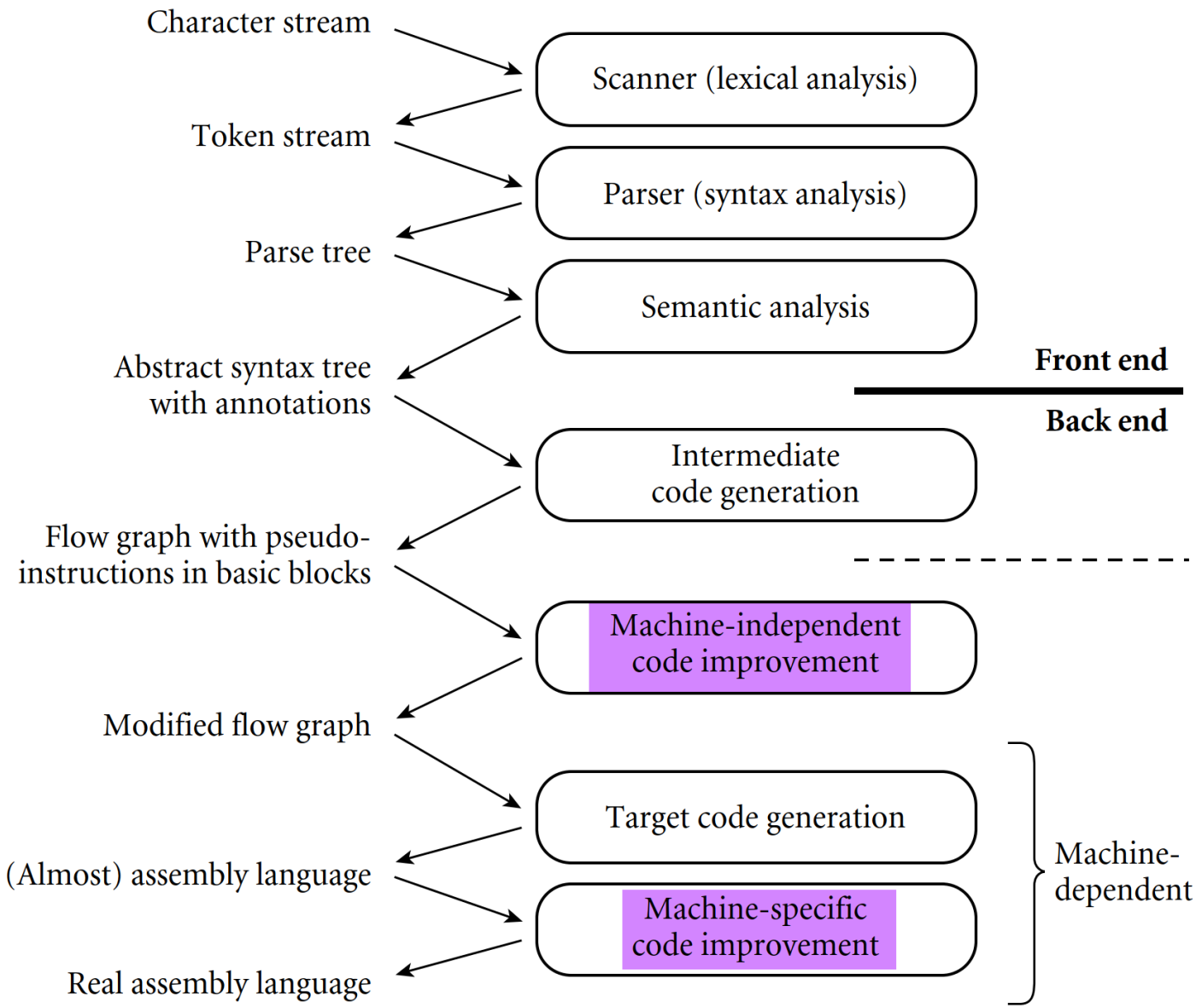
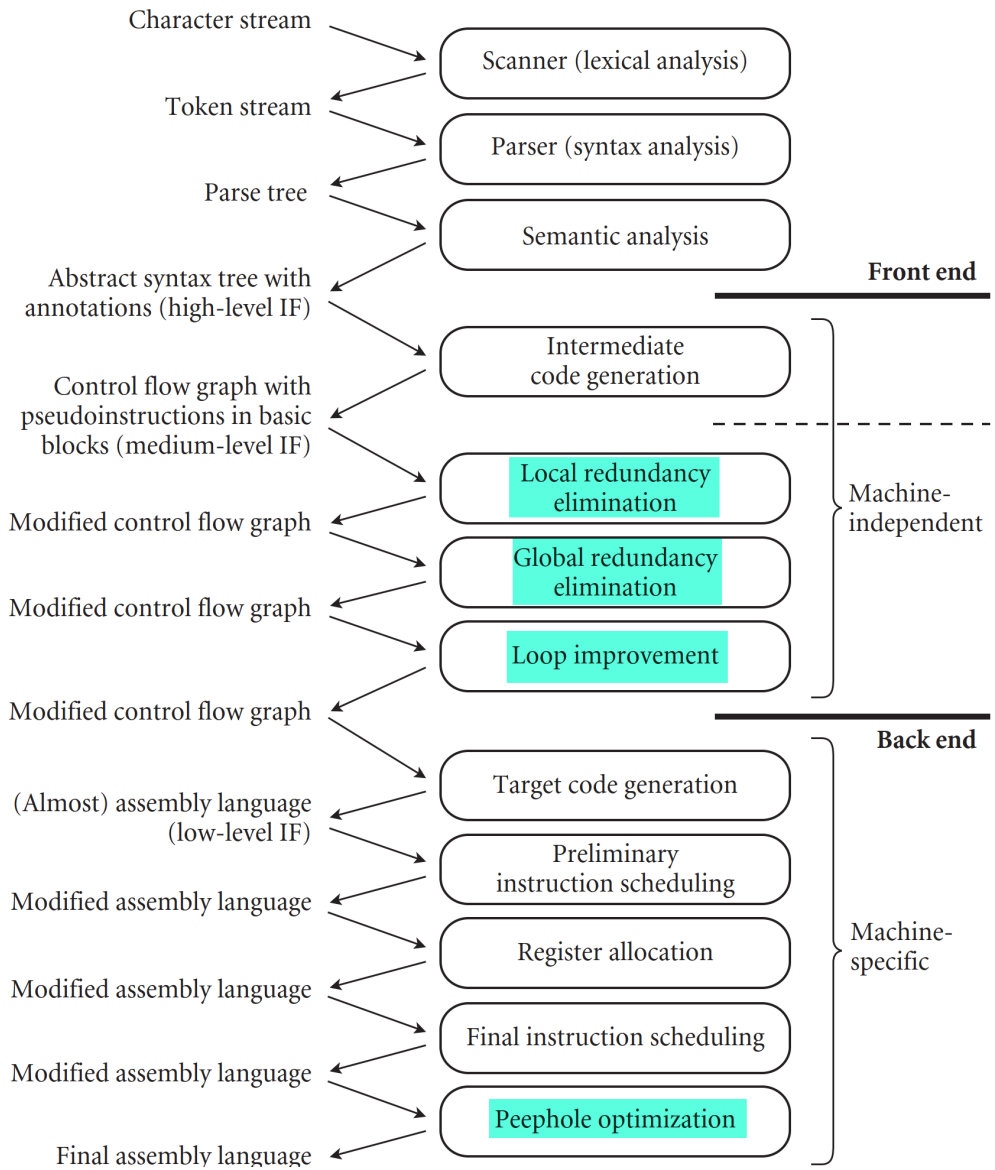
بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ــ دانشکدهٔ مهندسی برق و کامپیوتر
(نیم‌سال تحصیلی ۴۰۲۲)

