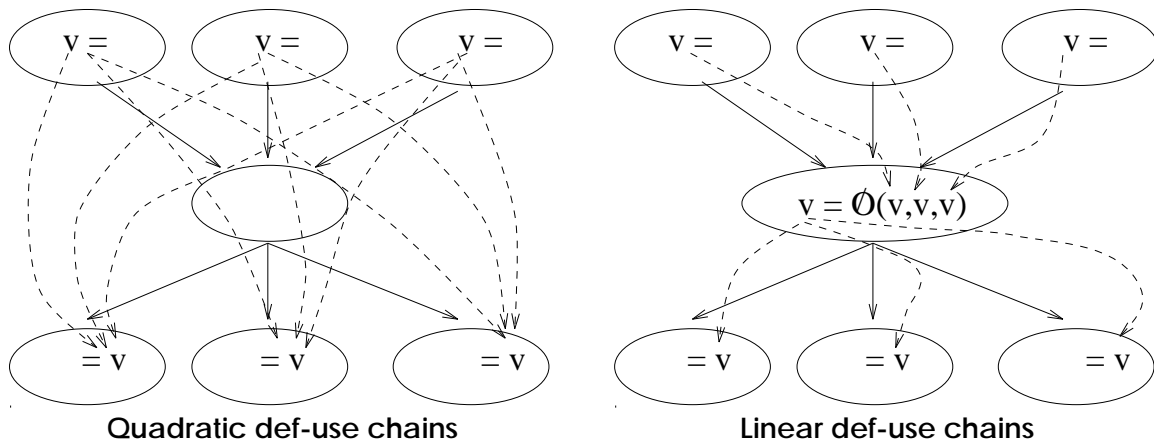


## Why is SSA good?

Data flow algorithms built on def-use chains gain asymptotic efficiency as shown below:



With each use reached by a unique def, program transformations such as code motion are simplified: motion of a use depends primarily on motion of its unique reaching def. Intuitively, the program has been transformed to represent directly the flow of values. We'll now look at some optimizations that are simplified by SSA form.

## SSA constant propagator [44]

### Original Program

```

i ← 6
j ← 1
k ← 1
repeat

  if (i = 6) then
    k ← 0
  else
    i ← i + 1
  fi

  i ← i + k
  j ← j + 1
until (i = j)
    
```

### SSA form

```

i1 ← 6
j1 ← 1
k1 ← 1
repeat

  i2 ← φ(i1, i5)
  j2 ← φ(j1, j3)
  k2 ← φ(k1, k4)
  if (i2 = 6) then
    k3 ← 0
  else
    i3 ← i2 + 1
  fi
  i4 ← φ(i2, i3)
  k4 ← φ(k3, k2)
  i5 ← i4 + k4
  j3 ← j2 + 1
until (i5 = j3)
    
```

Each name is initialized to the lattice value  $\top$ . Propagation proceeds only along edges marked *executable*. Such marking takes place when the associated condition reaches a non- $\top$  value. The value  $\top$  propagates along unexecutable edges.

## SSA constant propagator (cont'd)

### SSA Form

```

i1 ← 6
j1 ← 1
k1 ← 1
repeat
  i2 ← φ(i1, i5)
  j2 ← φ(j1, j3)
  k2 ← φ(k1, k4)
  if (i2 = 6) then
    k3 ← 0
  else
    i3 ← i2 + 1
  fi
  i4 ← φ(i2, i3)
  k4 ← φ(k3, k2)
  i5 ← i4 + k4
  j3 ← j2 + 1
until (i5 = j3)
    
```

