AFOSR - TR - 76 - 0603

ADA024966

An Approach to Global Register Allocation

Richard Karl Johnsson

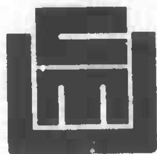
December 1975

Approved for public release; distribution unlimited.

DDC
RECEIVED
JUN 1 1976
D

DEPARTMENT
of
COMPUTER SCIENCE

AIR FORCE OFFICE OF SCIENTIFIC RESEARCH (AFSC)
NOTICE OF TRANSMITTAL TO DDC
This technical report has been reviewed and is
approved for public release IAW AFR 190-12 (7b).
Distribution is unlimited.
A. D. BLOSE
Technical Information Officer



Carnegie-Mellon University

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AFOSR - TR - 76 - 0603	2. GOVT ACCESSION NO.
	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): AN APPROACH TO GLOBAL REGISTER ALLOCATION	5. TYPE OF REPORT & PERIOD COVERED: Interim rept.
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Richard Karl Johnsson	8. CONTRACT OR GRANT NUMBER(s): F44620-73-C-0074
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Carnegie-Mellon University, Computer Science Dept., Pittsburgh, PA 15213	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61101D, AO-2466
11. CONTROLLING OFFICE NAME AND ADDRESS: Defense Advanced Research Project Agency, 1400 Wilson Blvd., Arlington, VA 22209	12. REPORT DATE: December 1975
	13. NUMBER OF PAGES: 138 (141 p.)
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Air Force Office of Scientific Research (NM), Bolling AFB, DC 20332	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

ARPA Order-2466

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Approved for public release; distribution unlimited.

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

see back of this page

403 081

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

The thesis presents an approach to the problem of global register allocation as performed by an optimizing compiler. The problem considered is actually the more general one of choosing what physical resource within the target machine will be used to hold the results of various computations in a running program. The results may be the values of common (redundant) subexpressions, partial results developed during expression evaluation, or variables declared by the programmer. An optimizing compiler can make better use of the resources of the target machine if these decisions are all considered together at or near the source level rather than being distributed throughout the compiler and operating at both source and object levels.

A decomposition of an optimizing compiler is presented with research focusing on one part of the compiler, namely the part which assigns the computed results to physical locations. The entities for which locations must be assigned by the compiler are uniquely identified by *temporary names* (TNs). The process of binding the TNs to actual locations is called TNBIND. A further decomposition of the TNBIND model yields several interesting problems. Three of these problems are considered in greater detail.

(1) Specifying the way in which particular language constructs interact with particular target machine capabilities.

(2) Determining the lifetimes of TNs, i.e., the segments of the program during which each TN contains a valid value. This is similar to what has been called live-dead analysis.

(3) Assigning a large number of TNs to the relatively few physical locations available. This is related to so-called "knapsack" or "cutting-stock" problems in operations research.

Several versions of the TNBIND model are incorporated into the Bliss-11 compiler and compared with each other and with the original compiler in terms of code quality and compilation time.

ACCESSION for
NTIS White Section ☑
DDC Buff Section ☐
UNANNOUNCED ☐
JUSTIFICATION ....
BY ....
DISTRIBUTION/AVAILABILITY CODES
Dist. AVAIL. and/or SPECIAL
A

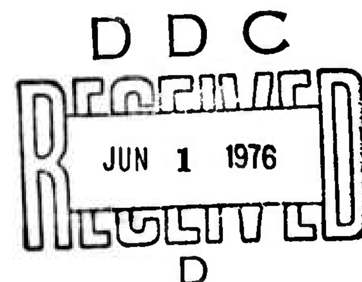
An Approach to Global Register Allocation

Richard Karl Johnsson

December 1975

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Submitted to Carnegie-Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.



This research was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense (Contract F44620-73-C-0074) and monitored by the Air Force Office of Scientific Research. This document has been approved for public release and sale; its distribution is unlimited.

Abstract

The thesis presents an approach to the problem of global register allocation as performed by an optimizing compiler. The problem considered is actually the more general one of choosing what physical resource within the target machine will be used to hold the results of various computations in a running program. The results may be the values of common (redundant) subexpressions, partial results developed during expression evaluation, or variables declared by the programmer. An optimizing compiler can make better use of the resources of the target machine if these decisions are all considered together at or near the source level rather than being distributed throughout the compiler and operating at both source and object levels.

A decomposition of an optimizing compiler is presented with research focusing on one part of the compiler, namely the part which assigns the computed results to physical locations. The entities for which locations must be assigned by the compiler are uniquely identified by *temporary names* (TNs). The process of binding the TNs to actual locations is called TNBIND. A further decomposition of the TNBIND model yields several interesting problems. Three of these problems are considered in greater detail.

(1) Specifying the way in which particular language constructs interact with particular target machine capabilities.

(2) Determining the lifetimes of TNs, i.e., the segments of the program during which each TN contains a valid value. This is similar to what has been called live-dead analysis.

(3) Assigning a large number of TNs to the relatively few physical locations available. This is related to so-called "knapsack" or "cutting-stock" problems in operations research.

Several versions of the TNBIND model are incorporated into the Bliss-11 compiler and compared with each other and with the original compiler in terms of code quality and compilation time.

Acknowledgements

I am deeply indebted to all the members of the Computer Science Department, faculty, staff, past and present students. Together they create an atmosphere which is highly conducive to academic and personal growth.

My special thanks go to my advisor, Professor William Wulf, for his ideas, suggestions, and tireless effort in the original programming of TNBIND for the Bliss-11 compiler. Thanks also to the other members of my committee, Professors Anita Jones, Mary Shaw, and David Casasent, for their helpful comments and criticisms. I must also thank Bruce Leverett for his help and cooperation while I was building my test versions of the compiler.

Finally my thanks go to Nancy and Elsa for their words of encouragement when needed and their coercion when required.

Contents

1. Introduction 1
 1.1 Background 2
 1.2 Issues and Subproblems 4
  1.2.1 Evaluation order 4
  1.2.2 Target path 5
  1.2.3 Machine requirements 6
  1.2.4 Run time environment 7
  1.2.5 User variables 7
  1.2.6 Interaction with control flow 8
 1.3 Previous work 9
  1.3.1 Index register allocation 9
  1.3.2 Evaluation order 15
  1.3.3 Global assignment 20
 1.4 Approach to the problem 24
 1.5 Thesis organization 27

2. A View of Global Register Allocation 28
 2.1 The global register allocation problem 28
 2.2 Local variables 30
 2.3 Input to TNBIND 31
 2.4 Actions of TNBIND 32
  2.4.1 Targeting and Preferencing 34
  2.4.2 Data gathering 35
  2.4.3 Lifetime determination 37
  2.4.4 Importance ranking 37
  2.4.5 Packing 38
 2.5 Summary of the model 38

3. Describing the Language and Machine 39
 3.1 A typical binary operator 40
  3.1.1 The necessary functions 40
  3.1.2 An example 44
  3.1.3 The store operator 46
 3.2 Other operators 47
  3.2.1 Unary operator 47
  3.2.2 If-then-else 48
  3.2.3 Simple loops 48
  3.2.4 Complex operations 49
 3.3 Cost computations 50
 3.4 Mechanically generating TNBIND 51

4. Determination of TN Lifetimes 53
 4.1 Definitions 55
 4.2 An example 57
 4.3 Reflection on lifetimes 64

4.4 Summary of Lifetimes 66

5. The Packing Problem 67
5.1 The problem 68
5.2 The procedure 71
5.2.1 The fathoming procedure 72
5.2.2 Backing up 74
5.2.3 Assigning another TN to a register 74
5.2.4 Honoring preferences 75
5.2.5 An intuitive view of the procedure 76
5.3 The practicality of obtaining an optimal solution 77
5.4 Formulation of the more general problem 78
5.5 Summary of packing 79

6. A Test Case 81
6.1 About Bliss 82
6.2 About the PDP-11 83
6.3 Bliss-11 implementation of TNBIND 84
6.3.1 Subroutine parameters 84
6.3.2 The store operator 86
6.3.3 Cost computations 87
6.3.4 Lifetimes 88
6.3.5 Ranking 89
6.3.6 Packing 89
6.3.7 A new kind of restriction 92
6.4 Measurements of TNBIND 93
6.4.1 The programs 95
6.4.2 The numbers 96
6.4.3 Discussion 99
6.4.4 Evaluation 103

7. Conclusion 105
7.1 The TNBIND Model - Review 105
7.2 More traditional optimizations 107
7.3 Contributions of the thesis 108
7.4 Future research 109
7.4.1 Better algorithms 109
7.4.2 Automatic TNBIND generation 109

A. The TNP Routines 111

B. The Basic Packing Routine 130

Bibliography 132

Chapter 1

Introduction

This thesis presents an approach to the problem of global register allocation as performed by an optimizing compiler. The problem is actually more general than merely choosing one register or another to hold a particular value. A compiler has the freedom and responsibility to choose what physical resource will be used to (temporarily) store information during the execution of a program. This is a freedom given up by a programmer when he changes from an assembler to a compiler. When a program is being executed, many values are computed, stored temporarily, used and then discarded. At the time of compilation a compiler must decide which values will be stored in which locations at runtime. The thesis does not include questions of dynamic storage allocation or other questions of large quantities of storage, but is restricted to consideration of problems usually associated with register allocation.

In most computers there are several classes of locations which may be used to hold information (e.g. registers, main memory, stack). In addition there are usually a number of (logically) identical locations within each class. Because the primary function of a compiler is to translate faithfully any algorithm presented to it, the primary goal must be to make a consistent set of decisions about where information will be stored, i.e. the translation must perform the algorithm described by the source program. This thesis is concerned with the secondary goal of an optimizing compiler, namely to select from the possible translations one that is somehow "better" than the others. The concern here is only with the decisions relating to the use of temporary storage. Other areas of optimization, especially common subexpression elimination and code motion will be discussed only as needed to provide context and motivation for the discussion of temporary storage.

1.1 Background

For many purposes the advantages gained by using an optimizing compiler are far outweighed by the cost of analyzing the alternative code sequences in such a compiler. In fact Hansen [Han74] has suggested that some types of programs are more naturally run in the environment of an adaptive compiler that initially interprets programs and only compiles or optimizes as execution frequency warrants. On the other hand, some applications demand highly efficient programs. In this latter category come operating systems, compilers, and other programs run constantly or very frequently but which are compiled infrequently. Traditionally such programs have been written in assembly languages. By programming at the machine instruction level the programmer is able to use all of his experience and knowledge of both the machine and the problem to make decisions about what code should be written to implement various parts of his algorithm. Until we can accurately build a model of human knowledge, reasoning and associative capacity into a compiler, there will continue to be small programs or parts of programs on which an experienced assembly language programmer can outperform any compiler (in terms of "goodness" of output code).

The problem with the traditional approach is twofold. 1) Assembly language programs do not lend themselves to understandability, modifiability and demonstration of correctness as many higher level language programs do. 2) There are not enough good assembly language programmers. An optimizing compiler is an attempt to solve this problem. The advantages of higher level languages (specifically understandability, modifiability, and demonstration of correctness) are widely accepted even by those who are not satisfied with the efficiency of the code

produced. The hope is that programs can be written in a high level language, and that an optimizing compiler can provide translations which execute with the efficiency of good assembly language programs.

This thesis will consider only that part of an optimizing compiler generally known as the register allocator. The problem considered is actually the more general one of deciding where information may be temporarily stored at program execution time; the locations chosen need not be registers. The idea is to make optimal decisions, but optimality is both hard to define and hard to measure. In terms of register allocation, optimality may mean minimum object code size, minimum execution time, minimum number of registers used, minimum number of loads and stores, some combination of these, or some other criteria. The word optimal was purposely omitted from the title of the thesis because there can be no optimal solution to the register allocation problem without a good deal of qualification on the meaning and measurements of optimality. The emphasis here is on providing a general framework in which particular optimization decisions car easily be made.

Although algorithms for specific aspects of the problem have been presented, there has been little or no consideration of the overall problem. A primary goal of this thesis is to present a model of register allocation which shows the interaction of the various subproblems and allows specific solutions to those subproblems to be easily incorporated into a realization of that model.

In the long term this research is directed toward compiler compilers -- programs which produce compilers from a description of the source language and the target machine. Just as formalization of language syntax led to advancements in mechanically generated parsers, it is expected that the formalization of the register

allocation problem will lead to advancements in this area of mechanically generated optimizers.

1.2 Issues and Subproblems

There are many issues in the overall register allocation problem. Most of these issues arise because we want to produce code that is not only correct, but also "good"; many of the issues become "non-issues" if we are concerned only with the generation of correct code. The issues include at least these:

Evaluation order
Target Path
Machine Requirements
Run time environment
Interaction with control flow

Each of these is discussed in more detail below.

1.2.1 Evaluation order

Since an expression must be evaluated by first evaluating its subexpressions and then combining them, some decision must be made about the order in which the subexpressions will be evaluated and where the resulting values will be stored until needed. A simple example will help illustrate the options. Suppose we are required to evaluate the expression

$$(a * b) + (c * d)$$

Using a three address notation, our evaluation order choices are

$a * b \rightarrow t_1$
$c * d \rightarrow t_2$
$t_1 + t_2 \rightarrow t_3$

and

$c * d \rightarrow t_1$

$$a * b \rightarrow t_2$$
$$t_1 + t_2 \rightarrow t_3$$

The point is that the evaluation of the expression requires that the two products be formed before their sum can be formed. Mathematically, there is no restriction placed on the relative order in which the products are formed. Indeed, if our hardware permits, the two products could be produced in parallel. In practice, however, the evaluation order may be somewhat restricted. The restrictions may be due to the language definition, e.g. the language might specify strict left to right evaluation of subexpressions. The evaluation order may also be restricted because of possible side-effects of an operand. The issue of evaluation order is how to decide which subexpression to evaluate first. In the example of Section 1.2.1 there is no obvious preference for choosing one or the other subexpression as the one to be evaluated first. In some cases choosing the correct evaluation order is the difference between "good" and "bad" code.

1.2.2 Target path

Although similar to evaluation order determination, the selection of target path is in fact an independent decision. In the example above we avoided the target path decision by using a three address notation. Very few computers have full three address capability. The last of the three address instructions above

$$t_1 + t_2 \rightarrow t_3$$

may have two realizations on most machines

$$t_1 \rightarrow t_3$$
$$t_2 + t_3 \rightarrow t_3$$

or

$$t_2 \rightarrow t_3$$
$$t_1 + t_3 \rightarrow t_3$$

The *target path* is the sequence of subnodes which are first loaded into result locations and then operated on by the indicated operators. We refer to an operand lying on the target path as the *target path operand*, although we frequently refer to an operand as being the target path since at any particular node the target path is uniquely identified by an operand. When the alternatives are made explicit, we see that the initial move instruction ($t_1 \rightarrow t_3$ or $t_2 \rightarrow t_3$) may be eliminated if it is possible that t_3 can be the same location as either t_1 or t_2. In the simple case, t_3 may be assigned to either location and we may make an arbitrary choice. In other cases more global context may restrict our decisions. In the above example, let us suppose that t_1 (i.e. a*b) is a redundant subexpression whose value, once computed, may be used several times without recomputation. In this case t_3 must not be the same location as t_1 since the value in t_1 must be preserved for later use. This additional information leads us to choose t_2 as the target path.

1.2.3 Machine requirements

Different computers place differing requirements on the operands of certain instructions. Aside from the special instructions unique to a given machine, these requirements are usually restrictions on the kind of location in which the operands of the normal unary and binary operators may reside. The available kinds of locations may include

main memory
register
top of stack
special register (e.g. floating point, index)
pair of registers

Any general model of the temporary storage management problem must provide for these differing requirements.

1.2.4 Run time environment

Language designers/implementors make decisions about the run time environment which must be considered by a general model of temporary storage management. These decisions are essentially extensions of the machine requirements. Some of the questions to be addressed here are

> How many registers are available for system temporaries?
>
> Which registers are safe and which are destroyed across subroutine calls?
>
> Are there display pointers? If so are they fixed or dynamically allocated?
>
> How are parameters to, and return values from, subroutines handled?
>
> What is the interaction with library and system functions including debuggers?

The model must allow for some degree of freedom in making these decisions.

1.2.5 User variables

Should variables declared and used by the programmer to temporarily hold values be given the same treatment as those generated by the compiler? Program size and execution time can be reduced if some user variables can reside in registers. If we are to consider allocating registers for user variables, it is logical that we do that at the same time we allocate registers for compiler temporaries. While machine restrictions and the run time environment force some decisions about which values must or must not be in registers, we may be able to produce a "better" program if we place some user variables in registers, even at the expense of keeping some compiler temporaries elsewhere.

1.2.6 Interaction with control flow

In order to generate "good" code we must make the best use of the resources available. In the case of registers, generally a scarce resource, this implies that we must be able to recognize precisely when a register contains valid information and when it does not. We say that a location contains a valid value when the value may be referenced before a new value is stored into the location. We can do this in much the same way a clever assembly language programmer does. When such a programmer needs a register for some value, he follows the control flow to determine where the registers currently in use are referenced and where values are stored into them. A register can contain several logically distinct quantities as long as no more than one of the quantities has a valid value at any point in the program. This issue concerns the graph theoretic properties of the program. Suppose that points x and y in the program flow graph are uses of some variable A, and that at point z a new value is stored into A. We want to ask questions such as "Is there a path from x to y that does not pass through z?" If the answer is no, then the variable A does not contain a valid value between points x and z.

Another aspect of this issue is the use of a register over a small piece of program to hold a variable whose value is normally kept in memory. When a variable is accessed frequently within a small segment of code, program performance may be improved by loading that variable into a register before the segment, thus eliminating a memory reference for each access, and storing it back into memory after the code segment. This optimization is most commonly applied to loops because the savings are multiplied by the number of times the loop is executed.

1.3 Previous work

This section presents some of the major results in the area of register allocation. Published work in the area has generally been limited to consideration of simple allocation problems in straight line programs and optimal evaluation order for expressions. Little published material is available on the problems of register allocation in the presence of control flow constructs. The material below summarizes the major results.

1.3.1 Index register allocation

Horwitz, et. al. [Hor66] discussed index register allocation in a paper oriented toward FORTRAN-like programs (and machines like the IBM 7090) which have simple array accessing mechanisms. An index is presumed to be a simple variable whose value must be retained either in a register or in memory at all times. Given the future index requirements of a program, the allocation of the index registers of the machine to the indices is considered. When all of the index registers contain values that will be needed again later in the program, a decision must be made to replace one of those values when a new index is required.

Horwitz considers the possibility that an index may be changed while it resides in a register. If an index is changed in a register, and subsequently that register must be allocated to another index, the changed value must be stored in memory. If the value is not changed, it is not necessary to store the value back into memory when the register is reallocated. This problem is analogous to the problem of page replacement in a virtual memory system. It is less expensive to replace a page which has not been changed since it was read from secondary storage because a valid copy still exists elsewhere.

For the purpose of this problem, a program can be considered to be a sequence of steps each of which requires a specific index. The fact that there may be steps in the program that do not require indices is not important. Consider the set of program steps and associated indices

step	index
1	x_1^*
2	x_2^*
3	x_3
4	x_1
5	x_2
6	x_2^*
7	x_4

where x^* means that index x is changed in the step where x^* appears. When a step calls for an index, that index must be in one of the index registers. The other index registers may contain any configuration of indices. The indices in the other index registers may or may not be in a modified state.

We may construct all of the allowable configurations for each step i, i.e. all combinations of n of the indices used by the program which include the index required by step i (where n is the number of index registers available). Consider the configurations to be nodes in a directed graph with branches from each configuration of the ith step to each configuration of the i+1st step. Each of these branches can be assigned a weight which is the cost of making the change in configurations between steps i and i+1 represented by the branch. The cost of changing between configurations is defined as the number of memory references required to make the change. Thus each new index which is loaded has a cost of one. Each starred index which is replaced has an additional cost of one. Changing an unstarred occurrence of

an index to a starred occurrence of the same index, or replacing an unstarred index require no memory references and therefore have a cost of zero.

Given this representation of the possible allocation of index registers, the problem becomes one of finding the shortest, i.e. least expensive, path through the graph from the first step to the last step. Although there are several algorithms for finding the shortest path through a directed graph, the number of calculations required for other than a small number of nodes makes these solutions impractical. Since it is necessary to find only one of the possibly many shortest paths through the graph, we may restrict attention to any subgraph which contains a shortest path. The bulk of the Horwitz paper is devoted to developing properties of these graphs which lead to rules for eliminating nodes and branches from consideration. Horwitz proves that the subgraph obtained by applying these rules does contain a shortest path, and gives a procedure for finding that path. Six rules are given for generating the subgraph from which an optimal index register allocation may be derived. Define $w(n_1,n_2)$ to be the cost of changing the configuration from that of node n_1 to that of node n_2. Define $W(n')$ to be the weight of a node given by $\min_n(W(n)+w(n,n'))$, i.e. the minimum over all n of the sum of the weight of n and the cost of changing from n to n'. The weight of the initial node is zero. Given these definitions we may summarize Horwitz's rules:

Rule 1: Generate only minimal change branches and eliminate any node which has no minimal change branches entering it. A minimal change branch is defined as a branch from node n at step i to node n' at step $i+1$ such that either nodes n and n' are identical or n' differs from n only in the index required at step $i+1$.

Rule 2: If n_1 and n_2 are nodes of step i and $W(n_1)+w(n_1,n_2) \leq W(n_2)$, eliminate n_2.

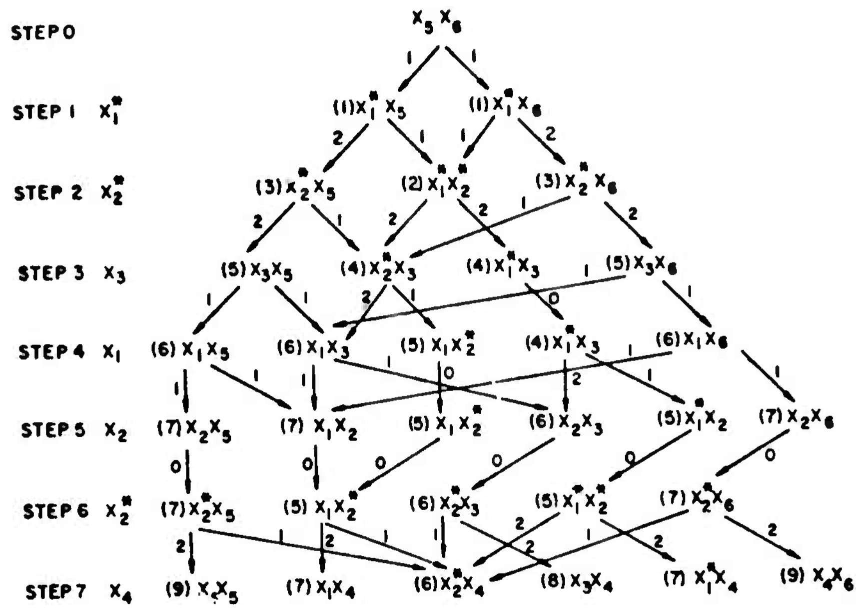


Figure 1-1. The result of applying Horwitz's Rule 1.

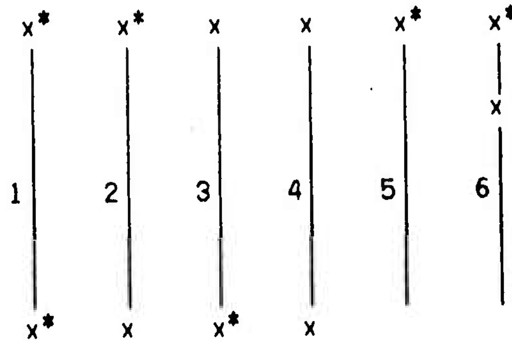


Figure 1-2. Luccio's six link types.

Rule 3: n_1 and n_2 are nodes at step i which differ in exactly one element. Let z_1 be the element of n_1 which is replaced by z_2 in n_2. Although the exact explanation is somewhat more complex, the idea is that node n_2 can be eliminated when $W(n_1) \leq W(n_2)$ and in the future z_1 will be used before z_2. This requires the ability to look ahead in the program.

Rule 4: This rule is a consequence of Rule 3 and prevents generation of nodes that would later be eliminated by Rule 3. If z_1 and z_2 are elements of a node n at step i and the next use of z_1 comes before the next use of z_2, do not form a node at step $i+1$ which replaces z_1 by the index required at step $i+1$.

Rule 5: Since we need only one shortest path, generate only one branch b into each node n' such that $W(n')=W(n)+w(b)$.

Rule 6: If a node n of step i which is not the last step has no branches leaving it, eliminate node n.

Figure 1-1 (reproduced from [Hor66]) shows the result of applying Rule 1 to the graph of the example program above when there are two index registers available. Step 0 is added to indicate the initial configuration which contains two indices not used in the program (x_5 and x_6). Each branch is labeled with the cost of the change between the indicated configurations and each configuration is labeled with the minimum cost to reach the configuration from step 0.

Luccio [Luc67] showed that Horwitz's rules may restrict the graph so that at some steps only one configuration is possible. The program steps before and after such a step may be treated separately. Luccio neatly describes his technique in terms of link diagrams. Six types of links are used to connect various combinations of starred and unstarred indices (Figure 1-2). Links of types 1, 2, 3, and 4 are built whenever a second occurrence of an index is seen. Links of types 5 and 6 are built following occurrences of starred indices and are maintained up to the current step. These are called temporary links since they will be changed to one of the other types when a succeeding occurrence of the particular index is encountered.

