Craw82]. The advantage of a code generator using a special register allocation phase, is that some global register assignment scheme can be used, while the Dataflow Graph Code Generator can use only local information. This disadvantage is largely solved by using the lazy generation technique (see section 6.4.3).

## 6.4.2. Condition codes

Apart of the register usage, the IG keeps up with the condition codes while generating object code. This is done to avoid redundant test (or compare) instructions.

### 6.4.2.1. Description of the condition code table

The condition code table for the VAX11 contains information about the following condition codes: the negative condition code (N), the zero condition code (Z), the overflow condition code (V) and the carry condition code (C). For an explanation of the condition codes, see for example [Digi81]. For each of the four condition codes, the table contains three fields.

- The *in-use* field, which specifies that the condition code may or may not be used to avoid redundant tests (i.e. it contains "yes" or "no").
- The *object* field, which specifies the object (i.e. the register or memory location) related to the condition code when it is in use (i.e. the in-use field contains "yes").
- The *contents* field, which specifies the content of the condition code (i.e. it contains 1, 0 or "unknown").

The contents field is used to replace a conditional branch by an unconditional branch.

#### 6.4.2.2. The validity of the condition code table

Like the register table, the condition code table is a description of a runtime situation, made during code generation. Therefore, also the condition code table will become invalid at points where this runtime situation can not be determined during code generation. Those points are the same as where the register table may become invalid, and therefore not discussed again.

### 6.4.3. Code generation

As stated before, the order in which assembly instructions are generated, must be an order in which the DGR could have been executed by a dataflow machine. To ensure this order, the assembly code is generated in an order implemented by a heuristic like the top-down method using a boundary. Where the PM could use backtracking, the IG can use a technique called *lazy generation*. When using lazy generation, the generated assembly code is put in a buffer. The code generator may change the assembly code in the buffer, before it is really generated, i.e. it leaves the buffer. This enables the code generator to produce better object code, because more contextual information is available. For example, special instructions can be used, like the VAX11's PUSHR which pushes multiple registers. Lazy generation can also perform (implicitely) optimizations which can not be performed by the Global Optimizer or the PM. Consider the program fragment

```
IF x = 0 THEN y := z+2 ELSE z := y+2;
x := x+1;
```

Figure 6.14(a) shows a possible assembly code sequence generated without using the lazy generation technique. Figure 6.14(b) shows the corresponding assembly code sequence generated using the lazy generation technique. The test instruction uses an autoincrement addressing mode, avoiding the increment instruction. Notice that this optimization can not be performed by a peephole optimizer, because a peephole optimizer uses a window of fixed length.

```
         TSTL  (r6)                     TSTL  (r6)+
         BNEQ  false                    BNEQ  false
 true :  ADDL  (r7),#2,(r8)     true :  ADDL  (r7),#2,(r8)
         BR    out                      BR    out
false :  ADDL  (r8),#2,(r7)    false :  ADDL  (r8),#2,(r7)
  out :  INCL  (r6)              out :

            (a)                            (b)
```

**Figure 6.14** An example using lazy generation

When it is cheaper the IG may, like the PM, introduce common subexpressions which were earlier eliminated by the Global Optimizer, e.g. small addressing modes.

#### 6.4.3.1. Code generation and register assignment

When an instruction is being generated, the code generator asks the register manager for a register to put the result of the instruction in. Suppose, the register manager allocates register r0. When another instruction is being generated using the result of the previous instruction as an operand, then this operand is found in register r0. However, some computers have runtime-routines or

instructions, which need specified registers. Suppose, a new instruction is being generated which needs its operand in, for example, register r1, then there are two possibilities: insert a move instruction from register r0 to r1, or tell the register manager to allocate register r1 for the first instruction. Which of the two is chosen depends on the costs and the context of the instructions.

Using the register table ensures that a value is never loaded from memory into a register when it is already present in that register. Nor will the value of a register be stored in a memory location when it is already present in that memory location.

A node (or subgraph) with more output arcs than input arcs will probably increase the use of registers. The code generator uses this kind of information to avoid possible register spilling.

When an assembly instruction is generated, the register table is updated. It should be noticed, that registers are not only directly affected by the instructions, but can also be affected indirectly by machine instructions or runtime-routines with have side-effects on registers, e.g. string manipulation instructions on the VAX11.

### 6.4.3.2. Code generation and construction blocks

The IG not only generates code for the covered dataflow graphs, but also for the block constructions. This last is done in a standard manner. The next figures show the kind of object code which is generated for a while-do block, a do-while block and an if-then-else block.

```
          jump label_1
label_2 : body
label_1 : control
          branch on true to label_2
```

**Figure 6.15** Object code for a while-do block

```
label_1 : body
          control
          branch on true to label_1
```

**Figure 6.16** Object code for a do-while block

```
          control
          branch on false to label_1
          body_true
          jump label_2
label_1 : body_false
label_2 :
```

**Figure 6.17** Object code for an if-then-else block

In a loop the initial and final register configuration of the loop body should match with each other, to avoid extra move instructions. At the entry of the loop body the initial register configuration

is known, and thereby the desired final register configuration, called the *boundary condition*.

If in an if-then-else block, an object is defined in the THEN-part as well as in the ELSE-part, then the same register (or memory location) should be used, to avoid extra move instructions. If in an if-then-else block, an object is defined in the THEN-part (or ELSE-part) which has a simulated definition (see chapter 4) in the ELSE-part (or THEN-part), then the same register (or memory location) should be used as the one used by the object prior to the block.

### 6.4.3.3. Code generation and condition codes

When dealing with a loop construction like in figure 6.16, a test instruction can may be avoided by using the condition code table. Suppose, the last instruction of the body is DEC z, then a test instruction like TST z is redundant. Also when dealing with an if-then-else construction, like in figure 6.17, a test instruction may be avoided. Suppose, the last instruction before the if-then-else construction is ADD r0,r1,r2, then a test instruction like CMP r2,#0 is redundant.

Most of the instructions have some effect on one or more of the condition codes. These effects are described in the TMD. When an instruction is generated, the condition code table is updated. For example, suppose a MOV instruction is generated, then the N, Z and V codes are updated and the C code remains unchanged.

When dealing with a basic block, i.e. a straight line of code, it is easy to maintain the condition code table. However, when construction blocks are involved, it gets harder. For example, in a construction like a WHILE-DO loop, the control-part has two previous straight lines of code, i.e. the code of the body-part and the code previous to the loop construction. Both lines of code produce a condition code table, the intersection of them is used as the new condition code table for the control-part. However, using the intersection as the new condition code table can sometimes result in rather poor object code. Consider the program fragment

```
z := x+y;
WHILE y ≠ 0 DO y := y-1;
```

Figure 6.18(a) shows the assembly code generated using the intersection as the new condition code table for the control-part of the WHILE-DO loop. Figure 6.18(b) shows an equivalent, yet better, assembly code sequence. Instead of just taking the intersection as the new condition code table and continuing generating code, a test instruction was inserted in the code previous to the WHILE-DO loop to produce a condition code table equivalent to the table produce by the loop body. Such insertions should, of course, only be used when it provides better code, otherwise just the intersection is used.

```
           ADD   x,y,z                    ADD   x,y,z
           JMP   label_1                  TST   y
label_2 :  DEC   y                        JMP   label_1
label_1 :  TST   y             label_2 :  DEC   y
           BNEQ  label_2       label_1 :  BNEQ  label_2

              (a)                            (b)
```

**Figure 6.18** Assembly code using the condition code table

### 6.4.3.4. The information transformation

The information as present in the CDGR is transformed to machine dependent information during code generation. An example is the transformation of the IR data-types to machine dependent

data-types. To perform this transformation, the GT contains (just like in the front-end) a Data-Type Transformation table which contains all the intermediate representation information and the corresponding machine dependent information. As an example, figure 6.19 pictures some of the transformations for the VAX11. This table is of course in reality implemented as a part of the IG-table.

| | | |
|---|---|---|
| small_integer | becomes | byte |
| normal_integer | becomes | word |
| long_integer | becomes | longword |
| long_long_integer | becomes | quadword |
| long_long_long_integer | becomes | octaword |
| normal_real | becomes | F-floating |
| long_real | becomes | D-floating or G-floating |
| long_long_real | becomes | H-floating |

**Figure 6.19** Examples of data-type transformations

### 6.4.3.5. Three and two operand instructions

Some computers, like the VAX11, have both 3-operand and 2-operand instructions. For example:

ADD operand1, operand2, destination

which means destination := operand1 + operand2, and

ADD operand1, destination

which means destination := operand1 + destination. A 2-operand instruction costs less than a 3-operand instruction. So, if a source operand is equal to the destination operand, then the 2-operand instruction should be used. Of course, only 3-operand instructions could be used and the PO could convert them to 2-operand instructions, whenever possible. However, this would increase the total compilation time and the code generator would not be an optimized code generator anymore.

When the IG has to choose between a 2-operand and a 3-operand instruction, it takes a look at the operands. Suppose one of them is placed in a register and its ref-count = 1, then the IG tries the 2-operand mode.

### 6.4.3.6. User-defined procedures

The usual technique for procedure entry is to have standard preludes to perform the housekeeping, and for procedure exit to have standard postludes. Those preludes and postludes should cover all the different types of procedures which can be defined by the user. Standard preludes are: save the environment, save the return address, jump to procedure body, set up local stack frame, etc. Standard postludes are: clear the local space, restore the environment of the caller and jump to the return address. Part of this housekeeping is performed in the call block of the DGR. A straightforward code generation of the call block is undesirable, because the same housekeeping code will be present at each call, thereby increasing the size of the program. Code to save the callers environment, for example, could be moved into the body of the procedure.

### 6.4.4. Storage allocation

In addition to register assignment, there is a phase called storage allocation. Storage allocation is performed by the *storage allocator* which reserves memory locations for data which can not be stored in registers. Just like the register manager, the storage allocator uses a description table. The table describes the objects stored in memory locations. Although it is not as important as in the case of registers, the storage allocator uses the memory locations as optimal as possible. Storage allocation is not discussed any further, because of its minor importance in this report.

## 6.5. The Peephole Optimizer

The Peephole Optimizer (PO) scans the assembly code sequence, generated by the Instruction Generator, in search of certain patterns of instructions and replaces such patterns by cheaper instructions. Although the PO can perform highly machine dependent optimizations, its performance will be rather poor because most of the optimizations are already performed. Not only the Global Optimizer performs optimizations, but also the front-end, the PM and the IG. The optimizations (implicitely) performed in the PM and the IG, are typical peephole optimizations, as will be shown in some examples later on. Experiments should point out the (un)usefulness of the PO.