کامپایلر

حسین فلسفین


Character stream
Scanner (lexical analysis)
Token stream
Parser (syntax analysis)
Parse tree
Semantic analysis
Front end
Abstract syntax tree with annotations
Back end
Intermediate code generation
Flow graph with pseudo-instructions in basic blocks
Machine-independent code improvement
Modified flow graph
Target code generation
Machine-dependent
(Almost) assembly language
Machine-specific code improvement
Real assembly language


Character stream
Scanner (lexical analysis)
Token stream
Parser (syntax analysis)
Parse tree
Semantic analysis
Front end
Abstract syntax tree with annotations (high-level IF)
Intermediate code generation
Control flow graph with pseudoinstructions in basic blocks (medium-level IF)
Local redundancy elimination
Modified control flow graph
Global redundancy elimination
Modified control flow graph
Loop improvement
Machine-independent
Modified control flow graph
Back end
Target code generation
(Almost) assembly language (low-level IF)
Preliminary instruction scheduling
Modified assembly language
Register allocation
Modified assembly language
Final instruction scheduling
Modified assembly language
Peephole optimization
Final assembly language
Machine-specific

Control Flow Graphs

The control flow graph (CFG) is a directed graph made up of nodes and edges with the edges representing possible paths of execution. Each node in the graph represents either a single instruction or a basic block.

A basic block is an instruction sequence with a single entry point at the first instruction and an exit point at the last instruction. Any jump (conditional or unconditional) instruction must be the last instruction in a block. If a block ends with a conditional jump, there are two edges coming out of the node representing that block.

The CFG is used as an IR in many compilers. It is a good basis for code optimization and it is not difficult to construct. In the following example, the code in the basic blocks has been shown in three-address code, but other representations are possible. For example, the three-address code can be converted into SSA form and then the CFG can be constructed. This approach offers further opportunities for optimization.