A link is said to cover all the steps along its extension excluding the extremes. Only the first extreme is excluded for temporary links so that they cover the current step. Luccio gives two rules for changing links of types 1, 2, 3, or 4 to links of definite allocation (type 0). The index corresponding to a link of type 0 must be kept in its register throughout the entire extension of the link.

If there are N registers available then

1. A link l of type 1 becomes type 0 if for each step k covered by l the number of other links of types 0, 1, 2, 3, or 4 covering k is less than N-1.

2. A link l of type 2, 3, or 4 becomes type 0 if for each step k covered by l the total number of other links covering k is less than N-1.

When the number of type 0 links covering a step k is N-1, the configuration for k is fixed. The n registers must contain the N-1 indices corresponding to the type 0 links and the index required by step k. At such steps the Horwitz method may be applied independently to the preceding and succeeding steps.

The Horwitz method is related to Belady's algorithm for page replacement in a virtual storage computer [Bel66]. Belady showed that in a paging environment, the page to be replaced should be the page whose next use is farthest in the future. In addition he noted that if a page has not been written into, it need not be written out (to secondary storage) but merely deleted. The ability to determine which page (register) is next used farthest in the future depends on knowing the future behavior of a program.

1.3.2 Evaluation order

Ikuo Nakata addressed the question of evaluation order in his paper describing the register allocation phase of a FORTRAN compiler for the HITAC-5020 [Nak67]. Nakata shows that the order of evaluation of the subexpressions of an expression can affect the number of temporary values that are required at any one time. Consider the expression a*b+(c+d)/(e+f). A straight forward code sequence to evaluate this expression is:

$$
\begin{aligned}
a * b &\rightarrow R_1 \\
c + d &\rightarrow R_2 \\
e + f &\rightarrow R_3 \\
R_2/R_3 &\rightarrow R_2 \\
R_1 + R_2 &\rightarrow R_1
\end{aligned}
$$

Suppose, however, that this expression must be evaluated on a computer with fewer than three registers. To use the same evaluation order with only two registers available would require that one of the intermediate results (namely a*b) be stored in some temporary memory location. On the other hand, by changing the order of evaluation of the subexpressions, the expression may be evaluated using only two registers and without storing intermediate results.

$$
\begin{aligned}
c + d &\rightarrow R_1 \\
e + f &\rightarrow R_2 \\
R_1/R_2 &\rightarrow R_1 \\
a * b &\rightarrow R_2 \\
R_2 + R_1 &\rightarrow R_1
\end{aligned}
$$

The central point of this example is that the subexpression (c+d)/(e+f) requires two intermediate values. Since those intermediate results are not needed after the division is performed, one of the registers may be used to compute a*b. Since the result of the evaluation of an expression occupies only one register, it follows that for any binary operator, the operand whose evaluation requires the larger number of registers should be evaluated first.

The number of registers required to evaluate the expression (a) <op> (b) where (a) and (b) are arbitrary expressions and <op> is some binary operator is given by the following analysis. Let l and m be the number of registers required to evaluate (a) and (b) respectively. If either (a) or (b) contains no operators (it is a constant or a simple variable) it requires zero registers. (Note that "require" here means the minimum number of registers necessary to evaluate an expression without storing any intermediate results).

There are two cases:

1. $l=m=p$. (a) can be evaluated first leaving the result in one of the l registers used. Evaluation of (b) will require one more than the p-1 registers remaining giving a total of p+1 registers for the expression.

2. $l \neq m$; $\max(l,m)=p$. In this case the operand requiring the larger number of registers is evaluated first leaving p-1 registers for the other operand. Since the other operand requires at most p-1 registers no additional registers are needed and the expression can be evaluated using only p registers.

In both cases p is a lower bound on the number of registers required and p+1 is an upper bound. In case 1 p+1 is a lower bound, and in case 2 p is an upper bound.

Nakata gives an algorithm for labeling the nodes of a tree with the number of registers required for evaluation of the node. Briefly, this algorithm assigns a label Ln to each node n of the tree such that if n is a leaf then Ln=0, otherwise the immediate descendants of n have labels l and r and Ln=min(max(l+1, r), max(l, r+1)).

Nakata's algorithm for code production involves first labeling the nodes of the tree by the above method, and then beginning at the root node, walking through the tree generating code to evaluate the expression represented. At each node the

operand requiring the larger number of registers is evaluated first. If the operands require the same number of registers, the left operand is evaluated first. Nakata does not consider formally the question of what to do when the number of simultaneous temporary values exceeds the number of registers. He does, however, offer some heuristics for deciding which temporary value should be stored. On most machines the left operand of a division or subtraction operation must be in a register, so the left operand of these operations should not be stored. This may conflict with the other assertion that the value to be stored should be the one whose use is farthest in the future, but Nakata conjectures that the efficiency of the code produced will not be significantly affected by the choice of either of these courses of action.

Using a graph theoretic approach, R. R. Redziejowski [Red69] later proved that Nakata's algorithm does use the minimum number of registers. Redziejowski transformed Nakata's tree into a "lineup" or linear sequence of vertices. Each vertex represents a single operation in the tree and an arc is drawn from vertex x to vertex y to represent a partial result which is computed at y and used at x. Choosing a feasible evaluation order is equivalent to ordering the sequence of vertices so that vertex y precedes vertex x if there is an arc from x to y. (This is equivalent to requiring that any partial result be computed before it is used.)

At any vertex x the number of partial results created before x and used after x is represented by the number of arcs passing over vertex x. Redziejowski calls this number the width of the lineup and develops an algorithm for producing a lineup of minimum width. Redziejowski's algorithm is in principle the same as Nakata's algorithm and therefore Redziejowski's proof of his algorithm can be considered as a

formal proof of Nakata's algorithm. Redziejowski generalizes the algorithm to include operators with more than two operands.

Sethi and Ullman [Set70] consider the more general problem of minimizing the number of program steps and/or the number of storage references in the evaluation of an expression with a fixed number of general registers. They exploit the associative and commutative properties of operators and assume that all elements are distinct (no common subexpressions) and that there are no non-trivial relations between operators (e.g. no distributive law).

Nakata's tree labeling scheme is modified slightly to account for commutative and non-commutative operators. This change assigns a label of one rather than zero to a leaf node which is the left descendant of its ancestor. The change means that the left and right operands of a binary operator may have different weights and accounts for the gains which may be made by exploiting commutativity.

First considering only non-commutative operators, Sethi and Ullman prove that their Algorithm 1 (which is essentially Nakata's algorithm) uses the minimum number of registers as well as the minimum number of loads and stores. Since the number of binary operators is not changed by the allowed transformations, a program which has a minimum number of loads and stores has a minimum number of program steps.

In Algorithm 2, Sethi and Ullman consider commutative operators by adding a step to Algorithm 1 which interchanges the left and right descendants of a commutative operator when the left descendant is a leaf and the right descendant is a non-leaf.

Associativity is treated only in conjunction with commutativity since in practice most associative operators are also commutative. The approach used by Sethi and

Ullman is to make the associative-commutative operators into n-ary operators, reorder the operands so that the one or two operands requiring the largest number of registers appear on the left, and then change back to binary operators associating to the left. This is conceptually similar to Redziejowski's treatment of n-ary operators.

Sethi and Ullman prove that each of their algorithms generates an evaluation sequence containing the minimum number of loads and stores under the assumptions of the algorithm. They then show that this leads to the conclusion that the algorithms also minimize the number of storage references.

In their conclusion, Sethi and Ullman point out that all of their algorithms can be performed in time proportional to the number of nodes in the tree. They also show that the algorithms can easily be modified to allow operations which require more than one register.

Beatty [Bea72] recasts the ideas of Sethi and Ullman in terms of axiom systems. Beatty extends the Sethi-Ullman algorithm for associative-commutative operators to include the unary minus and its relations to the other operators. These relations include the equalities

$$a-b = a+(-b)$$
$$-(a*b) = (-a)*b$$
$$-(a/b) = (-a)/b = a/(-b)$$

Beatty's proof of minimality is considerably more complicated than the Sethi-Ullman proof due to the properties of the unary minus.

More recently Bruno and Sethi [Bru74, Set75] have shown that the register allocation problem for straight line programs is polynomial complete when common subexpressions are not recomputed. The specific problems considered are (1) to use

the minimum number of registers without storing intermediate results and (2) to generate the minimum length code for a one register machine. While the optimal register allocation/evaluation order may be easily determined in some cases, the results of Bruno and Sethi tell us that in general there is no known nonenumerative solution.

1.3.3 Global assignment

The work thus far discussed has dealt only with the question of optimal use of registers in expressions. A paper by W. H. E. Day in the IBM Systems Journal [Day70] considers the much broader problem of global assignment of data items to registers. Before describing Day's work, it is necessary to explain the distinction between what Day calls global assignment and what he considers local assignment. Informally, a local assignment is one which makes assignments within basic blocks, i.e. without control flow. A global assignment considers larger contexts which include control flow. A more formal discussion follows.

Consider a programming language L. The terminal symbols of L are delimiters, operators, constants, and identifiers. The constants and identifiers are the data items of L. A program in L is a sequence of statements; a statement is a sequence of terminal symbols. Statements in L are either descriptive or executable, the latter specifying operations to be performed on data items. A data item is said to be *defined* in a statement when execution of the statement causes a new value to be assigned to the data item. A data item is *referred to* when the value of the data item is required for correct statement execution.

Let P be a program in L. A basic block in P is an ordered subset of elements of P which intuitively is "straight line code," i.e. a sequence of statements which can

only be entered by branching to the first statement and which can only be left by branching from the last statement. P_b is a representation of P as an ordered set of basic blocks. P_g is a representation of P as a directed graph with the elements of P_b as the vertices and a set of arcs representing the flow of control among the basic blocks of P. A region R_i is a strongly connected subgraph of P_g, and P_r is a representation of P as an ordered set of regions:

$$\begin{aligned} P_r &= \{R_1, R_2, \ldots, R_n\} \\ &= \{R_i \mid R_i \neq R_j \text{ for } i \neq j, \\ &\qquad R_i \cap R_j = \phi \text{ or } R_i \subset R_j \text{ for } i < j, \\ &\qquad R_n = P_g \} \end{aligned}$$

A computer has a set of registers G^* whose elements are g_i, and for most situations requiring the use of a register any available $g_i \in G^*$ may be assigned. Let d represent an element of P, P_b, or P_r and define:

$$G' = \{ g_i \mid g_i \in G^*, g_i \text{ is available for assignment everywhere in } d \}$$

$$N' = \{ n_i \mid n_i \text{ is a data item in } P,\ n_i \text{ may be assigned to registers in } d \}$$

Given these representations, Day offers the following definitions:

1. A local assignment is a (possibly multi-valued) mapping of $N \subseteq N'$ onto $G \subseteq G'$ for $d \in P_b$.

2. A global assignment is a (possibly multi-valued) mapping of $N \subseteq N'$ onto $G \subseteq G'$ for $d \in P_r$.

3. A one-one assignment is a one-one mapping of $N \subseteq N'$ onto $G \subseteq G'$. A one-one assignment defines a one-to-one correspondence between N and G.

4. A many-few assignment is a single-valued mapping of $N \subseteq N'$ onto $G \subseteq G'$ with cardinality(N) $\geq$ cardinality(G).

5. A many-one assignment is a many-few assignment in which cardinality(G) = 1.

A data item is *active* at a point in d if it may be referred to before being defined subsequent to that point. Two data items interfere in d if they are both active at some point in d. A necessary condition for the assignment of $N \subseteq N'$ to $g \in G$ in d is that n_i must not interfere with n_j for every $n_i, n_j \in N$, $i \neq j$.

Local assignment, as defined by Day, occurs entirely within basic blocks of a program. The methods described by Horwitz, Nakata, Sethi-Ullman, and Beatty provide algorithms which may be used to obtain optimal local assignments under the assumptions dictated by those authors. Local assignment is not, however, able to cope with data items which may be active on block entry or exit.

Global one-one assignment partially solves the problem of active data items at block boundaries by assigning data items to registers throughout an entire region. With this type of assignment, precautions need be taken only at region boundaries to assure that values of active data items are retained.

Assigning a data item to a register for an entire region may lead to inefficient use of the registers. With accurate program flow information, it is possible to determine the points at which a data item is active. When the active points of all data items are known, a set of data items which do not interfere may be determined and the elements of that set assigned to the same register. The availability of complete and accurate flow information is critical to efficient use of global many-one or many-few assignments. In the absence of flow information, many-one and many-few assignments degenerate to one-one assignments.

Day formulates global one-one, many-one, and many-few assignment problems

as integer programming problems. He makes the intuitively reasonable assumption there is some profit (>0) associated with the assignment of a data item to a register and that this profit depends on the frequency and context of the use of the data item. Day gives several algorithms for solving the assignment problems. Some of these give optimal results while others may produce non-optimal feasible results at a much lower cost in computational complexity. Day's formulations of the problems are summarized below.

The global one-one assignment is the simplest of the three problems since no interference data is required. Refering to the definition of a one-one assignment let n = cardinality(N') and m = cardinality(G') and let p be a vector of profits such that p_i is the profit associated with assigning $n_i \in N'$ to a register. Vector x is a selection vector such that $x_i = 1$ if $n_i \in N'$ is assigned to a register, otherwise $x_i = 0$. Let $\underline{1}$ be a vector of 1's of appropriate size so that $\underline{1}x$ produces the sum of the elements of x; then the problem is

maximize	$z = px$
subject to	$\underline{1}x \leq m$
where	$x_i \in \{0, 1\}$ and $p_i > 0$

The solution to the one-one assignment is simple: assign the m data items with the largest profits to registers.

The global many-one assignment problem is similar to the one-one problem except for the added restriction that no two data items which are assigned to the register may interfere. Day expresses this condition in terms of a matrix of data item interference values ($C \mid c_{ij} = 1$ if $n_i, n_j \in N', i \neq j$ interfere; $c_{ij} = 0$ otherwise).

The many-few assignment problem is an extension of the many-one assignment problem to more than one register. The problem is to select the best combination of

many-one assignments. Day explicitly excludes multi-valued mapping which might assign a single data item to different registers at different points in a region.

In his conclusion, Day reports the results of several tests of the actual execution characteristics of his algorithms for many-few assignment. The OPTSOL algorithm (which provides an optimal solution) requires much longer execution time for relatively little gain over the estimating algorithms. (Sample values: for one register and 48 data items t(optimal) = 6 sec., t(estimate) = 0.06 sec.). The total profits produced by the estimating algorithms are consistently greater than 90% of the profit produced by the OPTSOL algorithm and are significantly better than a one-one solution to the same problem. Day concludes that his algorithms are sufficiently fast to be included in an optimizing compiler.

1.4 Approach to the problem

In order to build the optimizing compiler mentioned earlier, it is necessary to have a general overall model of the resulting compiler. Once we have this model we can divide the task into subproblems along the lines of the phases of the resulting compiler and attack the subproblems individually. The overall structure of the compiler presumed in this thesis is the decomposition of the Bliss-11 compiler [Wul75]. The Bliss-11 compiler is decomposed into five major phases:

1. LEXSYNFLO -- lexical, syntactic and global flow analysis.
2. DELAY -- Program tree manipulation. Replacement of some nodes by simpler but equivalent nodes. Determination of evaluation order and target paths. General decisions about the code to be produced.
3. TNBIND -- Allocation of registers and other temporary storage.
4. CODE -- The actual code generation.
5. FINAL -- Peephole optimization and preparation of the code for final output.

In this thesis we will consider the TNBIND phase of the compiler. The name TNBIND comes from Temporary Name BINDing. A temporary name (TN) is a name assigned to any location to be used as temporary storage. A unique name is assigned for each logically distinct entity, although several names may represent the same physical location in the final code stream. It is assumed that the DELAY phase has made the evaluation order and target path decisions described above. TNBIND must make the actual bindings of TNs to locations after considering the other issues. Of particular importance are the machine requirements and characteristics. Working from the tree representation of the source program TNBIND determines the number and context of the uses of each TN and decides how to bind the TNs to the available locations so as to produce the "best" output code. There are two basic goals in studying the TNBIND phase of the generalized optimizing compiler structure. (1) We want to formalize some of the actions in a phase of compilation that is usually a collection of unrelated algorithms at best, and completely *ad hoc* at worst. (2) We also want to make the transition from a phase of a compiler for a particular language/machine combination to a general model of the temporary storage problem. The TNBIND model will include the assignment of TNs for user variables, and user variables will be considered as equal competitors with the compiler generated temporaries in the allocation of machine resources. The model will also place a great deal of emphasis on accurately determining the interaction of the TNs with control flow in the program. The restrictions placed on the final bindings of TNs to locations which arise from the machine requirements and the run time environment will be considered in a general way so that changes in the machine, the language, or the implementation can easily be incorporated into a new compiler.

The thesis can be seen as an extension of the work described in [Wul75]. We

take the relationship of TNBIND to the other phases of compilation as defined by the Bliss-11 decomposition to be the correct relationship. From that point we expand the TNBIND idea to a general model of temporary name binding applicable to a large class of languages and machines. We also consider new algorithms for the solution of two specific subproblems of the TNBIND model. The goal is to show how to produce a TNBIND phase of a compiler when presented with the language and the characteristics of the target machine. It is not proposed that the result of the research reported here should be a piece of a running compiler compiler system. Rather we will present a notation for describing a general model of a solution of the problem and indicate how the specifics of a particular language or machine may be incorporated in the model. Though it may be somewhat of an understatement, the step from the model presented here to the corresponding piece of a compiler compiler is "merely a matter of implementation."

The following description is an overview of the implementation suggested by the model. We assume here that our only choices for assigning TNs are registers and main memory. Each expression which must produce a value and each user variable is assigned a unique TN. Two values are calculated for each TN: the cost† of accessing the TN if it is assigned to a register and the same cost assuming the TN is assigned to a location in main memory. We also collect for each TN a list of all points in the program at which the value of the TN must actually exist in the assigned location. The optimal binding of tho TNs to the available locations is the one which produces a minimum cost program (i.e. the sum of the costs of each TN for the type of location to which it is bound is minimized) and no two TNs which must contain valid values at any one point are bound to the same location. The really hard parts of the problem are

† in terms of code size, memory references, etc.

determining, accurately, the points at which any given TN must contain a valid value and selecting a set of bindings to minimize cost without a complete enumeration of all feasible bindings. By changing the exact measures of cost, we can change the emphasis of the optimization. If the cost minimization procedure is effective we will produce at least very good, if not optimal, code as defined by our cost measures.

1.5 Thesis organization

Chapter 2 provides a detailed description of the model. This includes the decomposition of the problem into several subproblems including cost computations, lifetime determinations and the actual binding. Chapter 3 gives a notation for describing the pertinent facts about the language and the target machine as they relate to register allocation. This is the place where implementation decisions (e.g. subroutine call-return conventions) and machine specific information are encoded. The machine specific information needed here is not a description of each opcode, but rather more general information such as the relative cost of accessing registers and memory and what kinds of locations may (must) be used to hold the operands of various operators.

Chapters 4 and 5 describe solutions to two specific subproblems: the determination of lifetimes (also known as free-busy or live-dead analysis) and the problem of binding a large number of temporary locations to the limited physical resources of the target machine. Chapter 6 discusses the reimplementation of the TNBIND module of the Bliss-11 compiler as a test case of the thesis. Chapter 7 reviews the model and considers the possible directions for future research.

Chapter 2

A View of Global Register Allocation

This chapter will present a description of the TNBIND model of global register allocation. Global register allocation as used here means making decisions about which register (or other location) will be used to hold a particular value by considering a context larger than a single expression or statement. While it will not always be explicit in the following description, the intent of the model is to make all decisions in the context of a single subroutine. The model could be expanded to consider an entire compilation or, with suitable intermediate storage of data, a set of compilations, but the subroutine is the unit of program frequently considered by other optimizations and is large enough to provide the opportunity for interesting global decisions.

2.1 The global register allocation problem

Most register allocation done in actual compilers is local. As defined in Chapter 1, a local allocation is an allocation done entirely within a basic block. This type of allocation is much easier than a global allocation which makes allocations within a region. It is possible that an entire program may be a single basic block, in which case local and global allocations are identical, however such programs are a small minority and are rather uninteresting. In order to do the global allocation, we must have more information about the control and data flow of the program. In traditional compilers this information is not available at the time register allocation must be done.

Traditional register allocation methods such as those described by Hopgood [Hop69] operate on sequences of machine instructions. The instructions come from

the code generator with the necessary registers specified symbolically. The register allocator then associates the symbolic names with the actual registers of the machine. A local allocation is the logical choice in this situation because the information necessary to make global decisions is difficult to obtain from the machine code. Thus the model of compiling that places code generation before register allocation has to some extent dictated the use of local allocation methods.

The TNBIND model differs from the traditional view by placing the register allocation before the actual code generation. At this point the traditionalists will cry "How can you assign registers before you know what instructions are to be used?" The answer lies in the fact that the TNBIND model considers registers as more general than just a necessary part of a machine instruction. This is reflected in the choice of the term "temporary name" rather than "register" to refer to the entities in question. A temporary name is, quite simply, the name of a place that can be used to store information. It is possible to identify the values (information) that must be computed by looking at the parse tree of a program.† If we couple the knowledge of where these values are computed and where they are used with some basic knowledge about the machine, we can assign actual locations to hold each of the values without ever knowing the exact sequence of instructions that will be needed to perform the computation. Indeed it may be the case that we cannot decide on the exact instructions needed until we have determined whether certain values are being held in registers or not. Thus there is somewhat of a "chicken and egg" flavor to the problem. On the one hand we can argue that we cannot assign registers until we know what instructions are to be used, and on the other hand we argue that we

† In the case of a language construct that is not closely represented by the basic hardware of the machine, it may be necessary to add some information to the nodes of the parse tree during semantic analysis.

cannot decide on the best sequence of instructions until we know which registers will be used and which operands will be in registers.

The global allocation problem is hard. Sethi [Set75] has shown that the general problem is hard even for straight line programs. Not only is the problem of finding a good (i.e. near optimal) allocation polynomial complete, but merely determining whether a given function which associates registers with program nodes is a valid allocation is also a polynomial complete problem.

2.2 Local variables

The TNBIND model treats user declared local variables in the same way it treats temporary storage needed for expression evaluation. This is a logical extension since the programmer uses the abstract computational facility provided by the language in the same way the compiler uses the facilities provided by the target machine. The programmer expresses his operations in terms of the language primitives just as the compiler expresses the language operations in terms of the hardware primitives. The programmer uses his local variables to temporarily hold intermediate results. By treating the local variables like compiler generated temporary storage the programmer reaps the benefits of keeping some, if not all, of his results in registers. The TNBIND model differs from other views of register allocation by declaring, at the outset, that user variables will compete on equal basis with compiler variables for use of the registers.

2.3 Input to TNBIND

Most register allocation algorithms accept a skeleton code stream as input. The skeleton code stream has the actual machine instructions to be generated with the register fields filled in symbolically. It is the function of the allocator to map the symbolic names into physical assignments. Some authors have made a distinction between allocation and assignment; the former merely reserving one of a number of registers and the latter deciding which particular register. In the TNBIND model we use the terms interchangeably. A program may request more registers than are physically available, but it will not require more registers than are physically available. Because many of the operations performed in the TNBIND model are machine independent, having to deal with actual machine instructions introduces unnecessary complications. TNBIND operates on a program tree with machine dependence being supplied parametrically to its subphases as required. Each node of the tree which produces a real value† has been assigned a unique temporary name (TN) to hold the value. A TN is a symbolic representation of the location which will be used to hold the value at run time. TNs are generated from an infinite pool, although they will eventually have to be mapped onto the finite resources of the target machine. Local variables and common subexpressions are also assigned TNs. TNs which have particular restrictions on the type of location to which they may be ultimately bound carry an indication of this restriction. The restrictions are the result of the machine requirements and the implementation decisions as discussed in Chapter 1. In the case of user variables, the user may have restricted the options available to the compiler, e.g. the Bliss-11 compiler allows the programmer to declare that a

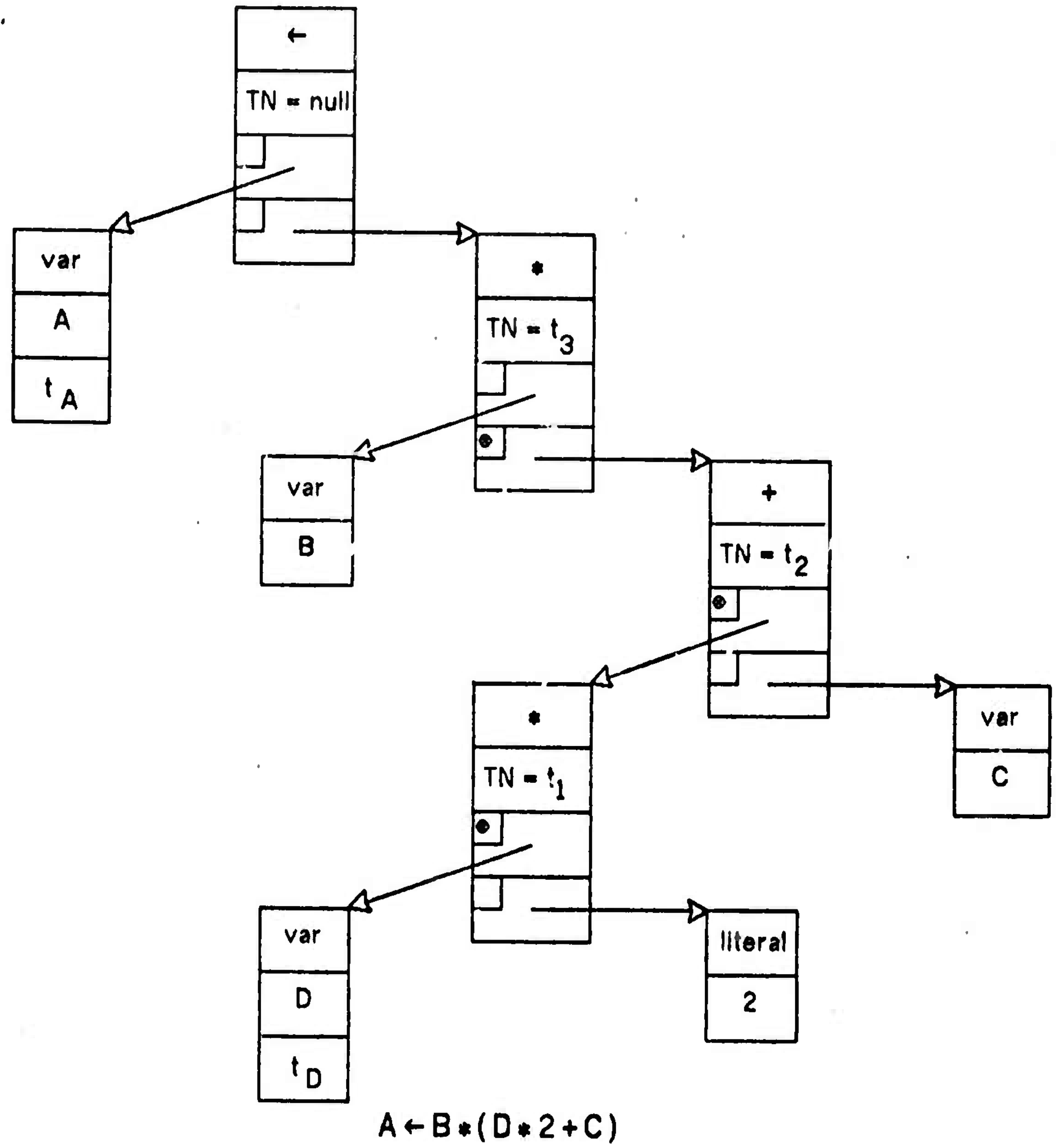
† A real value must be an actual bit representation of a result. A flow value, on the other hand, may be represented implicitly by changing the flow path. The value of a boolean, for example, may often be implicit in the selection of a subsequent flow path.

local variable must (or must not) be kept in a register. The restrictions may include concepts such as "must be a register," "must not be a register," "must not be register 0," or "must be an even-odd register pair." These restriction decisions are made by an earlier compilation phase which has knowledge of the semantics of the language as well as the characteristics of the machine. Earlier phases have also decided the evaluation order of subexpressions and have made the target path decisions. A global flow analysis phase has identified common subexpressions and noted the points at which each such expression must actually be evaluated. There may be more that one distinct common subexpression identified from each set of formally identical expressions [Ges72, Wul75].

