How Hard is Compiler Code Generation?

Alfred V. Aho and Ravi Sethi
Bell Laboratories
Murray Hill, New Jersey

1. Introduction

Over the past two decades great strides have been made in understanding how to design lexical and syntactic analyzers for programming-language compilers. Theory and practice have proceeded to the point where usable lexical and syntactic analyzers can be generated automatically from notations based on regular expressions and context-free grammars [Aho and Ullman (1977), Johnson (1975), Lesk (1975)].

Unfortunately, our understanding of code generation has not kept pace with the developments in the syntactic domain. Recently, however, a number of theoretical results have been obtained which suggest how well and how extensively code generation can be automated. This paper summarizes these results and discusses the problems that still remain to be solved.

2. The Problem

A program in a high-level language, by its very nature, does not specify the routine, hardware-specific computations needed to implement the program on a given machine; it is the function of the compiler to supply these details. In translating a source program into machine language, many compilers first translate the source program into an intermediate form, which is then subsequently transformed into the final object program. This paper discusses some of the difficulties of translating the intermediate-language program into machine code.

Intermediate Code

We assume that an *intermediate program* is a sequence of primitive statements such as:

A:= B **op** C
A:= **uop** C
A:= B
A:= B[C]
A[B]:= C
return A
goto L
if (condition) **goto** L

where condition is a Boolean variable, or a simple relational expression of the form A **relop** B.

The intermediate code represents a *flow graph* such as the one in Fig. 2.1, where each node represents a *straight-line (basic) block*—a sequence of statements in which control enters at the top and leaves at the bottom. Thus a branch statement can occur only at the end of a basic block.

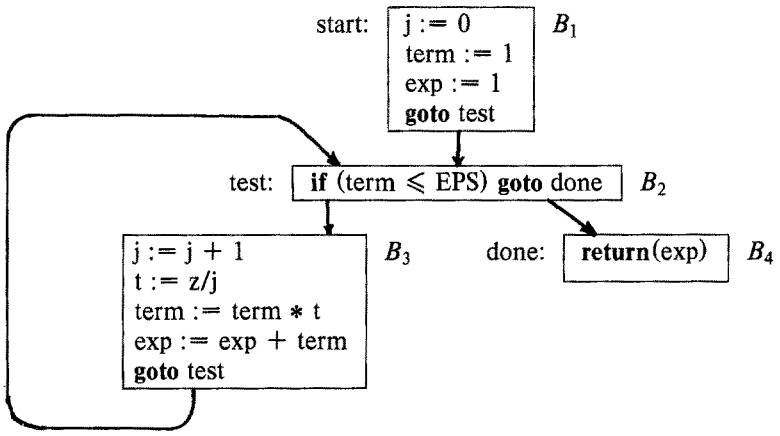


Fig. 2.1. Flow graph for a program computing e^z, $z \geqslant 0$, using

$$e^z = 1 + \frac{z}{1!} + \frac{z^2}{2!} + \frac{z^3}{3!} + \cdots$$

EPS is some constant to be specified.

Each basic block in a flow graph can be represented by a *dag* (directed acyclic graph) in which each interior node represents an operation and each leaf represents an initial value [Aho and Ullman (1972)]. A dag represents the set of expressions computed in a basic block, making explicit the partial order that must be satisfied by the source program. A compiler is free to evaluate the expressions in a basic block in any order consonant with this partial order. Unconditional gotos are not represented in the dag, and only the condition of a conditional goto is represented. The dag for block B3 from Fig. 2.1 is shown in Fig. 2.2. This dag happens to be a tree

The overall problem of code generation is to start with a flow graph and construct a machine-language program. We would like the resulting machine-language program to be optimal, but even if we qualify the word "optimal" suitably, we are far from being able to solve the overall problem completely. Some of the difficulties arise from the fact that generating optimal code from dags is an NP-complete problem for any realistic machine. Additional difficulties arise from the fact that target machines often have special hardware features that are hard to handle analytically.