The PO is table-driven, it uses a table of pairs

<source pattern, target pattern>

in which each source pattern describes a sequence of one or more assembler instructions (i.e. the instruction mnemonics and the operands) and optional conditions, and each target pattern describes an equivalent, yet cheaper, sequence of zero, one or more assembler instructions.

The PO maintains a window, which moves over the assembly code. If the window contains an instruction sequence which matches some source pattern in the table, then the sequence is replaced by the corresponding target sequence. Furthermore, the window moves a few instructions backwards, so that the last instruction in the window is the first instruction of the inserted sequence. This is essential because it is possible that source instructions that were rejected earlier now do match together with some of the inserted instructions. If the window contains no instruction sequences which match a source pattern in the table, then the window moves one instruction forward.

Pattern matching is performed in three steps:

- Find the source patterns in the table whose mnemonics match the mnemonics in the current window.
- Check that the operands of the source pattern match the operands of the instructions in the current window.
- Check that the condition is satisfied (i.e. it evaluates to TRUE).

As an illustration, the next figures show three examples for the VAX11. The original code sequences are pictured on the left and the optimized code sequences are pictured on the right. Notice that the optimizations in example 1 and example 3 are normally already performed by the PM, and that the optimization in example 2 is normally already performed by the IG. It is hard to find an example of a peephole optimization which is not already performed, assuming that such an example exists.

```
MOVL    (r0),r1
INCL    r0              MOVL    (r0)+,r1
```

**Figure 6.20** Peephole optimization example 1

The PO needs the register manager, as described in section 6.4., to perform its optimizations correctly. Suppose the PO scans a code sequence and it finds a load of a value into a register, which

```
INCL    r0
CMPL    r0,#0       INCL    r0
BLSS    label       BLSS    label
```

**Figure 6.21** Peephole optimization example 2

```
INCL    r0
CMPL    r0,#10
BLSS    label       AOBLSS  #10,r0,label
```

**Figure 6.22** Peephole optimization example 3

appears to be redundant within the peephole window. For example, with a window of size two, consider the sequence

```
MOV @#address, R0
ADD R0, R1
```

If the PO replaces the pattern by the cheaper instruction

```
ADD @#address, R1
```

then the load is removed while the value loaded by this load might be used later in the code, for example in the sequence

```
MOV @#address, R0
ADD R0, R1
MUL R0, R2
```

The PO can find such information in the register table.

The PO-table should not contain just a few hand-written optimizations, but all possible optimizations. To obtain all the possible peephole optimizations, the PO-table is generated out of the TMD. This automatic generation of peephole optimizers is extensively discussed in papers of Davidson and Fraser, for example [Davi80] and [Davi84].

# CHAPTER 7
# THE MACHINE DESCRIPTION

In this chapter, we formally define the target machine. A machine description specifies the types and accessibility properties of data on the machine, and the properties of the machine instructions. The actions performed by the machine instructions and addressing modes are described in terms of DGR primitive operators. The advantage of representing the instructions and addressing modes as dataflow graphs is that it is quite easy to match the target patterns against the source patterns which can be implemented by the machine instructions and addressing modes. Both the dataflow graphs in the TDGR and in the machine description use the DGR primitive operators to make this matching possible.

The target machine description (TMD) consists of four parts: the description of the machine architecture, the description of the set of addressing modes, the description of the machine instruction set and the description of the library of run-time routines.

## 7.1. The machine architecture

This part of the TMD describes the architecture of the target machine as far as relevant for the code generator. The description contains mostly information about the registers, like the identifier, the type, the size and a possible special designation of a register. The syntax of this description is presented in appendix C. As an example, appendix D describes the machine architecture of the VAX11.

## 7.2. The addressing modes

There are two classes of addressing modes. The first class, called *L-addressing modes,* consists of addressing modes which produce an address. These addressing modes are mostly used to obtain the destination operand of a machine instruction. The second class, called *R-addressing modes,* consists of addressing modes which produce a value. These addressing modes are mostly used to obtain the source operands of a machine instruction. Consider the Pascal fragment

x := y

which means "store the value of variable y at the address of variable x". To obtain the address of the variable x, i.e. the variable on the left, a L-addressing mode is used, and to obtain the value of variable y, i.e. the variable on the right, a R-addressing mode is used. The L-addressing modes are not only used to obtain destination operands, but also to obtain source operands for address instructions (e.g. MOVA and PUSHA of the VAX11). Although in the assembly languages there is normally no visual difference between L-addressing modes and R-addressing modes, there is quite a difference in the DGR-language.

The addressing modes are described using the same formalism as in the DGR-language. The description of an addressing mode contains not only a description of its graph but also a list of conditions, its assembly code and the cost function. The description of an addressing mode may be factorized into smaller graphs. This means that the graph of an addressing mode can have nodes which are graphs themselves. The syntax of this description language is presented in appendix C.

As an example, the addressing modes of the VAX11 are described in appendix E. This description uses a factorization as mentioned above.

## 7.3. The machine instruction set

The machine instruction set describes the machine instructions and some additional information. For each machine instruction, its description contains:

- A representation describing the operation performed by the machine instruction. This representation uses the same formalism as the DGR-language.
- A sequence of conditions, which specify for example the addressing modes which may be used as an operand.
- A list of specific registers, which are needed in the computation. The list is divided in a part containing the registers needed for input, a part containing the registers for internal use and a part containing the registers needed for output. This information is mostly needed when describing special instructions, like character string instructions.
- A list of specific register classes, which are needed in the computation. The list is divided, like above, in a part containing the classes needed for input, a part containing the classes for internal use and a part containing the registers needed for output. This information is needed on machines with instructions which only operate on special classes of registers, for example a multiply instruction which needs its first operand in an even register.
- The assembly code.
- A description of the condition codes.
- A cost function, as described in chapter 6.

The syntax of this description language is presented in appendix C. As an example, appendix F describes some machine instructions of the VAX11.

## 7.4. The library of runtime routines

This part of the TMD describes all the built-in functions (e.g. SIN, COS, etc.) and IO-routines that are supported by the target machine. The description of routines is equivalent to the description of machine instructions. Appendix C presents the description language of the routines.

It is rare for machines to provide instructions which can deal directly with all the requirements of high-level languages. Operations present in the DGR-language without a corresponding machine instruction are handled by special subroutines, called *primitive procedures* or *pseudo-instructions*. The code generator refers to the primitive procedures as though they are machine instructions. The cases in which primitive procedures are required, commonly include string manipulation and set operations. Primitive procedures are part of the library of runtime routines, and therefore they are called in the object code as any other routine. To improve the execution speed, the primitive procedures may be included in the set of machine instructions instead of in the library of runtime routines. Then the code of the primitive procedures is placed in-line in the object code, instead of being called as a routine.

Instructions not available on the target machine could also be expanded by the GT in the DGR. However, the compilation time of each program would increase.

# CHAPTER 8
# SUMMARY AND FUTURE RESEARCH

## 8.1. Summary

The design of the Dataflow Graph Code Generator has been presented. The code generator is part of a compiler which consists of a machine independent front-end, a machine and language independent global optimizer, and a language independent back-end. A construction of dataflow graphs, called the Dataflow Graph Representation, is used as an intermediate representation.

The front-end performs implicitely some optimizations during the construction process of the intermediate representation. These implicitely performed optimization techniques include: value propagation, constant folding and constant propagation.

The Dataflow Graph Representation is the language and machine independent interface between the front-end and the back-end. The DGR is a dataflow graph in which the nodes are dataflow graphs themselves. For the sake of clarity, a block description is used to describe the DGR. The DGR may be visualized as an abstract dataflow machine. Both retargetability and optimizations are supported by the DGR. The DGR can represent almost all common language constructions, including higher order pointers.

The Global Optimizer performs some optimization techniques on the DGR, including: common subexpression elimination, inline substitution and various loop optimization techniques.

In the back-end, the code generation algorithm is separated from the target machine data that drive the algorithm. The target machine data are stored in different tables. The back-end consists of four modules: the Graph Transformer, the Pattern Matcher, the Instruction Generator and the Peephole Optimizer.

The GT expands the compound nodes in the DGR into subgraphs of single nodes. The GT produces the Transformed Dataflow Graph Representation.

The PM covers all the basic blocks of the TDGR with subgraphs representing addressing modes, machine instructions and primitive procedures. In general more than one covering is possible, the PM tries to select an optimal covering. The PM uses a heuristic to implement the order in which the nodes in the TDGR are covered. Backtracking may be used to find a better covering for an already covered graph. The PM produces the Covered Dataflow Graph Representation.

The IG generates an assembly code sequence out of the CDGR. Normally there is more than one possible code sequence, the IG produces the one with the lowest costs. In order to produce correct and optimized code, the IG uses a register table and a condition code table. Both the tables are a description of a runtime situation, made during code generation. Lazy generation may be used to find a better object code sequence.

The PO performs peephole optimizations on the object code sequence. Its performance will be rather poor because most of the optimizations are already performed in the front-end, the Global Optimizer, the PM and the IG.

The target machine description specifies the types and accessibility properties of data on the target machine, and the properties of the machine instructions. The TMD uses the same graph description formalism as the DGR. A table constructor (not discussed in this report) produces, out of the TMD, the tables for the modules of the back-end.

## 8.2. Future research

As the reader would have noticed, a number of areas has been left for future research. In this section some of those areas receive special attention.

### 8.2.1. The DGR-language

The data-types of the operands and operators are specified in attributes. Each operand and operator is associated with some of these attributes. In this report, all the attributes are described next to each operand and operator. It would be more convenient if each operand and operator is associated with a predefined set of attributes. Modifications, like inserting an extra attribute, are then much easier to perform, only the set definition is modified instead of all the occurences.

Both the DGR and the TMD use primitive operators. Although the primitive operators are known, a graph has to be defined each time a primitive operator is used. It would be convenient if a library of such graphs is available.