Figure 2-1 shows the structure of the input to TNBIND for an assignment statement. Each operator node contains the operator, a TN (which may be null), and pointers to the operands. In the figure the target path operands are marked with the symbol "⊗" which appears in the pointer field. The leaf nodes of the tree structure are variables and literals. The leaf nodes which represent user variables to be treated by TNBIND also contain TNs. In the figure, the variables "A" and "D" have been assigned TNs.

2.4 Actions of TNBIND

Given the input described above, the function of TNBIND becomes one of binding the TNs to actual storage locations (registers, memory, etc.). This proceeds in several subphases as shown in Figure 2-2. In the figure each box represents one of the subphases. Boxes stacked vertically represent independent processes which may be done in parallel. Each element of a vertical stack is dependent on one or more of the elements of the stack on its left. Thus the vertical dimension



$A \leftarrow B*(D*2+C)$

Figure 2-1. The TNBIND input for an assignment statement.

represents potential parallelism while the horizontal dimension represents required sequential execution. Each of the subphases in Figure 2-2 is described below.

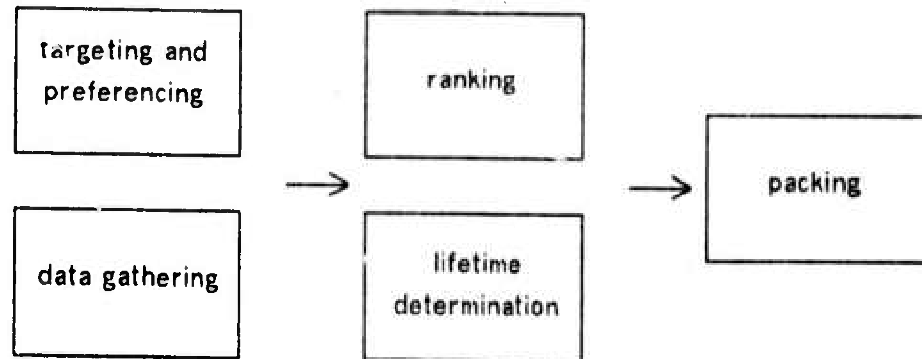


Figure 2-2. The subphases of TNBIND.

2.4.1 Targeting and Preferencing

Targeting is the process of trying to cause each result to be computed in the place the result will be needed, thereby eliminating non-productive data moves. The values of subexpressions are targeted or directed toward producing a final result in the TN in which the result will eventually reside. The targeting process attempts to take advantage of the opportunities recognized in the target path decision (Section 1.2.2).

The process is carried out in an execution-order tree walk. Each node passes its own TN to its target path subnode saying, in effect, "This is where I would like you to leave your result." The subnode considers the desirability of generating its result in the target TN. The considerations include whether the subnode is a common

subexpression or a user variable and whether the restrictions on the two TNs (ancestor and descendant) are compatible. If the target TN is a reasonable place to compute the result, the two TNs are "bound together," i.e. they will henceforth be considered to be the same TN. We refer to this successful operation as "making the target." The ancestor is informed of the decision made at the subnode. If the subnode could not make its target, the ancestor has the option of asking that some preference be given to assigning the two TNs to the same physical location in the later packing phase. The operation of expressing this preference is called *preferencing*. There may be several degrees of preference forming a spectrum from weak preference to strong preference to actual binding together.

2.4.2 Data gathering

In parallel with the targeting and preference class operations, the tree nodes are numbered and a flow graph is built. (Flow information collected during the flow analysis phase may not be accurate since later phases may have changed the order of expression evaluation. Flow information collected during flow analysis may be used if the data collected is sufficiently robust to provide the information needed by TNBIND. It is not important to know how the information is gathered.) The flow graph is represented as a sequence of linear blocks. A linear block is the largest piece of the program having one entry, one exit and no internal branches. Each linear block also has pointers to each of its possible successors.

A record is kept for each TN indicating at which nodes the TN is referenced. The references are separated into two classes: those that replace the value with an unrelated new value and those that use or modify the value. Changing one field of a variable which contains several packed fields is considered a modification. For the

purpose of this record, each node is assigned a *linear order number* (*lon*). (The *lon* values will be more precisely defined in Chapter 4.) This information is used together with the linear block information to determine at which nodes in the program the TN must contain its proper value. The determination can be made by a machine independent procedure which will be described in Chapter 4.

Another item of information recorded for each TN is the cost of its use. This cost is a function of the number of instructions, memory cycles, etc. which are required to perform the specified operations on the TN. A separate cost is kept for each class of storage to which the TN may be bound so that the savings in program size/speed resulting from any particular binding may be evaluated. The cost function may include some frequency of execution data depending on what information is available from the source language. The specification of the cost functions is essentially what determines the optimization criteria. In general a cost may have two parts: a static cost C_s and a dynamic cost C_d. C_s represents the cost in terms of code space. C_d represents the cost of performing the access and is multiplied by the number of times the access is performed. Thus the basic flavor of the optimization can be changed by changing the relative values of C_s and C_d. Increasing C_s relative to C_d tends to optimize for minimum code size while increasing C_d relative to C_s tends to optimize for minimum execution time (not necessarily implying an increase in code size). For any access a which is performed n_a times the cost is $C_s + n_a C_d$. The total cost for a TN is given by the expression

$$\sum_{a} (C_s + n_a C_d)$$

In practice it may be necessary to use an approximation for the number of executions of each access. When the data on branch probabilities is available it is considered in calculating n_a.

2.4.3 Lifetime determination

In using the concept of TN lifetimes, we recognize the fact that in general it is not necessary to retain the value of any TN throughout the execution of the program; it is only necessary that the value be available when required. The analysis of lifetimes has been given many names. Day [Day70] calls a data item "active" if it may be referenced before it is redefined. Allen has discussed "live-dead analysis" [All71b]. Other authors have referred to variables as being "free" or "busy" [Low69]. There is a basic difference in the emphasis of the analysis done by these authors and the analysis required by the TNBIND model. Past work has been concerned almost exclusively with determining whether it is feasible or desirable to keep a copy of a variable in a fast register over some part of a program. The TNBIND model is geared toward keeping TNs permanently in registers while identifying the points in the program at which the register may also be used to hold other values. A detailed description of the lifetime and the method of determining it is presented in Chapter 4.

2.4.4 Importance ranking

After the information described above has been collected the TNs are ordered according to their relative "importance." This is a measure of how important it is for to program optimization to bind a given TN to a particular kind of location. The importance of a TN is a function of the sum of C_s and C_d for each access as well as the restrictions on the binding of the TN. Those TNs restricted to a single storage class are given the highest priority to be bound to that storage class. The ranking phase is really nothing more than the creation of a number of sorted lists which will serve as input to the packing phase. The purpose of the ranking is to select the order in which the TNs will be considered by the packing phase. The most important TNs will be considered first.

2.4.5 Packing

The packing phase does the actual assignment of locations to the TNs (or TNs to locations). A perfect assignment is one which satisfies all of the restrictions and preferences and assigns each TN to a member of the most desirable of its allowed storage classes. In practice it is not always possible to make such perfect assignments. The function of the packing phase in this case is to minimize the increase in cost over the perfect solution. When it is not possible to assign all TNs to their desired locations, we attempt to select a minimum cost solution from the set of all possible assignments. This set may be very large. If there are k classes of storage to which TNs may be assigned and there are n TNs, then there are k^n possible assignments. Chapter 5 describes how this solution space may be searched efficiently for a minimum cost solution.

2.5 Summary of the model

The TNBIND model is summarized by the following statements:

1. A unique name (TN) is generated for each entity for which the compiler must choose a physical location.
2. Using knowledge available from the program tree and knowledge about the target machine, costs and patterns of use are determined.
3. TNs which should be bound to the same location are identified. A record is kept when it is desirable (from the optimization viewpoint) that two TNs share the same location, but such sharing may not be consistent with the semantics of the program or may not be the best decision in a more global context.
4. Lifetimes for each TN are determined from the use patterns and program flow information.
5. Finally, TNs are bound to physical locations such that the cost due to inability to make perfect assignments is minimized.

Chapter 3

Describing the Language and Machine

The TNBIND model relies on attributes of the target machine. It can be adapted to a new target machine by respecifying machine attributes including the types and number of temporary storage locations to be used and the characteristics of the various operations that the target machine can perform. The model treats programs at a much higher level than traditional register allocation methods. Since the details of the instruction set of the target machine are not explicitly encoded in the model, the respecification needed to effect a change of target machine is the relationship of target machine capabilities to language constructs rather than the format and semantics of the instructions.

The various language dependencies and machine dependencies affect the first phases of the TNBIND process. There is a specific routine for each node type. These routines perform the targeting, preferencing and data gathering functions for all of the TNs in the program unit being compiled. As mentioned in Chapter 2, we will assume that the unit of compilation is the subroutine. TNBIND considers the nodes of the program tree in execution order. That is, the first node considered is the first node for which code will be generated. In a tree representation of the program, the first nodes to be considered are the leaves. The descendants of a node represent the computations which must be performed before the computation specified by the node itself. TNBIND accomplishes the execution order examination of the modes by using a recursive tree-walk algorithm. At each node the algorithm is invoked to examine the subnodes. When the algorithms is invoked on a target path subnode, a target TN is passed. When the bottom of the tree is reached, the necessary TN

processing for the leaf node is performed, and the result is returned to the ancestor node. When all of the descendants of a node have been processed. The node itself is processed and then returns to its ancestor. In order to clarify this structure, we present below a few examples of the node types and the actions necessary to process them.

3.1 A typical binary operator

3.1.1 The necessary functions

Let us consider the actions that are necessary during the TNBIND processing of a typical binary operator. We will assume that this operator is one which is represented directly by the hardware, i.e. there is a hardware instruction which implements the basic operation. For example, the "ADD" instruction directly implements the "+" operator. The TNBIND processing is identical in form to the actual code generation process. By this we mean that the code generator traverses the tree in the same manner in order to generate code. The difference is that the TNBIND phase does not involve the actual instructions, but rather the control flow and the number and kinds of references made to temporarily stored data. The processing of a binary operator usually proceeds as follows:

1. process first operand
2. process second operand
3. move target path operand to temporary
4. operate on temporary with non-target operand
5. leave result in temporary

The TNBIND processing mimics the actions of the code which will eventually be produced. (If the target machine organization dictates some other sequence of evaluation for a binary operator, then the TNBIND processing will change accordingly.)

First the two operands are processed by TNBIND with the current node's TN being passed as a target to the target path operand. This processing occurs recursively until the bottom of the tree is reached. After the two operands have been processed, we look to see whether the target path operand "made its target." If not we must allow for the move of the value of the target path operand from the TN where it exists to the TN of the current node by increasing the costs of both TNs by the cost of the move. In this case we also record the fact that we prefer that the two TNs (target path TN and current node TN) be assigned to the same location. If they can be assigned to the same location by the later packing phase, the data move can be eliminated. We next update the cost values of the non-target TN and the current node's TN to include the cost of the operation. Lastly we decide whether to bind the current node's TN to the target that was passed in from above. Thus targeting decisions are made by the recipient of the target request, and targeting is done by the requestor when the targeting request is rejected.

In order to specify these actions we need to be able to talk conveniently about nodes, operands, TNs and costs. We need to define the basic operations to be performed and describe the sequence of these operations that should be performed for each node type.

The primitives we need for the typical binary operator described above are

TNP(node,target)
: node: a tree node
: target: a TN or 0
: Invoke the TNBIND processing on "node" passing "target" as the target. When no target is being passed, "target" is 0. This is the function invoked for each node by the recursive tree-walk algorithm. Common actions are performed by TNP and then the node-specific function for the node type is invoked.

PassTarget(node,target)

node: a tree node which is an operand of a binary operator

target: a TN

Evaluates to "target" if "node" is the target path operand. Otherwise the value is 0 meaning no target. This conveniently expresses the notion that the target is passed only to the target path operand without assuming which operand of a binary operator lies on the target path.

NoteUse(t,when)

t: a TN

when: a lon value

Add "when" to the list of nodes which are uses of "t".

NoteCreation(t,when)

t: a TN

when: a lon value

Add "when" to the list of nodes which are creations of "t".

Move(tfrom,tto)

tfrom: a TN or tree node

tto: a TN or tree node

If there is a TN associated with "tfrom" then invoke NoteUse passing the TN and the current lon value. Similarly for "tto" except invoke NoteCreation.

PrefMove(tfrom,tto)

tfrom: a TN or tree node

tto: a TN or tree node

Indicate that the TNs associated with "tfrom" and "tto" should be assigned to the same location if possible (preferencing), then invoke Move(tfrom,tto).

Operate(op,o1,o2)

op: a machine operation

o1: a TN or tree node

o2: a TN or tree node

Invoke NoteUse and/or NoteCreation as appropriate for the action of executing "o2 ← o2 op o1". For example, the call "operate(+,X,Y)" means that the action "X ← X + Y" is a use of both X and Y and therefore invokes NoteUse on both X and Y. For unary operators we omit "o2". Some operations are specified generically, e.g. "test" meaning "test for true or false".

Bind(t1,t2)

t1: a TN

t2: a TN

Bind the two TNs together. That is, force them to refer to the same location. This is the action when a node "makes its target."

In addition to these primitives we need a few notational conventions to simplify the explanation of the processing routines. We will use the notation "X[field name]" to represent the value of the named field of the item named X. Thus "q[first operand]" will refer to first operand of a node named q. We will also make use of the following abbreviations.

Abbr.	meaning
MyTN	node[temporary name]
Opr1	node[first operand]
Opr2	node[second operand]
OpType	node[operation]

We are now in a position to state the actions necessary to process a binary operator node in terms of the primitives.

```
TNP(Opr1,PassTarget(Opr1,MyTN))
TNP(Opr2,PassTarget(Opr2,MyTN))
```

PrefMove(node[target path],MyTN)
Operate(OpType,node[non target path],MyTN)

3.1.2 An example

Let us consider the processing of the assignment statement of Figure 2-1. The tree is reproduced in Figure 3-1 with names attached to the nodes so that we can refer to them easily. TN processing begins with node N_1, the assignment operator. The assignment operator invokes TNP on each of its operands, passing no targets since neither operand is flagged as the target path operand. Figure 3-2 shows the complete sequence of invocations. The calls to PassTarget are not shown; rather the result of the PassTarget evaluation is shown explicitly in subsequent calls to TNP.

In Figure 3-2 we see the TN processing recurring until the bottom of the tree is reached. This first happens when N_4 processes node D. After N_4 has processed both of its operands, it sees that its target request to node D was rejected and therefore invokes PrefMove to indicate that the value of D must be moved into t_1. At this time PrefMove updates the costs of t_1 and t_D, invokes NoteUse and NoteCreation on t_D and t_1 respectively, and adds each TN to the other's preference list. N_4 then invokes Operate to update the costs of the TNs involved in the operation. Finally N_4 invokes Bind to bind its own TN to the target TN.

After processing its second operand, N_3 sees that N_4 accepted the target request meaning that the value of N_4 will be left in t_2. N_3 invokes Operate and Bind and returns to N_2. N_2 invokes Operate and returns to N_1 which calls PrefMove to update the costs for the data move from t_3 into A.

The results of this processing are:

1. t_1, t_2, and t_3 will all refer to the same location. This means that the

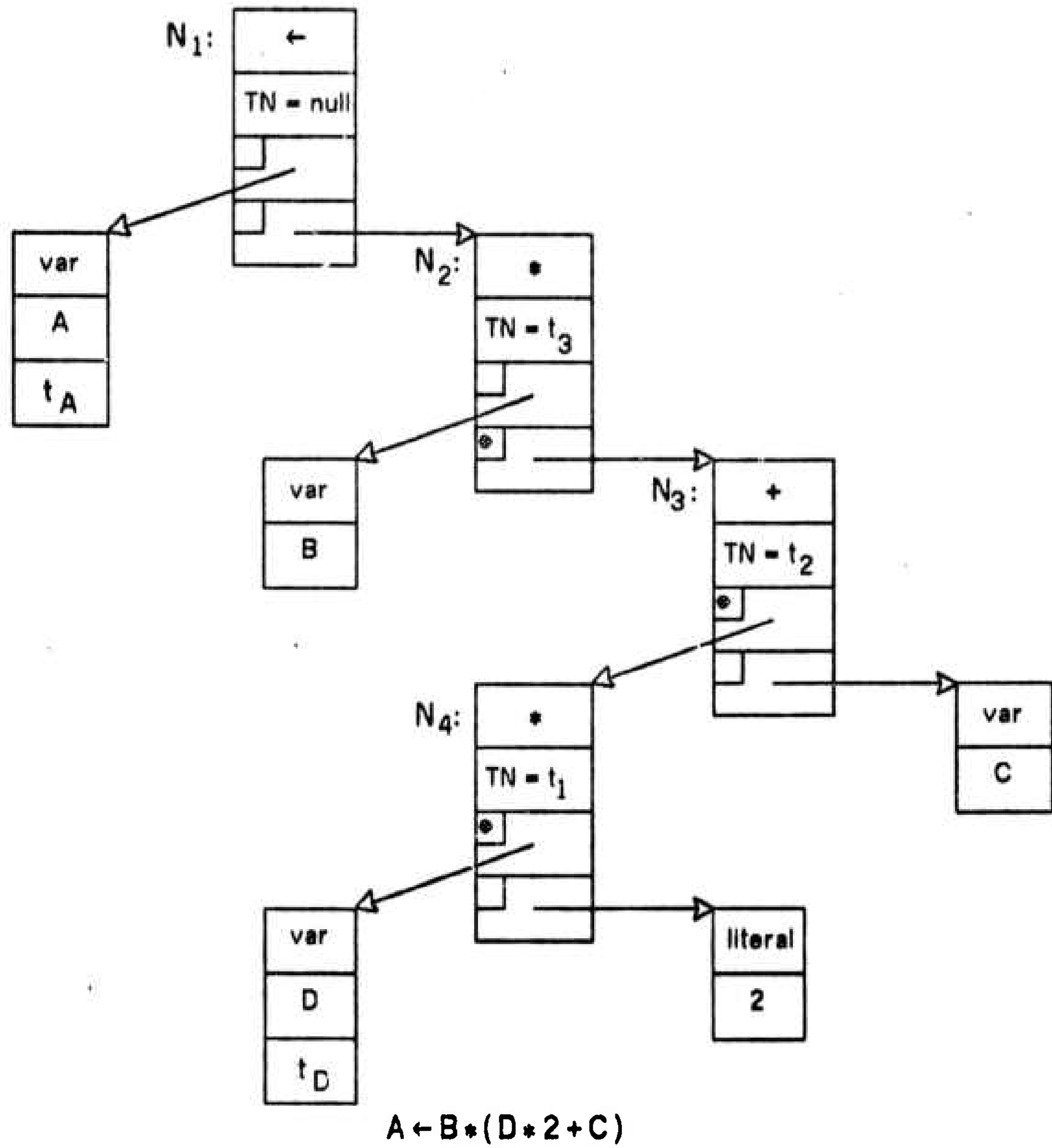


Figure 3-1. An expression tree to be processed by TNBIND.

controlling node	action
	$TNP(N_1,0)$
N_1	$TNP(A,0)$
N_1	$TNP(N_2,0)$
N_2	$TNP(B,0)$
N_2	$TNP(N_3,t_3)$
N_3	$TNP(N_4,t_2)$
N_4	$TNP(D,t_1)$
N_4	$TNP(2,0)$
N_4	$PrefMove(D,t_1)$
N_4	$Operate(*,t_1,2)$
N_4	$Bind(t_2,t_1)$
N_3	$TNP(C,0)$
N_3	$Operate(+,t_2,C)$
N_3	$Bind(t_3,t_2)$
N_2	$Operate(*,t_3,B)$
N_1	$PrefMove(t_3,A)$

Figure 3-2. A trace of TN processing actions.

expression on the right hand side of the assignment will be evaluated without any unnecessary data moves.

2. t_D is preferenced to t_1. If this use of D happens to be the end of a lifetime segment, then t_D and t_1 will be assigned to the same location thus eliminating the initial data move.

3. t_A is preferenced to t_3. If the lifetimes of t_A and t_3 have no points in common, t_A and t_3 will be assigned to the same location thus eliminating the final data move.

3.1.3 The store operator

The store operator is considered to be a special properties. These properties are due to the special optimizations which may be performed on store operations. In general we evaluate the right hand side leaving the result in some TN and then we move the value from the TN to the location named by the left hand side. We would

like to eliminate the final data move when the store operation is inherently simple. We say that a store operation is simple when it is possible to bind the TN of the right hand side to the location of the left hand side.† We define a predicate in order to identify the simple cases:

SimpleStore(left,right)
left: a tree node
right: a tree node
True if it is possible to bind the TN of "right" to the location of "left"; *false* otherwise.

A few of the operations which might be simple on some machines are
1. A←A+k (k a constant)
2. A←B+k
3. A←A op B(op one of some set of operators)
4. A←A op e(e an expression)

3.2 Other operators

3.2.1 Unary operator

With the notation established, we can easily describe the actions required by other types of tree nodes. The simplest of these is the typical unary operator:

TNP(Opr1,MyTN)
PrefMove(Opr1,MyTN)
Operate(OPTYPE,MyTN)

As noted in the discussion of binary operators, the actions taken in TN processing mirror the code that will be used to implement the operation.

† The exact definition of which store operations are simple is dependent on the capabilities of the target machine, but the concept is machine independent

3.2.2 If-then-else

The processing for more complex nodes, such as if-then-else, is also easily expressed in terms of our primitives.

```
TNP(node[boolean],0)
Operate(test,node[boolean])
TNP(node[then part],MyTN)
PrefMove(node[then part],MyTN)
TNP(node[else part],MyTN)
PrefMove(node[else part],MyTN)
```

Note that this presumes that the if-then-else has a value. If the language does not provide for an if-then-else expression then the calls to PrefMove and the passing of targets to the then and else parts may be eliminated.

3.2.3 Simple loops

The processing for a while loop is also very straightforward. Remembering that at this point we need not know what instructions will be generated to implement the loop, we need only process the subnodes of the while in the order in which the code will be executed. The fact that the body of the loop may be executed several times is irrelevant to the basic processing, but should be considered in the cost computations.

```
TNP(node[boolean],0);
Operate(test,node[boolean])
TNP(node[body],0)
```

Note that in this case we make no provision for the value of the while loop. If, for example, we wanted to specify that the value of the loop was the value of the last evaluation of the body, we could pass MyTN as a target to the body and insert a PrefMove at the end of the processing routine.

3.2.4 Complex operations

Up to this point we have discussed language constructs that are common to most algebraic languages and easily implemented on most computer hardware. One of the advantages of the TNBIND model of using temporary storage is the ability to handle new and more complex language constructs much closer to the language level rather than at the machine instruction level.