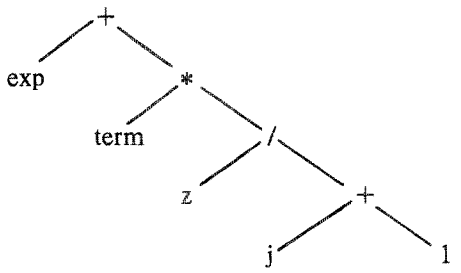


Fig. 2.2. The dag for the expression exp + term * (z/(j+1)).

Machine Models

A variety of machine models have been considered in the literature. Here we shall consider only the register-machine model and provide references for some of the other models. An *N-register machine* consists of a sequence of memory locations m_0, m_1, ... in which values can be stored, and a set of registers r_0, r_1, ..., r_{N-1} in which all computations are performed. The following types of instructions are available:

1. LOAD r, m
2. STORE r, m
3. $r \leftarrow r$ **op** m
4. $r \leftarrow r$ **op** r'

Here, r, r' are registers, m is a memory location or literal, and **op** is a machine-code operation.

Example 2.1. The following register-machine program computes the tree of Fig. 2.2.

1. LOAD r_1, j
2. $r_1 \leftarrow r_1 + 1$
3. LOAD r_2, z
4. $r_2 \leftarrow r_2 / r_1$
5. LOAD r_1, term
6. $r_1 \leftarrow r_1 * r_2$
7. LOAD r_2, exp
8. $r_2 \leftarrow r_2 + r_1$

Variations on the above instructions are possible. For example, we might have instructions of the form

$r_i \leftarrow r_j$ **op** r_i, or
$r_i \leftarrow r_j$ **op** r_k

Register machines have been considered in a number of papers, including Ershov

[1958], Anderson [1964], Nakata [1967], Redziejowski [1969], Sethi and Ullman [1970], Chroust [1971], Schneider [1971], Wasilew [1971], Beatty [1972], Stockhausen [1973], Aho and Johnson [1976] consider a class of machines that includes these models.

Another popular machine model is the stack machine, considered by Bruno and Lassagne [1975], and by Prabhala and Sethi [1977]. Finally, register-machine models in which each value may occupy either a one or two registers are considered by Aho, Johnson, and Ullman [1977a].

Some Questions

No matter what machine model is used, a number of fundamental questions arise. Here are a few that have received some theoretical attention.

1. *Automatic code generation.* Devise an algorithm that takes as input a description of a machine and delivers as output a good code generator for that machine. Although a number of preliminary reports have been issued on this subject [Miller (1970), Donegan (1973), Newcomer (1975)], practical automatic code generation is still beyond our present capabilities. We are much better prepared for the following:

2. *Automatic code generation from expression trees.* For expression trees we can generate optimal code for certain classes of machines. However, even for this restricted problem we have trouble generating optimal code for more complicated machines, such as even-odd register-pair machines which require double-length values to reside in even-odd register pairs.

3. *Common subexpressions.* One code-improvement technique is to identify common computations so they need to be performed only once. This process can complicate the process of code generation in that if common subexpressions in expression trees are merged to give dags, then generation of optimal code becomes an NP-complete problem on typical machines.

4. *Code generation in the presence of flow of control.* Good code generation requires knowing how to properly allocate machine resources during a computation. Assignment of registers to hold frequently-accessed values is one example of this kind of resource allocation problem. Since virtually all programs spend most of their time in loops, knowing the looping structure of a program is useful. A number of code-generation problems are concerned with the detection of loops in programs and the assignment of registers across loops.

5. *What is the best machine from a code-generation standpoint?* This is perhaps the most interesting question of all. Given a programming language, how do we design a machine for which we can generate efficient code efficiently? Notice that there are two parts to this problem. We want the code generator to produce object code that utilizes the target machine in an efficient manner. We also want the code generator itself to be efficient. Both questions have received relatively little attention.

3. Expression Trees

Consider, for the moment, an intermediate language in which every value can be held in a single register and every operator can be executed by a machine operation code. For such expression trees we can generate optimal code for register machines in time linearly proportional to the size (number of nodes) of the tree.

Consider the tree in Fig. 3.1. Once we bring a value from, say, the subtree for y into a register, then we may as well continue working on the subtree for y until a store occurs.