### 8.2.2. The Graph Transformer

In this report, the DGR and TDGR contain no comparison operators, therefore the object code sequence can unnecessary contain the same compare instruction more than once. Consider for example the program fragment

```
IF a < b THEN c := 1;
IF a > b THEN c := 0;
```

Although the DGR and TDGR contain the two different boolean operators <numeric and >numeric, the object code sequence will contain the same two 'compare a with b' instructions. Most of the unnecessary compare instructions can be removed during lazy generation, however this is a time consuming technique. A simple solution is to let the GT introduce compare operators in the TDGR. These compare operators are just common subexpressions and can easily be eliminated. In order to take advantage of this optimization, the target machine must provide a 'move from processor-register' instruction and a 'move to processor-register' instruction. Unfortunately, those special instructions are rather expensive. For example, a 'move data-register to processor-register' instruction on the MC68000 costs twice as much time as a 'compare data-register to data-register' instruction.

### 8.2.3. Implementation of the tables

The modules PM, IG and PO are table-driven. Each of these modules has its own table. The way in which the tables are used is described in chapter 6. In this section two different implementations of the tables are discussed: the sequential approach and the procedural approach. Which approach is the best, must still be investigated.

#### 8.2.3.1. The sequential approach

In the sequential approach, the table is a sequence of pairs <source,target>. The source-part describes the situation in which the target-part may be used. The advantage of this approach is that the table only contains data. Thus the data is separated from the algorithms, which makes the code generator easier to understand. The disadvantage of using a sequential table is that it is rather time consuming, although it can be speeded up by using for example hashing techniques. As an example, a

description of the PM-table is presented in figure 8.1 as a sequential table. The source pattern describes a graph, representing an addressing mode, a machine instruction or a primitive procedure. Not all the addressing modes are described in the PM-table. The PM has no sense of registers or memory locations, therefore it can not handle for example the VAX11's register addressing mode. Each source pattern can have attribute evaluation rules for the attributes of the DGR. Each source pattern has zero, one or more associated conditions. The target pattern describes a graph which is a part of the CDGR. The target pattern also gives the necessary attribute evaluation rules for the attributes of the CDGR.

```
<PM-table> ::= <rule> sequence.

<rule> ::= <source_pattern>,→,<target_pattern>.

<source_pattern> ::= <graph_description>
                     ,<attribute evaluation> sequence option
                     ,<condition> sequence option
                     .

<target_pattern> ::= <graph_description>
                     ,<attribute evaluation> sequence option
                     .
```

**Figure 8.1** A description of the PM-table

When the PM has to cover a node or group of nodes, it has to look up all the possibilities in the table. On a later implementation, the sequential search of the table can be speeded up by using a hash-function. The table is divided in different sections, each section associated with an unique entry (for example the MCIN as discussed in the "machine dependent method") which corresponds to a particular class of operators. When entering a section, a subgraph has to be selected. Each subgraph has a cost function, so the machine instruction with the lowest cost can be selected. The section is sequentially searched, therefore the subgraphs with the lowest costs are placed at the front.

### 8.2.3.2. The procedural approach

In the procedural approach, the table is not a sequence of items, but a collection of procedures. The advantage of this approach is that it is a faster implementation of the table than the sequential approach. The disadvantage is that the table is rather difficult to construct. As an example, figure 8.2 presents a part of the IG-table.

The example describes a part of the table which is used by the IG when code is being generated for an ADD-operator from the CDGR. After the code generator scans the ADD-operator, it calls the procedure add. Notice that although three possible machine instructions are implemented by this procedure, the ADD-operator is only matched once by the code generator, whereas in the sequential approach the operator may be matched three times. Obviously, the machine instructions in the table are sorted into increasing order of generality (or costs). This is done because the most specific machine instructions (INC and DEC) must be tried before the more general instructions (ADD).

### 8.2.3.3. A fast implementation of the tables

Thomas J. Pennello describes in [Penn86] an implementation of a fast LR Parser. In this implementation a speed-up of 6 to 10 can be achieved, at a cost of a factor 2 to 4 more in space.

> "This improvement is obtained by translating the parser's finite state control into assembly language. States become code memory addresses. The current input symbol resides in a register and a quick sequence of register-constant comparisons determines

```
PROCEDURE add (n : +numeric);
BEGIN
  a := operand1(n);
  b := operand2(n);
  c := operand3(n);
  IF type(a) = constant AND constant_value(a) = 1
  THEN BEGIN
         :
         code("INC",...)
  END
  ELIF type(b) = constant AND constant_value(b) = 1
  THEN BEGIN
         :
         code("INC",...)
  END
  ELIF type(a) = constant AND constant_value(a) = -1
  THEN BEGIN
         :
         code("DEC",...)
  END
  ELIF type(b) = constant AND constant_value(b) = -1
  THEN BEGIN
         :
         code("DEC",...)
  END
  ELSE BEGIN
         :
         code("ADD",...)
  END
END;
```

**Figure 8.2** A part of the IG-table using the procedural approach

the next state, which is merely jumped to. The parser's push-down stack is implemented directly on a hardware stack. The stack contains code memory addresses rather then the traditional state numbers."

LR parsers are nowadays not only used for context-free analysis of source programs but are also used in code generation (see section 1.3). It should be investigated how this technique of Pennello can be used in the implementation of the tables of the Dataflow Graph Code Generator.

Some other suggestions concerning the implementation of the tables are:

- Knowledge of programmer's tricks can be encoded in the tables. This knowledge-based approach of code generation is discussed in the Ph.D. thesis "Automatic Generation of Code Generators" (1977) of C. W. Fraser.
- The items of the tables should be present in the tables in an optimal order. The speed/size information given by the user can be used during the construction of the tables.

# APPENDIX A
# THE SYNTAX FORMAT

The syntax format is an extension of the familiar Backus-Normal-Form notation. This appendix shows a self-definition of the format. A non-terminal is represented by a character string enclosed in pointy brackets, "<" and ">". A terminal is directly represented as its written, or printed, representation. In this appendix however, a terminal is enclosed in quotes for the sake of clarity.

```
           <syntax> ::= <rule> sequence option.
             <rule> ::= <non-terminal>
                       ,<production-symbol>
                       ,<rule-body>
                       ,<end-symbol>
                       .
     <non-terminal> ::= "<",<identifier>,">".
<production-symbol> ::= "::=".
       <end-symbol> ::= ".".
        <rule-body> ::= <alternative> chain ";" option.
      <alternative> ::= <primitive> list.
        <primitive> ::= <primitive>,<special>
                       ;<terminal>
                       ;<non-terminal>
                       ;<compound>
                       .
         <terminal> ::= <identifier>.
         <compound> ::= <rule-body> pack.
          <special> ::= "sequence"
                       ;"chain",<primitive>
                       ;"list"
                       ;"pack"
                       ;"option"
                       .
```

Using the terminology from above, a syntax consists of a sequence of rules, where each rule is a non-terminal followed by "::=", followed by a serie of alternatives separated by a semicolon ";", followed by a full stop ".". An alternative is a serie of primitives separated by a comma ",". A primitive is a terminal, a non-terminal or a compound, which all may be followed by some special options.

A "sequence" indicates one or more repetitions of the primitive. A "chain" indicates a serie of primitives separated by the primitive on its right. The "chain" has a left associative performance, for example "a chain b chain c" means that "a" is chained with "b" and the result, e.g. the sentence "ababa", is chained with "c", e.g. producing the sentence "abababacababa". A "list" indicates a serie of primitives separated by a comma. A "pack" indicates that the primitive is surrounded by brackets, "(" and ")". An "option" indicates that the primitive may not occur.

# APPENDIX B
# DGR-LANGUAGE

In this appendix the syntax of the DGR-language is described.

```
<program_block> ::=
    program, <program_identifier>, {, <input_data>, →, <output_data>, } =
   ,[,first =, <module_identifier> sequence option
     ,body =, <module_block> sequence option
     ,flow =, nil
   ,]
    .

<module_block> ::=
    module, <module_identifier>, {, <input_data>, →, <output_data>, } =
   ,[,first =, (<procedure_identifier>; <construction_identifier>) sequence option
     ,body =, (<procedure_block>; <construction_block>) sequence option
     ,flow =, <module_identifier> sequence option
   ,]
    .

<procedure_block> ::=
    procedure, <procedure_identifier>, {, <input_data>, →, <output_data>, } =
   ,[,first =, (<procedure_identifier>; <construction_identifier>) sequence option
     ,body =, (<procedure_block>; <construction_block>) sequence option
     ,flow =, (<procedure_identifier>; <construction_identifier>) sequence option
   ,]
    .

<construction_block> ::=
    <while-do_block>
   ;<do-while_block>
   ;<case_block>
   ;<if-then-else_block>
   ;<call_block>
   ;<basic_block>
    .

<while-do_block> ::=
    while-do, <construction_identifier>, {, <input_data>, →, <output_data>, } =
   ,[,<control_block>
     ,first =, <construction_identifier> sequence option
     ,body =, <construction_block> sequence option
     ,flow =, <construction_identifier> sequence option
   ,]
    .
```

```
<do-while_block> ::=
    do-while, <construction_identifier>, {, <input_data>, →, <output_data>, } =
   ,[,<control_block>
     ,first =, <construction_identifier> sequence option
     ,body =, <construction_block> sequence option
     ,flow =, <construction_identifier> sequence option
   ,]
   .

<case_block> ::=
   case, <construction_identifier>, {, <input_data>, →, <output_data>, } =
   ,[,<control_block>
     ,(<entry_identifier>
      ,first =, <construction_identifier> sequence option
      ,body =, <construction_block> sequence option
      )sequence option
     ,flow =, <construction_identifier> sequence option
   ,]
   .

<if-then-else_block> ::=
    if-then-else, <construction_identifier>, {, <input_data>, →, <output_data>, } =
   ,[,<control_block>
     ,true
      ,first =, <construction_identifier> sequence option
      ,body =, <construction_block> sequence option
     ,false
      ,first =, <construction_identifier> sequence option
      ,body =, <construction_block> sequence option
     ,flow =, <construction_identifier> sequence option
   ,]
   .

<control_block> ::=
    control, {, <input_data>, →, <output_data>, } =
   ,[,first =, <construction_identifier> sequence
     ,body =, <construction_block> sequence
     ,flow =, nil
   ,]
   .

<call_block> ::=
    call, <construction_identifier>, {, <input_data>, →, <output_data>, } =
   ,[,first =, (<procedure_identifier>; <runtime-routine_identifier>)
     ,body =, nil
     ,flow =, <construction_identifier> sequence option
   ,]
   .
```

```
<basic_block> ::=
    basic, <construction_identifier>, {, <input_data>, →, <output_data>, } =
   ,[,first =, <graph_identifier>
     ,body =, <graph>
     ,flow =, <construction_identifier> sequence option
   ,]
   .

<graph> ::=
    graph, <graph_identifier>, {, <input_data>, →, <output_data>, } =
   ,[,<description>,]
   .

<input_data> ::= <data>.

<output_data> ::= <data>.

<data> ::= <object> chain ; pack list option.

<object> ::=
    <object_identifier>
   ,<object_type>
   ,<object_attributes>
   .

<object_type> ::=
    numeric
   ;boolean
   ;numeric_array
   ;boolean_array
   ;numeric_set
   ;boolean_set
   ;numeric_pointer
   ;boolean_pointer
   ;record
   ;file
   ;user_defined
   .

<object_attributes> ::=
    <class_attribute>
   ,<data-type_attribute>
   ,<attribute> sequence option
   .

<description> ::=
    <primitive>
   ;<functional_description>
   .
```

```
<primitive> ::=
    primitive_node
   ,type =
   ,<primitive_type>
   ,<attribute> sequence option
   ,<attribute_eval_rule> sequence option
   .

<attribute_eval_rule> ::= <attribute>, :=, <expression>.

<attribute> ::=
    <constant_value>
   ;<base>
   ;<size>
   ;<lowerbound>
   ;<upperbound>
   .

<functional_description> ::= <definition> sequence.

<definition> ::= <object_identifier>, =, <function>.

<function> ::= <graph_identifier>, <parameters>.

<parameters> ::= <object_identifier> list option pack.

<primitive_type> ::=
    <numeric_type>
   ;<boolean_type>
   ;<set_type>
   ;<special_type>
   .

<numeric_type> ::=
    +numeric
   ;-numeric
   ;*numeric
   ;/numeric
   ;=numeric
   ;≠numeric
   ;>numeric
   ;≥numeric
   ;<numeric
   ;≤numeric
   .

<boolean_type> ::=
    and
   ;or
   ;xor
   ;not
   .
```

```
<set_type> ::=
   union
  ;intersection
  ;difference
  .

<special_type> ::=
   dereference
  ;update
  ;constant
  ;halt
  ;conversion
  .

<class_attribute> ::=
   integer
  ;integer_subrange
  ;integer_constant
  ;real
  ;real_subrange
  ;real_constant
  ;boolean
  ;boolean_constant
  ;user_defined
  ;undefined
  .

<data-type_attribute> ::=
   small_integer
  ;normal_integer
  ;long_integer
  ;long_long_integer
  ;long_long_long_integer
  ;normal_real
  ;long_real
  ;long_long_real
  ;boolean
  ;user_defined
  ;undefined
  .

<program_identifier> ::= <identifier>.

<module_identifier> ::= <identifier>.

<procedure_identifier> ::= <identifier>.

<construction_identifier> ::= <identifier>.

<entry_identifier> ::= <identifier>.

<runtime-routine_identifier> ::= <identifier>.

<graph_identifier> ::= <identifier>.
```