Consider a language which implements lists of items as a primitive data type. There might be a language construct which allowed a sequence of statements to be performed on each element of a list. A programmer might write something like

forall I in L do begin s_1; s_2; s_3; . . .; s_n end;

The tree node for such a construct would have three subnodes: one for the item name (I), one for the list name (L) and one for the body. The processing in TNBIND might look like

```
NoteCreation(node[item name],node[lon])
Operate(end-of-list-test,node[item name])
TNP(node[body],0)
Operate(next-item,node[item name])
```

where the linear block information would show that the end-of-list-test was the successor of the next-item operation. This example is rather explicit in its use of a TN to hold each element of the list in turn. It is easy to imagine, however, that the same treatment could be given to other constructs. For example, it might be the case that the "forall" construct was only an internal representation produced by the semantic analysis phase so that the programmer's

L ← sqrt(L)

is transformed into

forall I in L do I ← sqrt(I)

3.3 Cost computations

A very important part of the TNBIND process is the computation of the costs associated with each TN. One of the simplest measures of cost to compute is object code size. It is easy to calculate the size of the object code required for a given access to a TN. This is a particularly interesting cost measure for a machine which an instruction which accesses a register is smaller that an equivalent instruction which accesses a memory location. On other machines some other measure may be more appropriate. The point is not what the measure is, but rather that the cost computation can be separated from the rest of the processing and modified independently as desired.

As discussed in Chapter 2, we want to collect relative cost measures; the absolute measures are of only marginal interest. In terms of a code size measure, the cost data we want to get information such as "Assigning variable X to a register instead of a memory location will save 8 words of code." We can then say that the cost of (failing to allocate a register for) X is 8 words. If code size is our only measure then it makes no difference whether the 8 is the difference between 2 and 10 or between 100 and 108.

The costs are calculated in very much the same way that the TN processing is carried out, i.e. by having separate routines to handle the specific information about each node type and a driver routine to handle the common information. The cost calculations can be included in the TN processing routines, or they can be separated into a separate pass ofer the tree.

The cost computations rely most heavily on the attributes of the target machine. The data that must be available for the cost computations includes the following values for each of the storage classes to be considered.

The cost of a simple access of a TN.

The cost of isolating a subfield of a TN.

The cost of implementing non primitive functions, e.g. the cost of performing the "exclusive or" operation on a machine which does not have a corresponding hardware function.

In traversing the tree, all references to a TN are analyzed and the costs are accumulated. Those TNs which must be assigned to a register are assigned arbitrarily high costs to assure that they are treated first in the packing process. The TN processing phase attempts to keep the lifetimes of such TNs as small as possible, relying on the preferencing operation to eliminate loads and stores when possible.

3.4 Mechanically generating TNBIND

It should be possible to generate the TNBIND model of a compiler mechanically from a description of the input language and the target machine. This is not to say that there is an obvious algorithm which accepts BNF and ISP descriptions as input and produces program text as output, but rather that there is a systematic way of using knowledge about the language and the machine to generate the necessary TNBIND routines.

One possibility for specifying the language is to provide functional descriptions of the operators which look very much like the descriptions in this chapter. That is, specify for each node type the sequence of actions it should take during program execution. The information is not very specific, but rather a much more general description of the order in which the subnodes are evaluated and how the values of the subnodes are to be used. Any initial attempt to specify a new language for TNBIND will most certainly involve writing out the processing routines for each node

type in the tree representation of the language. In TNBIND we are concerned not with the syntax of the input language but with the semantics.

Several pieces of information about the target machine obviously must be supplied. These include the types of locations to be used for temporary storage and the number of locations of each type. TNBIND must also know the functional characteristics of the various operations which can be performed by the target machine. For example, if the target machine is of the "general register" variety without no memory-to-memory instructions, TNBIND must know that there must be a TN associated with the right hand side of every assignment, and that that TN must be bound to a register. One candidate for specifying such information is Newcomer's attribute sets and transformations [New75]. Building TNBIND requires both the knowledge of what the attributes and transformations are and the costs of making the transformations. The appropriate cost measures and values are not readily determined from classical descriptions of computer hardware.

The advantage of the TNBIND model is that it provides a mold into which we can fit descriptions of languages and machines. In this role the model serves to point out options and keep the treatment of operators uniform.

Chapter 4

Determination of TN Lifetimes

One of the assumptions made in discussing the problem of register allocation is that it is in fact a problem. As computer hardware advances are made, the number of registers which can reasonably be made available increases. As long as there are more registers available than there are data items to store in them, there is no real problem. On the other hand, in such cases, we begin to think of new ways to use the registers to increase the efficiency of our programs. When machines moved away from the single accumulator model, it became possible to consider using registers for purposes other than expression evaluation.

A number of articles have appeared describing methods of program optimization through judicious use of the registers. Lowery and Medlock [Low69] describe the analysis done in the FORTRAN H compiler to keep the values of frequently used variables in registers within loops. This type of optimization is frequently referred to as "load-store motion" [All71a].

A good deal of work has been done in the area of program control flow for the purpose of finding the paths along which the value of a variable may be retained in a register. Allen [All70] and Beatty [Bea71] discuss the use of graph theory and the concepts of regions and intervals of a graph to determine the aspects of control flow relevant to register allocation.

The current state of computer hardware provides us with (in most cases) more than enough registers to evaluate the most complex expressions occuring in programs. In order to improve the efficiency of our programs we would like to keep

the values of frequently used variables in registers where they can be accessed with more speed and sometimes with shorter instructions. A simple approach is to first allocate the registers necessary for expression evaluation and then use the remaining registers, if any, to hold one variable each, inserting the necessary load and store instructions into the code at the beginning and end of the program segment being considered (the most frequent piece of program used with this method is the loop as in [Low69]). A more ambitious goal is to multiplex several variables into each register. This idea has been discussed in [Day70] and [Bea71]. The approach taken in the TNBIND model is of the multiplexing variety, but with a different emphasis. In the TNBIND model variables are either assigned to a register throughout their lifetimes or they are not. This is conceptually much simpler than a model which loads and stores several variables during the course of executing a program. The disadvantage is that some variables used heavily in the inner loop of a program may not be assigned to registers by TNBIND. We might encourage the assignment of registers for variables used in loops, or we might seek a way to incorporate the loop optimizations into the TNBIND model. The latter possibility is discussed in Chapter 7.

This chapter describes a method of determining the lifetime of temporary names. We want a characterization of the lifetimes which is not only accurate but also very precise so that we can make maximal use of the registers by assigning several TNs to each register. The lifetime of a TN is the set of those segments of the program during which the value of the TN must be available. The complexity involved in determining the lifetime of a TN is related to the type of TN, e.g. TNs which are compiler generated temporaries generally have much shorter and more easily determined lifetimes than TNs which are user variables. A few definitions will help in the discussion of lifetime determination.

4.1 Definitions

A *program point*, p, is used to label an instant in the execution of a program. The term is also applied to the static program and essentially names a particular object code instruction. A program point represents a node in the graphical representation of the program. A *successor* of a node is an element of the set of instructions which may be executed immediately after the instruction represented by the node. In terms of the program tree, the successors of a node are either brothers or immediate ancestors.

A *flow path* is a sequence of program points $p_1, \ldots, p_n$ such that for all i ($1 \leq i < n$) p_{i+1} is an immediate successor of p_i. A flow path describes a possible sequence of nodes in the program flow graph. The length of a flow path is the number of transitions necessary to move from the initial point to the final point of the path, i.e., the length is one less than the number of points in the path.

A *use* of a TN is any reference to the TN which requires the value stored in it. This includes simple loading of the value or assigning a new value which is a function of the old value. The latter kind of reference may be called a *modification*.

A *creation* of a TN is a reference which stores a new value (not a function of the old value) into the TN.

A *lon*, or *linear order number*, is a unique number assigned to each node in the program graph. The lon values are used to name the program points. The values increase along any flow path through the program graph except in the case of loops. For loops the successor of the loop has a lon which is greater than the lon of every

node in the loop.† Within a linear block the lon values are consecutive.

A TN is *alive* at a point p_i *iff* there exists a flow path $p_i, \ldots, p_n$ such that p_n is a *use* of the TN and for every p_j ($i<j<n$) on the path, p_j is not a creation of that TN. The initial point of the path, p_i, need not reference the TN at all. The essential idea is that the value of the TN must be preserved at every point along the path.

The lifetime of a TN is the set of points in the program at which the TN is *alive*. The kind of information we need to determine lifetimes is similar to that used in global common subexpression recognition. In order to recognize common subexpressions we need to know when the value of a variable changes so that we can find all of the expressions involving that variable and mark them as changed. This is essentially an analysis of program flow and has been discussed by several authors [All70, Coc70, Ges72].

For the purpose of describing the lifetime determination, we will assume the following information is available: (1) a description of the linear blocks† of the program in terms of their starting and ending lon values, (2) the starting lon values of each successor, and (3) a list for each TN indicating, by lon value, the nodes which are creations or uses of the TN.

A *connection matrix* is a matrix of binary values representing the successor relationships among the nodes of a graph. The connection matrix C of a graph of n nodes is an n x n matrix ($c_{ij}=1$ if node j is an immediate successor of node i, $c_{ij}=0$

---

† The assumption here is that we are not burdened with the unrestricted control structures that can be constructed by using an arbitrary goto. This is not to say that the method to be described will not work for such structures. Rather the simplying assumption merely makes the exposition of the method less complex.

† A linear block has exactly one entry, one exit, and no internal branches. It is, in the most restrictive sense, "straight line" code.

otherwise). If the nodes of a program graph are numbered by lon as defined above then some generalizations about the corresponding connection matrix may be stated. Within a linear block the only successor of a node is the node with the next highest lon value, i.e., if node i is <u>not</u> the last node of a linear block then $c_{ij}=1$ iff $j=i+1$. Consider two linear blocks LB_1 and LB_2 composed of nodes a, a+1, . . ., b and d, d+1, . . ., e respectively. By the definition of linear block we know that the two sequences have no points in common and that, except possibly at the endpoints, no node of one block is a successor of any node in the other block. In the connection matrix

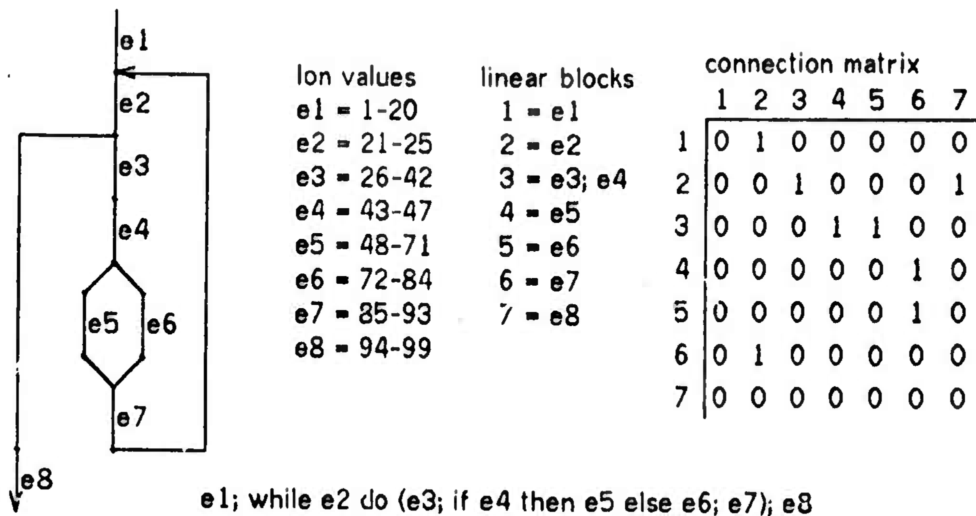
$$a \leq i < b \quad \wedge \quad d \leq j < e \quad \text{implies} \quad c_{ij} = 0 = c_{ji}$$

Moreover, if we think about a matrix representing the connections among linear blocks, we know that LB_2 is a successor of LB_1 iff $c_{bd}=1$ and LB_1 is a successor of LB_2 iff $c_{ea}=1$. By knowing the composition of the linear blocks we can readily convert back and forth between the full connection matrix and the matrix of linear block connections.

4.2 An example

A sample program graph with lon values, linear blocks, and the linear block connection matrix is shown in Figure 4-1. The example program has 99 tree nodes broken down into 7 linear blocks. Consideration of the connection matrix of linear blocks instead of the connection matrix of nodes yields a significant reduction in the amount of data required. Unfortunately this abstraction does not contain all of the information from the original connection matrix, although the lost information may be recovered.

Primarily because of the increased storage efficiency, we would like to be able



	1	2	3	4	5	6	7
1	0	1	0	0	0	0	0
2	0	0	1	0	0	0	1
3	0	0	0	1	1	0	0
4	0	0	0	0	0	1	0
5	0	0	0	0	0	1	0
6	0	1	0	0	0	0	0
7	0	0	0	0	0	0	0

e1; while e2 do (e3; if e4 then e5 else e6; e7); e8

Figure 4-1. The example program.

to perform matrix operations on the full connection matrix by manipulating the smaller linear block connection matrix. In order to do this we must transform the linear block connection matrix into a more accurate representation of the full connection matrix. In Figure 4-2 the connection matrix has been modified to include entries labeled "S" (for sequential) along the main diagonal. The connection matrix is now a shorthand for the much larger node-level connection matrix. In our shorthand matrix, each element represents a submatrix of the full connection matrix. The interpretation of the values in the shorthand is

$0 \rightarrow m_{i,j} = 0 \ \forall i,j$
$1 \rightarrow m_{1,n} = 1$, $m_{i,j} = 0$ for other i,j; n is the number of rows in the submatrix
$S \rightarrow m_{i,i+1} = 1$, $m[i,j] = 0$ for $j \neq i+1$

Figure 4-3 shows an expanded section of the matrix from Figure 4-2. The rows and columns in Figure 4-3 are labeled with lon values. The lines within the matrix show the partitioning.

	1	2	3	4	5	6	7
1	S	1	0	0	0	0	0
2	0	S	1	0	0	0	1
3	0	0	S	1	1	0	0
4	0	0	0	S	0	1	0
5	0	0	0	0	S	1	0
6	0	1	0	0	0	S	0
7	0	0	0	0	0	0	S

Figure 4-2. Connection matrix with "S" entries.

	16	17	18	19	20	21	22	23	24	25
16	0	1	0	0	0	0	0	0	0	0
17	0	0	1	0	0	0	0	0	0	0
18	0	0	0	1	0	0	0	0	0	0
19	0	0	0	0	1	0	0	0	0	0
20	0	0	0	0	0	1	0	0	0	0
21	0	0	0	0	0	0	1	0	0	0
22	0	0	0	0	0	0	0	1	0	0
23	0	0	0	0	0	0	0	0	1	0
24	0	0	0	0	0	0	0	0	0	1
25	0	0	0	0	0	0	0	0	0	0

Figure 4-3. Expanded section of connection matrix. (Linear block 2 and a portion of linear block 1).

A second modification to the matrix is based on the information we wish to obtain from the program, namely what nodes in the program are on flow paths leading to any use of a particular TN. In this context any creation of the TN effectively terminates the path. More simply, a creation node has no predecessors. This is reflected in the matrix by setting the columns corresponding to the creation nodes to 0 (indicating that they cannot be successors of any node). In the shorthand matrix this is done by first making the creation nodes separate linear blocks. If we consider a TN with creations at nodes 6, 67, and 80 (in linear blocks 1, 4, and 5) the resulting

shorthand matrix will appear as shown in Figure 4-4. The columns labeled 1b, 4b, and 5b are the new blocks, each consisting of a single node. Although the matrix we must now consider has increased in size from 7x7 to 13x13, it is a far cry from the potential 99x99 matrix we would have if we considered each node separately. It is important to keep the size of the matrix to a minimum since from this point on the algorithm must be performed separately for each TN.

	1a	1b	1c	2	3	4a	4b	4c	5a	5b	5c	6	7
1a	S	0	0	0	0	0	0	0	0	0	0	0	0
1b	0	0	1	0	0	0	0	0	0	0	0	0	0
1c	0	0	S	1	0	0	0	0	0	0	0	0	0
2	0	0	0	S	1	0	0	0	0	0	0	0	1
3	0	0	0	0	S	1	0	0	1	0	0	0	0
4a	0	0	0	0	0	S	0	0	0	0	0	0	0
4b	0	0	0	0	0	0	0	1	0	0	0	0	0
4c	0	0	0	0	0	0	0	S	0	0	0	1	0
5a	0	0	0	0	0	0	0	0	S	0	0	0	0
5b	0	0	0	0	0	0	0	0	0	0	1	0	0
5c	0	0	0	0	0	0	0	0	0	0	S	1	0
6	0	0	0	1	0	0	0	0	0	0	0	S	0
7	0	0	0	0	0	0	0	0	0	0	0	0	S

Figure 4-4. Connection matrix with creations partitioned.

The matrix we are considering (actually the complete matrix it represents) shows only the connections along paths of length one. That is, $c_{ij}=1$ when there is a flow path of length one from node i to node j. In order to be able to answer the question about where a TN is alive, we must know about all paths of any length. We know from the use of connection matrices in graph theory that the boolean product of the connection matrix with itself will produce a matrix showing paths of length two. Let us consider why this is so. First recall that the boolean product of matrices is

just like the algebraic product except that multiplication and addition are replaced by the <u>and</u> and <u>or</u> operations respectively. Thus each element of the product matrix is produced by <u>or</u>ing together several elements which have been produced by <u>and</u>ing corresponding elements of the original matrices.

The boolean product P of matrices A and B is defined by

A is of order n x k
B is of order k x m
P is of order n x m

$$P = A * B \text{ iff } p_{ij} = \bigvee_{r=1}^{k} (a_{ir} \wedge b_{rj})$$

By this definition we see that $p_{ij}=1$ iff $\exists x \ni a_{ix}=1$ and $b_{xj}=1$. For the product of a connection matrix with itself we have

$$p_{ij} = \bigvee_{r=1}^{n} (c_{ir} \wedge c_{rj})$$

$$= 1 \text{ iff } \exists x \ni c_{ix}=1 \text{ and } c_{xj}=1$$

This is exactly the statement that there is some program point p_x such that there is a flow path p_i, p_x, p_j in the graph of the program; i.e., there is a path of length two.

If we call the original connection matrix C^1 indicating that it reflects paths of length one then $C^1*C^1=C^2$ which reflects paths of length two. Similarly $C^3=C^1*C^2$, $C^4=C^1*C^3$, etc. Forming the element-wise <u>or</u> of two or more of these matrices produces a new matrix whose elements indicate paths of any of the constituent lengths, i.e., C^1 <u>or</u> C^2 indicates paths of length two or less. The boolean sum C^1 <u>or</u> C^2 <u>or</u> . . . <u>or</u> C^∞ is the matrix we are seeking, the one which indicates whether there is a path of any length between any pair of nodes. In practice it is only necessary to accumulate terms of the sum until it converges since in a finite graph there is a finite maximum length path which does not contain cycles. If the graph contains n nodes then the sum will converge in no more than n steps since a path of length n+1 or

greater must contain a cycle. The converged sum is called the *closure* or *transitive closure* of the connection matrix. Clearly we can find the closure of a connection matrix by forming the products and oring them together. This solution is unattractive, however, since the operation of finding the product is itself an n^3 algorithm. Steven Warshall [War62] devised and proved an algorithm which will determine the closure of a connection matrix in time proportional to n^2. The basic approach of the algorithm is to operate on rows of the connection matrix. When processing the ith row, any row which has a 1 in the ith column is ored into the ith row. Warshall shows that when the rows and columns are selected in the proper order the closure is formed in a single pass over the matrix. The algorithm performs nicely on our shorthand matrices when we define the or operation on the shorthand elements as

$$0 \vee X = 0$$
$$1 \vee X = 1$$
$$S \vee S = S$$

where X may take on any of the three values (0, 1, S) and our interpretation of the symbols in the shorthand for the closure is changed to

$$0 \rightarrow m_{ij} = 0 \ \forall i,j$$
$$1 \rightarrow m_{ij} = 1 \ \forall i,j$$
$$S \rightarrow m_{ij} = 1 \ \forall j,i$$

Remembering that the elements of the shorthand matrix correspond to the linear blocks of the original graph, it is easy to understand the definition and the change of notation. In the closure the element representing a linear block (the S elements) will have at least all 1's above the main diagonal because there is a path from any node in the block to any later node in the block. The 1 elements in the connection matrix represented connections between blocks, but clearly if there is a path from one node of one block to a node of a second block, then there must be a

path from every node in the first block to every node in the second. The S entries serve only to remind us of the interior detail of the connections of the nodes. Indeed if there is a path from a linear block back to itself then the entry for the block will change from S to 1 (S v 1 = 1) indicating that there is a path from every node in the block to every other node in the block. Figure 4-5 shows the closure of the matrix from Figure 4-4.

	1a	1b	1c	2	3	4a	4b	4c	5a	5b	5c	6	7
1a	S	0	0	0	0	0	0	0	0	0	0	0	0
1b	0	0	1	1	1	1	0	0	1	0	0	0	1
1c	0	0	S	1	1	1	0	0	1	0	0	0	1
2	0	0	0	S	1	1	0	0	1	0	0	0	1
3	0	0	0	0	S	1	0	0	1	0	0	0	0
4a	0	0	0	0	0	S	0	0	0	0	0	0	0
4b	0	0	0	1	1	1	0	1	1	0	0	1	1
4c	0	0	0	1	1	1	0	S	1	0	0	1	1
5a	0	0	0	0	0	0	0	0	S	0	0	0	0
5b	0	0	0	1	1	1	0	0	1	0	1	1	1
5c	0	0	0	1	1	1	0	0	1	0	S	1	1
6	0	0	0	1	1	1	0	0	1	0	0	S	1
7	0	0	0	0	0	0	0	0	0	0	0	0	S

Figure 4-5. Closure of partitioned matrix.

Let us reflect for a moment on the information contained in the closure of the connection matrix. By zeroing the columns associated with creations, we have assured that no path in the matrix passes through a creation node. The interpretation of the elements of the closure is that the i,j element is 1 *iff* there is a path from node i to node j that does not pass through a creation. If there exists a j such that node j is a use and element i,j of the matrix is a 1, then it follows that the TN is alive at node i. The lifetime of the TN is obtained as a bit vector by oring together the columns of

the closure matrix that correspond to the uses of the TN. Like the other operations, this too can be performed in terms of the shorthand matrix.

Suppose now that our hypothetical TN has uses at nodes 32, 74, and 99. In the matrix of Figure 4-5 these nodes are in linear blocks 3, 5a, and 7. Oring those columns together produces

0 1 1 1 1 0 1 1 S 1 1 1 S

The S entries come from blocks 5a and 7 and indicate that the TN is alive in those blocks before (and including) the last use in each block. The 1 entries indicate that the TN is alive throughout blocks 1b, 1c, 2, 3, 4b, 4c, 5b, 5c, and 6. In particular the TN is not alive in blocks 1a and 4a and in block 5a after the use at node 74.

4.3 Reflection on lifetimes

Let us reflect on the importance of this method. The result is very important. By repeating the process for each TN we produce a precise specification of the lifetime of each TN. This means that we can determine easily and accurately whether two TNs interfere. Two TNs are said to *interfere* or *conflict* with each other if there is any program point at which they are both alive, i.e. if the element-wise and of the lifetime vectors for the two TNs contains any non zero elements. This is exactly the knowledge needed to make efficient use of the registers in the compiled code. We may assign two TNs to the same location only if they do not interfere. Note also that the method is completely independent of any language or target machine. Once we are given the linear blocks and their successors along with the creation and use points the rest is a mechanical process. Warshall's algorithm for finding the closure allows us to transform a matrix into its closure in a single pass over its elements.

At this point let us ask why it is necessary and/or desirable to expend the

computational effort required to generate the connection matrices and their closures. There are much simpler algorithms which will produce good approximations of the lifetimes with much less effort. First, we must remember that our ultimate goal is to produce compilers which generate the best possible code (the metric being determined by the implementor). The simpler algorithms will always lead us to safe decisions; that is, we will never be told that two TNs do not interfere when in fact they do. The problem is that the approximate lifetimes may exclude the best solution from the set of feasible solutions. As long as the extra effort is not unreasonable, our goal requires that we use the exact solutions to such problems. Second, we recognize that the simpler algorithms do yield exact solutions in many cases. To exploit this fact we use the simpler methods when possible by dividing the TNs into two classes: interesting and uninteresting. The uninteresting TNs are those whose lifetimes may be determined exactly by a simple method. Their lifetimes always consist of a single segment of the program graph from the first reference to the last reference. The method described in this chapter is used only for the interesting TNs, those which might have lifetimes composed of disjoint program segments. In general the interesting TNs are user variables and common subexpressions; the uninteresting TNs are compiler generated temporaries. We make the division into the two classes by declaring that any TN which has more than one creation or is referenced in more than one linear block is interesting. One of the two conditions is necessary (but not sufficient) to produce a lifetime of disjoint segments.

4.4 Summary of Lifetimes

In this chapter we have presented a method of determining the lifetime of a TN, that is, the points in the program at which the value contained in the TN must be preserved. The method depends only on collection of data about creation and use points and on knowledge of the flow of control. Because we would like to have a very precise representation of the lifetimes, we want to consider each program node separately. However, even with Warshall's algorithm, finding the closure of the connection matrix is an n^2 process and n grows very quickly. This realization led to the development of a shorthand matrix on which the closure could be performed.