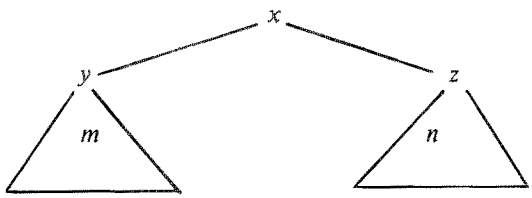


Fig. 3.1. On a register machine, if we compute the subtree for y using m registers, hold the value of y, and then compute z, we need $\max(m, n+1)$ registers.

For a large class of machine models Aho and Johnson [1976] prove a "normal-form theorem" to the effect that there is an optimal program which computes an entire subtree of a node before starting work on another subtree of that node.

More precisely, a program $P = P_1P_2 \ldots P_kQ$, is in *normal form* if P has no "useless" instructions, and

(i) STORE's occur exactly at the end of each of the program fragments $P_1,P_2,\ldots,P_k$, and all instructions in P_i compute portions of the subtree for the node stored by the last instruction in P_i.

(ii) Within P_i and Q the instructions for computing any subtree are contiguous.

In Fig. 3.1 let m and n be the least number of registers required to compute the subtrees for y and z, respectively. From the normal-form theorem, we know that we can compute either y before z or z before y in an optimal program. Ershov [1958] notes that we need $max(m,n)$ registers for x if $m \neq n$ and $m+1$ registers for x if $m=n$. (Nakata [1967] has a similar observation; Redziejowski [1969] supplies a proof.) Basically, there are two possibilities for computing the subtrees of x and we must consider both of them.

We can *label* each node with an integer which gives the least number of registers required to compute the node using a *simple N-register machine* having LOAD, STORE, and the following instructions

$r \leftarrow r$ **op** m
$r_i \leftarrow r_j$ **op** r_k

(It is important to note that while the right operand may be in a memory location,

the left operand must be in a register.) The labeling algorithm proceeds as follows: Label left leaves by 1 and right leaves by 0. For a nonleaf node the label is the greater of the labels of its children if these labels are unequal, and one greater than the label of its children otherwise.

Incidentally, on most machines, if we computer first one subtree then the other, we are treating the registers as a stack. Prabhala and Sethi [1977] show that the labeling scheme can be parameterized to work over a class of stack machines.

What to Store?

While it is relatively easy to determine the minimal number of registers needed to compute a node, it is not always obvious what should be done when we run out of registers. Clearly some node has to be stored, but which one?

The Dynamic Programming Algorithm

A general technique for finding optimal programs, including nodes to be stored, can be based on dynamic programming [Aho and Johnson (1976)]. This technique is applicable to a wide class of register machines in which each instruction is of the form $r \leftarrow E$, where E is some expression in which each leaf refers to a register or to a memory location. A typical example of such an instruction is $r \leftarrow r + \mathbf{ind}(m)$ which, graphically depicted, is:

This instruction results in the contents of memory location m' being added to register r, where m' is the value contained in location m. Instructions such as $r_i \leftarrow r_i + r_j$ and $r \leftarrow r + m$ also fit this model.

For the tree in Fig. 2.2 the last instruction can be $r_i \leftarrow r_i + r_j$ or $r \leftarrow r + m$, or any other instruction with + at the root that "covers" the tree of Fig. 3.2 An instruction $r \leftarrow E$ is said to *cover* a tree T if by pruning some of the subtrees of T we get a tree differing from E at the leaves only. The instruction $r \leftarrow r + m$ covers the tree in Fig. 2.2 The instruction also specifies that the left son of the root should appear in a register and the right son should appear in memory. With instruction $r_i \leftarrow r_i + r_j$, which also covers the tree, the left and right sons of the root should both appear in registers.

We can generate an optimal program by considering all possibilities systematically. Look at all instructions that cover a tree. Each instruction leaves us with a set of subtrees that have to be computed into registers or memory as specified by the instruction.