### Pass 1

```

i1 ← 6
j1 ← 1
k1 ← 1
repeat
  i2 ← φ(i1, i5) = (6 ∧ ⊤) = 6
  j2 ← φ(j1, j3) = (1 ∧ ⊤) = 1
  k2 ← φ(k1, k4) = (1 ∧ ⊤) = 1
  if (i2 = 6) then
    k3 ← 0
  else
    /* Not executed */
  fi
  i4 ← φ(i2, i3) ⇒ (6 ∧ ⊤) = 6
  k4 ← φ(k3, k2) ⇒ (0 ∧ ⊤) = 0
  i5 ← i4 + k4 ⇒ (6 + 0) = 6
  j3 ← j2 + 1 ⇒ (1 + 1) = 2
until (i5 = j3 ⇒ (6 = 2) = false)
    
```

## SSA constant propagator (cont'd)

### Pass 1

```

i1 ← 6
j1 ← 1
k1 ← 1
repeat
  i2 ← φ(i1, i5) = (6 ∧ ⊤) = 6
  j2 ← φ(j1, j3) = (1 ∧ ⊤) = 1
  k2 ← φ(k1, k4) = (1 ∧ ⊤) = 1
  if (i2 = 6) then
    k3 ← 0
  else
    /* Not executed */
  fi
  i4 ← φ(i2, i3) ⇒ (6 ∧ ⊤) = 6
  k4 ← φ(k3, k2) ⇒ (0 ∧ ⊤) = 0
  i5 ← i4 + k4 ⇒ (6 + 0) = 6
  j3 ← j2 + 1 ⇒ (1 + 1) = 2
until (i5 = j3 ⇒ (6 = 2) = false)
    
```

### Pass 2

```

i1 ← 6
j1 ← 1
k1 ← 1
repeat
  i2 ← φ(i1, i5) = (6 ∧ 6) = 6
  j2 ← φ(j1, j3) = (1 ∧ 2) = ⊥
  k2 ← φ(k1, k4) = (1 ∧ ⊤) = ⊥
  if (i2 = 6) then
    k3 ← 0
  else
    /* Not executed */
  fi
  i4 ← φ(i2, i3) ⇒ (6 ∧ ⊤) = 6
  k4 ← φ(k3, k2) ⇒ (0 ∧ ⊤) = 0
  i5 ← i4 + k4 ⇒ (6 + 0) = 6
  j3 ← j2 + 1 ⇒ (⊥ + 1) = ⊥
until (i5 = j3 ⇒ (6 = ⊥) = ⊥)
    
```

Our solution has stabilized. Even though  $k_2$  is  $\perp$ , that value is never transmitted along the unexecuted edge to the  $\phi$  for  $k_4$ .

## SSA value numbering [3, 39]

```
a ← read()
v ← a + 2
c ← a
w ← c + 2
t ← a + 2
x ← t - 1
```

For the above program, constant propagation will fail to determine a compile-time value for  $v$  and  $w$ , because the behavior of the  $read()$  function must be captured as  $\perp$  at compile-time.

Nonetheless, we can see that  $v$  and  $w$  will hold the same value, even though we cannot determine at compile-time exactly what that value will be. Such knowledge helps us replace the computation of  $(c + 2)$  by a simple copy from  $v$ .

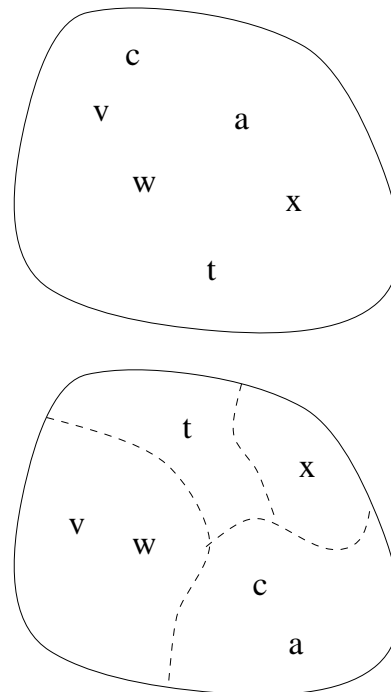
Value numbering attempts to label each computation of the program with a number, such that identical computations are identically labeled.