Chapter 5

The Packing Problem

This chapter will describe an algorithm which assigns the TNs to the physical locations available within the target machine. The lifetime information generated by the procedure described in the previous chapter is taken as an Input to the packing algorithm. The costs associated with each TN and the restrictions placed on the assignment of the TN to physical locations are the other inputs. In the ideal case, it is possible to assign each TN to a location which minimizes its cost; in practice, this is frequently not possible. Thus the packing algorithm must handle two slightly different but related problems. The first problem is related to problems known in operations research as "cutting-stock" or "knapsack" problems [Gil66]. The locations to be used for temporary storage represent the stock from which pieces (TNs) are to be cut or the knapsacks into which items (TNs) are to be packed. The measure of space in both cases is in terms of program points. The TNs not only require a given number of points but particular points. In this respect the TN packing problem is more constrained than the cutting-stock or knapsack problems. If a TN is placed into a location, it must be at a fixed position and orientation in the space of program points within that location. In a cutting-stock problem the task is to cut pieces from a piece of stock. The exact position and orientation of the pieces is not specified. The second problem deals with the selection of TNs to be assigned to the preferred locations (usually registers). The problem is to minimize the increase in cost over the cost that would have realized if all TNs could have been packed into their preferred locations.

The algorithm described considers only two classes of storage -- registers and

memory. The algorithm is also based on the implicit assumption that most TNs should be assigned to registers if possible. These assumptions simplify the algorithm considerably and are a reasonably accurate characterization of current computers. Extension to more than two classes of storage will be considered after the algorithm is presented. The algorithm described will produce an optimal solution to the packing problem, i.e. will minimize the increase in cost due to inability to assign all TNs to registers.

5.1 The problem

The problem of assigning the TNs to the set of available registers is the many-few allocation discussed by Day [Day70]. That is, there are a number of TNs which must be assigned to relatively few locations. When the number of TNs is not larger than the number of locations then the solution to the problem is trivial. Added complications are the preferences noted during TN processing and the restrictions on the locations to which certain TNs may be assigned. Let us first consider the many-one allocation problem, that is, reduce the problem to consider only one register.

Assume that the TNs are represented by a sequence T in which the ith element T_i represents the ith TN in some ordering. Define a vector p of which each element p_i is the profit associated with assigning the ith TN to a register. The profit is taken to be the difference between the two cost measures calculated for the TN during temporary name processing as described in Chapter 2. A selection vector x is used to identify which TNs have been selected for assignment to the register (x_i=1 if T_i is assigned to the register, x_i=0 otherwise).

Two TNs are said to interfere if their lifetimes have any points in common. Total interference exists among the TNs in a set N if n_i interferes with n_j for all

$n_i, n_j \in N$, $i \neq j$. Let N_i^* be a subset of N such that there is total interference among the elements of N_i^*. Let N^{**} be a set of subsets giving a complete description of the total interference data; $N^{**} = \{N_1^*, N_2^*, \ldots, N_m^*\}$. This implies that if n_j interferes with n_p then there exists at least one i such that $N_i^* \in N^{**}$ and $\{n_j, n_p\} \subseteq N_i^*$. Let A be an interference matrix (which has dimension (m x k) when there are k TNs) such that $a_{ij} = 1$ if $n_j \in N_i^*$; otherwise $a_{ij} = 0$. The selection vector x is a solution to the assignment problem. The solution is feasible if no two TNs selected by x interfere. This condition is expressed by $Ax \leq 1$, i.e. each element of the product of A and x is ≤ 1 meaning that there is at most one TN occupying the register at any program point.

The optimal solution to the many-one assignment problem is the solution of the integer programming problem:

maximize	$z = px$
subject to	$Ax \leq 1$
where	$x_i \in \{0,1\}$
	$a_{ij} \in \{0,1\}$
	$p_i > 0$

This problem is described by Balinski [Bal65] as a weighted set matching problem. There is no known solution to this class of problems aside from examining all possible selection vectors and evaluating the objective function of each. Fortunately it is not necessary to actually calculate the values for each solution (remember that for n TNs there are 2^n solutions).

The following definitions are due to Day [Day70] and Geoffrion [Geo67]. A *complete solution* is an assignment of a binary value to each element of x. A *partial solution* S_p is an assignment of binary values to some of the elements of x with the other elements remaining free. A *completion* of S_p is an assignment of binary values to each of the free elements of S_p. *Explicit enumeration* is the process of excluding

a complete solution from the set of possible optimal feasible solutions. *Implicit enumeration* is the process of excluding a set of solutions from the set of possible optimal feasible solutions without the explicit enumeration of each element of the set being excluded. Let z' be the value of the objective function (px) for the most profitable feasible solution yet obtained. To *fathom* a partial solution S_p is to determine that either there is no feasible completion of S_p with profit greater than z' or that there is a most profitable feasible completion of S_p with profit $z''>z'$. If a partial solution S_p is fathomed, then the set of completions of S_p is implicitly enumerated. The key to finding an optimal feasible solution is finding an effective fathoming procedure.

Before discussing the solution further, it is interesting to note that the procedure used to find an optimal solution to the many-one problem may be used to find an optimal solution to the many-few problem. An obvious approach to the many-few problem is to treat it as a sequence of many-one problems. With this approach an optimal assignment is found for one of the registers. The remaining TNs are then used as input to a second many-one problem for a second register. The process continues until either the supply of registers or TNs is exhausted. This approach has two disadvantages: it may produce a non optimal solution and, more importantly, it does not allow the preference data to be considered.

A more general approach to the many-few problem is to expand the many-one problem by a factor of m, the number of registers to be considered. The structure of this problem is identical to that of the many-one problem with only the makeup of the matrices changing. The problem is stated as

$$\begin{array}{ll} \text{maximize} & z = p^*x^* \\ \text{subject to} & A^*x^* \le 1 \\ \text{where} & x_i \in \{0,1\} \\ & a_{ij} \in \{0,1\} \\ & p_i > 0 \end{array}$$

Here p^* is a sequence $p_1, p_2, \ldots, p_m$ where each p_i is identical to the original profit vector p of the many-one problem. Similarly x^* is a sequence $x_1, x_2, \ldots, x_m$ where x_i is the selection vector for the ith register. A^* is a matrix composed of submatrices.

A^*_{1j} = I, an identity matrix of order (n x n)

when $i=j+1$, A^*_{ij} = A, the original interference matrix; order (p x n)

otherwise, $A^*_{ij'}$ = 0, $i \neq j$ a zero matrix of order (p x n).

The form of matrix A^* is shown in Figure 5-1. The addition of the identity elements assures that no TN can be assigned to more than one register. In the abstraction of the integer programming problem this constraint is necessary to eliminate solutions which might achieve a high profit by assigning a particularly profitable TN to all of the registers.

5.2 The procedure

The complete enumeration of the solutions to the assignment problem is obtained by a branch and bound procedure [Mit70]. Such methods use a branching procedure to generate an ordered sequence of partial solutions and a set of bounding rules to provide a lower bound on the value of the objective function for each possibly optimal feasible completion of each partial solution. The procedure, shown in Figure 5-2, begins with no TNs assigned to registers. One by one TNs are

$$A^* = \begin{bmatrix} I & I & \cdots & I \\ A & 0 & \cdots & 0 \\ 0 & A & \cdots & 0 \\ \vdots & \vdots & \ddots & \\ 0 & 0 & & A \end{bmatrix}$$

Figure 5-1. The expanded interference matrix.

assigned to a register (maintaining the feasibility of the partial solution) until the resulting partial solution is fathomable. At this point the complete solution with the largest value for the objective function is remembered and all completions of the current partial solution are implicitly enumerated. The last TN assigned in the partial solution is then removed from its register and the procedure is repeated from the fathoming step.† The process terminates when all solutions have been enumerated, i.e. when the initial (empty) partial solution is fathomable.

5.2.1 The fathoming procedure

The fathoming procedure is fairly simple. If the current partial solution is S_p, then let us define S_b to be the completion of S_p which assigns to registers all TNs which are free in S_p. If the value of the objective function associated with S_b is not greater than the largest objective function value yet observed, then there is no completion of S_p which will produce a larger value and S_p is fathomed. Otherwise S_p is not fathomed and the procedure of Figure 5-2 will try to assign the next TN to a

† If there is a possibility that assigning the removed TN to a different register may produce a better feasible completion then this may be tried.

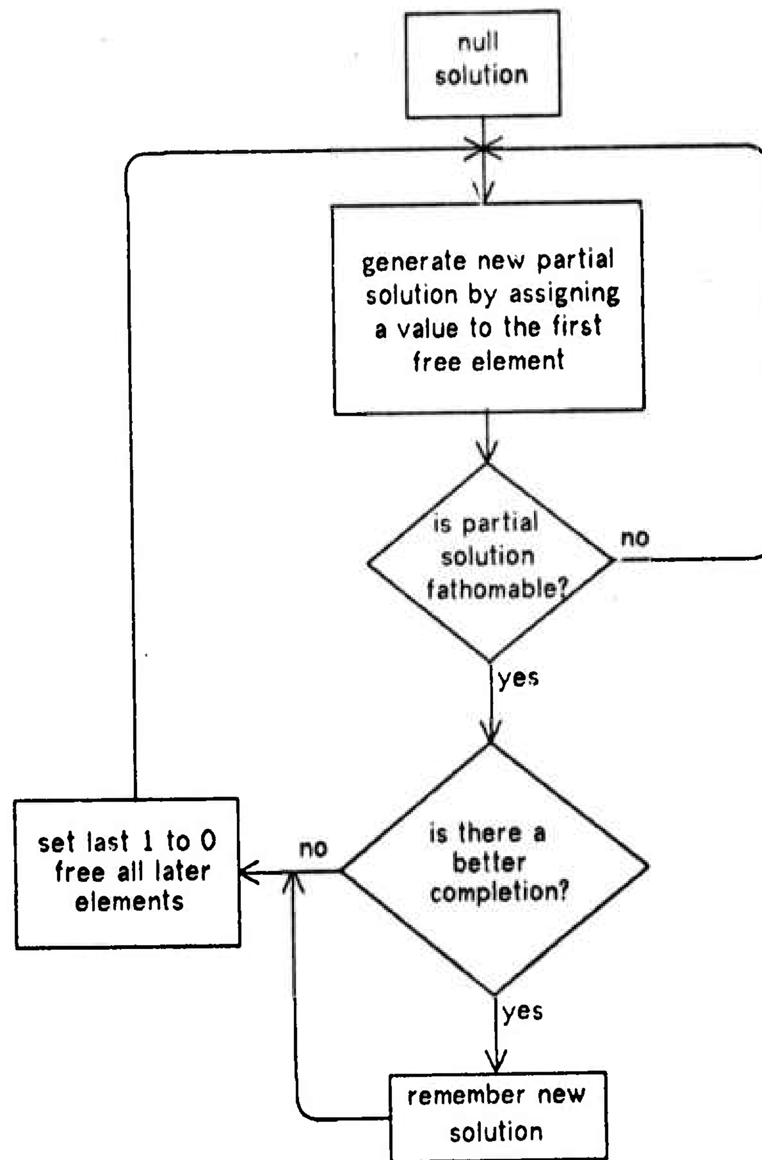


Figure 5-2. The Branch and Bound Algorithm.

register. Regardless of whether the attempt to assign the next TN to a register is successful, the result is a new partial solution derived from the old solution by assigning a value to one of the free elements of the old selection vector. This new partial solution is then presented to the fathoming procedure.

5.2.2 Backing up

When a partial solution is fathomed, the next step is to "undo" the most recent assignment of a TN to a register reflected in S_p, i.e. the last 1 is changed to 0. All elements of S_p following the changed element are free. An attempt is now made to fathom the new partial solution. Following this procedure, the partial solutions fathomed have more and more free elements. When a partial solution containing k free elements is fathomed, 2^k solutions are implicitly enumerated.

5.2.3 Assigning another TN to a register

The process of attempting to assign a new TN to a register consists of trying several steps to find a location to which the TN can be assigned without interfering with any TN already assigned. In the following discussion of the steps, a register is said to be *open* if there are any TNs assigned to it and *closed* otherwise.

1. If the TN has a preference for assignment then try to assign the TN to one of the prefered locations. (This step is expanded in Section 5.2.4.)
2. Try to find an open register to which the TN can be assigned.
3. If there are any closed registers, open one and assign the TN to it.
4. If steps 1 through 3 fail to assign TN to a location then the attempt is unsuccessful. The new partial solution will have the corresponding element of the selection vector equal to 0.

5.2.4 Honoring preferences

As discussed in Chapter 2, some TNs are "preferenced" to other TNs. The intent of a preferencing operation is to indicate that there is some additional benefit to be gained by assigning the two TNs to the same location, be it register or memory. In terms of the generated code, failing to honor a preference means that at some point a value will have to be moved from one TN to the other.

Suppose the TN being considered by the assignment algorithm is T and that T has preferences $P_1, \ldots, P_n$.[†] The algorithm for honoring preferences iterates through the P_i's. If any P_i has been assigned to a register and T can be assigned to that register, then the assignment is made and the preference is honored. It may happen that none of the preferences has been assigned to a register, or that T is unable to fit into any of the registers to which P_i's have been assigned. In these cases the preference usually cannot be honored. However, it may also happen that some preference P_j has been assigned to a memory location. If the profit associated with T is not greater than the cost of a move from memory to a register, then assigning T to the preferred memory location will result in a better overall solution. If none of the preferences have been assigned to locations or T cannot fit into any of the preferred locations then the preference cannot be honored.

Note that at this point the only TNs which have been assigned to memory locations are those which are restricted to be in memory. No feasible solutions which might have higher profit are lost when a preference for a memory location is honored as long as the profit loss due to such an assignment is not greater than the cost of a move from memory to a register.

---

† In a typical Bliss-11 program, the value of n is small (2 or less). Constructs can be generated, however, which result in large numbers of preferences.

5.2.5 An intuitive view of the procedure

The above procedure can be viewed as a packer operating on a vector of TNs. The TNs in the vector are arranged in order of decreasing importance. As we have described importance this means that the TNs we would most like to have assigned to registers are considered first. In this way we are more likely to be able to assign the important TNs to registers because the registers are "less crowded" when the important TNs are considered. The packer considers each TN in the vector in turn, assigning as many as possible to registers. When the end of the vector is reached, an initial complete solution has been generated. The packer has assigned as many TNs as possible to registers, given the order in which they were considered. The only fact that keeps us from stating that this is an optimal solution is the possibility that removing some set of TNs from the registers might allow the assigning of some other set of TNs with a larger combined importance.

After producing the initial complete solution, the packer works backward in the vector reconsidering the assignments. At each step the packer asks whether removing the current TN from its register might lead to a more profitable complete solution. Sometimes the answer is that there can be no more profitable completion and the packer continues backward in the vector. At other times there is a more profitable completion and the packer moves forward in the vector again to investigate the feasibility of the more profitable completion. When there is no more profitable completion, all of the solutions in all completions are implicitly enumerated. The algorithm terminates when there are no solutions (i.e. completions of the empty partial solution) which can produce a larger profit than the most profitable solution already obtained.

5.3 The practicality of obtaining an optimal solution

The algorithm described above is essentially the same as the OPTSOL algorithm described by Day for solving the many-one assignment problem. Day proves that the algorithm terminates only after enumerating all solutions and yields an optimal solution. Mitten [Mit70] also gives a proof of the general branch-and-bound procedure. Day reports experimental results of using OPTSOL to solve 400 randomly generated assignment problems. The results show that the time required to solve a problem is a function of both the size of the problem (number of TNs) and the density of nonzero elements in the interference matrix C. Execution time was directly related to problem size and inversely related to C matrix density. The disheartening result is that the average execution over all densities of C are large and grow very rapidly. For one register the following average execution times (on an IBM 360/65) were observed

number TNs	time (sec)
24	<1
32	2
48	6
64	46

We see that for moderate sized problems† the execution times quickly become unreasonable. Day indicates that for more than one register the times grow much faster. Appealing to the argument of Section 4.3, we cannot expect that the branch-and-bound algorithm to guarantee an optimal solution in a reasonable amount of time. What we can do is measure the rate of progress of the algorithm toward termination or the fraction of the total number of solutions enumerated. Care in choosing the

† In real programs observed by the Bliss-11 compiler, the number of TNs per routine averages less than 20 because routines tend to be small. However routines having 50 or more TNs are fairly common.

order in which TNs are added to partial solutions will increase the probability that an optimal or near optimal solution is found early in the search. The branch-and-bound procedure is more effective if a feasible, near optimal solution is taken instead of the empty solution as the starting point. Day gives a procedure for finding such a solution quickly and notes that the profit of the initial solution is almost always greater than 90% of the profit of an optimal feasible solution. Day suggests that the initial solution is a close enough approximation to an optimal solution to make it usable and can be computed quickly enough to make it practical. It should be noted here that Day's results are for randomly generated profits and interference matrices. The initial solution described in Chapter 6 for real programs almost always produces an optimal solution.

5.4 Formulation of the more general problem

Section 5.1 describes the formulation of the general integer programming problem for assigning TNs to one of several registers by replicating the profit vector and interference matrix for each register. In this section we show how this same reasoning can be used to express a much more general problem. Thus far we have assumed that there are only two classes of storage available for TNs, a limited number of identical registers and an unlimited number of memory locations.

Suppose that the machine for which we wish to compile code has more than two classes of storage. For example the Univac 1108 has 47 registers available to user programs. Of this number 11 are index registers, 12 are arithmetic accumulators, 4 are both index registers and accumulators, 17 are fast memory locations, 2 have special functions in certain instructions and 1 is modified by real time clock interrupts. In such a machine, there are several levels of trade-offs when trying to decide which of the many classes of storage should be used for any particular TN.

Consider a machine with n storage locations divided into m classes. As before we will consider main memory to be an additional class of unlimited size and we will implicitly assume that main memory is the least desirable class of storage for all TNs. For each TN compute a set $r = \{r_1, \ldots, r_m\}$ of m profit values such that r_i is the profit realized by assigning the TN to a member of class i instead of to memory. This is identical to the profit vector calculated in Section 5.1 if the only two classes of storage are class i and memory. Thus we consider all profits to be relative to the worst case solution. The problem is now

maximize	$z = p^*x^*$
subject to	$A^*x^* \leq 1$
where	$x_i \in \{0,1\}$
	$a_{ij} \in \{0,1\}$
	$p_i > 0$

as before, but the subvectors of p^* are no longer identical. Recall that in the formulation of the many-few register problem of Section 5.1, the subvectors of p^* were replications of the single profit vector. In the more general problem we choose for the ith subvector the vector of profits associated with the ith storage location. In terms of the abstract linear problem the solution to this problem is no more difficult that the solution of the register-memory problem of the same size. In practice, however, as the number of classes of storage increases, the number and complexity of the restrictions placed on TNs tends to increase.

5.5 Summary of packing

The packing problem, as described in this chapter, is the problem of binding a large number of TNs to a relatively small number of locations. While we are concerned with the solution of the general packing problem, we have concentrated on the situations in which there are only two classes of storage, one generally desirable

and one generally undesirable. We have shown that the problem can be formulated as an integer programming problem and that the form of the problem is essentially the same regardless of whether we consider one register, several registers, or several classes of storage each containing several distinct locations. Section 5.2 described a branch-and-bound procedure which yields an optimal solution to the problem. We have also noted that while the number of possible solutions to be examined by the procedure is exponential in the number of TNs, the algorithm terminates much sooner if we can determine a feasible, near optimal solution from which to begin the search of the solution space.

Chapter 6

A Test Case

This chapter will describe the implementation of the TNBIND algorithms in the compiler for Bliss-11, a dialect of Bliss [Wul71] for the PDP-11. The essential features of both the language and the machine are described below. The standard version of the Bliss-11 compiler uses an initial version of TNBIND and thus provides a good comparison for some of the more advanced ideas. The fact that the language is Bliss does not have a significant effect on the implementation; the language dependencies for any Algol-like language would be nearly identical. The target machine has significant effect on the TNBIND algorithms, but mainly influences the cost measures and the kinds of targeting and preferencing done.

The decision to use Bliss-11 as a test has both advantages and disadvantages. On the positive side, the existing compiler is highly modular and it was easy to "unplug" the existing register allocation phase and "plug in" the new TNBIND with relatively few changes to the rest of the compiler. This would not have been the case with a compiler which started with a more traditional register allocation method. It is also intended that the new TNBIND will become a part of the standard compiler thus providing real world benefits. On the negative side, in order to produce a working compiler, TNBIND had to incorporate all of the functions provided by the original version. These included assigning TNs to those nodes which needed them, generating linear block information, and assigning labels for the final code stream. The fact that the TNBIND algorithm has been implemented for only one language-machine pair raises the question of whether the algorithms would perform as well on other languages or, more importantly, other machines. TNBIND does not include some

of the more traditional register optimizations because these were not critical to the goals of Bliss-11. Chapter 7 will discuss how these operations can be easily incorporated into the TNBIND philosophy.

6.1 About Bliss

Bliss falls into the class of Algol-like languages. It has the features normally associated with such languages: block structure, recursive routines, loops, conditionals, local variables. Bliss was designed as a systems implementation language and therefore presumes little or no runtime support. The emphasis is on flexibility and runtime efficiency. Bliss provides the programmer with the ability to perform arbitrary address calculations at runtime and to address parts of words when appropriate. More complete descriptions of the language may be found in [Wul71], [DEC74], and [Wul75].

The programming style of Bliss tends to be considerable different from Algol or other languages. Because there in little overhead in routine calls, programmers are encouraged to write small routines. In addition the control structures observed in Bliss programs tend to be well structured because there is no goto statement in Bliss. The control structure of Bliss makes the linear block analysis quite easy. The small size of the routines usually means that the numbers of linear blocks and TNs are small. These two factors combine to make the first pass implementation of TNBIND acceptable in terms of computing time required. As routines get larger and have more linear blocks and TNs, the time required for the TNBIND algorithms increases dramatically. On the one hand this tells us that we will need better algorithms in order to deal effectively with large routines. On the other hand we can argue that smaller routines and simpler control structures are the "wave of the future," and

therefore it is not unreasonable to design algorithms which perform better for small routines than for large routines.

6.2 About the PDP-11

The PDP-11, manufactured by Digital Equipment Corporation [DEC71], falls into the class of "mini-computers." It has a 16-bit word length which is addressed in units of 8-bit bytes; thus a 16-bit word can address 2^{16} bytes or 2^{15} words. Instructions come in 0- 1- and 2-operand formats and may be 1, 2, or 3 words in length. Each operand uses six bits of the first instruction word and may require one additional word to hold a 16-bit address or index quantity. The PDP-11 has eight registers; six of these are general registers, one is used by both hardware and software as a stack pointer, and one is the program counter. Instructions whose operands are in registers are both faster and smaller. The fact that any instruction can have any of its operands in registers or memory locations means that almost all TNs are of the kind that can be assigned to either a register or memory location. This makes the TN packing a simple cost minimizing procedure with few restrictions on the locations to which any TN may be assigned.

Bliss makes use of locations in the stack for local storage when there are no registers available. Because the PDP-11 hardware uses the same stack, e.g. to save the processor state during an interrupt sequence, all stack locations must be allocated before they are used. A stack location is allocated when it is at or below the location pointed to by the stack pointer register. Stack locations are allocated explicitly by adding a constant to the stack pointer register or implicitly by pushing parameters.

6.3 Bliss-11 implementation of TNBIND

In the Bliss-11 compiler TNBIND is presented with a tree representation of a routine to be compiled. The tree is traversed in a recursive depth-first tree walk. There is a separate routine for each type of tree node. Several of these routines were shown in Chapter 3. A switching routine, TNP, performs some common functions (like updating the lon value) and calls the appropriate routine for each node. language. A complete listing of the TNP routines for Bliss-11 appears in appendix A. The interesting parts of TNBIND are those that are affected by decisions made by the language designer/implementer. Below we discuss two areas in which non obvious processing is done in the Bliss-11 version of TNBIND.

6.3.1 Subroutine parameters

In most machines there are several ways to call subroutines and pass arguments. In Bliss-11 the programmer may specify whether arguments to a subroutine are to be passed in registers or on the stack. The caller is responsible for removing parameters from the stack after a call, while the callee is responsible for restoring the contents of any registers used by its code (except for the register in which the value of the subroutine is returned). In processing a subroutine call node, TNBIND performs the following actions for each argument.

1. Call TNP to do TN processing for the argument expression.
2. Generate a new TN and assign it to the location in which the argument will be passed (either a specific register or a stack location).
3. Preference the TN of the argument to the TN of the location.

TNBIND thus simulates the machine code implementation of the subroutine call which might be expressed as

```
for each argument do
        begin
        compute value;
        store value in proper place
        end;
call subroutine
```

Lifetimes are generated at this time for the TNs which identify the argument locations. These lifetimes extend from the lon value at which the argument will be stored to the lon value of the actual call. This prevents the argument locations from being used by any computations necessary to produce succeeding argument values. These TNs are part of the initial state input to the packing phase.

After the subroutine call, the locations used for any arguments passed on the stack remain allocated. Rather than deallocate these locations immediately, the caller keeps track of the number of such locations on the stack at all times and adjusts the stack pointer only when two or more flow paths join and the stack depth is not the same on all of the paths. This has two effects: the number of instructions necessary to adjust the stack pointer after subroutine calls is reduced, and the locations on the stack may be used for temporary storage by later calculations or local variables. These locations are called dynamic temporaries because they are dynamically allocated and deallocated at run time. Since the allocation of these temporarles is a byproduct of a routine call, there are no additional instructions needed to allocate these locations. Thus we have a group of dynamically allocated locations whose allocation and deallocation overhead costs are zero.

Temporary locations, i.e. registers and stack locations, may be either "open" or "closed". In general a location is "open" when it is available for use in a particular routine. When refering to registers, "open" means that some TN has been assigned to

the register and consequently the contents of the register will have to be saved and restored. When refering to stack locations, "open" means that the location is allocated, i.e., below the current value of the stack pointer. Registers are either open or closed throughout a routine, but stack locations may be open and closed at several points within the routine. Each location is represented by a list of the TNs assigned to the location. A location is closed when its list is empty; dummy TNs are used to mark the locations as open. When a dynamic temporary is closed, a dummy TN with a lifetime corresponding to the closed period is added to the appropriate list, thus making it unavailable for use by other TNs during the closed period.

