# Compiler Design for Efficient Code Generation and Program Optimization

A. Rudmik
Queen's University

E.S. Lee
University of Toronto

## I Introduction

This paper describes the design of a compiler for an extended PASCAL with particular emphasis on code generation and optimization. The design objectives are: compiler reliability, code generator portability and object code efficiency.

Designing a compiler to meet the above objectives usually requires compromises in the design. To achieve the objective of compiler reliability, one frequently tries to simplify the compiler design in order to reduce the complexity that must be mastered in the implementation. Unfortunately, the objective of producing efficient code conflicts with the objective of simplifying the compiler design. The design presented in this paper attempts to maximize the attainment of these objectives with minimal compromises.

The PASCAL language is augmented with erasable extensions so that programs written using the extensions can easily be converted into standard PASCAL programs. The three extensions to PASCAL are modules, initial procedures, and open procedures. The modules are used for data abstraction and program modularization. The compiler checks the interfaces between separately compiled modules and reports all inconsistencies. Data initialization is specified by initial procedures. The initial procedures are interpreted at compile time, thus initializing program data variables. The compiler replaces the call to an open procedure with the code for that procedure. All three of the above extensions have a significant impact on the compiler design.

## II Compiler Structure

The PASCAL compiler translates the source program into an intermediate code, which is subsequently translated into machine specific object code. One advantage of this approach is that, through the

proper choice of intermediate code the language dependent program analysis and machine dependent code generation can be factored into separate passes. Another advantage is that "optimizing" transformations can be performed on this intermediate code to improve the generated object code. The PASCAL compiler currently consists of eight passes.

Pass1 performs the lexical and syntactic analysis, generating as output a standardized high level control flow graph.

Pass2 analyzes the constant, type, and variable declarations and maps the program identifiers into unique name indices according to PASCAL scope rules.

Pass3 traverses the control flow graph and checks the program for semantic correctness. In the process of performing the semantic analysis, simple transformations are performed on the flow graph in instances where the graph generated in pass1 does not represent the semantics of the program.

Pass4 replaces the calls to open procedures by the corresponding procedure bodies. Temporary storage locations are assigned to local variables declared within open procedures.

Pass5 performs global machine and language independent optimizing transformations on the intermediate code.

Pass6 performs a "live-dead" variable analysis and encodes this information in the flow graph to be used by the register allocator in pass7.

Pass7 translates the intermediate code into machine specific object code.

Pass8 performs machine dependent peephole optimizing transformations on the object code.

The inclusion of passes 4, 5, 6, and 8 are optional.

The first pass translates the source program into a standardized intermediate program representation. All subsequent passes, with the exception of pass8 are filters, transformers, or generators accepting as input this intermediate code. Passes two through seven are driven by a common parsing algorithm.

The object code generated by pass7 can be further optimized by a peephole optimizer. The peephole optimizer is modelled after the one described by Fraser [9]. The major departure from Fraser's approach is that the peephole optimizer transforms the object code directly rather than an assembler form of the program.

The compiler structure adopted has several advantages:

1) The high level control flow graph can be generated using a simple syntax directed translation based on any favorite parsing technique.

2) The semantic checking is performed on a language independent program representation. This permits major syntactic changes in the language without requiring changes in the semantic routines (unless there are corresponding semantic changes).

3) The machine specific code generator can be implemented in a simple straight forward manner without worrying about local or global code inefficiencies.

4) The compiler processes at most three different program representations: the source program, the high level flow graph, and the object code. The small number of program representations simplify the implementation of each pass and permit many of the compiler passes to be optional. A compiler based on this model can be implemented quickly without sacrificing the potential of generating good code.

## III Intermediate Code

The intermediate code produced by the first pass is a high level control flow graph representation of the program [17,18]. The control flow graph is constructed while parsing the program. Given a parser that recognizes the innermost nested phrases first, the PASCAL compiler constructs a control structure graph (CSG) representing the control flow semantics for each phrase recognized by the parser. The nodes in the CSG represent computations and the directed edges represent the control flow paths. The operators contained in the nodes of the CSG specify the location of data, define data constants, and perform computations of the program.

For a large class of programming languages one can identity a small set of control structures which have identical control flow semantics. Programs in these languages can then be represented by flow graphs which are constructed by concatenating and nesting CSG's. The intermediate code constructed in this manner is called a hierarchical directed graph (HDG). If the CSG's are restricted to directed graphs with exactly one entry and exactly one exit edge, then an HDG is defined to be an ordered directed graph where each node of the HDG contains either an HDG or an operator.