- Prior to SSA form, value numbering algorithms were applied only within basic blocks (i.e., no branching) [2].
- Early value numbering algorithms relied on textual equivalence to determine value equivalence. The text of each expression (and perhaps subexpression) was *hashed* to a value number. Intervening defs of variables contained in an expression would kill the expression. This approach could not detect equivalence of  $v$  and  $w$  in the example to the left, since  $(a + 2)$  is not textually equivalent to  $(c + 2)$ .

It seems that  $x$  ought to have the same value as  $v$  and  $w$ , but our algorithm won't discover this, because the "function" that computes  $x$  ( $\lambda n.n - 1$ ) differs from the "function" that computes  $v$  and  $w$  ( $\lambda n.n + 2$ ).

## SSA value numbering (cont'd)

- We essentially seek a *partition* of SSA names by value equivalence, since value equivalence is reflexive, symmetric, and transitive.
- We'll initially assume that all SSA names have the same value.
- When evidence surfaces that a given block may contain disparate values (names), we'll talk about *splitting* the block. Generally, the algorithm will only split a block in two. However, the first split is more severe, in that names are split by the functional form of the expressions that compute their value.

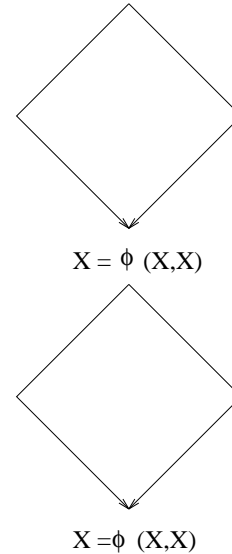
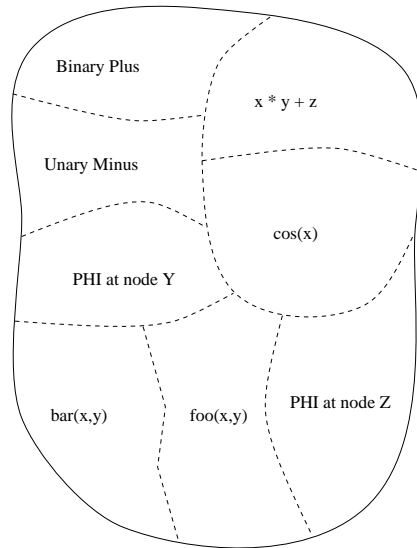


Above are shown the initial and final partitions for the example on the previous page.

## SSA value numbering (cont'd)

After construction of SSA form, we split by the function name that computes values for the assigned variables. We thus distinguish between binary addition, multiplication, etc.

One further point is that  $\phi$ -functions at different nodes must also be distinguished, even though their function form appears the same. This is necessary because a branch taken into one  $\phi$ -function is not necessarily the same branch taken into another, unless the two functions reside in the same node.



## SSA value numbering example

```

if (condA) then
  a1 ← α
  if (condB) then
    b1 ← α
  else
    a2 ← β
    b2 ← β
  fi
  a3 ← φ(a1, a2)
  b3 ← φ(b1, b2)
  c2 ← *a3
  d2 ← *b3
else
  b4 ← γ
fi
a5 ← φ(a1, a0)
b5 ← φ(b0, b4)
c3 ← *a5
d3 ← *b5
e3 ← *a5
    
```

For brevity, symbols  $\alpha$ ,  $\beta$ , and  $\gamma$  represent syntactically distinct function classes in the program shown to the left.

In the figures that follow, we'll see that  $c_2$  and  $d_2$  have the same value, while  $c_3$  and  $d_3$  do not. Thus, program optimization will save a memory fetch by using the value of  $c_2$  for  $d_2$ .