6.3.2 The store operator

SimpleStore (Section 3.1.3) is very important in Bliss-11. Because the PDP-11 can perform general memory-to-memory operations, a great many store operations are simple. A store operation may be simple even when the left hand side is an expression or an own or external variable. The important feature of the SimpleStore predicate is that the machine specific details of which operations are simple is encoded entirely within the predicate. It is not necessary, for example, to modify the routine processing the "+" operator to take account of the fact that many "+" operations are simple. The routine processing the store operator evaluates the SimpleStore predicate for each store operation and performs the binding when possible. When the store is not simple, the right hand side is preferenced to the left hand side just in case the lifetimes of the TN's involved are such that the binding is possible after all.

6.3.3 Cost computations

The Bliss-11 version of TNBIND counts memory references as the cost of accessing a TN. The counts include both the reference to the actual value and the reference to memory which is necessary to pick up an address or index value in a multi-word instruction. This measure tends to minimize both execution time and code size by putting heavily used TNs in registers, which results in faster and shorter instructions. The size of the object code is used as a static indicator of the effectiveness of the optimizations.

A TN assigned to a register requires no memory references for a direct access and one memory reference for an indirect access. A TN assigned to a memory (stack) location has a cost of two for a direct access (one to get the stack offset and one to access the stack location) and three for an indirect access. Other forms of access are similarly assigned costs. Indexing, for example, has a cost of two if the TN is in a register because the hardware will handle the addition. If indexing is required for a TN in memory, the hardware function will have to be simulated by software which has a cost of between five and nine depending on whether there is a free register at the moment the indexing is required.†

During the TN processing tree walk, a minimum and maximum cost is computed for each TN. The cost of a TN is the sum of the costs computed for that TN by the

---

† Because TNBIND cannot always assign all TNs to registers, the code generation phase of compilation is sometimes presented with these nasty problems. In the case of indexing the code generator will load the value of the TN into a free register if there is one and then proceed normally. If no register is available, then code must be generated to explicitly simulate indexing. This predicament is a result of the decision that a TN is assigned to a single location throughout its life. There may be a register available at the time the indexing is needed, but unless the register is available throughout the lifetime of the TN, it will not be used by TNBIND. It is comforting to note that in real programs it is seldom, if ever, necessary to index by a memory location.

nodes that reference it. The cost computed at a node is the product of the cost of a single reference and the number of references required to execute the function represented by the node.

6.3.4 Lifetimes

During the TN processing, creation and use points (in terms of lon values) are noted for each TN through calls on NoteCreation and NoteUse as described in Chapter 4. NoteCreation and NoteUse also collect some additional information to simplify the lifetime computations.

The first (smallest lon value) and last access are noted for each TN, and TNs which have more than one creation or are accessed in more than one linear block are flagged as possibly having lifetimes composed of disjoint program segments. As described in Chapter 4, this partitions the TNs into interesting and uninteresting subsets. The lifetime of an uninteresting TN is simply all of the lon values between the first and last accesses. Most compiler generated temporaries, notably those used for expression evaluation, fall into the uninteresting subset because, in general, they have one creation and one or more uses within a single linear block. The interesting TNs are those whose lifetimes may consist of noncontiguous sequences of lon values. These TNs are subjected to the complete lifetime computation described in Chapter 4.

There are two types of TNs which are not treated by the lifetime phase because their lifetimes are determined when they are generated. These are the TNs which hold the arguments to a routine call and the dummy TNs used to close the dynamic temporaries.

6.3.5 Ranking

The ranking phase computes the difference between the maximum and minimum cost measures for each of the TNs which may be assigned either to a register or to a memory location. This value is the profit associated with assigning the TN to a register. The output of the ranking phase is a list of all of these TNs sorted by profit. Those TNs which must be assigned to a register but which do not require a specific register are added to the list after being assigned arbitrarily large profit values.

6.3.6 Packing

The packing phase is responsible for the actual binding of TNs to locations. The objective is to maximize the sum of the profits of the TNs which are assigned to registers. A secondary goal in cases where all TNs are assigned to registers is to minimize the number of registers, since the contents of any register used will have to be saved and restored.

The packing phase first assigns all TNs which require specific registers. These TNs represent arguments passed to other routines in specific registers, the values returned by other routines, or a request from the programmer to put some variable in a specific register. The packing phase then deals with the sorted list of TNs prepared by the ranking phase. The algorithm used is a slightly modified implementation of the algorithm described in Chapter 5.

The TNs are considered to be elements of a vector sorted by profitability with the most profitable TN first. Corresponding to this vector is a binary vector indicating (with 1's) which TNs are assigned to registers. The binary number formed

by the ones and zeros of this vector can be taken as a numbering of the solutions of the problem of which TNs are bound to registers. Thus when there are n TNs to be considered, there are 2^n solutions. Ignoring for a moment the TNs which are restricted to specific registers, there are really $(m+1)^n$ solutions where m is the number of registers available. Clearly not all of these solutions are feasible. The goal of the packing algorithm is to select the most profitable feasible solution.

There are several approaches to selecting a feasible solution. Below are listed three possible approaches. The three are given in order of increasing complexity and computing time required. This is also the order of increasing probability of producing an optimal solution since each method considers a larger portion of the solution space than the preceding method.

1. Beginning with the most important TN, try to bind each TN in turn to some register. When a TN is bound to a register (using a spatial analogy we say it "fits") it remains there and the corresponding element of the binary vector is set to one. At the end of the TN vector the solution represented by the binary vector, i.e. the one we have built during the algorithm, is used.

2. Same as 1 except that when a TN will not fit into any register we try to move the TNs with which it conflicts into other registers. This operation, called *reshuffling*, is aimed at correcting decisions made earlier in the algorithm. Whenever we find a register into which a TN will fit, we bind it there without regard for whether it might also fit into some other register.

3. Same as 1 initially. When the end of the vector of TNs is reached, we back up in the vector and set the corresponding binary vector element to zero. We remember the profit value and the bindings of the most profitable solution yet obtained. If it might be possible to obtain a more profitable solution by binding the TN just removed to a different register or by not binding it to a register at all, then trying this alternate solution by moving forward in the sequence again as described in 1.

The *reshuffling* operation described is method 2 is interesting. The term is derived from the effect on the TN assignments. No TNs are removed from registers or added to registers. The TNs assignments to specific registers are merely permuted or shuffled. The reshuffling in method 2 is called "bottom level" because it is invoked at the bottom of the packing search. In method 3 the reshuffling is implicit in the search algorithm, but is performed at the top level, i.e. a TN assignment is only changed to another register when all when the search is backing up.

Method 3 is equivalent to the branch and bound procedure described in Chapter 5. It will always produce an optimal solution, but may explore a large fraction of the $(m+1)^n$ solutions in the process. Success in using procedure 3 depends on being able to decide quickly whether a new path is worth exploring. It is easy to tell whether the best completion of a partial solution will produce a solution with a larger profit than any solution yet obtained. The key is to able to determine the profit of the best feasible completion without generating the completion. For the purposes of measurement and comparison, all three methods were implemented. The results are reported in Section 6.4.

When a particular TN, *t*, is to be packed by the packing procedure, the available locations are considered in a particular order. The order of consideration is

1. Register preference. If *t* has been preferenced to some *t'* and *t'* has already been bound to a register, then the register to which *t'* is bound is considered. If reshuffling is being done at the bottom level then it is invoked if necessary.

2. Stack preference. If *t* is preferenced to some *t'* which has been bound to a stack location, then the stack location is considered only if the profit associated with the TN is no larger than the cost of loading or storing a register (see Section 5.2.4).

3. Open register. The open registers are considered in an arbitrary order. When applicable, bottom level reshuffling is invoked.

4. Closed register. A closed register is selected and opened. Ideally this step is only invoked when the profit associated with t is greater than the cost of saving and restoring the register. In practice it is almost always correct to open another register because the register, once opened, can be used for TNs to be packed later.

After the packing algorithm terminates there may be TNs which have not been bound. These TNs are bound to stack locations. Dynamic temporaries are used when possible. As a last resort the TN is bound to a static temporary, a stack location which is explicitly allocated/deallocated at routine entry/exit.

6.3.7 A new kind of restriction

Because there are usually enough memory locations there is no incentive to expend large amounts of computing time to produce an optimal packing of the remaining TNs into the smallest number of memory locations.†

Because of the nature of the variable length instructions of the PDP-11, it is sometimes desirable to load the address of a frequently used variable or a frequently called routine into a register and then reference the variable or routine indirectly through the register. The semantic analysis phase can generate TNs to hold the addresses when the number of accesses passes some threshold. A problem arises, however, if the solution produced by the packing algorithm does not bind one of these TNs to a register. If a TN used to hold a fixed address is bound to a memory location, the resulting code is worse (by any metric) than if the TN had not been

† Clearly we can produce programs which require any arbitrarily large number of memory locations to hold TNs. The concern here is to do very well under ordinary circumstances and not worry too much about "pathological" programs.

generated. In order to accommodate this type of TN, a new type of restriction was devised. Just as some TNs are marked "must be a register" or "must be a specific register," these TNs generated to hold addresses are marked "register or forget it" meaning that if binding to a register is not possible then the TN should not be bound to a stack location but rather should be bound to the variable in question. This type of TN opens the way for including many of the classic register optimizations into the TNBIND philosophy. Because these optimizations are not included in the Bliss-11 compiler, they will not be discussed here; Chapter 7 will describe these optimizations and how they would be included in TNBIND. The possibility of including these optimizations increases the credibility of TNBIND as a method for register allocation for other languages and particularly for other machines.

6.4 Measurements of TNBIND

This section will present a comparison of several variations on the TNBIND philosophy. The intent is to measure TNBIND, in terms of quality of output code and compilation speed, as compared with more traditional methods of register allocation. We also want to explore the space of possible TNBIND algorithms not included in the Bliss-11 compiler. It is not reasonable to build several register allocators merely in order to make these comparisons. Instead we simulate the actions that would be taken by several different register allocators by "turning off" various pieces of the TNBIND optimizations.

The compilers we will be considering are:

1. PB - The "production version" of the Bliss-11 compiler. This compiler is in use daily for many programming projects. It incorporates an initial version of TNBIND with simplified lifetimes and has been

carefully tuned to produce high quality code over a period of several years.

2. TB - The compiler with all TNBIND features enabled. This includes the matrix closure to produce exact lifetime representations and the full branch and bound search for an optimal solution to the packing problem. (As a practical matter, the search was limited to a small fraction of the solution space by limiting the number of solutions that were examined. Because of the nature of the search, finding an optimal solution for very large routines could take many hours of computing time.)

3. TBR - The same as TB except that simplified reshuffling is done at the bottom of the search rather than the top. That is, when a TN will not fit into any register or will not fit into a preferred register, a search is made to see if moving any one TN would allow the fit. This action brings the reshuffling closer to problem it attempts to solve at the cost of not enumerating all possible solutions.

4. TBS - Instead of the complete enumeration in the packing phase, TBS uses a simple, one pass algorithm in which the packing of each TN is attempted only once along with the bottom level reshuffling of TBR. The TNs are considered in order of decreasing cost.

5. DUMB1 - All of the "clever" parts of TNBIND are disabled. In particular
   a. All lifetimes are taken to be continuous from first access to last access.
   b. TNs are considered for packing in order of first access. This simulates a register allocator making a pass over the object code assigning registers as they are needed.
   c. All "register or forget it" TNs are forgotten.
   d. Preferencing is turned off
   e. There is no attempt at reshuffling.
   f. No user declared local variables are assigned to registers.

6. DUMB2 - This is DUMB1 with most of the other optimizing features of the compiler turned off. This gives some idea of the code that might be compiled by an unsophisticated compiler. The rationalization for this is that Bliss-11 has many optimization features that tend to be partially redundant, i.e., when one optimization is disabled, another

optimization may partially compensate to prevent an accurate assessment of the effect of the disabled optimization. Optimizations are not generally additive; the whole is less than the sum of the parts. The additional features disabled in DUMB2 are

a. Common subexpression recognition.
b. Peephole optimization. The last phase of compilation which performs transformations on the object code to produce equivalent, but more efficient, sequences of instructions.

There are some optimizations in the compiler that cannot be easily disabled for the purpose of these measurements. The most notable of these is the extensive special case analysis performed in the code generation phase.

6.4.1 The programs

Five programs with a total of 128 routines were selected for the measurements. With one exception these are "real" programs as opposed to programs written explicitly to test the compiler. The programs were written by five different authors and thus represent somewhat different programming styles. The authors have spent varying amounts of time tuning their programs to the optimizations performed by the standard compiler. The five programs are described briefly below.

CODGEN. The code generation phase of a FORTRAN compiler. Although this is a "real" program in the sense described above, it has been used as a benchmark of the progress of Bliss-11 over the last three years.

SPOOK. A daemon process which periodically examines the state of an operating system. This program was under development when the measurements were taken.

EVLSTK. The stack evaluation module of an interactive interpreter.

TXQMS. A simple queue management system. This is one of a series of

programs designed to test the original Bliss-11 compiler.

KA612. One module of a powerful interactive debugging system designed for use with Bliss-11 programs. This program contained the largest routine in any of the test programs (>500 words of code).

6.4.2 The numbers

The cost measures used for Bliss-11 are designed to minimize code size for two reasons: (1) in a computer with a small addressing space such as the PDP-11 minimal code size is important, and (2) in the PDP-11 minimum code size almost always means minimum run time. In the discussions that follow, the terms "less code" and "better code" are taken to be synonymous.

Figure 6-1 shows the code sizes produced by PB and the three TBx compilers. Each column shows the absolute number of words of code produced and the ratio of this number to the code size produced by PB. PB is taken to be a state-of-the-art compiler against which the other compilers are measured. The DUMBx compilers are not shown in this figure because they are not really comparable to the optimizing versions.

Figure 6-2 shows the processing times in seconds for the TNBIND portions of the four optimizing compilers as well as the ratios to the PB times. Again, the DUMBx compilers are not included because their times are not representative of the times that would be observed from an unsophisticated compiler. This is because the DUMBx compilers contain all of the generality of the more complex compilers with only the benefits removed.

In figure 6-3 we see the ratios of the overall compile times to the PB compile

	PB	TB	TBR	TBS
CODGEN	2234(100)	2235(100)	2226(100)	2224(100)
SPOOK	627(100)	654(104)	649(104)	649(104)
EVLSTK	1245(100)	1190(96)	1181(95)	1188(95)
TXQMS	302(100)	299(99)	301(100)	302(100)
KA612	2089(100)	2141(102)	2138(102)	2136(102)
total	6497(100)	6519(100)	6495(100)	6499(100)

Figure 6-1. Code sizes (percent of PB size).

	PB	TB	TBR	TBS
CODGEN	14.6(100)	69.7(477)	35.2(241)	32.9(225)
SPOOK	3.5(100)	12.4(354)	8.3(237)	7.7(220)
EVLSTK	6.5(100)	75.6(1163)	27.7(426)	26.8(412)
TXQMS	1.5(100)	9.7(647)	3.3(220)	3.1(207)
KA612	17.2(100)	339.5(1974)	310.6(1806)	247.7(1440)

Figure 6-2. Seconds of TNBIND run time (percent of PB time).

	PB	TB	TBR	TBS
CODGEN	100	154	126	125
SPOOK	100	131	128	128
EVLSTK	100	274	173	154
TXQMS	100	176	132	137
KA612	100	314	309	280

Figure 6-3. Ratio of total compile times (percent).

time. This is a measure of how the more complex TNBIND operations affect the total compile time.

Figure 6-4 shows a breakdown of the differences in routine size by overall size of the routine. The numbers represent the difference produced by the "best" of the TBx compilers using PB as a base. By "best" we mean that the size difference reported is the difference between PB and the smallest amount of code produced by and on the TBx compilers. All 128 routines are included.

Size from PB	number observed	size difference / number of routines										
1-10	30	no differences										
11-25	35	-5	-1	0	+3	+4						
		1	4	28	1	1						
26-50	29	-3	-2	-1	0	+1	+3	+4	+6	+7		
		1	1	18	4	1	1	2	1			
51-100	15	-6	-5	-4	-3	-2	0	+6	+15			
		1	1	1	2	1	7	1	1			
101-200	14	-36	-11	-6	-5	-4	-3	-2	-1	0	+1	+6
		1	1	2	1	1	1	1	1	3	1	1
>200	5	-23	0	+1	+6	+20	+24					
		1	0	1	1	1	1					

Figure 6-4. Differences in code size by routine.

	code size		run time	
	DUMB1	DUMB2	DUMB1	DUMB2
CODGEN	2679(120)	2888(129)	15.6(107)	14.6(100)
SPOOK	760(121)	894(143)	4.5(129)	4.2(120)
EVLSTK	1651(133)	1924(155)	9.0(138)	9.1(140)
TXQMS	414(137)	480(159)	2.2(147)	2.2(147)
KA612	2889(129)	3642(151)	19.1(111)	19.0(110)

Figure 6-5. Code sizes and TNBIND runtimes for DUMBx compilers.

Figure 6-5 shows the code sizes and compile times produced by the DUMBx compilers along with the percentage of the PB sizes and times. These numbers do not represent actual compilers and therefore should not be taken to be the results one might obtain from compilers producing code with the same degree of sophistication. In particular the run times of the DUMBx compilers are slightly larger than the run times for PB. The run times for the DUMBx compilers are useful only for

comparisons with the TBx times to see the costs the particular operations omitted in the DUMBx compilers.

6.4.3 Discussion

We see from Figure 6-1 that there is relatively little difference in the code sizes produced by the PB, TB, TBR, and TBS compilers. This may be attributed to the fact that the TBx compilers actually add relatively little to the TNBIND processing already performed by PB. The point of the comparison is that the TNBIND in PB has grown piecemeal over the last three years while the TNBIND in the TBx compilers is the product of about a month's work. Working from the general model it was possible in a short time to produce a TNBIND that compares favorably with a finely tuned production compiler.

Although we might expect that the code quality (size) would get progressively better as we moved from TBS through TBR to TB, we see from Figure 6-1 that this is not always true. One reason for this anomaly is the interaction of optimizations in the compilers. TNBIND can make optimization decisions only on the basis of the cost measures supplied. While these decisions are made on a global scale as far as the register allocation is concerned (see discussion of global vs. local allocations in Section 2.1), the decisions are local in the sense that they are made without interaction with other phases of compilation. Even optimal solutions to the allocation problem based on the cost measures defined within TNBIND may not produce the best code when considered in the context of the other compiler optimizations.

It is interesting to note that even in the worst case the code produced by DUMB2 is less than twice the size of the code produced by the other compilers. One

would expect that a truly unsophisticated compiler would be worse than the optimizing compilers by a factor of three of four at least. The fact that the DUMB compilers do as well as they do in the size comparison is a tribute to the optimization and extensive special case analysis done in the other phases of the compiler.

The data in Figure 6-2 is more disturbing. The TBx compilers require much more time than the PB compiler to perform the same task and produce approximately the same results. It is expected that the TBx compilers would require more time since they are doing more computation, but the differences are large both relatively and absolutely. Consider the case of KA612 which uses about 17 seconds for TNBIND in PB and more than five minutes for TNBIND in TB. Closer examination of the timings shows that the bulk of the difference is accounted for by the lifetime computations. PB uses a simplified lifetime characterization which is computed on-the-fly (in linear time) during its TNP phase. There is no increase in the cost of lifetime determination due to routine size. TB uses the full matrix closure algorithm of Chapter 4 for all interesting TNs. This algorithm experiences an n^2 increase in running time as the size (number of linear blocks) of a routine increases. Nearly half of the TNBIND time used in KA612 is accounted for in the lifetime determination for a single routine. This routine, the largest in the measured programs, has 160 linear blocks and 70 interesting TNs. The resulting code for the routine is 575 words from PB and 595 words from TB. The effect of the lifetime computation on run time can be seen in the differences between the TBS values in Figure 6-2 and the values for DUMB1 from Figure 6-5. It must be noted at this point that the DUMBx compilers are not "lightening fast" for two basic reasons. (1) The TNBIND phase accounts for only about 10% of the total compile time and therefore even reducing the TNBIND time to zero would not produce drastic changes overall. (2) The DUMBx compilers have to

pay the price of many of the more sophisticated algorithms, e.g. lifetimes and packing, without reaping the benefits. The DUMBx compilers were generated mainly to show code size and not compile times.

The other major difference in runtime between PB and TB is the complete enumeration packing algorithm. The packing done by PB is essentially the same as that done by TBS. The only basic difference is that PB uses a more elaborate reshuffling scheme. PB will reshuffle several TNs and may recur several times in trying to make room for a new TN. TBS will try to reassign only one TN and will not recur if the reassignment is not possible. The difference between the TB and TBR columns in Figure 6-2 is the cost of the complete enumeration.† In only a few cases did TBR examine more than the initial complete solution. The difference between the TBR and TBS columns is the overhead of maintaining the mechanism for the enumeration algorithm. TBS does no backtracking or state saving; its solution is the initial solution from which TB and TBR begin to search.

In Figure 6-3 we see that the compile time for the TBx compilers increased over the PB time by just over a factor of three in the worst case and less than a factor of two in most cases. This performance is quite respectable, considering that this was the first implementation of a general model of register allocation.

Figure 6-4 shows how the TBx compilers compared with PB. An example of how to read the table is probably the best explanation of what the data represents. The routines are separated into groups based on the size of the code produced by

† As noted above, the enumeration was actually limited to a fraction of the solution to avoid the exponential computing time that would be required by large problems. In compiling programs to collect the test data, the search was limited to examining 50n solutions where n is the number of TNs to be packed. This limit was reached by 17 of the 128 routines.

the PB compiler. Lets consider the routines which were compiled into 11 to 25 words of code by PB. This is the second group of numbers in the table. The second column tells us that there 35 routines in this size range. In the third column we see that the TBx compilers produced code which was five words smaller than the PB code one time, one word smaller four times, the same size 28 times, three and four words larger one time each. The intent of this table is to report all of the size differences without making separate entries for each of the 128 routines. The routines are separated into groups by size to give some idea of the percentage differences.

An interesting statistic in gauging the effectiveness of the various parts of the TNBIND procedure is the relative score of the four optimizing compilers on the 128 test routines. Each compiler is given a point when it is the simplest compiler to produce the smallest amount of code for a routine. No points are given if all compilers produce the same code. Simplicity is measured by how many of the sophisticated TNBIND algorithms are included; thus PB is the simplest followed by TBS, TBR and TB. If for some routine the code sizes produced by the compilers are

PB	20 words
TB	18
TBR	18
TBS	19

then TBR would get one point. Computing this measure over the 128 test routines gives the following result:

PB	19
TB	6
TBR	5
TBS	12

This result is interesting because it points out the importance of precise lifetimes as opposed to the sophisticated packing algorithm. There were 23 routines for which

better code was produced by one of the TBx compilers. Twelve times out of that 23 the best code was produced by the addition of only the more precise lifetime characterization. The only essential difference between PB and TBS is the lifetime characterization. In the test cases, TBS was actually actually a restricted version of TB. It is reasonable to conclude that adding the more precise lifetimes to the PB compiler would result in a compiler better overall than any of the four tested. In reaching this conclusion we assert that the cases in which PB produced better code than any of the TBx compilers are due to a combination of tuning and chance. The tuning argument states that is is reasonable that a finely tuned simple algorithm will sometimes produce better results than a more sophisticated general algorithm. The question of chance brings us back to the issue of compensating optimizations. The peephole optimization phase of Bliss-11 can have drastic effects on the amount of code produced by combining unrelated sequences of instructions which happen to be identical. When a TN will fit into more than one register, we arbitrarily choose the first one we find. It might be that some other choice would allow or disallow combining sequences of instructions. Examination of the code produced by the various compilers showed that the largest differences (both positive and negative) were mainly due to the action of the peephole optimization rather than the TNBIND actions. Since the packing algorithms of PB and the TBx compilers are different, it is not necessary that these arbitrary choices would be the same in any two compilers.

6.4.4 Evaluation

One of the goals of this thesis is to explore extensions to register allocation algorithms. In pursuing that goal we have looked specifically at the TNBIND model and the lifetime and packing phases of that model. The test results have shown that

it is possible to produce according to the general TNBIND model a compiler which compares favorably in code quality with a state-of-the-art optimizing compiler.

The TBx compilers cannot match the PB compiler in execution speed, but the increase is not (with one exception) an unreasonable tradeoff for the efficiency in production of the compiler. A slowdown by a factor of 2-10 is tolerable for a first try at using the general model. Although there are no statistics generally available, one would expect that similar degradations in performance were experienced by the first automatically produced parsers.

Another point to consider in evaluating the results is that while the lifetime computation is an n^2 algorithm in the size of the routine being compiler, the trend in programming is toward smaller routines. Indeed a compiler which produces very good code but requires long compilation for large complex routines can be a factor in encouraging programmers to write smaller, more easily understood routines. In addition, studies such as that by Knuth [Knu71] have shown that programs in general are very simple implying that the complex cases which require large amounts of computing should be relatively rare.

Chapter 7

Conclusion

In this chapter we consider again the overall view of what is usually called register allocation. The view is on a higher level than that on which register allocation is usually discussed, and does not explicitly include some of the optimizations frequently performed by optimizing compilers. We will show that the TNBIND philosophy can easily be extended to incorporate other optimizations which can be expressed at the source level of the program.