Branching constructs, such as calls, returns, exits, and goto's are represented by branching operators and program labels. Branching operators cause a transfer of control from one node to another without the requirement of an edge to represent the branch.

```
s := 0;
while i < k do
   begin
      t := j*10 + i;
      s := s + a[t];
      i := i + 1
   end
```

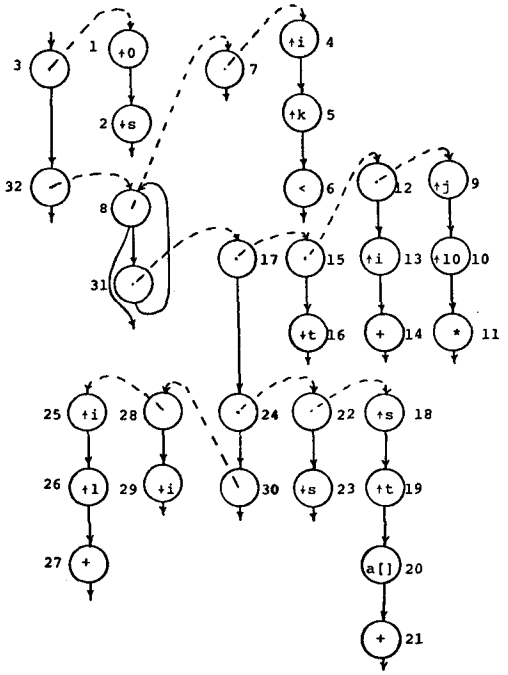


figure 1: HDG program representation

The HDG program representation will be used as a theoretical model for describing the data flow analysis algorithms. Figure 1 illustrates a simple PASCAL program and it's HDG representa-

tion. The nesting of CSG's are indicated by broken edges from a node to the HDG contained in the node.

The HDG in figure 1 is constructed from two CSG's: a simple node having a single entry and a single exit edge, and the while loop CSG. The operators contained within the terminal nodes of a CSG are instructions to an abstract stack machine. The instruction ↑j pushes the value contained in the memory location "j" onto the evaluation stack; ↓t stores the value on top of the evaluation stack into the memory location "t" and pops the stack. The operation "a[]" replaces the top value in the stack with the value of a[t].

The HDG is implemented as an n-ary tree where each CSG is represented by a labelled spanning subtree. The tree representation has the property that the label on a particular node completely specifies the number of subtrees subtended by the node as well as the control flow semantics associated with the node and its subtrees. This implementation of the intermediate code contains two kinds of labels: labels which indicate the destination of branching operators and labels indicating the particular CSG represented by a node. In this paper the two uses of labels will be differentiated by the context in which they are used.

## IV Declaration Analysis

The following correspondance between data types and control structures have been observed: scalars and assignment statements, arrays and for loops, records and compound statements, variants and case statements. The compiler constructs an HDG representation for data type definitions in a similar manner to the control structures in the program body. The semantic analysis of PASCAL constant, type, and variable declarations are performed by a collection of recursive tree walking routines.

The top down parsing of the HDG is appropriate for the processing of PASCAL type declarations (for example, nested record definitions). The problem of finding identical type definitions simply becomes the problem of finding identical subtrees.

The identifiers used in the declarations are mapped into name indices according to PASCAL scope rules. The HDG representing the code body is traversed and all identifier names are replaced by the name indices.

## V Opening Procedures

Frequently, the use of data abstraction necessitates the definition of short procedures and functions to operate on data defined by the abstraction. In many

cases the call linkage is a significant execution time overhead when using these operations. The open procedure extension to PASCAL allows the operation to be compiled inline.

The transformation of replacing call sites by inline procedure and function code is implemented as a transformation on the intermediate code. The HDG for each procedure is stored on a temporary file. When the compiler encounters a call to an open procedure, the compiler locates the HDG for the procedure and replaces the call by the HDG. The open procedure becomes an unnamed block having a new scope level. The actual parameters in the procedure are assigned to the corresponding formal "value" parameters. The HDG for the procedure is traversed replacing all occurances of var parameters by their corresponding actual parameters.

## VI Global Program Optimization

The analysis performed by the optimizer is based on high level data flow analysis techniques described by Rosen[17] and Rudmik[18]. The main advantage of using the HDG program representation is that the CSG's can be analyzed by the compiler designer prior to implementing the optimizer. The conditions for performing a code motion transformation can be derived in a systematic manner[18] for each CSG. The control flow paths created by branching operators and program labels are treated as restrictions on the conditions of the optimizing transformations.