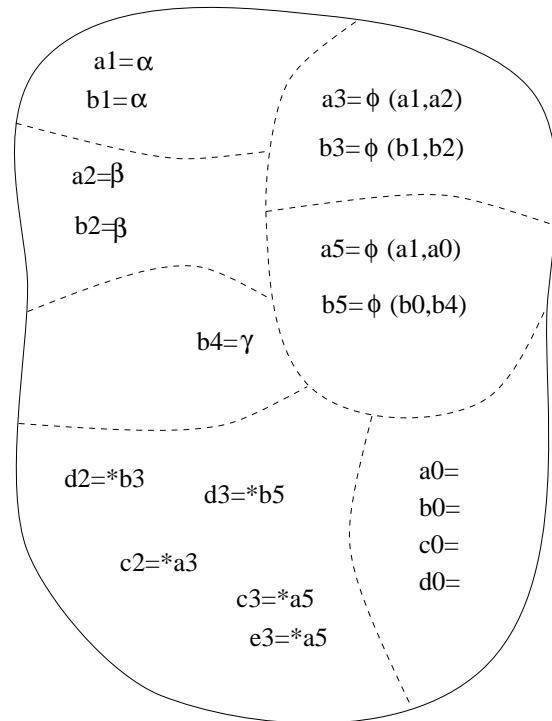
Note that if  $b$  is declared *volatile* in the language  $C$ , then this optimization would be incorrect, because each reference to  $b$  should be realized. How can one account for volatility in this optimization? Perhaps by assuming that volatile variables cannot have the same value.

It would be difficult and expensive to express all possible defs of a volatile variable in SSA form.

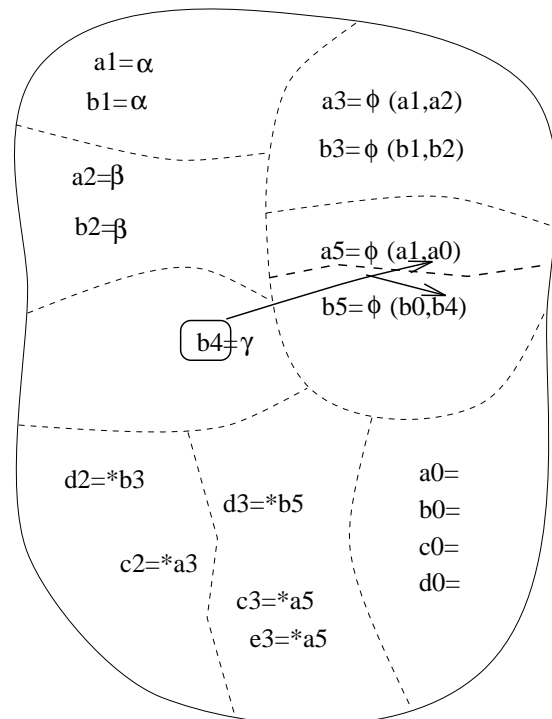
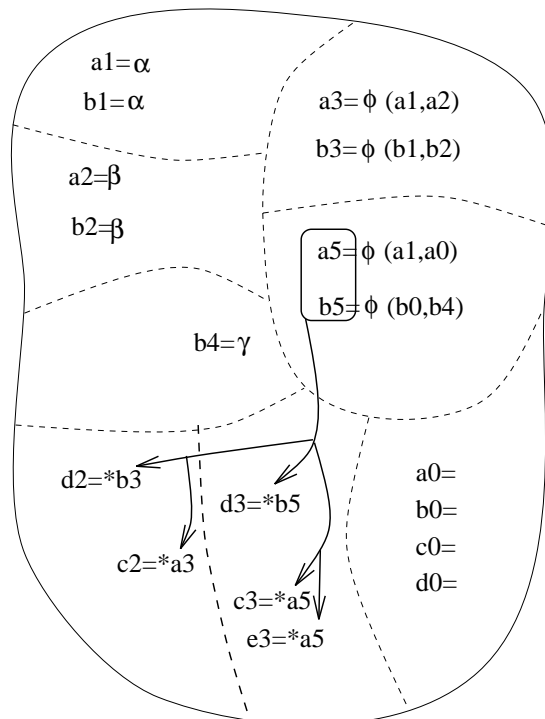
## SSA value numbering example (cont'd)

Here we see the initial partition of SSA names:

- The syntactic classes  $\alpha$ ,  $\beta$ , and  $\gamma$  are distinguished;
- $\phi$ -functions at different nodes are distinguished;
- The initial value of each variable  $v_0$  is considered identical;
- Within each syntactic class, values are considered identical.