This chapter also summarizes the contributions of the thesis and considers directions for future research in this area.

7.1 The TNBIND Model - Review

The TNBIND model of temporary storage allocation is more closely related to the source level of the program than to the object level. As a result, the model is easily adaptable to new languages or machines. The basic points of the model are

1. Information local to some part of a program is computed, stored, and used over contexts larger than a single statement or expression.

2. Within some piece of program which can reasonably be considered as a unit, a compiler has the responsibility to allocate physical resources to hold the computed information.

3. There is no inherent difference between the intermediate results produced by a compiler in the process of evaluating arithmetic expressions and the intermediate results produced by a programmer in the process of performing some complex algorithm.

Chapter 2 expanded these ideas to a model of implementation. That model described the assignment of unique names (TNs) to the items of information to be

stored, the collection of data to determine lifetimes and preferences, and finally the association of each TN with some physical location in the target machine.

The advantage of the model is that it deals with representations of the source program which still contain much information about the semantics and control flow of the program. This high level analysis makes global allocations easier. Recall from chapter 1 that a local allocation occurs entirely within linear blocks while global allocations consider larger contexts. With a local allocation, interblock transfer of information must be via memory locations which can be fixed throughout the program segment being considered. Since registers are a scarce resource (relative to main memory), traditional compilation techniques have reserved the use of registers to those data items which are used only in local contexts.

Traditional register allocation methods deal with actual machine instructions for which the allocator must assign registers. Without knowledge of what registers will be available, the code generator must compile code for the worst case. In the TNBIND model the register allocator, presented with the constraints, chooses a set of bindings of TNs to locations which will minimize the cost (maximize the profit) of the resulting object program. The code generator is then able to produce code which is specifically tailored to the TN bindings.

The TNBIND model was based on the compiler structure of Bliss-11. A question naturally arises as to what effect the adoption of this compiler structure has on the TNBIND concept. The lifetime, ranking, and packing subphases of TNBIND are independent of any change in the compiler structure. Those subphases require only that flow information be available and that certain decisions, e.g. evaluation order, have been made. If we can determine the connection matrix and the creation/use

points of the TNs, we can perform the lifetime, ranking, and packing operations at any point in the compilation. In that sense a good deal of TNBIND is independent of the compiler structure.

On the other hand, if we had not started with the Bliss-11 decomposition we might not have developed the model as we did. One can argue, we believe, that the Bliss-11 structure is the "right" structure for a highly optimizing compiler.

7.2 More traditional optimizations

There is no reason why any optimization of register utilization which can be expressed at the source level of a program cannot be incorporated into the TNBIND model. One such optimization which has been discussed frequently is the use of a register to hold the value of a frequently accessed variable during the execution of a loop. This is done by inserting instructions to load the value into a register before the loop and to store the value after the loop. If it can be determined that the value of the variable cannot be changed within the loop then the store instruction can be eliminated. This is the optimization described by Lowery and Medlock [Low69]. To incorporate this optimization into an implementation of TNBIND we simply assign a TN to hold the value of the variable during the loop and add the necessary load and store as assignment nodes in the parse tree. The TN is is marked "register or forget it" so that failure to assign a register to the TN will result in the TN being bound to the location of the variable. Presumably the code generator in a compiler which includes these optimizations is clever enough to ignore requests to assign the value of a variable to itself.

The addition of such optimizations at the level of TNBIND rather than at the source level means that the TNs generated for the optimization are considered as

equal competitors for the registers available to the allocator. We can then assert with reasonable confidence that choosing registers for some such TNs and not for others is the correct decision.†

7.3 Contributions of the thesis

This thesis has presented a high level model of a global register allocation process. The model in fact deals with the more general problem of assigning locations to hold computed values during the execution of a compiled program. The model is not restricted to the situation which allows only two classes of locations to be used for such temporary storage. The model defines a cost which is associated with each entity to be stored and each class of storage location. The concept of targeting values to particular locations is not new, but its application in terms of the model is different. Preferencing is a new concept and is very useful in providing direction to what would otherwise be an arbitrary decision.

The fact that the registers are bound before code generation means that meaningful special case analysis can be profitably applied during code generation. The TNBIND model also provides a starting point from which to generate this phase of a compiler automatically. The basic knowledge needed by TNBIND is a list of what operands of what operations must or must not be in a particular class of storage. The cost measures can be as simple or as complex as the compiler designer wishes.

It may be observed that some of the ideas reported here are also presented in [Wul75]. In the thesis we have taken the basic compiler structure of Bliss-11 as given, but we have not taken the details of the implementation. We have developed

† Correct in the sense that the decisions minimize the cost of the object program as we have defined the cost measures.

the general model of register allocation and explored some new dimensions of the solution space. The emphasis of the thesis has been on providing a framework within which specific solutions to particular problems can be explored. The intent is that the register allocator of a compiler should be produced (either manually, mechanically, or more likely a combination of the two) according to the TNBIND model and then tuned to the pecularities of the language and machine. It is expected that this tuning will usually take the form of adjusting the cost measures and introducing preferencing. Preferencing is a particularly powerful technique, because it allows us to indicate the code we want to generate in the normal case without precluding what we must do in the worst case.

7.4 Future research

7.4.1 Better algorithms

As shown by the timing data in chapter 6, the lifetime determination can become very costly for large routines. On the other hand, adding the more precise lifetime characterization to the Bliss-11 compiler produced better code in about 10% of the routines examined in chapter 6. We need to develop better algorithms for dealing with very large problems. Equally as important is the development of fallback positions of lifetime characterizations which are accurate but possibly less precise and less costly to determine.

7.4.2 Automatic TNBIND generation

One of the long range goals for research in this area is to enable the automatic production of optimizing compilers. Some parts of TNBIND are independent of both

the language and the machine and thus could be copied directly to new environments (modulo the language in which the compiler is implemented). In this category are the lifetime determination and the basic cost collecting functions. The actual cost values depend on the target machine and the designer's criteria for optimization.

The TNP phase requires knowledge about the language to be compiled and some way of determining what code sequences might be used to implement each construct in the language. Some work in the latter area has been done by Newcomer [New75], but further work is necessary to iron out the differences in the interfaces expected by the various phases of the resulting compiler.

As suggested in chapter 1, the step from the TNBIND model to a program which produces a running TNBIND is "merely a matter of implementation." At the current time, however, a good deal of judgement is required to make the transition.

Appendix A

The TNP Routines

Listed below are the actual unadulterated TNP routines from the Bliss-11 compiler. The interesting point is not exactly what each routine does, but that the operations for most node types are highly similar. The routines become complex only we wish to exploit special features of the hardware (e.g. in the store operator) or to make clever implementation decisions (e.g. routine calls). The macros TLRDEF and NDRDEF are used to consolidate the common features of most of the TNP routines.

```
macro TLRDEF(NAME,BODY)=
        routine ID(TL)NAME(NODE,TARGET)=
            ( map GTVEC NODE, TNWORD TARGET; BODY; .NODE[REGF])$;


macro NDRDEF(NAME,BODY)=
        routine ID(ND)NAME(NODE,TARGET)=
            ( map GTVEC NODE, TNWORD TARGET; BODY; novalue)$;


TLRDEF(NULL,(ACCESS(.MYTN); NOTEDTD));


NDRDEF(B,(
        local TOP1,TOP2,TAR,NTAR;
        TOP1←TOP2←0;
        if .NODE[TPATH]
            then !reverse order
                begin
                TAR←.NODE[OPR2]; NTAR←.NODE[OPR1];
                TOP2←.MYTN; TOP1←0;
                end
            else !normal order
                begin
                TAR←.NODE[OPR1]; NTAR←.NODE[OPR2];
                TOP1←.MYTN; TOP2←0;
                end;
        TNP(.NODE[OPR1],.TOP1);
        TNP(.NODE[OPR2],.TOP2);
        PREFMOVE(.TAR,.MYTN);
        OPERATE(.NTAR,.MYTN);
        ));
```

```
NORDEF(U,(
        TNP(.NODE[OPR1],.MYTN);
        PREFMOVE(.NODE[OPR1],.MYTN);
        ));


NORDEF(UX,(
        TNP(.NODE[OPR1],0);
        PREFMOVE(.NODE[OPR1],.MYTN);
        ));


NORDEF(PTR,(
        TNP(.NODE[OPR1],0);
        PREFMOVE(.NODE[OPR1],.MYTN);
        if .NODE[SIZEF] eql 8 then
            if .NODE[POSF] eql 0 then
                ACCESS(.NODE[OPR1]);
        ));


NORDEF(DOT,(
        TNP(.NODE[OPR1],.MYTN);
        if (.NODE[MODE] eql INDEXED) or (.NODE[MODE] eql INDEXED+DEFERRED) then
            MAKESAME(.MYTN,.NODE[OPR1]);
        ));


routine ISBIT(LEX)=
        begin
        map LEXEME LEX;
        bind GTVEC NODE=LEX;
        if .LEX[LTYPF] eql GTTYP
            then if .NODE[NODEX] eql SBITOP
                then (TNNEEDED)[1] + 1
        end;


NORDEF(REL,(
        local LOP,ROP;
        bind BITLEFT =ISBIT(.NODE[OPR1]),
             BITRIGHT=ISBIT(.NODE[OPR2]);
        LOP←ROP←0;
        if BITLEFT then LOP←.MYTN else
        if BITRIGHT then ROP←.MYTN;
        TNP(.NODE[OPR1],.LOP);
        TNP(.NODE[OPR2],.ROP);
        if BITLEFT then (PREFMOVE(.NODE[OPR1],.MYTN); COMPARE(.MYTN,.NODE[OPR2])) else
        if BITRIGHT then (PREFMOVE(.NODE[OPR2],.MYTN); COMPARE(.MYTN,.NODE[OPR1])) else
            (COMPARE(.NODE[OPR1],.NODE[OPR2]); MOVE(0,.MYTN));
        ));
```

```
NORDEF(FPAR,(
        local GTVEC UOP:PARM;
        bind LEXEME PARMLEX=PARM;
        TNP(PARM←.NODE[OPR1],if .NODE[CSCOMPL] leq MAGIC2 then .MYTN else 0);
        PREFMOVE(.NODE[OPR1],.MYTN);
        ));


NORDEF(LOADNODE,(
        local UOP,LEXEME OP1;
        TNP(OP1←.NODE[OPR1],0);
        if .OP1[LTYPF] neq BNDVAR
            then PREFMOVE(.OP1,.MYTN)
            else MOVE(.OP1,.MYTN);
        ));


NORDEF(ADDSUB,(
        local TOP1,TOP2,TAR,NTAR;
        TOP1←TOP2←0;
        if .NODE[TPATH]
            then !reverse order
                begin
                TAR←.NODE[OPR2]; NTAR←.NODE[OPR1];
                TOP2←.MYTN; TOP1←0;
                end
            else !normal order
                begin
                TAR←.NODE[OPR1]; NTAR←.NODE[OPR2];
                TOP1←.MYTN; TOP2←0;
                end;
        TNP(.NODE[OPR1],.TOP1);
        TNP(.NODE[OPR2],.TOP2);
        if (NOT .NODE[RCMTF]) and (NOT .NODE[RCMOF])
            then MAKESAME(.MYTN,.TAR)
            else PREFMOVE(.TAR,.MYTN);
        OPERATE(.NTAR,.MYTN);
        ));


routine BINDSTORE(LOP,ROP)=
        begin map GTVEC LOP:ROP;
        bind LEXEME LLOP=LOP;
        ROP←BASETN(.ROP);
        if .ROP leq 7 then return NOVALUE;
        if .ROP[REQD] neq 0 then return NOVALUE;
        if .LLOP[LTYPF] eql LITTYP
          then begin
                ROP[REQD]←MEMREQDB;
                ROP[TNLITBIT]←1;
                ROP[TNLITLEX]←.LOP;
                ROP[BNDTYP]←0;
                return NOVALUE
                end;
        ROP[REQD]←MEMREQDB;
        ROP[REGF]←.LOP;
        NOTEUSE(.LOP,.ROP[LONFU]);
```

```
        NOVALUE
        end;


routine FINDLEX(LEX,TREE)=
        begin
        map LEXEME LEX, GTVEC TREE;
        bind LEXEME TLEX=TREE;
        macro FINDNCSE(NODE)=
                (map GTVEC NODE;
                 if .NODE[REGF] leq 7 then return 0 else
                 if .GT[.NODE[REGF],BNDTYP] neq BNDNCSE then return 0 else
                 NODE←.GT[.NODE[REGF],OFFSETF];
                 if .NODE[TYPEF] eql GRAPHT then return 0;
                 NODE<LTYPF>←BNOVAR)$;


        if .LEX[LTYPF] eql GTTYP then FINDNCSE(LEX);
        if .TLEX[LEXPART] eql .LEX[LEXPART] then return 1;
        if .TLEX[LTYPF] eql GTTYP then
          if .TREE[NODEX] eql SYNNULL then (FINDNCSE(TREE); return FINDLEX(.LEX,.TREE))
                                   else
            incr I from 0 to .TREE[NODESIZEF]-1 do
                if FINDLEX(.LEX,.TREE[OPERAND(.I)]) then return 1;
        0
        end;


routine FINDLEFT(LN,RN)=
        begin local X,Y;
        map GTVEC LN:RN; bind LEXEME LRN=RN;
        if .LRN[LTYPF] neq GTTYP then return 0;
        if .RN[NODEX] eql SDOTOP then return (.RN[OPR1] eql .LN);
        if NOT (ONEOF(.RN[NODEX],BIT5(SADDOP,SMINOP,SANDOP,SOROP,SSWABOP))
                or ONEOF(.RN[NODEX]-2,BIT6(SSHIFTOP-2,SROTOP-2,
                                   SMAXOP-2,SMINNOP-2,SEQVOP-2,SXOROP-2)))
          then return 0;
        if .RN[TPATH] then (X←.RN[OPR2];Y←.RN[OPR1])
                    else (Y←.RN[OPR2];X←.RN[OPR1]);
        if .RN[NODESIZEF] eql 2 then
          if FINDLEX(.LN,.Y) then return 0;
        if (Y←FINDLEFT(.LN,.X)) neq 0 then .Y+1 else 0
        end;
```

```
routine ISNEGNOT(LN,RN)=
        begin
        map GTVEC LN:RN;
        local GTVEC LRN:LLRN;
        bind LEXEME LNLEX=LN:RNLEX=RN:LRNLEX=LRN:LLRNLEX=LLRN;
        if .RNLEX[LTYPF] neq GTTYP then return 0;
        if .RN[NODEX] neq SNEGOP then
            if .RN[NODEX] neq SNOTOP then return 0;
        LRN←.RN[OPR1];
        if .LRNLEX[LTYPF] neq GTTYP then return 0;
        if .LRN[NODEX] neq SDOTOP then return 0;
        LLRN←.LRN[OPR1];
        if EQLPOSSIZE(.LN,.LLRN) then return 1;
        0
        end;


routine SIMPLESTORE(LN,RN)=
        begin
        !
        ! A "SIMPLE" STORE IS, BY DEFINITION, ONE WHICH DOES NOT
        ! NEED A SPECIAL TEMPORARY FOR THE RNS.
        !
        ! VALUE RETURNED:
        !       -1 :: WE HAVE A STORE OF THE FORM
        !                 (EXPR1) ← .(EXPR1) OP (EXPR2),
        !              OR (EXPR1) ← NOT .(EXPR1)
        !              OR (EXPR1) ← - .(EXPR2);
        !                 THE 'RCMTF' BIT OF THE 'OP' (OR 'NOT' OR '-') NODE
        !                 SHOULD BE TURNED OFF.
        !        1 :: WE HAVE SOME OTHER KIND OF SIMPLE STORE, E.G.
        !                 VAR1 ← .VAR2 + 3;
        !                 THE 'RCMTF' BIT OF THE 'OP' NODE SHOULD BE LEFT AS IS.
        !        0 :: THE STORE WE ARE DEALING WITH IS NOT SIMPLE.
        !
        macro
            ADDORSUB=ONEOF(.RN[NODEX],BIT2(SADDOP,SMINOP))$,
            ANDORIOR=ONEOF(.RN[NODEX],BIT2(SANDOP,SOROP))$,
        SPECIALCASES=
            NOT ONEOF(.RN[NODEX],(BIT3(SPLUSOP,SXOROP,SEQVOP) or
                                  BMSKX(SGTROP,6) or
                                  BMSKX(SGTRUOP,6)))$;
        map GTVEC LN:RN;
        local GTVEC LRN:LLRN, RRN;
        bind LEXEME LNLEX=LN:RNLEX=RN:LRNLEX=LRN:LLRNLEX=LLRN;
        bind SIMPLEVAL=3;
        routine SIMPLOP(NODE)= (map GTVEC NODE;
                        if .NODE[NODEX] leq MAXOPERATOR then 1 else
                        if .NODE[NODEX] eql SYNNULL then 1 else
                        if .NODE[NODEX] eql SSTOROP then
                            SIMPLOP(if .NODE[TPATH] then .NODE[OPR1] else .NODE[OPR2]));


        routine ISPSOK(LN,RN)=
            begin
            map GTVEC LN:RN;
            bind LEXEME LNLEX=LN:RNLEX=RN;
            local SSP;
```

```
        macro ISBISORBIC=
            (if ANDORIOR then
               (local LEXEME RRN:LRN;
                if .RN[TPATH]
                  then (LRN←.RN[OPR2]; RRN←.RN[OPR1])
                  else (LRN←.RN[OPR1]; RRN←.RN[OPR2]);
                (1 and (.RRN[KNOTF] eqv (.RN[NODEX] eql SANDOP)))
                + .LRN[KNOTF]*2  ))$;
        macro ISINCORDEC=
            (if ADDORSUB then
               if abs(EXTEND(.RN[OFFSETF])) eql 1 then
                 (.RN[RCAF] or .RN[RCSF]))$;
        macro ISCOMORNEG=
            (if .RN[NODEX] eql SPLUSOP then
               (.RN[RCCF] or .RN[RCNTF]))$;


        SSP←.LNLEX[SSPF];
        if .SSP leq PF016
            then TRUE
        else if .SSP leq PF08
            then (if ISINCORDEC then TRUE
                  else if ISCOMORNEG then TRUE
                  else ISBISORBIC)
        else (ISBISORBIC eql 1)
        end;


    if .RNLEX[LTYPF] neq GTTYP then return 0;
    if .LNLEX[LTYPF] eql GTTYP then
        begin macro PROCEED=exitblock$;
        if .LN[NODEX] eql SYNPOI then
            if ISPSOK(.LN,.RN) then PROCEED;
        if NOT SIMPLOP(.LN) then return 0
        end;
    if .LNLEX[LTYPF] eql BNDVAR then
        begin local X;
        if .LNLEX[LEXPART] eql .PCREG then return 0;
            ! LATER THIS WILL BE EXTENDED TO ALL
            ! VOLATILE LOCATIONS.
        if NOT ISPSOK(.LN,.RN) then return 0;
        if (X←FINDLEFT(.LN,.RN)) neq 0
            then if .X leq (SIMPLEVAL+1)/(1+(.LN[MODE] neq GENREG)) then return -1
        end;
    if .RN[NODEX] eql SPLUSOP then
        return
            if .RN[OCCF] eql 1
                then if .RN[CSCOMPL] eql 0 then 1 else
                     if ISNEGNOT(.LN,.RN[OPR1]) then -1;
    if .RN[NODEX] eql SFSTORE then return 1;
    if .RN[NODEX] leq MAXOPERATOR then
        if .RN[NODEX] eql SDOTOP then return 0 else
        if .RN[NODEX] eql SSWABOP then return 1 else
            if .RN[NODESIZEF] eql 2 then
                begin
                macro PROCEED=exitblock$;
                if .RN[TPATH]
                    then (LRN←.RN[OPR2]; RRN←.RN[OPR1])
```

```
                    else (LRN←.RN[OPR1]; RRN←.RN[OPR2]);
                if .RN[REGF] eql .LRN[REGF] then
                    if .RN[CSCOMPL] gtr SIMPLEVAL then PROCEED;
                if FINDLEX(.LN,.LRN) then
                    if SPECIALCASES then PROCEED else return 0;
                if .LNLEX[SSPF] gtr PF016 then PROCEED;
                if .LRNLEX[LTYPF] eql GTTYP then
                  if .LRN[NODEX] eql SFSTORE then return 0;
                return 1 - FINDLEX(.LN,.RRN)
                end;
      if begin
         macro TRUNC(X)=(X-MAXOPERATOR)$;;
         ONEOF(TRUNC(.RN[NODEX]),BIT4(TRUNC(SYNIF),TRUNC(SYNCASE),
                                      TRUNC(SYNSEL),TRUNC(SYNLABEL)))
         end
        then return 1
        else if .RN[NODEX] leq MAXOPERATOR then
          begin
          LRN←if .RN[TPATH] then .RN[OPR2] else .RN[OPR1];
          if .LRNLEX[LTYPF] neq GTTYP then return 0;
          if .LRN[NODEX] eql SDOTOP then
              begin
              LLRN←.LRN[OPR1];
              if .LLRN eql .LN then return -(SPECIALCASES)
                else if EQLPOSSIZE(.LLRN,.LN)
                        then if SPECIALCASES
                                then (LRN[CODED]←1;
                                      LRN[MODE]←0;
                                      LRN[REGF]←.RN[REGF];
                                      return -1);
              end;
          end;
    0
    end;


routine TRYSIMPLESTORE(NODE,LN,RN)=
    begin
    map GTVEC NODE:LN:RN;
    bind LEXEME LNLEX=LN:RNLEX=RN;
    if .NOSIMPLESTORE then return 0;
    while 1 do
        begin
        macro DOBINO=(
                    case (SIMPLESTORE(.LN,.RN)+1) of
                      set
                          %-1%  (BINDSTORE(.LN,.RN[REGF]); RN[RCMTF]←false);
                          % 0%  0;
                          % 1%  if not .NOSIMPLESTORE then BINDSTORE(.LN,.RN[REGF])
                      tes
                  )$;
        if .RNLEX[LTYPF] neq GTTYP then return NOVALUE;
        if .RN[NODEX] leq MAXOPERATOR then return DOBINO;
        select .RN[NODEX] of
          nset
            SYNIF:
                begin
                if DOBINO then
```

```
            (TRYSIMPLESTORE(.NODE,.LN,.RN[OPR3]);
             TRYSIMPLESTORE(.NODE,.LN,.RN[OPR4]) );
            return NOVALUE
            end;


        SYNCASE:
            begin
            if OOBIND then
              incr I from 2 to .RN[NODESIZEF]-2 do
                TRYSIMPLESTORE(.NODE,.LN,.RN[OPERAND(.I)]);
            return NOVALUE
            end;


        SYNSEL:
            begin
            if OOBIND then
              incr I from 2 to .RN[NODESIZEF]-3 by 2 do
                TRYSIMPLESTORE(.NODE,.LN,.RN[OPERAND(.I)]);
            return NOVALUE
            end;


        SYNCOMP:
            CONTINUE RN←.RN[OPERAND(.RN[NODESIZEF]-1)];


        SYNLABEL:
            (OOBIND; return NOVALUE);


        SFSTORE:
            OOBIND;


        always:
            return NOVALUE
      tesn;
    end;
NOVALUE
end;
```

```
NORDEF(STORE,(
        local LEXEME LOP:ROP, GTVEC T, T1,T2;
        T1←T2←0;
        if .NODE[TPATH]
            then (LOP←.NODE[OPR2]; ROP←.NODE[OPR1]; T1←.MYTN)
            else (LOP←.NODE[OPR1]; ROP←.NODE[OPR2]; T2←.MYTN);
        TNP(.NODE[OPR1],.T1);
        TNP(.NODE[OPR2],.T2);
        if (T←.MYTN) neq 0 then
            begin
            if .T[REQD] eql MEMREQDB then
                if .T[REGF] eql .ROP[ADDRF] then
                    T←0;
            end;
        PREFMOVE(.ROP,.MYTN);


        if .T eql 0
            then
                begin
                TRYSIMPLESTORE(.NODE,.LOP,.ROP);
                PREFMOVE(.ROP,.LOP);
                end
            else PREFMOVE(.MYTN,.LOP);


        ));


macro INITPARMDESC=(LNKGD←.LNAME[LNKGDESCF]; PARMNO←0)$,
          NEXTPARMDESC=
            begin
            if (PARMNO←.PARMNO+1) gtr .LNKGD[LNKGSIZEF]
                then PT←STACKPARM
                else
                    (PT←.LNKGD[PARMTYPE(.PARMNO)];
                     PL←.LNKGD[PARMLOC(.PARMNO)])
            end$;


routine SPANPARMS=
        begin
        local GTVEC T, ITEM L;
        decr I from .CALLSTK[CURD] to 0 do
            FORALLTN(T,CALLSTK[LSELEM(.I)],
                L←.T[TNLIFELIST];
                if .L[LIFESTOP] lss .LON then L[LIFESTOP]←.LON
                );
        NOVALUE
        end;
```

```
macro
        INITCALL=INITSTK(CALLSTK)$,
        PUSHCALL=PUSHSTK(CALLSTK)$,
        NOTECALL=ADDTOTOS(CALLSTK,TNREP(.SUBNODE[REGF]))$,
        POPCALL=POPSTK(CALLSTK)$,
        RELEASECALL=RELEASESPACE(GT,.CALLSTK,STKSIZE)$;


TLRDEF(CALL,(
        external LIFEBLK;
        bind SWITCHREGISTER= 177570;
        bind STVEC LNAME=NODE[OPR1];
        local GTVEC SUBNODE:TN:LNKGD, DLON,N,PARMND,PT,PL,FSP;


        TNP(.NODE[OPR2],0);


        INITPARMDESC;


        FSP←1;
        if .NODE[NODESIZEF] gtr 2 then
                begin
                PUSHCALL;
                incr I from 2 to .NODE[NODESIZEF]-1 do
                    begin
                    NEXTPARMDESC;
                    SUBNODE←.NODE[OPERAND(.I)];
                    DLON←.LON+1;
                    TNP(.SUBNODE,0);
                    TN←.SUBNODE[REGF];
                    TN[TNLIFELIST]←LIFEBLK(.LON,.LON);
                    NOTECALL;
                    SPANPARMS();
                    case .PT of
                        set
                        .


                        %0. stack parm %
                                begin
                                if .FSP then
                                    begin local D;
                                    FSP←0; D←.DTOSTK[LDTD];
                                    N←.DTEMPS[CURD];
                                    SAVDTD;
                                    if ((.DTEMPS[CURD] neq .D)
                                        or (.NODE[NODESIZEF] eql 3)
                                        or (.NODE[NODESIZEF] eql 5)) then
                                            if TRYSPDYTEMP(.TN,.N) then
                                                exitcase (TN[REQD]←MEMREQDB)
                                    end;
                                if not TRYSPDYTEMP(.TN,N←.N+1)
                                    then OPENDYTEMP(.TN,.DLON,0);
                                DTDSTK[LDTD]←.N;
                                TN[REQD]←MEMREQDB
                                end;
```

```
                    %1: specific register %
                          begin
                          local LEXEME SUBSUB,TTN;
                          SUBSUB←.SUBNODE[OPR1];
                          if .SUBSUB[LTYPF] eql GTTYP then
                              begin
                              TTN←.GT[.SUBSUB,REGF];
                              if .TTN geq 8 then
                                  TN[TNPERMIT]←.TTN;
                              end;
                          TNSRREQD(.TN,.PL);
                          end;


                    %2: (literal) memory %
                          begin
                          TN[TNLITBIT]←TRUE;
                          TN[TNLITLEX]←LITLEXEME(.PL);
                          TN[REQD]←MEMREQDB
                          end;


                    %3: (named) memory %
                          begin
                          TN[REGF]←.PL;
                          TN[REQD]←MEMREQDB
                          end


                    tes;
                if not .FSP then
                    KILLPDTEMPS(.SUBNODE,
                        (if (.DTEMPS[CURD] eql .N) or (.I eql .NODE[NODESIZEF]-1)
                            then .N else .N+1))
                end;
            POPCALL
            end;
    NOTEDTD;
    if not .FSP then POPDTD;
    if (.DTEMPS[CURD] geq (STKSIZE-.MAXPARMS)) or .CUTBACKSTACK
        then KILLDYTEMPS(.NODE);


    if .LNAME[LNKGTF] neq INTRRPTLNKGT then
        begin
        if .MYTN lss 8 then MYTN←GETTN();
        TNSRREQD(.MYTN,VREGNUM);
        end;
    if (TN←.MYTN) neq 0 then
        if .NODE[XOPR2] eql .LXHALT
            then
                begin
                TN[REQD]←MEMREQDB;
                TN[TNLITBIT]←TRUE;
                TN[REGF]←SWITCHREGISTER;
                end
            else
```

```
                TNSRREQD(.TN,VREGNUM);


        NOTEUSE(.NODE[OPR2],.LON);
        NEXTLON;
        NOTECREATION(.TN,.LON);


        UPDATELON;
        SETBOUND;
        ));


routine LOADR0(NODE)=
        begin
        map GTVEC NODE;
        local GTVEC TX;
        if (TX←.NODE[REGF]) eql 0 then return 0;
        TX[BNDTYP]←BNDREG;
        TX[REQD]←SRREQDB;
        TX[REGF]←VREGNUM;
        TX[BNDLSTHDR]←REGS[VREGNUM];
        if ISCLOSED(.REGS[VREGNUM])
            then OPENLIST(REGS[VREGNUM]<0,0>);
        .TX
        end;


TLRDEF(routine,(
        external LBPATCH,LBRETLIST;
        local STVEC RNAME:LNAME:LDESC, GTVEC TN:TL;
        LBRETLIST←0;
        RNAME←.NODE[OPR2];
        RNAME[RETLAB]←.NODE;
        LDESC←.GT[LNAME←.RNAME[LNKGNMF],LNKGDESCF];
        if not .ANYENAB then
            begin
            decr I from 5 to 0 do NULLLST(REGS[.I]);
            incr I from 1 to .LDESC[LNKGSIZEF] do
                if .LDESC[PARMTYPE(.I)] eql REGPARM
                    then OPENLIST(REGS[.LDESC[PARMLOC(.I)]]);
            end;
        if not EMPTY(.RNAME[REGFORMLST]) then
            begin
            local LSTHDR L,ITEM I;
            I←L←.RNAME[REGFORMLST];
            until (I←.I[RLINK]) eql .L do
                begin
                TL←GETTN();
                PREFMOVE(.TL,.GT[.I[RDATITEM(1)],REGF]);
                TL[BNDLSTHDR]←REGS[.I[LDATITEM(1)]];
                TL[REQD]←MEMREQDB;
                TL[REGF]←.I[LDATITEM(1)];
                UPDATE(.I[RDATITEM(1)],1,2)
                end
            end;
```

```
        if (TN←.NODE[REGF]) neq 0 then TN←LOADR0(.NODE);
        if .SIMPLELIFE then NODE[REGF]←GETTN();
        TNP(.NODE[OPR1],0);
        LBPATCH(.LBRETLIST);
        UPDATELON;
        PREFMOVE(.NODE[OPR1],NODE[REGF]←.TN);
        ));


NDRDEF(ANDOR,(
        local LOP,ROP,OP1;
        local ITEM LBLOP:LBROP;
        bind LEXEME LEX2=NODE[OPR2];
        LOP←ROP←0;
        if .NODE[TPATH]
            then (ROP←.MYTN;OP1←.NODE[OPR2])
            else (LOP←.MYTN;OP1←.NODE[OPR1]);
        TNP(.NODE[OPR1],.LOP);
        if FLOWRES then
            begin
            LBLOP←NEWBLK(2);
            LBLOP[LBSUCC(1)]←.CBSTART;
            SAVDTD;
            end;
        TNP(.NODE[OPR2],.ROP);
        if FLOWRES then
            begin
            LBROP←NEWBLK(1);
            LBROP[LBSUCC(1)]←LBLOP[LBSUCC(2)]←.CBSTART;
            if .DTEMPS[CURD] neq .DTOSTK[LDTD] then
                (KILLDYTEMPS(.NODE[OPR2]); SETNOTFPARM);
            POPDTD;
            return
            end;
        if not RESREQ then return;
        PREFMOVE(.OP1,.MYTN);
        ));


NDRDEF(COMP,(
        incr I from 0 to .NODE[NODESIZEF]-2 do
            TNP(.NODE[OPERAND(.I)],0);
        TNP(.NODE[LASTOPERAND],.MYTN);
        if RESREQ then PREFMOVE(.NODE[LASTOPERAND],.MYTN);
        ));
```

```
NORDEF(if,(
        local DTUNEVEN;
        local ITEM LBBOOL:LBTHEN:LBELSE;
        BINDLST(.NODE[OPR1]);
        TNP(.NODE[OPR2],0);
        LBBOOL←NEWBLK(2);
        LBBOOL[LBSUCC(1)]←.CBSTART;
        SAVDTD;
        TNP(.NODE[OPR3],.MYTN);
        LBTHEN←NEWBLK(1);
        LBBOOL[LBSUCC(2)]←.CBSTART;
        RESETDTD;
        TNP(.NODE[OPR4],.MYTN);
        LBELSE←NEWBLK(1);
        LBTHEN[LBSUCC(1)]←LBELSE[LBSUCC(1)]←.CBSTART;
        DTUNEVEN←(.DTEMPS[CURD] neq .DTDSTK[MDTD]);
        MINDTD;
        if .DTUNEVEN then
            begin
            KILLFORKDYTEMPS(.NODE[OPR3]);
            KILLFORKDYTEMPS(.NODE[OPR4]);
            SETNOTFPARM
            end;
        POPDTD;
        if RESREQ then
            (PREFMOVE(.NODE[OPR3],.MYTN); PREFMOVE(.NODE[OPR4],.MYTN));
        BINDLST(.NODE[OPR5]);
        ));


NORDEF(case,(
        local T,RES,DTUNEVEN,GTVEC SUBNODE;
        local ITEM LBSEL:LBFRK, FRKX, GTVEC LBFRKV;
        DTUNEVEN←0;
        BINDLST(.NODE[OPR1]);
        TNP(.NODE[OPR2],0);
        LBSEL←NEWBLK(.NODE[NODESIZEF]-3);
        LBFRKV←GETSPACE(GT,.NODE[NODESIZEF]-3);
        SAVDTD;
        incr I from 2 to .NODE[NODESIZEF]-2 do
            begin
            SUBNODE←.NODE[OPERAND(.I)];
            LBSEL[LBSUCC(.I-1)]←.CBSTART;
            TNP(.SUBNODE,.MYTN);
            LBFRKV[.I-2,0,36]←LBFRK←NEWBLK(1);
            PREFMOVE(.SUBNODE,.MYTN);
            if .I geq 3 then
                if .DTEMPS[CURD] neq .DTDSTK[MDTD] then DTUNEVEN←1;
            if .I neq .NODE[NODESIZEF]-2 then RESETDTD else MINDTD;
            end;
        incr I from 1 to .NODE[NODESIZEF]-3 do
            (LBFRK←.LBFRKV[.I-1,0,36]; LBFRK[LBSUCC(1)]←.CBSTART);
        RELEASESPACE(GT,.LBFRKV,.NODE[NODESIZEF]-3);
        if .DTUNEVEN then
            begin
            incr I from 2 to .NODE[NODESIZEF]-2 do
                KILLFORKDYTEMPS(.NODE[OPERAND(.I)]);
            SETNOTFPARM
```

```
            end;
        POPOTO;
        BINOLST(.NODE[OPERAND(.NODE[NODESIZEF]-1)]);
        ));


routine FLOOP(NODE,TARGET,TYPE)=
        begin
        ! TNP for while-do, until-do, do-while, and do-until
        ! cases 0 through 3 of type respectively
        map GTVEC NODE;
        local L1,L2;
        local ITEM LBBOOL:LBBODY,LOOPTOP;
        BINOLST(.NODE[OPR1]);
        BINOLST(.NODE[OPR2]);
        SAVOTO;
        ENTLOOP;
        LBBOOL←NEWBLK(1); LBBOOL[LBSUCC(1)]←LOOPTOP←.CBSTART;
        TNP(.NODE[OPR3],0);
        if .TYPE/2
            %D-W/U% then (LBBODY←NEWBLK(1); LBBODY[LBSUCC(1)]←.CBSTART)
            %W/U-D% else (LBBOOL←NEWBLK(2); LBBOOL[LBSUCC(1)]←.CBSTART);
        if (not .TYPE^(-1))
          or (bind LEXEME OP4=NODE[OPR4]; .OP4[LTYPF] neq GTTYP)
            then (RESETOTO; KILLDYTEMPS(.NODE[OPR3]));
        TNP(.NODE[OPR4],0);
        if .TYPE/2
            %D-W/U% then
                        begin
                        LBBOOL←NEWBLK(2);
                        LBBOOL[LBSUCC(1)]←.LOOPTOP;
                        LBBOOL[LBSUCC(2)]←.CBSTART
                        end
            %W/U-D% else
                        begin
                        LBBODY←NEWBLK(1);
                        LBBODY[LBSUCC(1)]←.LOOPTOP;
                        LBBOOL[LBSUCC(2)]←.CBSTART
                        end;
        RESETOTO; XITLOOP;
        KILLDYTEMPS(.NODE[OPR4]);
        KILLDYTEMPS(.NODE);
        POPOTO;
        if .MYTN neq 0 then MOVE(LITLEXEME(-1),.MYTN);
        end;
```

```
NDRDEF(WD,FLDDP(.NDDE,.TARGET,0));


NDRDEF(UD,FLDDP(.NDDE,.TARGET,1));


NDRDEF(DW,FLDDP(.NDDE,.TARGET,2));


NDRDEF(DU,FLDDP(.NDDE,.TARGET,3));


NDRDEF(IDLDDP,(
        local L, GTVEC CV;
        local ITEM LBPRE:LBBDDY;
        TNP(.NDDE[DPR2],0);
        TNP(.NDDE[DPR3],0);
        TNP(.NDDE[DPR4],0);
        BINDLST(.NDDE[DPR5]);
        BINDLST(.NDDE[DPR6]);
        PREFMDVE(.NDDE[DPR2],.NDDE[DPR1]);
        CV←.NDDE[DPR1];
        SAVDTD;
        ENTLDDP;
        LBPRE←NEWBLK(2); LBPRE[LBSUCC(1)]←.CBSTART;
        TNP(.NDDE[DPR7],0);
        LBBDDY←NEWBLK(2);
        LBBDDY[LBSUCC(1)]←.LBPRE[LBSUCC(1)];
        LBBDDY[LBSUCC(2)]←LBPRE[LBSUCC(2)]←.CBSTART;
        DPERATE(.NDDE[DPR3],.NDDE[DPR1]);
        CDMPARE(.NDDE[DPR1],.NDDE[DPR4]);
        RESETDTD; XITLDDP;
        KILLDYTEMPS(.NDDE[DPR7]);
        PDPDTD;
        if .MYTN neq 0 then MDVE(LITLEXEME(-1),.MYTN);
        ));


TLRDEF(label,(
        external LBPATCH;
        TLCDMMDN(.NDDE,.TARGET);
        LBPATCH(.NDDE[LBPATCHLIST]);
        KILLPDTEMPS(.NDDE,.DTEMPS[CURD]);
        SETNDTFPARM;
        ));
```

```
TLRDEF(leave,(
        local GTVEC N;
        N←.ST[.NODE[OPR2],LINKFLD];
        TNP(.NODE[OPR1],.N[REGF]);
        MUSTPATCH(N[LBPATCHLIST]);
        PREFMOVE(.NODE[OPR1],.N[REGF]);
        NOTEDTD;
        ));


TLRDEF(RLEAVE,(
        external LBRETLIST;
        local GTVEC RNTN;
        RNTN←.NODE[OPR2]; RNTN←.RNTN[RETLAB];
        TNP(.NODE[OPR1],.RNTN[REGF]);
        PREFMOVE(.NODE[OPR1],.RNTN[REGF]);
        MUSTPATCH(LBRETLIST);
        NOTEDTD;
        UPDATELON;
        ));


TLRDEF(SYNNULL,(
        local GTVEC PAR;
        bind LEXEME LEX=NODE;
        PAR←.NODE[CSPARENT];
        if not .PAR[BOUND] then
            begin
            if not .PAR[DELAYED] then NONBOGUS(PAR);
            if .PAR neq .LEX[ADDRF] then
                TLLIST(FASTLEXOUT(GTTYP,.PAR),0);
            end;
        NOTEDTD;
        UPDATELON;
        ACCESS(.NODE);
        ));


NDRDEF(select,(
        local OTHEREND,OTUNEVEN,SAVDTC, LEXEME L, GTVEC OTHERTN;
        local ITEM LBSEL:LBLEFT:LBRIGHT;
        OTUNEVEN←0;
        OTHEREND←LITVALUE(.NODE[LASTOPERAND]);
        if .OTHEREND eql 0
            then OTHERTN←0
            else NODE[OPERAND(.NODE[NODESIZEF]-2)]←LEXOUT(TNTYP,OTHERTN←GETTN());
        TNP(.NODE[OPR1],0);
        LBSEL←NEWBLK(1);
        LBLEFT←0;
        MOVE(LITLEXEME(0),.OTHERTN);
        incr I from 1 to .NODE[NODESIZEF]-3 do
            begin
            L←.NODE[OPERAND(.I)];
            if .I
                then !left part
                    begin
                    if .LBLEFT neq 0
```

```
                    then LBLEFT[LBSUCC(2)]←.CBSTART
                    else LBSEL[LBSUCC(1)]←.CBSTART;
                if .L[LTYPF] neq SELTYP then (TNP(.L,0);COMPARE(.NODE[OPR1],.L));
                LBLEFT←NEWBLK(2);
                end
            else !right part
                begin
                if .I eql 2 then SAVDTC←.DTEMPS[CURD];
                SAVDTD;
                LBLEFT[LBSUCC(1)]←.CBSTART;
                TNP(.L,.MYTN);
                LBRIGHT←NEWBLK(1);
                LBRIGHT[LBSUCC(1)]←.CBSTART;
                if .DTEMPS[CURD] neq .SAVDTC then DTUNEVEN←1;
                RESETDTD;
                KILLDYTEMPS(.L);
                POPDTD;
                end;
        if .I leq .OTHEREND then ACCESS(.OTHERTN)
        end;
    LBLEFT[LBSUCC(2)]←.CBSTART;
    if .DTUNEVEN
        then SETNOTFPARM
        else
            incr I from 2 to .NODE[NODESIZEF]-3 by 2 do
                begin
                local LEXEME OP;
                bind GTVEC SUBNODE=OP;
                OP←.NODE[OPERAND(.I)];
                if .OP[LTYPF] eql GTTYP
                    then SUBNODE[DTDELETE]←DTDONTCARE
                end;
    ));


TLRDEF(ENABLE,(
    SETBOUND;
    LOADRD(.NODE);
    NOTEDTD;
    UPDATELON;
    TNP(.NODE[OPR1],0);
    ));
```

```
NDRDEF(SIGNAL,(
        external LBPATCH,LBRETLIST;
        local GTVEC TL;
        TNP(.NODE[OPR1],0);
        LOADR0(.NODE);
        MUSTPATCH(LBRETLIST);
        PREFMOVE(.NODE[OPR1],.MYTN);
        ));
```