The optimizing pass of the compiler accepts as input an HDG representation of the program and generates as output a reordered and restructured HDG. The optimized HDG must be functionally equivalent to the input HDG. The graph theoretic model of the HDG is used to describe the control flow and data flow analysis algorithms. The tree representation facilitates efficient analysis of the intermediate code using simple tree walking algorithms. This approach does not preclude application to languages having unstructured branching constructs or to programs that use them, but simply requires that the use of these constructs be infrequent.

Each node in the HDG has data flow information associated with it, called a node descriptor. This information is computed by traversing the HDG. The space required to store the node descriptors is bounded by the number of variables used in the program. Therefore, the data flow analysis is linear in both space and time with the size of the program segment being optimized. Furthermore, the optimizer does not retraverse previously optimized substructures; the node that subtends these substructures already has node descriptors computed in the process of optimizing the computation represented by

the node. This technique of information reduction further improves the space and time efficiency of the data flow analysis algorithms.

There are two kinds of node descriptors. First, the node exit descriptor describes certain data flow properties of the node that are true after evaluating the computation represented by the node. The optimizer assigns a node exit descriptor to each terminal node (most deeply nested node - only containing an operator) of the HDG, based on the property of the operator contained in the node. The node exit descriptors for all other nodes are derived from the descriptors of the terminal nodes. For a given node n, the information specified by the node exit descriptor is selected by the following functions.

MOD(n) - specifies the set of variables whose data values can be modified by the execution of node n.

REF(n) - specifies the set of variables whose data values can be referenced in node n.

BRANCH(n) - specifies the set of program labels referenced by branching operators in node n.

LABEL(n) - specifies the set of program labels defined in node n.

If node n contains the CSG instance g, then the node exit descriptor for node n is determined by performing the union of the node exit descriptors over all the nodes of the CSG g immediately contained within node n. Figure 2 presents the node exit descriptors for the HDG in figure 1. The notation for specifying the node exit descriptors is (a,b) where a = MOD(n) and b = REF(n).

| node | | node | |
|---|---|---|---|
| 1 | ({},{}) | 18 | ({},{s}) |
| 2 | ({s},{}) | 19 | ({},{t}) |
| 3 | ({s},{}) | 20 | ({},{a}) |
| 4 | ({},{i}) | 21 | ({},{}) |
| 5 | ({},{k}) | 22 | ({},{a,s,t}) |
| 6 | ({},{}) | 23 | ({s},{}) |
| 7 | ({},{i,k}) | 24 | ({s},{a,s,t}) |
| 8 | ({},{i,k}) | 25 | ({},{i}) |
| 9 | ({},{j}) | 26 | ({},{}) |
| 10 | ({},{}) | 27 | ({},{}) |
| 11 | ({},{}) | 28 | ({},{i}) |
| 12 | ({},{j}) | 29 | ({i},{}) |
| 13 | ({},{i}) | 30 | ({i},{i}) |
| 14 | ({},{}) | 31 | ({i,s,t},{a,i,j,s,t}) |
| 15 | ({},{i,j}) | 32 | ({i,s,t},{a,i,j,k,s,t}) |
| 16 | ({t},{}) | | |
| 17 | ({t},{i,j}) | | |

figure 2: Node exit descriptors

The second kind of descriptor, called a node entry descriptor, describes data flow properties that are known to be true on entering a node. The node entry information is defined with respect to a preceeding reference node in the HDG. This information is computed by traversing a directed subgraph of the HDG from the reference node to the specified node. The node entry information is obtained from the node exit descriptors calculated previously and by propagating information in a forward direction (as specified by the control flow semantics of the HDG). There are three kinds of node entry descriptor information used by the optimizer.

First, the successor-predecessor node entry descriptor denoted MODsp(m,n) is computed by traversing a subgraph of the

HDG along all possible directed paths from node m to node n.

Second, the excluded node entry descriptor denoted MODex(m,n,X) is computed in a manner similar to MODsp(m,n) except that nodes in set X are not visited.

Third, the included node entry descriptor denoted MODinc(m,n) is obtained by traversing the subgraph of the HDG from node m to node n along all directed paths that contain exactly one node m and exactly one node n.

The node entry descriptors REFsp, REFex, and REFinc can be computed in a similar manner.