Dynamic programming allows us to check all the possibilities mentioned in the above paragraph. Working up from the leaves, for each node n we determine the minimum cost of computing the node into a register, using i registers, or into memory using all registers. For instruction $r \leftarrow E$, match the root of E with the

subtree at node n, looking for subtrees that match the leaves of E. From the normal-form theorem any subtrees matching leaves labeled m in E can be precomputed into memory using all registers for the computation. Let the remaining subtrees be $s_1, s_2, \ldots, s_j$. These subtrees must be computed into registers.

For all permutations π of $1, 2, \ldots, j$, sum the cost of computing $s_{\pi(1)}$ with i registers, $s_{\pi(2)}$ with $i-1$ registers, and so on. Picking the cost for the cheapest permutation gives us the least cost for using $r \leftarrow E$ to compute node n. Picking the cheapest instruction among all instructions covering the subtree at node n, we can find the least cost for computing node n with i registers.

Treating the number of registers and the number of instructions as a constant, the amount of work done by the dynamic programming approach is linear in the number of nodes in the tree. The constant of proportionality depends on the set of instructions.

The advantage of the dynamic programming algorithm is that it can be used as a universal code generation technique; it can be used to produce optimal code for any machine in which all registers are interchangeable. (Knuth [1977] provides a generalization to classes of symmetric registers.) The disadvantage is that the constant of proportionality can be high and, more seriously, that it cannot handle noninterchangeable registers, such as even-odd register pairs.

An Algorithm for Simple Register Machines

If we are given the instruction repertoire of a machine in advance, we may be able to significantly reduce the amount of time needed to generate optimal code. By way of example, consider a simple N-register machine with LOAD, STORE, and instructions of the form:

$r \leftarrow r$ **op** m
$r_i \leftarrow r_j$ **op** r_k

Sethi and Ullman [1970] show how an integer label, giving the number of registers needed to compute the node without stores, can be used to locate nodes to be stored. The idea is to find the lowest node x such that left son y and right son z of x both require all N registers. Precompute the right subtree for z using all N registers. Form a new tree T' by replacing node z by a leaf. Repeat the process on T'. This procedure will construct optimal code for all trees on the above machine in linear time.

Once we have a procedure for generating code as above we can take advantage of algebraic laws to reduce the "cost" of a tree. Sethi and Ullman [1970] and Beatty [1972] show how the usual algebraic laws like associativity and commutativity can be accommodated. The distributive law has not been handled adequately.

Bouncing

With the increasing use of small word-length machines, it is important to treat efficiently multiple-precision quantities that take two or more registers. Unfortunately, the normal-form theorem may not apply in this case, and an optimal program can "bounce" back and forth between subtrees of a node, computing a little

of each subtree on each bounce. The number of bounces depends on the form of double-register instructions. For machines in which a double quantity can occupy any pair of registers (the unrestricted register-pair machine) the number of bounces in an optimal program can be limited to two, and a modified linear-time dynamic programming algorithm can be developed. For even-odd register pair machines, however, the number of bounces in an optimal program can be proportional to the size of the tree, and no polynomial-time optimal code-generation algorithm is known. We refer the reader to [Aho, Johnson, and Ullman (1977a)] for details.

4. Dags

When an expression contains common subexpressions, the dag for the expression will no longer be a tree, but will contain a shared node for each common subexpression. Dags can be constructed relatively easily. In practice, the "value number" method [Cocke and Schwartz (1970), Aho and Ullman (1972)] can construct the dag for an expression in linear time on the average.

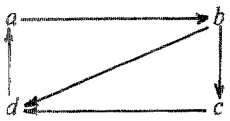
Height-One Dags

Although it is easy to construct a dag for an expression, generating optimal code from a dag is considerably harder. Bruno and Sethi [1976] show that the problem of generating optimal code from a dag for a one-register machine is NP-complete. Aho, Johnson, and Ullman [1977b] show that the problem remains NP-complete even if the only sharing is at height-one nodes (nodes whose descendants are leaves).