<object_identifier> ::= <identifier>.

# APPENDIX C
# SYNTAX OF THE TMD

This appendix describes the syntax of the target machine description as discussed in chapter 7. This description language uses the same formalism as the DGR-language.

```
<machine_architecture> ::=
  ,machine_architecture =
  ,[,number_of_registers, =, <integer>
   ,size_of_registers, =, <integer>
   ,bits_per_word, =, <integer>
   ,nr_of_addressable_units_per_word, =, <integer>
   ,bits_per_unit, =, <integer>
   ,max_integer, =, <integer>
   ,min_integer, =, <integer>
   ,max_character, =, <integer>
   ,general_purpose_registers, =, <register> list option
   ,address_registers, =, <register> list option
   ,data_registers, =, <register> list option
   ,status_register, =, <register> list
   ,program_counter, =, <register> list
   ,stack_pointer, =, <register> list
   ,frame_pointer, =, <register> list
   ,argument_pointer, =, <register> list
   ,increment_size, =, <integer> list
  ,]
  .

<add_graph> ::=
   add_graph, <add_graph_identifier>, {, <input_data>, →, <output_data>, } =
  ,[,<descriptions>,]
  ,[,<match_conditions>,]
  ,[,<register_conditions>,]
  .

<addressing_mode> ::=
   <mode_type>, <mode_identifier>, {, <input_data>, →, <output_data>, } =
  ,[,<descriptions>,]
  ,[,<match_conditions>,]
  ,[,<register_conditions>,]
  ,[,<assembly_code>,]
  ,[,<cost_function>,]
  .
```

```
<instruction> ::=
   instruction, <instruction_identifier>, {, <input_data>, →, <output_data>, } =
  ,[,<descriptions>,]
  ,[,<match_conditions>,]
  ,[,<add_mode_conditions>,]
  ,[,<register_conditions>,]
  ,[,<assembly_code>,]
  ,[,<condition_codes>,]
  ,[,<cost_function>,]
  .

<routine> ::=
   routine, <routine_identifier>, {, <input_data>, →, <output_data>, } =
  ,[,<descriptions>,]
  ,[,<match_conditions>,]
  ,[,<register_conditions>,]
  ,[,<assembly_code>,]
  ,[,<condition_codes>,]
  ,[,<cost_function>,]
  .

<mode_type> ::= L-mode; R-mode.

<descriptions> ::= <description> chain or.

<match_conditions> ::= <object_condition> sequence option.

<object_condition> ::=
   <object_identifier>, =, <object_identifier>
  ;<object_identifier>, ≠, <object_identifier>
  ;<object_identifier>, is, constant
  ;<object_identifier>, in, size
  .

<add_mode_conditions> ::= <add_mode_condition> sequence option.

<add_mode_condition> ::= add_mode, <object_identifier> pack, =, <set_of_addressing_modes>.

<set_of_addressing_modes> ::= <mode_identifier> sequence option.

<register_conditions> ::=
   <need_registers> option
  ,<use_registers> option
  ,<register_relation> sequence option
  .

<need_registers> ::= need, <io_registers>.

<use_registers> ::= use, <io_registers>.

<io_registers> ::=
   <input_registers> option
  ,<internal_registers> option
  ,<output_registers> option
  .
```

<input_registers> ::= input, <register> sequence.

<internal_registers> ::= internal, <register> sequence.

<output_registers> ::= output, <register> sequence.

<register_relation> ::=
<object_identifier>, =, <object_identifier>
;<object_identifier>, ≠, <object_identifier>
;<object_identifier>, is, register
;<object_identifier>, in, <register_class>
.

<register_class> ::=
data_register
;address_register
;even_register
;odd_register
.

<register> ::= <register_identifier>.

<assembly_code> ::= (<string>; <code>) sequence option..

<code> ::= code, <object_identifier> pack.

<condition_codes> ::=
N =, <negative_code>
,Z =, <zero_code>
,O =, <overflow_code>
,C =, <carry_code>
.

<negative_code> ::= <string>.

<zero_code> ::= <string>.

<overflow_code> ::= <string>.

<carry_code> ::= <string>.

<cost_function> ::= <space>, <time>, <references>.

<space> ::= <integer>; <unknown>.

<time> ::= <integer>; <unknown>.

<references> ::= <integer>; <unknown>.

<unknown> ::= ?.

<add_graph_identifier> ::= <identifier>.

<mode_identifier> ::= <identifier>.

<instruction_identifier> ::= <identifier>.

<routine_identifier> ::= <identifier>.

<register_identifier> ::= <identifier>.

<object_identifier> ::= <identifier>.

The non-terminals <input_data>, <output_data> and <description> are not described in this appendix, they are already described in appendix B. Only the production rule

<function> ::= <graph_identifier>, <parameters>.

in appendix B must be modified into the rule

<function> ::= (<graph_identifier>; <add_graph_identifier>), <parameters>.

The non-terminals <negative_code>, <zero_code>, <overflow_code> and <carry_code> are not fully described in this appendix, we refer to them as strings as presented in [Digi81].

# APPENDIX D
# THE MACHINE ARCHITECTURE

In this appendix the machine architecture of the VAX11 is described. The machine architecture is a part of the target machine description, as described in chapter 7.

```
machine_architecture =
        [number_of_registers = 16
         size_of_registers = 32
         bits_per_word = 32
         nr_of_addressable_units_per_word = 4
         bits_per_unit = 8
         max_integer = 2147483647
         min_integer = -2147483647
         max_character = 127
         general_purpose_registers = r0, r1, ... r10, r11
         address_registers = nil
         data_registers = nil
         status_register = PSL
         program_counter = PC, r15
         stack_pointer = SP, r14
         frame_pointer = FP, r13
         argument_pointer = AP, r12
         increment_size = 1, 2, 4, 8, 16]
```

# APPENDIX E
# ADDRESSING MODES OF THE VAX11

In this appendix the addressing modes of the VAX11 are described. This hand-written description is a part of the target machine description as described in chapter 7. For the convenience of the user, the description may be factorized, as in this appendix. The Table Constructor, however, will probably introduce an other factorization in the table.

The description is started with some elementary graphs, which are used to describe the addressing modes. There are two classes of addressing modes: the L-addressing modes and the R-addressing modes (for an explanation see chapter 7).