The following are examples of node entry descriptors for node 12 in figure 1:

```
MODsp(3,12)        = {i,s,t}
MODex(3,12,{12})   = {i,s,t}
MODinc(3,12)       = {}
REFsp(3,12)        = {a,i,j,k,s,t}
REFex(3,12,{12})   = {a,i,k,s,t}
REFinc(3,12)       = {i,k}
```

The side effects of procedure calls and variable aliases are computed using algorithms developed by Banning [5]. These algorithms are implemented by performing a single pass through the HDG representation of the program. The information required by Banning's algorithms is easily obtained from the HDG.

PASCAL requires that reference variables refer to anonymous memory locations. A reference variable can be treated as an index into a large array which is disjoint from other program variables. The optimizer associates a pseudo variable name with each distinct reference type declared in the program. This approach allows us to handle PASCAL pointers in a straight forward manner.

All optimizing transformations must preserve functional equivalence in programs that terminate normally. The transformed program is defined to be functionally equivalent to the source program, if for each set of values of the input variables the same set of values of the output variables is produced by both programs. The definition of functional equivalence as applied to the HDG must include the effects of a transformation on the data flow of the program. A transformation is defined to preserve functional equivalence on the HDG if the following properties of the HDG are preserved:

1) For each set of values of the input variables the assignment of values to the output variables is not changed.

2) A transformation must not introduce a new error condition or eliminate an existing error condition.

The expression "j*10" represented by node 12 can be moved backward out of the while loop provided that the following conditions are satisfied. The operator "*" denotes set intersection.

1) BRANCH(12) = Ø and LABEL(12) = Ø, node 12 does not contain branching operators or program labels.

2) MOD(12) = Ø

3) MODsp(3,12) * REF(12) = Ø

4) either
   4.1) BRANCHsp(3,12) = Ø and LABELsp(3,12) = Ø, node 12 is on every path from node 3 to node 12.

   or
   4.2) Let node k be a successor node of node 3 and a predecessor node of node 12. If node k contains a branching operator then the destination of the branch must be on a path from node 3 to node 12. Similarly, if node k contains a label then all the branching operators that reference the label must be on paths from node 3 to node 12.

5) either
   5.1) The evaluation of node 3 must imply a subsequent evaluation of node 12.

   or
   5.2) All evaluations of node 12 are safe (will not cause an error) where node 12 is moved as a direct successor to node 3.

All of the above conditions can be evaluated by examining the HDG graph structure. This approach does not perform the data flow analysis along paths constructed from branching operators, but rather treats branching operators as restrictions on the transformations. The advantage of this approach is that the data flow analysis can be performed efficiently with a slight lose in optimizer effectiveness for programs containing branching operators.

The conditions 1, 2, 3, 4.1, and 5.1 can be evaluated efficiently (ie. in linear time). Condition 4.2 requires that the entire procedure containing the branch label be analyzed (or the entire program if goto out of a procedure is permitted). Condition 5.1 requires that the while loop be transformed into a repeat-until loop. In some cases condition 5.2 can be satisfied if subrange information is available to the optimizer.

The generalized conditions for code motion transformations can be derived for arbitrary CSG's [18]. The properties of a specific CSG are then applied to the generalized conditions to obtain the conditions for a specific code motion transformation. The conditions for moving an expression such as "j*10" backward out of a loop can be obtained from the generalized backward code motion transformation. The diagram of figure 3 illustrates the transformation of moving an arbitrary node n contained within the CSG g, backward onto the entry edge of the CSG. Node m is a reference node.

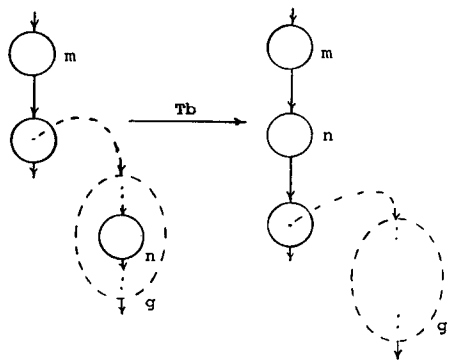


Figure 3.

The conditions for transformation Tb are:

1) Node n must represent a statement.

2) BRANCH(n) = Ø and LABEL(n) = Ø

3) MODsp(m,n) * REF(n) = Ø

4) MODex(m,n,{n}) * MOD(n) = Ø

5) either
   5.1) BRANCHsp(m,n) = Ø
        and LABELsp(m,n) = Ø

   or
   5.2) Let node k be a successor node of node m and a predecessor node of node n. If node k contains a branching operator then the destination of the branch must be on a path from node m to node n. Similarly, if node k contains a label then all the branching operators that reference the label must be on paths from node m to node n.