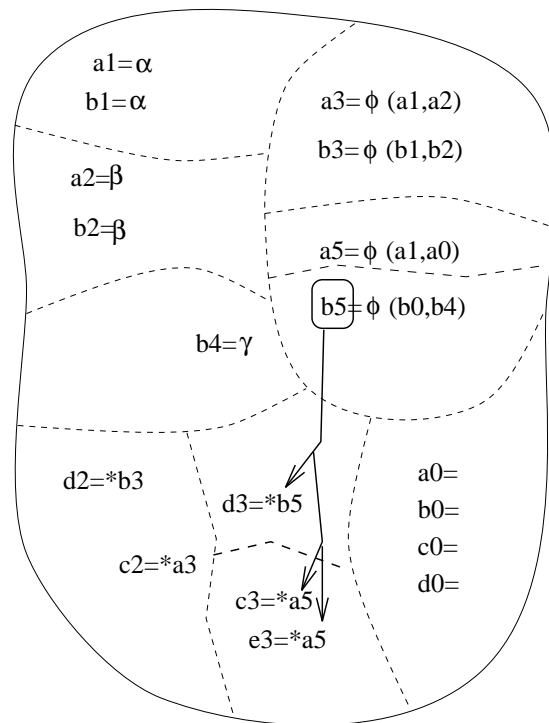


## SSA value numbering example (cont'd)



On the left, the block with  $a_5$  splits the five names shown into two subblocks; on the right,  $b_4$  splits  $a_5$  from  $b_5$ .

## SSA value numbering example (cont'd)



Finally,  $b_5$  splits  $c_3$  from  $d_3$ . Here, note that we could have used either  $a_5$  or  $b_5$  to do the job. Asymptotic efficiency is gained by choosing  $b_5$ , because there are fewer uses of that name than of  $a_5$ .

In summary, the algorithm is as follows:

1. Let  $W$  be a worklist of blocks to be used for further splitting.
2. Pick and remove (arbitrary) block  $D$  from  $W$ .
3. For each block  $C$  properly split by  $D$ ,
  - (a) If  $C$  is on  $W$ , then remove  $C$  and enqueue its splits by  $D$ ;
  - (b) Otherwise, enqueue the split with the fewest uses.
4. Loop to step 2 until  $W$  is empty.

## Register allocation

- Optimal register allocation is NP-hard.
- Trivial approaches can be really bad: using the most recently freed register is provably worst for pipelined machines.
- Many approaches begin by assuming an infinite number of *virtual registers* for assignment to values. These are then covered by actual registers during allocation.

### Chaitin-Chandra

Each variable (or expression) is assigned a virtual register for the duration of a procedure. Actual registers are allocated by coloring an interference graph, using Chandra's heuristic. Where allocation fails, some expressions are chosen for *spilling*: these are not kept in registers but loaded on demand and immediately stored afterwards.

### Chow-Hennessey

The maximum number of live variables is computed. Some register allocation can clearly succeed if there are sufficient registers to cover max-live. However, this may involve allocating the same variable to two different registers in different live ranges. This necessitates swapping registers for a given variable where control flow merges.

---

Knope and Zadeck give a method that slashes rather than spills: variables are kept intermittently in registers.

## Possible course and project coverage

I've found that if one tries to cover all the parsing methods in sequence, then the projects tend to fall behind because the necessary lectures haven't been given. My solution is to alternate between the two kinds of lectures. I've tried to develop examples that serve as glue. One such example is the grammar that structures left and right values. While this is a good introduction to type checking, the grammar also illustrates the limitations of SLR parsing.