For the sake of clarity, the attributes of the objects are not described. It is assumed that the function 'code' produces a string representing the assembly code associated with its parameter.

```
add_graph const { → (v1 numeric)} =
        [primitive_node]
        [v1 is constant]
        [ ]

add_graph content {(Rn numeric) → (v1 numeric)} =
        [primitive_empty]
        [ ]
        [Rn is register]

add_graph post-incr {(v1 numeric) → (Rn numeric)} =
        [t1 = const( )
         Rn = add(v1,t1)]
        [t1 in size]
        [Rn is register]

add_graph pre-decr {(v1 numeric) → (v2 numeric),(Rn numeric)} =
        [t1 = const( )
         v2 = subtr(v1,t1)
         Rn = v2]
        [t1 in size]
        [Rn is register]

add_graph displ {(v1 numeric) → (v2 numeric)} =
        [t1 = const( )
         v2 = add(v1,t1)]
        [ ]
        [ ]
```

```
add_graph defer {(v1 numeric) → (v2 numeric)} =
        [primitive_node]
        [ ]
        [ ]

add_graph non-defer {(v1 numeric) → (v2 numeric)} =
        [primitive_empty]
        [ ]
        [ ]

add_graph index {(Rx numeric),(v1 numeric) → (v2 numeric)} =
        [t1 = const( )
         t2 = mult(Rx,t1)
         v2 = add(v1,t2)]
        [t1 in size]
        [Rx is register, Rx ≠ PC]

add_graph non-index {(v1 numeric) → (v2 numeric)} =
        [primitive_empty]
        [ ]
        [ ]

add_graph indirect {(v1 numeric) → (v2 numeric)} =
        [v2 = defer(v1)]
        [ ]
        [ ]

add_graph non-defer-non-index {(v1 numeric) → (v2 numeric)} =
        [t1 = non-defer(v1)
         v2 = non-index(t1)]
        [ ]
        [ ]

add_graph non-defer-index {(v1 numeric),(Rx numeric) → (v2 numeric)} =
        [t1 = non-defer(v1)
         v2 = index(Rx,t1)]
        [ ]
        [ ]

add_graph defer-non-index {(v1 numeric) → (v2 numeric)} =
        [t1 = defer(v1)
         v2 = non-index(t1)]
        [ ]
        [ ]

add_graph defer-index {(v1 numeric),(Rx numeric) → (v2 numeric)} =
        [t1 = defer(v1)
         v2 = index(Rx,t1)]
        [ ]
        [ ]
```

```
add_graph non-defer-non-index-indirect {(v1 numeric) → (v2 numeric)} =
        [t1 = non-defer-non-index(v1)
         v2 = indirect(t1)]
        [ ]
        [ ]

add_graph non-defer-index-indirect {(v1 numeric),(Rx numeric) → (v2 numeric)} =
        [t1 = non-defer-index(v1,Rx)
         v2 = indirect(t1)]
        [ ]
        [ ]

add_graph defer-non-index-indirect {(v1 numeric) → (v2 numeric)} =
        [t1 = defer-non-index(v1)
         v2 = indirect(t1)]
        [ ]
        [ ]

add_graph defer-index-indirect {(v1 numeric),(Rx numeric) → (v2 numeric)} =
        [t1 = defer-index(v1,Rx)
         v2 = indirect(t1)]
        [ ]
        [ ]

L-mode register-deferred {(Rn numeric) → (v1 numeric)} =
        [t1 = content(Rn)
         v1 = non-defer-non-index(t1)]
        [ ]
        [Rn ≠ PC]
        ["(" code(Rn) ")"]
        [? ? ?]

L-mode register-deferred-indexed {(Rn numeric),(Rx numeric) → (v1 numeric)} =
        [t1 = content(Rn)
         v1 = non-defer-index(Rx,t1)]
        [ ]
        [Rn ≠ PC]
        ["(" code(Rn) ")[" code(Rx) "]"]
        [? ? ?]

L-mode autoincrement {(Rm numeric) → (v1 numeric),(Rn numeric)} =
        [t1 = content(Rm)
         v1 = non-defer-non-index(t1)
         Rn = post-incr(t1)]
        [ ]
        [Rm = Rn]
        ["(" code(Rn) ")+"]
        [? ? ?]
```

```
L-mode autoincrement-deferred {(Rm numeric) → (v1 numeric),(Rn numeric)} =
        [t1 = content(Rm)
         v1 = defer-non-index(t1)
         Rn = post-incr(t1)]
        [ ]
        [Rm = Rn]
        ["@(" code(Rn) ")+"]
        [? ? ?]

L-mode autoincrement-deferred-indexed {(Rm numeric),(Rx numeric) → (v1 numeric),(Rn numeric)} =
        [t1 = content(Rm)
         v1 = defer-index(Rx,t1)
         Rn = post-incr(t1)]
        [ ]
        [Rm = Rn, Rn ≠ Rx]
        ["@(" code(Rn) ")+[" code(Rx) "]"]
        [? ? ?]

L-mode autoincrement-indexed {(Rm numeric),(Rx numeric) → (v1 numeric),(Rn numeric) } =
        [t1 = content(Rm)
         v1 = non-defer-index(Rx,t1)
         Rn = post-incr(t1)]
        [ ]
        [Rm = Rn, Rn ≠ Rx]
        ["(" code(Rn) ")+[" code(Rx) "]"]
        [? ? ?]

L-mode autodecrement {(Rm numeric) → (v1 numeric),(Rn numeric)} =
        [t1 = content(Rm)
         t2, Rn = pre-decr(t1)
         v1 = non-defer-non-index(t2)]
        [ ]
        [Rm = Rn, Rn ≠ PC]
        ["-(" code(Rn) ")"]
        [? ? ?]

L-mode autodecrement-indexed {(Rm numeric),(Rx numeric) → (v1 numeric),(Rn numeric) } =
        [t1 = content(Rm)
         t2, Rn = pre-decr(t1)
         v1 = non-defer-index(Rx,t2)]
        [ ]
        [Rm = Rn, Rn ≠ PC, Rn ≠ Rx]
        ["-(" code(Rn) ")[" code(Rx) "]"]
        [? ? ?]

L-mode displacement {(Rn numeric) → (v1 numeric)} =
        [t1 = content(Rn)
         t2 = displ(t1)
         v1 = non-defer-non-index(t2)]
        [ ]
        [ ]
        [code(displ) "(" code(Rn) ")"]
        [? ? ?]
```

```
L-mode displacement-indexed {(Rn numeric),(Rx numeric) → (v1 numeric)} =
        [t1 = content(Rn)
         t2 = displ(t1)
         v1 = non-defer-index(Rx,t2)]
        [ ]
        [ ]
        [code(displ) "(" code(Rn) ")[" code(Rx) "]"]
        [? ? ?]

L-mode displacement-deferred {(Rn numeric) → (v1 numeric)} =
        [t1 = content(Rn)
         t2 = displ(t1)
         v1 = defer-non-index(t2)]
        [ ]
        [ ]
        ["@" code(displ) "(" code(Rn) ")"]
        [? ? ?]

L-mode displacement-deferred-indexed {(Rn numeric),(Rx numeric) → (v1 numeric)} =
        [t1 = content(Rn)
         t2 = displ(t1)
         v1 = defer-index(Rx,t2)]
        [ ]
        [ ]
        ["@" code(displ) "(" code(Rn) ")[" code(Rx) "]"]
        [? ? ?]

L-mode absolute & relative {(a numeric) → (v1 numeric)} =
        [v1 = non-defer-non-index(a)]
        [ ]
        [ ]
        ["@#" code(a)]
        [? ? ?]

L-mode relative-deferred {(a numeric) → (v1 numeric)} =
        [v1 = defer-non-index(a)]
        [ ]
        [ ]
        ["@" code (displ) "#" code(a)]
        [? ? ?]

L-mode absolute-indexed & relative-indexed {(a numeric),(Rx numeric) → (v1 numeric) } =
        [v1 = non-defer-index(Rx,a)]
        [ ]
        [ ]
        ["#" code(a) "[" code(Rx) "]"]
        [? ? ?]

L-mode relative-deferred-indexed {(a numeric),(Rx numeric) → (v1 numeric)} =
        [v1 = defer-index(Rx,a)]
        [ ]
        [ ]
        ["@" code(displ) "#" code(a) "[" code(Rx) "]"]
        [? ? ?]
```

```
R-mode literal & immediate { → (v1 numeric)} =
        [v1 = const( )]
        [ ]
        [ ]
        ["#" code(v1)]
        [? ? ?]

R-mode register {(Rn numeric) → (v1 numeric)} =
        [t1 = content(Rn)
         v1 = direct(t1)]
        [ ]
        [Rn ≠ PC, Rn ≠ SP]
        [code(Rn)]
        [? ? ?]

R-mode register-deferred {(Rn numeric) → (v1 numeric)} =
        [t1 = content(Rn)
         v1 = non-defer-non-index-indirect(t1)]
        [ ]
        [Rn ≠ PC]
        ["(" code(Rn) ")"]
        [? ? ?]

R-mode register-deferred-indexed {(Rn numeric),(Rx numeric) → (v1 numeric)} =
        [t1 = content(Rn)
         v1 = non-defer-index-indirect(Rx,t1)]
        [ ]
        [Rx is register, Rn ≠ PC]
        ["(" code(Rn) ")[" code(Rx) "]"]
        [? ? ?]

R-mode autoincrement {(Rm numeric) → (v1 numeric),(Rn numeric)} =
        [t1 = content(Rm)
         v1 = non-defer-non-index-indirect(t1)
         Rn = post-incr(t1)]
        [ ]
        [Rm = Rn]
        ["(" code(Rn) ")+"]
        [? ? ?]

R-mode autoincrement-deferred {(Rm numeric) → (v1 numeric),(Rn numeric)} =
        [t1 = content(Rm)
         v1 = defer-non-index-indirect(t1)
         Rn = post-incr(t1)]
        [ ]
        [Rm = Rn]
        ["@(" code(Rn) ")+"]
        [? ? ?]
```

```
R-mode autoincrement-deferred-indexed {(Rm numeric),(Rx numeric) → (v1 numeric),(Rn numeric)} =
        [t1 = content(Rm)
         v1 = defer-index-indirect(Rx,t1)
         Rn = post-incr(t1)]
        [ ]
        [Rm = Rn, Rn ≠ Rx]
        ["@(" code(Rn) ")+[" code(Rx) "]"]
        [? ? ?]

R-mode autoincrement-indexed {(Rm numeric),(Rx numeric) → (v1 numeric),(Rn numeric) } =
        [t1 = content(Rm)
         v1 = non-defer-index-indirect(Rx,t1)
         Rn = post-incr(t1)]
        [ ]
        [Rx is register, Rm = Rn, Rn ≠ Rx]
        ["(" code(Rn) ")+[" code(Rx) "]"]
        [? ? ?]

R-mode autodecrement {(Rm numeric) → (v1 numeric),(Rn numeric)} =
        [t1 = content(Rm)
         t2, Rn = pre-decr(t1)
         v1 = non-defer-non-index-indirect(t2)]
        [ ]
        [Rm = Rn, Rn ≠ PC]
        ["-(" code(Rn) ")"]
        [? ? ?]

R-mode autodecrement-indexed {(Rm numeric),(Rx numeric) → (v1 numeric),(Rn numeric) } =
        [t1 = content(Rm)
         t2, Rn = pre-decr(t1)
         v1 = non-defer-index-indirect(Rx,t2)]
        [ ]
        [Rx is register, Rm = Rn, Rn ≠ PC, Rn ≠ Rx]
        ["-(" code(Rn) ")[" code(Rx) "]"]
        [? ? ?]

R-mode displacement {(Rn numeric) → (v1 numeric)} =
        [t1 = content(Rn)
         t2 = displ(t1)
         v1 = non-defer-non-index-indirect(t2)]
        [ ]
        [ ]
        [code(displ) "(" code(Rn) ")"]
        [? ? ?]

R-mode displacement-indexed {(Rn numeric),(Rx numeric) → (v1 numeric)} =
        [t1 = content(Rn)
         t2 = displ(t1)
         v1 = non-defer-index-indirect(Rx,t2)]
        [ ]
        [Rx is register]
        [code(displ) "(" code(Rn) ")[" code(Rx) "]"]
        [? ? ?]
```

```
R-mode displacement-deferred {(Rn numeric) → (v1 numeric)} =
        [t1 = content(Rn)
         t2 = displ(t1)
         v1 = defer-non-index-indirect(t2)]
        [ ]
        [ ]
        ["@" code(displ) "(" code(Rn) ")"]
        [? ? ?]

R-mode displacement-deferred-indexed {(Rn numeric),(Rx numeric) → (v1 numeric)} =
        [t1 = content(Rn)
         t2 = displ(t1)
         v1 = defer-index-indirect(Rx,t2)]
        [ ]
        [Rx is register]
        ["@" code(displ) "(" code(Rn) ")[" code(Rx) "]"]
        [? ? ?]

R-mode absolute & relative {(a numeric) → (v1 numeric)} =
        [v1 = non-defer-non-index-indirect(a)]
        [ ]
        [ ]
        ["@#" code(a)]
        [? ? ?]

R-mode relative-deferred {(a numeric) → (v1 numeric)} =
        [
        [v1 = defer-non-index-indirect(a)]
        [ ]
        [ ]
        ["@" code(displ) "#" code(a)]
        [? ? ?]

R-mode absolute-indexed & relative-indexed {(a numeric),(Rx numeric) → (v1 numeric) } =
        [v1 = non-defer-index-indirect(Rx,a)]
        [ ]
        [Rx is register]
        ["#" code(a) "[" code(Rx) "]"]
        [? ? ?]

R-mode relative-deferred-indexed {(a numeric),(Rx numeric) → (v1 numeric)} =
        [v1 = defer-index-indirect(Rx,a)]
        [ ]
        [Rx is register]
        ["@" code(displ) "#" code(a) "[" code(Rx) "]"]
        [? ? ?]
```