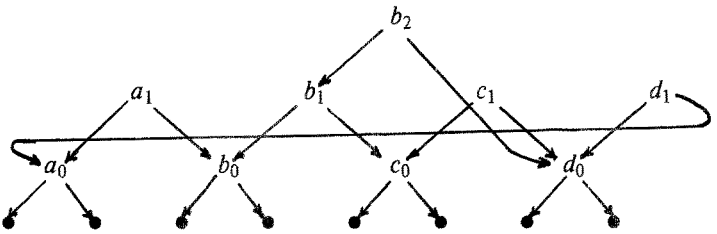
We can easily transform an instance of the well-known NP-complete problem, feedback node set (FNS), to an instance of optimal code generation for one-register machines. The FNS problem is: Given a directed graph G, find a smallest set of nodes F (a feedback node set) such that removing F from G eliminates all cycles from G.

Given the instance G of FNS, we can construct a dag D representing an expression to be evaluated as in Fig. 4.1. D has a height 1 node x_0 for each node x of G. At node x of G number all the edges leaving x in some order. If there are j edges leaving x, construct a left chain $x_1, x_2, \ldots, x_j$, where x_{i-1} is the left son of x_i, $1 \leqslant i \leqslant j$. If the ith edge leaving x in G is (x, y), then make y_0 a right son of x_i.

It can now be shown that we can construct a minimal feedback node set F for G from an optimal program P for D, and conversely, thereby establishing that optimal code generation is NP-complete.



Graph G



Corresponding dag D.

Fig. 4.1. Removing node b breaks all cycles in G.
Precomputing and storing node b_0 in D results in an optimal program.

Infinite-Register Machines

The same proof technique can be used to show that even if the number of registers is unlimited, optimal code generation remains NP-complete. That is, consider a dag in which the only sharing is of leaves, which we assume are labeled by register names rather than by memory locations. The only machine instructions are of the form

$$r_i \leftarrow r_i \textbf{ op } r_j$$
$$r_i \leftarrow r_j$$

Deciding an evaluation order for a dag in which the number of register copy instructions is minimized is NP-complete.

Heuristic Techniques

As with any NP-complete problem, there are two possible approaches to finding reasonable solutions in practice:

1) Find heuristic techniques that give approximate, but not necessarily optimal, answers.

2) Find useful special cases for which exact polynomial-time algorithms can be found.

Both approaches have been used on the code-generation problem for dags. One heuristic which is often used in practice [Waite (1974)] is to transform a dag

into a forest of trees in which each multiply-used interior node is made into a root. Multiply-used leaves are treated as distinct nodes. Each tree in this forest can be evaluated optimally and stored.

Another heuristic suggested for dags is the top-down greedy algorithm [Aho and Ullman (1973), Aho, Johnson, and Ullman (1977b)]. Here a dag is partitioned into a sequence of "left chains" (sequences of left children), which are evaluated bottom-up. It can be shown that this approach never generates code that is more than 3/2 times optimal.

As an example of a special case for which a polynomial-time algorithm can be developed, for a one-register machine there is an algorithm that is linear in the size of a dag and exponential only in the amount of sharing [Aho, Johnson, and Ullman (1977b)].

Series-Parallel Graphs

Most of the dags that occur in practice have a fairly simple structure. Complexity results that apply to all dags are therefore not necessarily applicable to dags that appear in practice. A subclass of dags that may be more representative is the class of series-parallel graphs (see Simon and Lee [1971] for a definition). Abdel Wahab and Kameda [1977] consider a scheduling problem that is related to code generation. Their results imply that the minimal number of registers needed to compute a series-parallel graph can be found in $O(n\log n)$ time. However, much remains to be done to solve the code generation problem for series-parallel graphs.

Related Problems

Code generation is really one instance of a very general resource allocation problem. Consider a *pebble game* defined as follows. We are given a dag D and a supply of pebbles. The rules of the game are:

1) a pebble can always be placed on a leaf.
2) if all sons of a node x have pebbles on them, then a pebble can be placed on node x.
3) a pebble can be removed from a node at any time.

Notice that the rules permit a pebble to be placed on a node more than once. In the code-generation framework we place a pebble exactly once on each node. We invite the reader to simulate the distinctions between registers and memory by using pebbles of different colors.