I begin with a few warm up assignments

1. The Chinese menu problem (10 days).
2. A finite-state machine problem, such as a reserved keyword or table generator (12 days).
3. Prefix expression evaluation, by recursive descent and a simple YACC grammar (2 weeks).

And then start the sequence of assignments that leads to a finished compiler.

1. Symbol tables, starting with the C grammar (2 weeks).
2. Abstract syntax trees (10 days).
3. Semantic analysis: left and right values, type checking (10 days).
4. Preliminary code generation: simple expressions (10 days).
5. Final code generation (2-3 weeks).

## Conclusions

- Much of the work in crafting a compiler has been automated, by parser generators, tokenizing tools, attribute grammars, and automatic code generators.
- Partly due to these tools, creating the runtime library now occupies a large portion of compiler construction time.
- To understand and better appreciate the automatic tools, I believe in giving students practice in creating some components "by hand".
- It's important to get all the way through code generation in a one-semester course.
- I prefer to leave program optimization for a second course.
- In a projects course of this nature, one is often loathe to assign (or grade) homework. I have found the following strategy works well: <sup>a</sup> Give out a list of 6 problems that the students should be able to work. One week later, give one of the six problems as a quiz, where the problem is determined by the roll of a die.

---

<sup>a</sup>Thanks to Ken Goldman and Sally Goldman for this idea.

## In summary

Sung to the tune:  
"I am the very model  
of a modern Major General"

We start with some descriptions of our languages fanatical  
That specify the syntax and the attributes grammatical  
Through Yacc and Lexx our BNF is processed quite dramatical  
By front ends that we generate these parsers automatical.

They shift, reduce, and scrutinize our errors problematical  
And sometimes honest programs get transformed into the radical  
But what the heck we know our derivations are canonical

*And you'll admit our diagnostics are the most laconical.*

And you'll admit our diagnostics are the most laconical,  
Yes you'll admit our diagnostics are the most laconical,  
Because we know our diagnostics are the most laconical.

Code motion, hoisting, commoning and all the transforms you'd expect  
Your program's faster even if the output isn't quite correct.  
But most of us believe our transformations are canonical  
And you'll admit our diagnostics are the most laconical.

## Some textbooks

**Aho, Sethi, Ullman: *Compilers: principles, techniques, and tools*, Addison-Wesley, 1988 (affectionately called "The Dragon Book").** A long-time favorite in classes and as a reference book. Good coverage of popular parsing methods, though Earley's method is not presented. Good description of runtime storage organization. No real connections are given to projects, and no tools are provided (but references to extant tools are given). No real insight given on how to disambiguate a grammar.

**Fischer and LeBlanc: *Crafting a Compiler with C*, Benjamin/Cummings, 1991.** There are actually two versions of this book, one with C and one based on ADA. This is an excellent textbook (my favorite), with excellent coverage of parsing, semantic analysis, and code generation. The text meshes nicely with tools provided by the authors.

**Mason and Brown: *lex & yacc*, O'Reilly & Associates, 1992.** A good companion for the tools it covers. I list this as an optional reference book.

**Waite and Carter: *An Introduction to Compiler Construction*, HarperCollins, 1993.** A relatively new book, strong on code generation, but thin in parsing and semantic analysis. Biased toward the VAX instruction set. A consistent well-integrated text. Very weak on grammars: the text uses syntax diagrams.

**Waite and Goos: *Compiler Construction*, Springer-Verlag, 1984.** A now-dated but good text, fairly broad and not overly deep in any one area.

---

My favorite in teaching has been Fischer and LeBlanc. That book is currently undergoing revision, which will merge the language-specific versions and include much new material, for example on program optimization. I am joining them as a coauthor in this revision.