Appendix B

The Basic Packing Routine

Listed at the end of this appendix is the code for the basic loop of the packing procedure. The routine EVAL is called recursively to evaluate the maximum profit that can be obtained by packing subsequent TNs. Below are descriptions of the important routines called by EVAL.

POSTUPDATE — remembers the current complete solution as the most profitable thus far examined. When the algorithm terminates, the last solution remembered is an optimal solution.

REALIZABLE — tries to pack its second argument into a register. The first argument is true when REALIZABLE is called during reshuffling. The value returned is true if the packing succeeded and false otherwise.

BESTCASEWITHOUT — returns the value of the most profitable completion of the current partial solution without the current TN. The most profitable completion may not be feasible.

UNDESIRE — removes its argument from the current solution.

```
routine EVAL(PASTVAL,ME)=
    begin
    map TNREPR ME;
    label MOVERIGHT;
    macro BOTTOMOFTREE(X)=(.X[RLINK] eql LHEAD<0,0>)$;
    macro TRYUPDATE=(if -.MYVAL eql .GLBLMAX then POSTUPDATE())$;
    local TEMP,MINACCEPT,MAXSON,PREFMISSED,TNREPR SON;
    bind MYVAL=PASTVAL;


    EVALCNT←.EVALCNT+1;
    ME[TNFLD(TNREGSTRIED)]←.RESERVED;
    if NOT REALIZABLE(false,.ME[TNPTR]) then return 0;
    if .SIMPLERESHUFFLE
        then PREFMISSED←0
        else PREFMISSED←(.ME[TNFLD(PREFF)] neq 0) and (.ME[TNFLD(BNDTYP)] neq BNDPREF);
    ME[TNFLD(TNCONFNUM)]←NOCONFSEEN;
    if (MYVAL←.PASTVAL+.ME[TNFLD(TNCOST)]-.PREFMISSED) gtr .GLBLMAX
        then (GLBLMAX←.MYVAL; MYVAL←-.MYVAL);
    MINACCEPT←.GLBLMAX-abs(.MYVAL);
    MAXSON←0;
    if BOTTOMOFTREE(ME)
        then (TRYUPDATE; UNDESIRE(.ME[TNPTR]); DORESHUF←1)
        else
        do  begin
            SON←.ME;
            while (SON←.SON[RLINK]) neq LHEAD<0,0> do
              MOVERIGHT:
                begin
                if .EVALCNT gtr .MAXREPS then (CUTOFF←true; exitloop);
                if .SON[TNFLD(BNDLSTHDR)] neq 0 then leave MOVERIGHT;
                if .SON[TNFLD(TNCSUM)] leq .MINACCEPT then exitloop;
                if (TEMP←EVAL(abs(.MYVAL),.SON)) gtr .MAXSON then
                    (if (MAXSON←.TEMP) gtr .MINACCEPT then MINACCEPT←.TEMP);
                if BESTCASEWITHOUT(.SON[TNPTR]) leq .SON[TNFLD(TNCOST)] then exitloop;
                end;
            TRYUPDATE;
            UNDESIRE(.ME[TNPTR]);
            if .TEMP eql .SON[TNFLD(TNCSUM)] then
                if .SON[LLINK] eql .ME then ! optimal completion, don't try any more
                    exitloop;
            if .SIMPLERESHUFFLE then exitloop;
            end
        until NOT REALIZABLE(false,.ME[TNPTR]);
    return(.ME[TNFLD(TNCOST)]+.MAXSON-.PREFMISSED)
    end;
```

Bibliography

All70 Allen, F. E., "Control Flow Analysis," **SIGPLAN Notices**, July 1970.

All71a Allen, F. E. and John Cocke, "A Catalogue of Optimizing Transformations," **Design and Optimization of Compilers**, (R. Rustin, ed.), Prentice-Hall, 1971, 1-30.

All71b Allen, F. E., "Control and Data Flow Analysis," Computer Science Dept. colloquium series, Carnegie-Mellon Univ., April 1971.

Bal65 Balinski, M. L., "Integer Programming: Methods, Uses, Computation," **Management Science** 12, 3 (November 1965), 253-313.

Bea71 Beatty, James C., "A Global Register Assignment Algorithm," **Design and Optimization of Compilers**, (R. Rustin, ed.), Prentice-Hall, 1971, 65-88.

Bea72 Beatty, James C., "An Axiomatic Approach to Code Optimization for Expressions," JACM 19,4 (October 1972), 613-40.

Bel66 Belady, L. A., "A Study of Page Replacement Algorithms for a Virtual Storage Computer," **IBM Systems Journal** 5,2 (1966), 78-101.

Bru74 Bruno, John and Ravi Sethi, "Register Allocation for a One-Register Machine," Computer Science Dept. Technical Report No. 157, Penn State University (October 1974).

Coc70 Cocke, John and J. T. Schwartz, **Programming Languages and their Compilers**, Courant Institute of Mathematical Sciences, New York University, New York, 1970.

Day70 Day, W. H. E., "Compiler Assignment of Data Items to Registers," **IBM Systems Journal** 9,4 (1970), 281-317.

DEC71 Digital Equipment Corp., **PDP-11/20/15/R20 Processor Handbook**, Maynard, Mass., 1971.

DEC74 Digital Equipment Corp., **Bliss-11 Programmer's Manual**, Maynard, Mass., 1974.

Geo67 Geoffrion, Arthur M., "Integer Programming by Implicit Enumeration and Balas' Method," **SIAM Review** 9,2 (April 1967), 178-190.

Ges72 Geschke, Charles M., "Global Program Optimizations," Ph.D. thesis, Computer Science Department, Carnegie-Mellon University, 1972.

Gil66 Gilmore, P. C. and R. E. Gomory, "The Theory and Computation of Knapsack Functions," **Operations Research** 14,6 (1966) 1045-74.

Han74 Hansen, Gilbert J., "Adaptive Systems for the Dynamic Run-Time Optimization of Programs," Ph.D. thesis, Computer Science Department, Carnegie-Mellon University, 1974.

Hop69 Hopgood, F. R. A., **Compiling Techniques**, American Elsevier, New York, 1969, 91-103.

Hor66 Horwitz, L. P., R. M. Karp, R. E. Miller and S. Winograd, "Index Register Allocation," JACM 13,1 (January 1966), 43-61.

Knu71 Knuth, Donald E., "An empiracle study of FORTRAN programs," **Software--Practice and Experience** 1,2 (April/June 1971), 105-133.

Low69 Lowery, E. S. and C. W. Medlock, "Object Code Optimization," **CACM 12,1** (January 1969), 13-22.

Luc67 Luccio, F., "A Comment on Index Register Allocation," **CACM 10,9** (September 1967), 572.

Mit70 Mitten, L. G., "Branch-and-Bound Methods: General Formulation and Properties," **Operations Research 8**, 1 (January-February 1970), 24-34.

Nak67 Nakata, Ikuo, "On Compiling Algorithms for Arithmetic Expressions," **CACM** 10,8 (August 1967), 492-94.

New75 Newcomer, Joseph M., "Machine-independent Generation of Optimal Local Code," Computer Science Department Ph.D. thesis, Carnegie-Mellon University, May 1975.

Red69 Redziejowski, R. R., "On Arithmetic Expressions and Trees," **CACM 12,2** (February 1969), 81-84.

Set70 Sethi, R. and J. D. Ullman, "The Generation of Optimal Code for Arithmetic Expressions," JACM 17,4 (October 1970), 715-728.

Set75 Sethi, Ravi, "Complete Register Allocation Problems," **SIAM Journal on Computing** 4,3 (September 1975), 226-248.

War62 Warshall, Stephen, "A Theorem on Boolean Matricies," **JACM 9,1** (January 1962), 11-12.

Wul71 Wulf, W. A., D. B. Russell and A. N. Habermann, "BLISS: A Language for System Programming," **CACM** 1,12 (December 1971), 780-790.

Wul75 Wulf, W., R. Johnsson, C. Weinstock, S. Hobbs and C. Geschke, **The Design of an Optimizing Compiler**, American Elsevier, New York, 1975.