The pebble game was used by Paterson and Hewitt [1970] and Walker and Strong [1973] to study the connection between flowchart schemes and recursive schemes. In a flowchart, the number of locations to which a computation can refer is fixed. However, Paterson and Hewitt [1970] show that there exist recursive schemes for which there is no bound on the number of locations that might be referred to during the computation of the scheme, thereby establishing that there are recursive schemes that are not flowchartable.

Hopcroft, Paul, and Valiant [1975] use pebble games to show that space is a strictly more powerful resource than time for multitape Turing machines. Their

basic idea is to represent a Turing machine computation taking time t by a dag D whose size depends on t. They relate the number of pebbles needed by dag D to the amount of space needed to perform the same Turing-machine computation. Since a dag with n nodes can be pebbled using $O(n/\log n)$ pebbles, it can be shown that $\text{DTIME}(t\log t) \leqslant \text{DSPACE}(t)$. Paul, Tarjan, and Celoni [1976] have shown that there exist n-node dags for which $O(n/\log n)$ is a lower bound on the number of pebbles needed. Most pebbling schemes that try to use few pebbles do a lot of recomputation.

While we know that $O(n/\log n)$ pebbles are enough, Sethi [1975] shows that it is at least NP-hard to determine if say k pebbles are enough. This problem is widely suspected to be PSPACE-complete, but the proof is elusive.

Evidence that a lot of pebbles are needed to compute certain dags is used by Cook [1974] and Cook and Sethi [1976] to support Cook's conjecture that $O((\log n)^k)$ space, for any k, is not enough to recognize languages recognizable in polynomial time.

5. Loops

So far we have considered only the evaluation of straight-line blocks. Once we consider the evaluation of an entire flow graph, a host of additional questions arise. Many of these problems concern loops. Since most programs spend most of their time in relatively small portions of the code, it is reasonable to try to identify these heavily-traveled regions (called the inner loops) of a program and to generate as good code for these regions as possible.

We shall consider here the problem of how to assign values to registers across loops (the *register-assignment problem*). Fortran H uses the rather simple technique of noting which variables, constants, and base addresses are referenced most frequently within a loop and assigning as many of them as possible to individual registers across the loop [Lowry and Medlock (1969)].

A common simplifying assumption made in the register-assignment problem is to fix in advance the order of evaluation for the nodes of a dag. This order of evaluation may well have been determined using the techniques discussed in the last two sections. Beatty [1974] starts with such an evaluation order and partitions the register-assignment problem into two phases. The first phase determines whether a value should be in a register or in memory at any program point; the second phase decides in which registers values are to reside. Day [1970], Harrison [1975], Yhap [1975], and Kim and Tan [1976] investigate various aspects of the register-assignment problem.

Returning to basic blocks, suppose that we are given an evaluation order for the nodes of a basic block. How hard is it to find an optimal program that uses this evaluation order? The answer to this question depends on the machine model. Marill [1962] gives a simple algorithm that works for interchangeable registers. Horowitz *et al.* [1966] minimize usage of index-registers—a problem that is perhaps best viewed as a paging problem. At any time we may have at most M pages in memory. The evaluation order specifies the order in which pages will be referenced or changed. A page that is not changed need not be stored back into memory. Horowitz *et al.* give an algorithm for minimizing the number of times a

page is loaded or stored. Luccio [1967] suggests improvements, but the overall algorithm is still inherently exponential. Kennedy [1972] extends the results of Horowitz *et al.*

If we change the cost criterion and do not differentiate between referencing or changing a page, then Belady [1966] provides a linear algorithm: when necessary, store the page whose next reference is furthest away.

To extend these register-assignment strategies to flow graphs, we run into the problem of not knowing which branches will actually be taken. Beatty [1974] describes one approach to this problem in which the number of intervening instructions between references is used in a heuristic.

6. Conclusions

In this paper we have considered only a small number of the many problems that arise in the design of a code generator for a major programming language. We have seen that some of the factors that affect the difficulty of generating good code are the characteristics of the intermediate language and the register structure of the underlying machine.

What of the Future?

An important side-effect of studying code generation is the insight it casts upon machine design. We have seen that some language constructs are difficult to implement efficiently even on the simplest of machines. We have also seen that even expression trees are difficult to implement on certain kinds of register machines.