## Reference books

**Aho and Ullman:** *The Theory of Parsing, Translation, and Compiling (two volumes)*, Prentice-Hall, 1973. Not really a textbook, but an excellent reference.

**Bauer and Eickel:** *Compiler Construction: An Advanced Course*, Springer-Verlag, 1976. Notes from a course taught in 1974. A good teaching reference, but not a textbook. The chapters are separately authored.

**Hopcroft and Ullman:** *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979. A formal text on language recognition. A good reference for teaching.

**Martin:** *Introduction to Languages and the Theory of Computation*, McGraw-Hill, 1991. An excellent reference on automata theory. Terrific coverage of undecidability.

**Wirth:** *Algorithms + Data Structures = Programs*, Prentice-Hall, 1976. Really a book on other topics, but includes great coverage of recursive-descent compilers for PASCAL. Also uses syntax diagrams.

## References

- [1] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling, Volume II*. Prentice-Hall, Inc., 1973.
- [2] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [3] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of values in programs. *Conf. Rec. Fifteenth ACM Symp. on Principles of Programming Languages*, pages 1-11, January 1988.
- [4] Andrew W. Appel. Garbage collection. In Peter Lee, editor, *Topics in Advanced Language Implementation*, chapter 4. The MIT Press, 1991.
- [5] M. Auslander and M. Hopkins. An overview of the PL.8 compiler. *Proc. SIGPLAN'82 Symp. on Compiler Construction*, pages 22-31, June 1982. Published as *SIGPLAN Notices* Vol. 17, No. 6.
- [6] R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein. The Program Dependence Web: A representation supporting control-, data-, and demand driven interpretation of languages. *Proc. SIGPLAN'90 Symp. on Compiler Construction*, pages 257-271, June 1990. Published as *SIGPLAN Notices* Vol. 25, No. 6.
- [7] Keith D. Cooper, Ken Kennedy, and Linda Torczon. The impact of interprocedural analysis and optimization in the rn programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):491-523, October 1986.
- [8] R. Cytron, A. Lowry, and F. K. Zadeck. Code motion of control structures in high-level languages. *Conf. Rec. Thirteenth ACM Symp. on Principles of Programming Languages*, pages 70-85, January 1986.
- [9] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, October 1991.
- [10] Ron K. Cytron and David Shields. FRIL - a fractal intermediate language. Technical report, Washington University in St. Louis, 1993. Report number WUCS 93-51 from Washington University in St. Louis and wuarchive ftp.
- [11] J. W. Davidson and C. W. Fraser. Code selection through object code optimization. *ACM Trans. on Programming Languages and Systems*, 6(4):505-526, October 1984.
- [12] D. M. Dhamdhere. Practical adaptation of the global optimization algorithm of morel and renvoise. *ACM Trans. on Programming Languages and Systems*, 13(2):291-294, April 1991.
- [13] D. M. Dhamdhere, B. K. Rosen, and F. K. Zadeck. How to analyse large programs efficiently and informatively. *Proc. SIGPLAN'92 Symp. on Compiler Construction*, June 1992. to appear.
- [14] Dhananjay M. Dhamdhere and Uday P. Khedker. Complexity of bi-directional data flow analysis. *Conf. Rec. Twentieth ACM Symp. on Principles of Programming Languages*, 1993.
- [15] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319-349, July 1987.
- [16] Charles N. Fischer and Jr. Richard J. LeBlanc. *Crafting a Compiler with C*. Benjamin/Cummings Publishing Company, Inc., 1991.