6) either
   6.1) The evaluation of node m must imply a subsequent evaluation of node n,

        and MOD(n) * REFinc(m,n) = Ø

   or
   6.2) For all nodes k that are successors of node m, MOD(n) * REF(k) = Ø, or all variables in REF(k) must be modified on every path from node m to node k,

        and program output variables are not members of MOD(n),

        and the evaluation of node n is safe for node n a direct successor of node m.

The generalized conditions for the elimination of useless and redundant code, and forward code motion can be expressed in a similar manner to transformation Tb.

The optimizer also searches the intermediate code for specific frequently used code patterns and replaces these code patterns with special operators or CSG's. The semantics of the new operators or CSG's are defined to be equivalent to the structures they replace. These simple structural transformations simplify the task of generating improved code for frequently used constructs in the code generator pass.

The global optimizer performs the following transformations[2,6,7,15,24]: constant propagation, folding, elimination of redundant statements and expressions, backward code motion of statements and expressions, elimination of useless code, strength reduction, recombining statements, and opening procedures.

## VII Register Allocation

The code generator performs a simple yet effective register allocation to reduce the number of memory references. The following criteria is used to for allocate registers to program variables.

1) If no free registers are available, the least recently used register is deallocated and the register is made available.

2) On encountering a node where a variable is indicated as being "dead", the register allocated to the variable is deallocated and made available.

The data flow analyzer in pass6 determines whether a variable is "live" at a particular node in the HDG and encodes this information in the node descriptor. The register allocator in pass7 uses this information to allocate registers.

The machine dependent peephole optimizer [9] cannot be used in conjunction with the register optimizer. The problem

is that the peephole optimizer will inadvertently eliminate the use of a register in certain code sequences. This problem is currently under investigation. The peephole optimizer does an excellent job on improving the code produced using a naive code generator but has a small effect on the code generator described in section IX.

## VIII Storage allocation and Initialization

The seventh pass in the compiler allocates and initializes storage for program variables and compiler generated temporary variables. Initialization of data storage is specified by code contained within initial procedures. The intermediate code for these procedures is constructed in exactly the same manner as for normal procedures. An interpreter traverses the intermediate code initializing the appropriate memory locations specified by the left hand side of assignment statements. All PASCAL statements and expressions with the exception of procedure and function calls (including procedures "new" and input and output procedures) are permitted within the initial procedure. This approach to data initialization gives the programmer great flexibility in writing his initialization code. The initial procedures can be easily converted into standard procedures thus preserving PASCAL compatability.

## IX Code Generation

The code generator is a set of mutually recursive tree traversing procedures. There are three different ways in which the code generator can realize the generation of efficient machine code. The first technique uses knowledge of the current control structure for which we are generating code to propagate information down through the HDG. This technique provides additional context information that can be used by the code generator at lower levels of the HDG.

The second technique is to delay the binding to machine code instructions and to propagate information up through the HDG. It can be shown that the combination of the above two techniques permits the generation of locally optimal code for zero, one, and two address computers.

In generating code for machines with span-dependent branching instructions, significant savings in code space and execution time can be obtained by using the shorter form of an instruction whenever possible[22]. The PASCAL code generator produces near optimal branching code by incorporating a backtracking scheme into the tree walking code generators. Although this technique requires exponential time in the worst case, empirical evidence indicates that in practice the algorithm is nearly linear (backtracking required in 1.5 percent of the cases).

This property is attributed to the fact that modern structured programming methods encourage the writing of small code units which are amenable to this kind of optimization.

A code generator can be implemented quickly using a naive code generation strategy. This code generator would always emit code upon encountering an operator in the HDG. The code generator can later be extended to incorporate the above improvements without having to rewrite it. This makes the compiler portable without sacrificing the potential of generating good code. Both code generation strategies have been used for the PDP 11/45. The naive code generator was 20 percent smaller than the more sophisticated one and produced code requiring 40 to 45 percent more space.