It is quite apparent that it is much harder to generate good code for the complicated machines currently used than the simple theoretical machines considered in the literature. We suspect, therefore, that it will be a long while before we have the understanding to design a universal code-generator generator that can deliver production-quality code generators for the diverse machines available today. Nevertheless, we feel that it is important to bring theory and practice closer together. By developing more powerful theoretical tools and by carrying over some of the attractive features of theoretical models into concrete machine designs, we can foresee new machine designs for which it will be easier to devise good code generators automatically. It is this goal that we hope will be nourished by a study of code generation.

Bibliography

Abdel-Wahab, H. M., and T. Kameda [1977]. Scheduling to minimize maximum cumulative cost subject to series-parallel precedence constraints, *Operations Research,* to appear.

Aho, A. V., J. E. Hopcroft, and J. D. Ullman [1974]. *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, Mass.

Aho, A. V., and S. C. Johnson [1976]. Optimal code generation for expression trees, *J. ACM* **23:3**, 488-501.

Aho, A. V., S. C. Johnson, and J. D. Ullman [1977a]. Code generation for machines with multiregister operations, *Proc. Fourth ACM Symposium on Principles of Programming Languages,* pp. 21-28.

Aho, A. V., S. C. Johnson, and J. D. Ullman [1977b]. Code generation for expressions with common subexpressions, *J. ACM* **24:1**, 146-160.

Aho, A. V., and J. D. Ullman [1972]. Optimization of straight-line programs, *SIAM J. Computing* **1:1**, 1-19.

Aho, A. V., and J. D. Ullman [1973]. *The Theory of Parsing, Translation, and Compiling.* Vol. 2: *Compiling,* Prentice-Hall, Englewood Cliffs, N. J.

Aho, A. V., and J. D. Ullman [1977]. *Principles of Compiler Design,* Addison-Wesley, Reading, Mass.

Anderson, J. P. [1964]. A note on some compiling algorithms. *Comm. ACM* **7:3**, 149-150.

Beatty, J. C. [1972]. An axiomatic approach to code optimization for expressions, *J. ACM* **19:4**, 613-640. *Errata,* **20:1** (Jan. 1973) 188, and **20:3** (July 1973) 538.

Beatty, J. C. [1974]. Register assignment algorithm for generation of highly optimized object code, *IBM J. Res. Develop.* **18:1**, 20-39.

Belady, L. A. [1966]. A study of replacement algorithms for a virtual storage computer, *IBM Systems J.* **5:2**, 78-101.

Bruno, J., and T. Lassagne [1975]. The generation of optimal code for stack machines, *J. ACM* **22:3**, 382-396.

Bruno, J., and R. Sethi [1976]. Code generation for a one-register machine, *J. ACM* **23:3**, 502-510.

Chroust, G. [1971]. Scope conserving expression evaluation, IFIP71, TA-3, 178-182.

Cocke, J., and J. T. Schwartz [1970]. *Programming Languages and Their Compilers, Preliminary Notes, Second Revised Version,* Courant Institute of Mathematical Sciences, New York.

Cook, S. A. [1974]. An observation on time-storage tradeoff, *JCSS* **9:3**, 308-316.

Cook, S. A., and R. Sethi [1976]. Storage requirements for deterministic polynomial time recognizable languages, *JCSS* **13:1**, 25-37.

Day, W. H. E. [1970]. Compiler assignment of data items to registers, *IBM Systems J.* **9:4**, 281-317.

Donegan, M. K. [1973]. An approach to the automatic generation of code generators, Laboratory for Computer Science and Engineering, Rice University, Houston, Texas.

Ershov, A. P. [1958]. On programming of arithmetic operations, *Comm. ACM* **1:8**, 3-6.

Harrison, W. [1975]. A class of register allocation algorithms, RC-5342, IBM Thomas J. Watson Research Center, Yorktown Heights, New York.

Hopcroft, J. E., W. J. Paul, and L. G. Valiant [1975]. On time versus space and related problems, *Proc. 16th Annual Symposium on Foundations of Computer Science,* pp. 57-64.