- [17] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.
- [18] C.W. Fraser and D.R. Hanson. A retargetable compiler for ansi c. *SIGPLAN Notices*, 26(10):29–43, 1991.
- [19] C.W. Fraser, R.R. Henry, and T.A. Proebsting. Burg-fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76, 1992.
- [20] Mahadevan Ganapathi and Charles N. Fischer. Affix grammar driven code generation. *ACM Transactions on Programming Languages and Systems*, 7(4):560–599, October 1985.
- [21] R. S. Glanville and S. L. Graham. A new method for compiler code generation. *Conf. Rec. Fifth ACM Symp. on Principles of Programming Languages*, pages 231–240, January 1978.
- [22] L. H. Holley and B. K. Rosen. Qualified data flow problems. *IEEE Trans. on Software Engineering*, SE-7(1):60–78, January 1981.
- [23] Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. *SIGPLAN Notices*, 27(7), 1992. SIGPLAN PLDI Conference Proceedings.
- [24] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.
- [25] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.
- [26] M.E. Lesk and E. Schmidt. Lex—a lexical analyzer generator. In *UNIX Programmer's Manual 2*. AT&T Bell Laboratories, 1975.
- [27] T. J. Marlowe. *Data Flow Analysis and Incremental Iteration*. PhD thesis, Dept. of Computer Science, Rutgers U., October 1989.
- [28] John C. Martin. *Introduction to Languages and the Theory of Computation*. McGraw-Hill, Inc., 1991.
- [29] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *Software Engineering Notes*, 9(3), 1984. Also appears in Proceedings of the ACM Symposium on Practical Programming Development Environments, Pittsburgh, PA, April, 1984, and in SIGPLAN Notices, Vol. 19, No 5, May, 1984.
- [30] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 1994.
- [31] T. W. Reps and T. Teitelbaum. *The Synthesizer Generator*. Springer-Verlag, NY, NY, 1989.
- [32] T. W. Reps and T. Teitelbaum. *The Synthesizer Generator Reference Manual*. Springer-Verlag, NY, NY, 3rd edition, 1989.
- [33] Dennis Ritchie. A tour through the unix c compiler. Technical report, AT&T Bell Laboratories, 1979.
- [34] B. K. Rosen. High level data flow analysis. *Comm. ACM*, 20(10):712–724, October 1977.
- [35] B. K. Rosen. Data flow analysis for procedural languages. *J. ACM*, 26(2):322–344, April 1979.
- [36] B. K. Rosen. Monoids for rapid data flow analysis. *SIAM J. Computing*, 9(1):159–196, February 1980.

- [37] B. K. Rosen. Degrees of availability as an introduction to the general theory of data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis*, chapter 2, pages 55–76. Prentice Hall, 1981.
- [38] B. K. Rosen. A lubricant for data flow analysis. *SIAM J. Computing*, 11(3):493–511, August 1982.
- [39] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. *Conf. Rec. Fifteenth ACM Symp. on Principles of Programming Languages*, pages 12–27, January 1988.
- [40] Vivek Sarkar. Determining average program execution times and their variance. *Proceedings of the 1989 SIGPLAN Conference on Programming Language Design and Implementation*, 24(7):298–312, July 1989.
- [41] R. T. Teitelbaum and T. Reps. The Cornell Program Synthesizer: a syntax-directed programming environment. *Comm. ACM*, 24(9):563–573, September 1981.
- [42] G. A. Venkatesh. A framework for construction and evaluation of high-level specifications for program analysis techniques. *Proc. SIGPLAN '89 Symp. on Compiler Construction*, pages 1–12, 1989. Published as *SIGPLAN Notices* Vol. 24, No. 7.
- [43] Jr. Vincent A. Guarna, Dennis Gannon, David Jablonowski, Allen D. Malony, and Yogesh Gaur. Faust: An integrated environment for the development of parallel programs. Technical report, U. of Il.-Center for Supercomputing Research and Development, November 1988. CSR D Rpt. No. 825 to appear in *IEEE Software*, 1989.
- [44] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Trans. on Programming Languages and Systems*, 13(2):181–210, April 1991.
- [45] P. T. Zellweger. *Interactive Source-Level Debugging of Optimized Programs*. PhD thesis, Xerox Palo Alto Research Center, Palo Alto, Ca. 94304, May 1984.