# APPENDIX F
# MACHINE INSTRUCTIONS OF THE VAX11

In this appendix some machine instructions of the VAX11 are described as an example. This hand-written description is a part of the target machine description as described in chapter 7.

The first two instructions, described in this appendix, implement the arithmetic addition of two small_integers. In the instruction ADDB3, the first operand is added to the second operand and the sum operand is replaced by the result. In the instruction ADDB2, the first operand is added to the sum operand and the sum operand is replaced by the result.

```
instruction ADDB3
        {(v1 numeric integer small_integer)
         (v2 numeric integer small_integer) →
         (v3 numeric integer small_integer)} =
        [v3 = addition(v1,v2)]
        [ ]
        [add_mode(v1) = all
         add_mode(v2) = all
         add_mode(v3) = all_but_literal_&_immediate]
        [ ]
        ["ADDB3" code(v1) "," code(v2) "," code(v3)]
        [N = v3 LSS 0
         Z = v3 EQL 0
         V = overflow
         C = carry]
        [1 ? 0]

instruction ADDB2
        {(v1 numeric integer small_integer)
         (v2 numeric integer small_integer) →
         (v3 numeric integer small_integer)} =
        [v3 = addition(v1,v2)]
        [v2 = v3]
        [add_mode(v1) = all
         add_mode(v2) = all_but_literal_&_immediate
         add_mode(v3) = all_but_literal_&_immediate]
        [ ]
        ["ADDB2" code(v1) "," code(v2)]
        [N = v3 LSS 0
         Z = v3 EQL 0
         V = overflow
         C = carry]
        [1 ? 0]
```

The next instruction described is the INCB instruction, which implements the increment of a small_integer. The value one is added to the operand and the operand is replaced by the result.

```
instruction INCB
            {(v1 numeric integer small_integer) →
             (v2 numeric integer small_integer)} =
            [t1 = constant_1( )
             v2 = addition(v1,t1)]
            [v1 = v2]
            [add_mode(v1) = all_but_literal_&_immediate
             add_mode(v2) = all_but_literal_&_immediate]
            [ ]
            ["INCB" code(v1)]
            [N = v2 LSS 0
             Z = v2 EQL 0
             V = overflow
             C = carry]
            [1 ? 0]
```

# APPENDIX G
# EXAMPLE

This appendix describes the compilation of a Pascal program fragment as an example. The Pascal program fragment is translated by the front-end into a DGR. The Dataflow Graph Code Generator translates this DGR into assembly code for the VAX11.

The Pascal fragment used in this appendix, was already introduced in chapter 4 as an example of the implementation of an array. Figure G.1 shows this Pascal fragment. The array "a" is declared as an array of integers. All the scalar variables, present in the fragment, are declared as integers. Boundary checking is omitted in this example, it is assumed that the subscripts "i", "j" and "k" are valid subscripts for the array.

```
w    :=  a[i];
a[j] :=  x;
y    :=  a[k];
z    :=  a[j];
```

**Figure G.1** The Pascal program fragment

The front-end translates the fragment into a dataflow graph. Figure G.2 pictures this dataflow graph. Notice that although aliasing is possible, the dereference and update operators ensure that the object code will be generated in a correct order. In this example, the fact is used that an array is a 4-tuple

<base, lowerbound, upperbound, size>

The constant operators "s" in the dataflow graph produce the value size(a) and the constant operators "b" produce the value base(a)-lowerbound(a)*size(a). Figure G.3 shows the dataflow graph described in the DGR-language. The primitive nodes of this graph are shown in figure G.4 to figure G.9 .

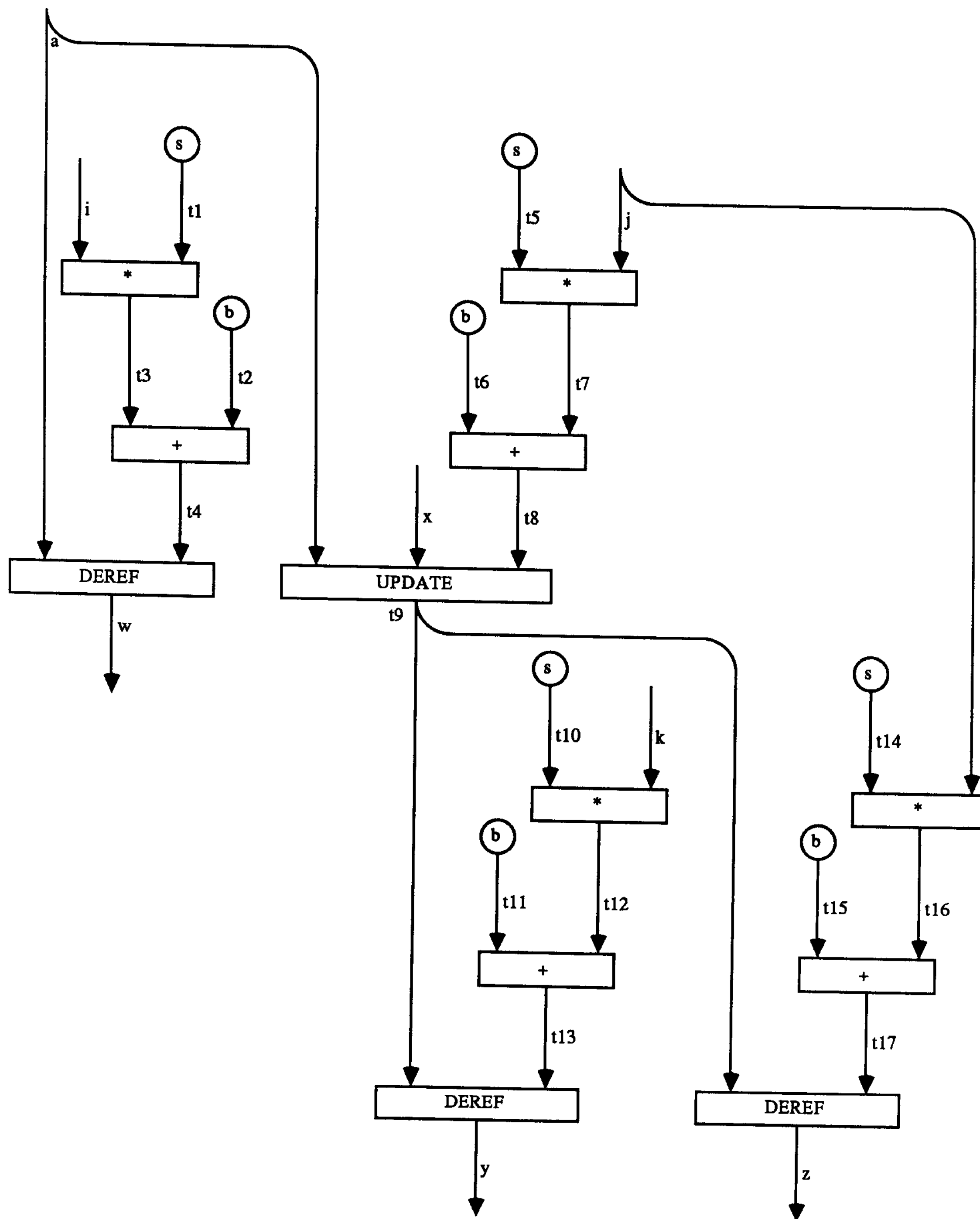



**Figure G.2** The dataflow graph produced by the front-end

```
graph example
   {(a numeric_array integer normal_integer),
    (i numeric integer normal_integer),
    (j numeric integer normal_integer),
    (x numeric integer normal_integer),
    (k numeric integer normal_integer) →
    (w numeric integer normal_integer),
    (y numeric integer normal_integer),
    (z numeric integer normal_integer),} =
   [t1 = constant_s( )
    t2 = constant_b( )
    t3 = multiply(t1,i)
    t4 = addition(t2,t3)
    w = deref(a,t4)
    t5 = constant_s( )
    t6 = constant_b( )
    t7 = multiply(t5,j)
    t8 = addition(t6,t7)
    t9 = update(a,t8,x)
    t10 = constant_s( )
    t11 = constant_b( )
    t12 = multiply(t10,k)
    t13 = addition(t11,t12)
    y = deref(t9,t13)
    t14 = constant_s( )
    t15 = constant_b( )
    t16 = multiply(t14,j)
    t17 = addition(t15,t16)
    z = deref(t9,t17)]
```