Horowitz, L. P., R. M. Karp, R. E. Miller, and S. Winograd [1966]. Index register allocation, *J. ACM* **13:1**, 43-61.

Johnson, S. C. [1975]. YACC—yet another compiler-compiler, CSTR-32, Bell Laboratories, Murray Hill, N. J.

Kennedy, K. W. [1972]. Index register allocation in straight-line code and simple loops, in Rustin, R. (ed.) *Design and Optimization of Compilers,* Prentice-Hall, Englewood Cliffs, N. J., pp. 51-64.

Kim, J., and C. J. Tan [1976]. Register assignment algorithm - II, Report RC 6262, IBM Thomas J. Watson Research Center, Yorktown Heights, New York.

Knuth, D. E. [1977]. A generalization of Dijkstra's algorithm, Dept. of Computer Science, Stanford University.

Lesk, M. E. [1975]. LEX—a lexical analyzer generator, CSTR-39, Bell Laboratories, Murray Hill, N. J.

Lowry, E. S., and C. W. Medlock [1969]. Object code optimization, *Comm. ACM* **12:1**, 13-22.

Luccio, F. [1969] A comment on index register allocation, *Comm. ACM* **10:9**, 572-574.

Marrill, T. [1962]. Computational chains and the simplification of computer programs, *IRE Trans. on Electronic Computers,* **EC-11:2**, 173-180.

Miller, P. L. [1970]. Automatic Code-Generation from an Object-Machine Description, MAC TM 18, Massachusetts Institute of Technology, Cambridge, Mass.

Nakata, I. [1967]. On compiling algorithms for arithmetic expressions, *Comm. ACM* **10:8**, 492-494.

Newcomer, J. M. [1975]. Machine-independent generation of optimal local code, Ph. D. Thesis, Computer Science Department, Carnegie-Mellon University.

Paterson, M. S., and C. E. Hewitt [1970]. Comparative schematology, *Record of Project MAC Conference on Concurrent Systems and Parallel Computation,* pp. 119-128.

Paul, W. J., R. E. Tarjan, and J. R. Celoni [1976]. Space bounds for a game on graphs, *Proc. 8th Annual ACM Symposium on Theory of Computing,* pp. 149-160.

Prabhala, B., and R. Sethi [1977]. A comparison of instruction sets for stack machines, *Proc. 9th Annual Symposium on Theory of Computing.*

Redziejowski, R. R. [1969]. On arithmetic expressions and trees, *Comm. ACM* **12:2**, 81-84.

Schneider, V. B. [1971]. On the number of registers needed to evaluate arithmetic expressions, *BIT* **11:1**, 84-93.

Sethi, R. [1975]. Complete register allocation problems, *SIAM J. Computing* **4:3**, 226-248.

Sethi, R., and J. D. Ullman [1970]. The generation of optimal code for arithmetic expressions, *J. ACM* **17:4**, 715-728.

Simon, R., and R. C. T. Lee [1971]. On the optimal solution to AND/OR series-parallel graphs, *J. ACM* **18:3**, 354-372.

Stockhausen, P. F. [1973]. Adapting optimal code generation for arithmetic expressions to the instruction sets available on present-day computers, *Comm. ACM* **16:6**, 353-354. *Errata,* **17:10** (Oct. 1974) 591.

Ullman, J. D. [1976]. The complexity of code generation, in J. F. Traub (ed.) *Algorithms and Complexity,* Academic Press, New York, pp. 53-70.

Waite, W. M. [1974]. Optimization, in Bauer and Eickel (eds.) *Compiler Construction: An Advanced Course,* Springer-Verlag, New York, pp. 549-602.

Walker, S. A., and H. R. Strong [1973]. Characterization of flowchartable recursions, *JCSS* **7:4**, 404-447.

Wasilew, S. G. [1971]. A compiler-writing system with optimization capabilities for complex order structures, Ph. D. Thesis, Northwestern University, Evanston, Ill.

Yhap, E. F. [1975]. General register assignment in presence of data flow, RC-5645, IBM Thomas J. Watson Research Center, Yorktown Heights, New York.