## X Summary and Conclusions

The PASCAL compiler currently generates code for the PDP 11/45 and is self compiling on a PDP 11/03 with 28k words of memory. The code produced without optimization, using the sophisticated code generator, requires 7 to 10 percent more space that the C compiler for similar programs. Significant code space reductions were obtained by invoking the code optimizers in the different passes of the PASCAL compiler. The table below summarizes the effectiveness of these optimizations on reducing code space.

| Kind of optimization | Space saving |
|---|---|
| 1) Global optimization | 18 - 24 % |
| 2) Machine dependent optimization | |
| a) Span dependent instructions | 8 - 15 % |
| b) Registers (minimize memory accesses) | 8 - 12 % |
| 3) Peephole optimization | 5 - 8 % |
| Total space savings | 30 - 45 % |

The execution time of the optimized programs was improved by a factor of 1.5 to 3.2. The optimizer was effective in eliminating redundant range checking code; a program compiled and optimized with range checking, typically executed faster than unoptimized programs without range checking. The optimizer was especially effective in optimizing programs that used complex record data types.

## Acknowledgments

The authors wish to thank Mary Lepage and Sundaram Ramakesavan for their excellent programming help.

## References

[1] Aho, A.V. and Johnson, S.C., Optimal code generation for expression trees. J. ACM. 23,3 (July 1976).

[2] Allen, F.E. Program optimization. Ann. Rev. Automatic Programming. 5 (1979), pp. 239-307.

[3] Allen, F.E. Interprocedural data flow analysis. Information Processing 74, North-Holland Pub. Co., Amsterdam. (1974), pp. 398-402.

[4] Allen, F.E. et al. The experimental

compiling systems project. IBM Research report, RC 6718 (#28922) (June 1977).

[5] Banning J. An efficient way to find the side effects of procedure calls and aliases of variables, Sixth annual ACM Symposium on principles of prog. lang, (Jan.1979), pp. 29.

[6] Carter, J.L. A case study of a new code generation technique for compilers. Comm. ACM. 20, 12 (1977), pp. 914-920.

[7] Cocke, J. Global common subexpression elimination. SIGPLAN Notices (ACM) 5,7(July 1970), pp. 20-24.

[8] Cocke, J. An algorithm for reduction of operator strength. Comm. ACM. 20, 11 (Nov. 1977), pp. 850-856.

[9] Fraser C.W. A Compact, Machine-Independent Peephole Optimizer, Sixth annual ACM Symposium on Principles of prog. lang., (Jan.1979), pp. 1.

[10] Glanville, R.S. and Graham, S.L. A new method for compiler code generation. Conf. rec. Fifth annual ACM symposium on principles of prog. lang. (Jan. 1978), pp. 231-240.

[11] Graham, S.L., and Wegman, M. A fast and usually linear algorithm for global flow analysis. J. ACM 23, 1(Jan. 1976), pp. 172-202.

[12] Hecht, M.S., and Ullman, J.D. A simple algorithm for global data flow analysis problems. SIAM J. 4 (1975), pp. 519-532.

[13] Kennedy, K. Node listings applied to data flow analysis. Second ACM symp. on Principles of Programming Languages. Palo Alto, Calif., (1975), pp. 10-21.

[14] Kildall, G.A. A unified approach to global program optimization. ACM symp. on Principles of Programming Languages, Boston, Mass., (1973), pp. 194-206.

[15] Knuth, D.E. An empirical study of FORTRAN programs. Software - Practice and Experience 1,2 (April-June 1971), pp. 105-133.

[16] Loveman, D.B. Program improvement by source to source transformation. Third ACM Symp. on Principles of Programming Languages, Atlanta, Ga., (1976), pp. 140-152.

[17] Rosen, K.B. High-level data flow analysis. Comm. ACM 20, 10(1977), pp. 712-724.

[18] Rudmik, A. On the Generation of Optimizing Compilers, Ph.D. thesis, Dept. Elect. Eng. University of Toronto, 1975.

[19] Schaefer, M. A Mathematical Theory of Global Program Analysis. Prentice-Hall, Englewood-Cliffs, N.J., (1973).

[20] Scheifler, R.W. An analysis of inline substitution for a structured programming language. Comm. ACM. 20, 9 (Sept. 1977), pp. 647-654.

[21] Schneck, P.B., and Angel, E. A FORTRAN to FORTRAN optimising compiler. Computer J. 16 (1973), pp. 322-330.

[22] Szymanski, T.G. Assembling code for machines with span dependent instructions. Comm. ACM. 21, 4 (April 1978), pp. 300-308.

[23] Ullman, J.D. Data flow analysis. Second USA-Japan Computer. Conf., (1975), pp. 335-342.

[24] Wulf, W.A., et al. The Design of an Optimizing Compiler.

[25] Zelkowitz, M.V., and Bail, W.G. Optimization of structured programs, Software and Practice 4 (1974), pp. 51-57.