**Figure G.3** The dataflow graph described in the DGR-language

```
graph constant_s
   {( → (v1 numeric integer_constant normal_integer size(a))} =
   [primitive_node
    type = constant]
```

**Figure G.4** The graph constant_s described in the DGR-language

```
graph constant_b
   {( → (v1 numeric integer_constant normal_integer base(a)-lowerbound(a)*size(a))} =
   [primitive_node
    type = constant]
```

**Figure G.5** The graph constant_b described in the DGR-language

```
graph multiply
   {(v1 numeric integer normal_integer)
    (v2 numeric integer normal_integer) →
    (v3 numeric integer normal_integer)} =
   [primitive_node
    type = *numeric]
```

**Figure G.6** The graph multiply described in the DGR-language

```
graph addition
   {(v1 numeric integer normal_integer)
    (v2 numeric integer normal_integer) →
    (v3 numeric integer normal_integer)} =
   [primitive_node
    type = +numeric]
```

**Figure G.7** The graph addition described in the DGR-language

```
graph deref
   {(v1 numeric_array integer normal_integer)
    (v2 numeric integer normal_integer) →
    (v3 numeric integer normal_integer)} =
   [primitive_node
    type = dereference]
```

**Figure G.8** The graph deref described in the DGR-language

```
graph update
   {(v1 numeric_array integer normal_integer)
    (v2 numeric integer normal_integer)
    (v3 numeric integer normal_integer) →
    (v4 numeric_array integer normal_integer)} =
   [primitive_node
    type = update]
```

**Figure G.9** The graph update described in the DGR-language

The dataflow graph produced by the front-end, is transformed by the Global Optimizer into an optimized dataflow graph. In this example, the Global Optimizer can only perform the common subexpression elimination technique. The common subexpressions in this example are the constant "s" operators, the constant "b" operators and the subgraphs producing the address of a[j]. Figure G.10 pictures the optimized dataflow graph and figure G.11 shows the DGR description of the optimized dataflow graph. The primitive nodes of this graph are the same as those shown in figure G.4 to figure G.9 .

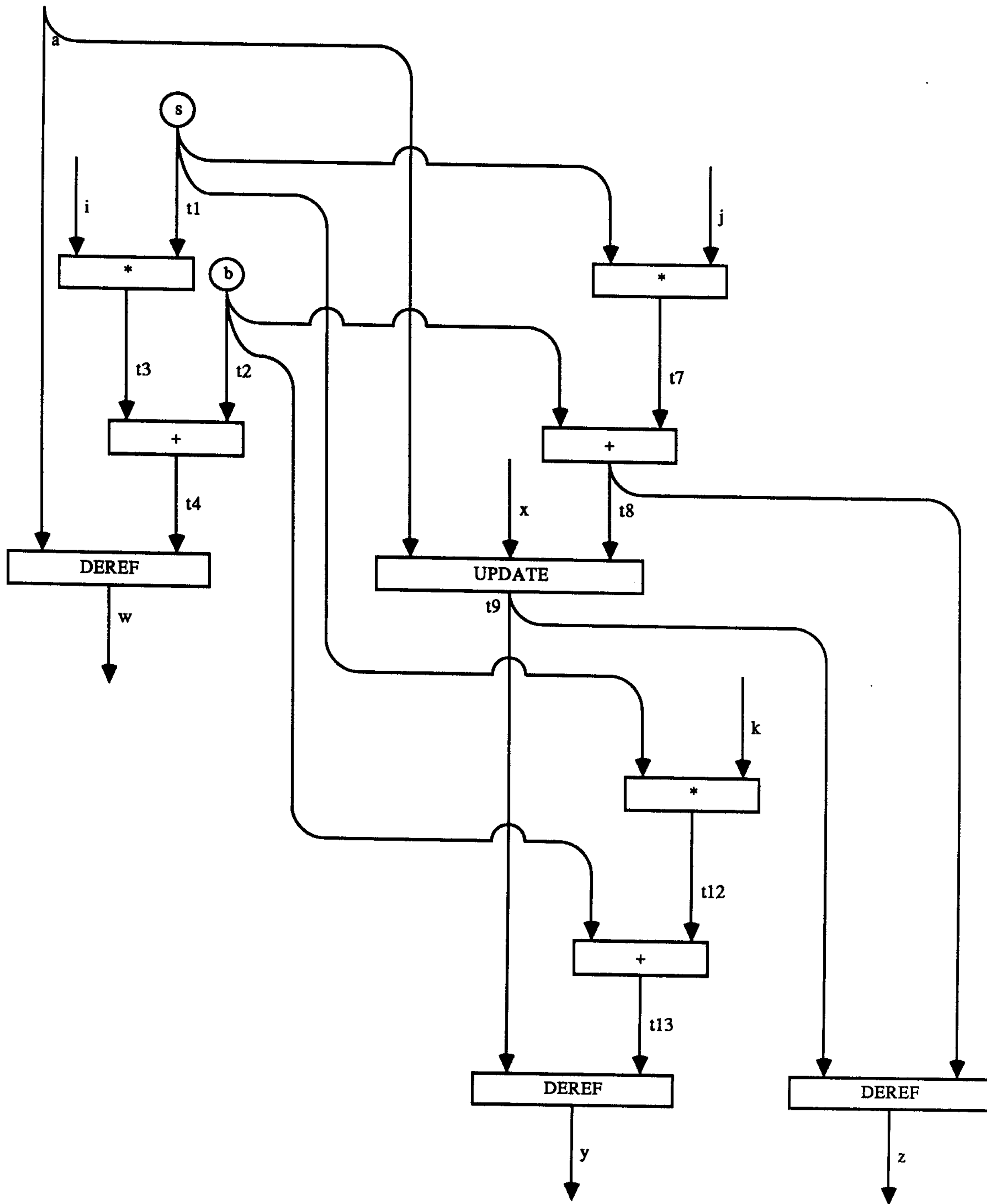



**Figure G.10** The dataflow graph produced by the Global Optimizer

```
graph example
  {(a numeric_array integer normal_integer),
   (i numeric integer normal_integer),
   (j numeric integer normal_integer),
   (x numeric integer normal_integer),
   (k numeric integer normal_integer) →
   (w numeric integer normal_integer),
   (y numeric integer normal_integer),
   (z numeric integer normal_integer),} =
  [t1 = constant_s( )
   t2 = constant_b( )
   t3 = multiply(t1,i)
   t4 = addition(t2,t3)
   w = deref(a,t4)
   t7 = multiply(t1,j)
   t8 = addition(t2,t7)
   t9 = update(a,t8,x)
   t12 = multiply(t1,k)
   t13 = addition(t2,t12)
   y = deref(t9,t13)
   z = deref(t9,t8)]
```

**Figure G.11** The optimized dataflow graph described in the DGR-language

The PM covers the dataflow graph with subgraphs representing addressing modes and machine instructions. The PM selects the covering with the fewest costs. Suppose, the PM selects the covering as pictured in figure G.12 . Notice that the arcs, implementing the datastructures in the dataflow graph, are also present in the covered graph to ensure that the object code will be generated in a correct order. Figure G.13 pictures the graph produced by the PM described in the CDGR-language. The primitive nodes of this graph are shown in figure G.14 to figure G.18 .

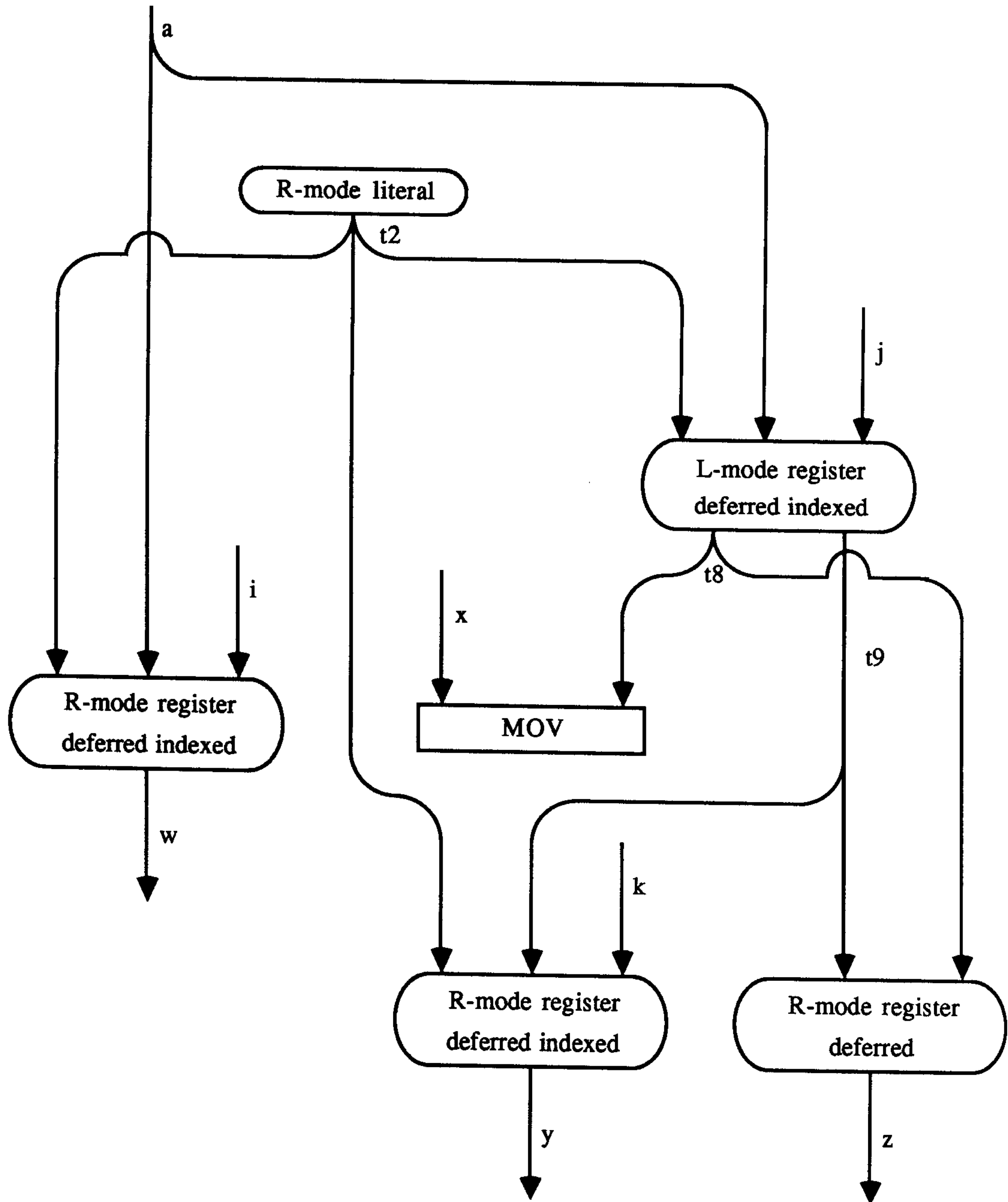



**Figure G.12** The graph produced by the PM

```
graph example
    {(a numeric_array integer normal_integer),
     (i numeric integer normal_integer),
     (j numeric integer normal_integer),
     (x numeric integer normal_integer),
     (k numeric integer normal_integer) →
     (w numeric integer normal_integer),
     (y numeric integer normal_integer),
     (z numeric integer normal_integer),} =
    [t2 = R-mode-1( )
     w = R-mode-2(t2,a,i)
     t8,t9 = L-mode-3(t2,a,j)
      = move(x,t8)
     y = R-mode-2(t2,t9,k)
     z = R-mode-3(t9,t8)]
```

**Figure G.13** The graph produced by the PM described in the CDGR-language

```
graph R-mode-1
    {( → (v1 numeric integer_constant normal_integer
            base(a)-lowerbound(a)*size(a))} =
    [addressing mode
     type = R-mode-literal]
```

**Figure G.14** The graph R-mode-1 described in the CDGR-language

```
graph R-mode-2
    {(v1 numeric integer normal_integer),
     (a1 numeric_array integer normal_integer),
     (v2 numeric integer normal_integer) →
     (v3 numeric integer normal_integer)} =
    [addressing mode
     type = R-mode-register-deferred-indexed]
```

**Figure G.15** The graph R-mode-2 described in the CDGR-language

```
graph R-mode-3
    {(a1 numeric_array integer normal_integer),
     (v1 numeric integer normal_integer) →
     (v2 numeric integer normal_integer)} =
    [addressing mode
     type = R-mode-register-deferred]
```

**Figure G.16** The graph R-mode-3 described in the CDGR-language

```
graph L-mode-3
    {(v1 numeric integer normal_integer),
     (a1 numeric_array integer normal_integer),
     (v2 numeric integer normal_integer) →
     (v3 numeric integer normal_integer),
     (a2 numeric_array integer normal_integer)} =
    [addressing mode
     type = L-mode-register-deferred-indexed]
```

**Figure G.17** The graph L-mode-3 described in the CDGR-language

```
graph move
    {(v1 numeric integer normal_integer),
     (v2 numeric integer normal_integer) → } =
    [instruction
     type = MOV]
```

**Figure G.18** The graph move described in the CDGR-language

The IG generates an assembly code sequence out of the CDGR. Figure G.19 shows the assembly code produced by the IG.

```
MOV  #b,ro
MOV  i,r1
MOV  (r0)[r1],w
MOV  j,r1
MOVA (r0)[r1],r2
MOV  x,(r2)
MOV  k,r1
MOV  (r0)[r1],y
MOV  (r2),z
```

**Figure G.19** The assembly code produced by the IG

# REFERENCES

**[Alle76]** F. E. Allen, J. Cocke. A Program Data Flow Analysis Procedure. Communications of the ACM, 1976, vol.19, no.3, pp.137-166.

**[Arvi82]** Arvind, K. P. Gostelow. The U-Interpreter. Computer, 1982, vol.15, no.2, pp.42-49.

**[Bal85]** H. E. Bal. The design and implementation of the EM Global Optimizer. 1985. Rapport nr IR-99. Subfaculteit Wiskunde en Informatica, Vrije Universiteit Amsterdam.

**[Ball79]** J. E. Ball. Predicting the Effects of Optimization on a Procedure Body. ACM SIGPLAN notices, 1979, vol.14, no.8, pp.214-220.

**[Bird82]** P. L. Bird. An Implementation of a Code Generator Specification Language for Table Driven Code Generators. ACM SIGPLAN notices, 1982, vol.17, no.6, pp.66-55.

**[Bohm84]** A. P. W. Bohm. Dataflow Computation. 1984. Mathematisch Centrum, Amsterdam.

**[Born79]** R. Bornat. Understanding and Writing Compilers. 1979. The MacMillan Press Ltd, London.

**[Brow74]** P. J. Brown. Macro Processors. 1974. John Wiley & Sons, London.

**[Catt82]** R. G. G. Cattell. Formalization and Automatic Derivation of Code Generators. Computer Science: Systems Programming, no.3. 1982. UMI Research Press, Ann Arbor, Michigan.

**[Craw82]** J. Crawford. Engineering a Production Code Generator. ACM SIGPLAN notices, 1982, vol.17, no.6, pp.205-215.

**[Davi80]** J. W. Davidson, C. W. Fraser. The Design and Application of a Retargetable Peephole Optimizer. ACM TOPLAS, 1980, vol.2, no.2, pp.191-202.

**[Davi84]** J. W. Davidson, C. W. Fraser. Automatic Generation of Peephole Optimizations. ACM SIGPLAN notices, 1984, vol.19, no.6, pp.111-116.

**[Denn83]** J. B. Dennis, W. Y-P. Lim, W. B. Ackerman. The MIT Data Flow Engineering Model. Information processing 83, IFIP Congress series, 1983, vol.9, pp.553-560.

**[Digi81]** Digital Equipment Corporation. VAX-11 Architecture Handbook. 1981. Maynard, Massachutts.

**[Gana82]** M. Ganapathi, C. N. Fischer, J. L. Hennersy. Retargetable Compiler Code Generation. ACM Computing Surveys, 1982, vol.14, no.4, pp.573-592.

**[Gana85]** M. Ganapathi, C. N. Fischer. Affix Grammar Driven Code Generation. ACM TOPLAS, 1985, vol.7, no.4, pp.560-599.

[Glan78] R. S. Glanville, S. L. Graham. A New Method for Compiler Code Generation. POPL, 1978, no.5, pp.231-260.

[Gost80] K. P. Gostelow, R. E. Thomas. Performance of a Simulated Dataflow Computer. IEEE Transactions on Computers, 1980, vol.c-29, no.10, pp.905-919.

[Grah80] S. L. Graham. Table-Driven Code Generation. Computer, 1980, vol.13, no.8, pp.25-36.

[Grah82] S. L. Graham, R. R. Henry, R. A. Schulman. An Experiment in Table Driven Code Generation. ACM SIGPLAN notices, 1982, vol.17, no.6, pp.32-63.

[Grah84] S. L. Graham, P. Aingrain, R. R. Henry, M. K. McKusick, E. Pelegri-LLopart. Experience with a Graham-Glanville Code Generator. ACM SIGPLAN Notices, 1984, vol.19, no.6, pp.13-26.

[Gurd83] J. Gurd, I. Watson. Preliminary Evaluation of a Prototype Dataflow Computer. Information processing 83, IFIP Congress series, 1983, vol.9, pp.565-551.

[Harl84] D. M. Harland. Polymorphic Programming Languages. Computers and their applications 25. 1984. Ellis Horwood Limited, Chichester.

[Hech75] M. S. Hecht, J. D. Ullman. A Simple Algorithm for Global Data Flow Analysis Problems. SIAM JOURNAL on COMPUTING, 1975, vol.6, no.6, pp.519-532.

[Hech77] M. S. Hecht. Flow Analysis of Computer Programs. 1977. The Computer Science Library. Elsevier north-Holland, New York.

[Jens75] K. Jensen, N. Wirth. Pascal User Manual and Report. 1975 second edition. Springer-Verlag, Berlin Heidelberg.

[John77] R. G. Johnson, R. A. Mueller. Automated generation of cross-system software for micro-computers. Computer, 1977, vol.10, no.1, pp.23-31.

[Land82] R. Landwehr, H. S. Jansohn, G. Goos. Experience with a Automatic Code Generator Generator. ACM SIGPLAN notices, 1982, vol.17, no.6, pp 56-66.

[Lemo83] K. A. Lemone, M. E. Kaliski. Assembly Language Programming for the VAX-11. 1983. Little, Brown and Company, Boston.

[Lune83] H. Lunell. Code Generator Writing Systems. Linkoping Studies in Science and Technology, Dissertations no.96. 1983. Software Systems Research Center, Linkoping, Sweden.

[Maur83] P. M. Maurer, A. E. Oldehoeft. The use of combinators in translating a purely functional language to low-level data-flow graphs. Computer Languages, 1983, vol.8, no.1, pp.27-65.

[Patn84] L. M. Patnaik, P. Bhattacharya, R. Ganesh. DFL : A Data Flow Language. Computer Languages, 1984, vol.9, no.2, pp.97-106.

[Penn86] T. J. Pennello. Very Fast LR Parsing. ACM SIGPLAN notices, 1986, vol.21, no.7, pp.145-151.

[Scha73] M. Schaefer. A Mathematical Theory of Global Program Optimization. 1973. Prentice-Hall, Englewood Cliffs, N.J.

[Sche77] R. W. Scheifler. An Analysis of Inline Substitution for a Structured Programming Language. Communications of the ACM, 1977, vol.20, no.9, pp.667-656.

**[Tane81]** A. S. Tanenbaum. Computer Networks. 1981. pp.668-676. Prentice-Hall, Englewood Cliffs, N.J.

**[Veen85]** A. Veen. The Misconstrued Semicolon. 1985. Centrum voor Wiskunde en Informatica, Amsterdam.

**[Wadg85]** W. W. Wadge, E. A. Ashcroft. Lucid, the Dataflow Programming Language. 1985. APIC Studies in Data Processing no.22. Academic Press.

**[Wats82]** I. Watson, J. Gurd. A Practical Dataflow Computer. Computer, 1982, vol.15, no.2, pp.51-57.

**[Wulf75]** W. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs, C. M. Geschke. The Design of an Optimazing Compiler. 1975. Elsevier Computer Science Library. American Elsevier Publishing Co., New York.