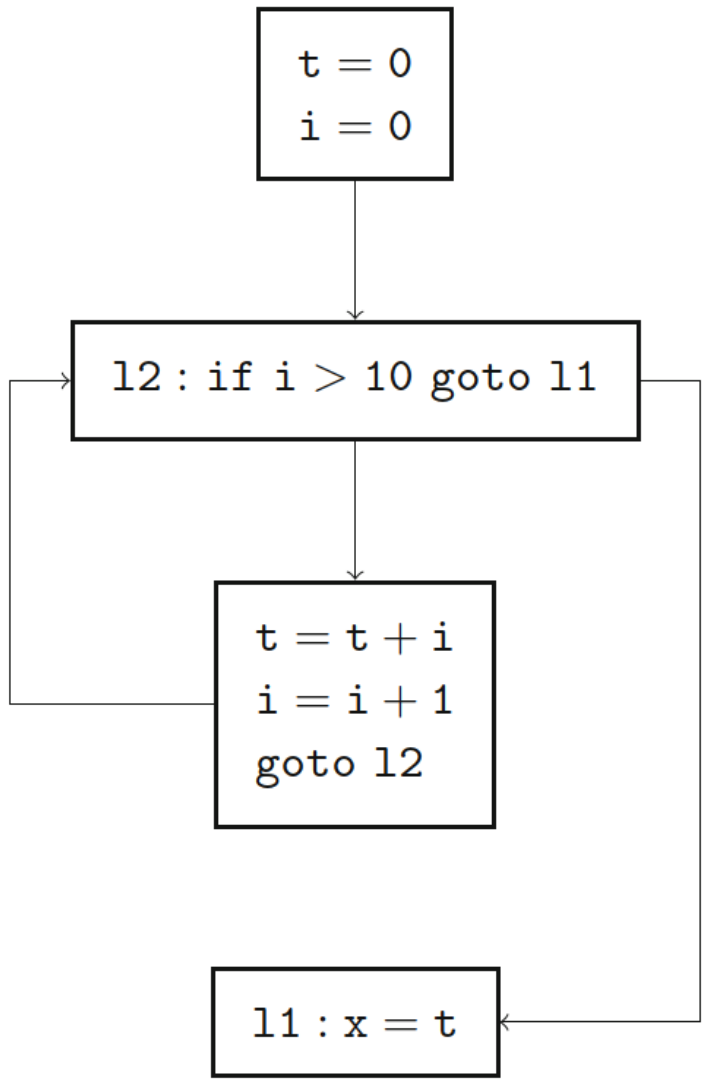
Example:

```
t = 0;
i = 0;
while (i <= 10) {
   t = t + i;
   i = i + 1;
}
x = t;
```

3AC

```
i = 0
l2:
if i > 10 goto l1
t = t + i
i = i + 1
goto l2
l1:
x = t
```

CFG



چند نکتهٔ مهم دربارهٔ روند بهینه‌سازی

☞ *It is important to remember that the term optimization when used in the context of compiler design does not refer to the search for the "best" code. Instead it is concerned with the generation of "better" code.*

☞ *Here, we concentrate on target machine-independent optimization, operating on the intermediate representation generated by the semantic analyzer.*

☞ *Machine-independent optimization can be very effective and there are many ways in which code improvements may be made at this stage of compilation. It is helpful to think about this optimization as a collection of techniques applied in turn to the intermediate representation, in effect a set of IR filters.*

☞ *Producing the CFG is often the first step in the intermediate representation optimization process. The next step is to examine each basic block individually and perform local optimization. This local optimization is completely independent of the flow of control between basic blocks.*

8.5 Optimization of Basic Blocks

We can often obtain a substantial improvement in the running time of code merely by performing local optimization within each basic block by itself. More thorough global optimization, which looks at how information flows among the basic blocks of a program, is covered in later chapters, starting with Chapter 9. It is a complex subject, with many different techniques to consider.

Local Optimization

A basic block is a maximal-length sequence of instructions that will always execute in its entirety (assuming it executes at all). In the absence of delayed branches, each basic block in assembly language or machine code begins with the target of a branch or with the instruction after a conditional branch, and ends with a branch or with the instruction before the target of a branch. As a result, in the absence of hardware exceptions, control never enters a basic block except at the beginning, and never exits except at the end. Code improvement at the level of basic blocks is known as local optimization. It focuses on the elimination of redundant operations (e.g., unnecessary loads or common subexpression calculations), and on effective instruction scheduling and register allocation.

Global Optimization

At higher levels of aggressiveness, production-quality compilers employ techniques that analyze entire subroutines for further speed improvements. These techniques are known as ***global optimization****. They include* ***multi-basic-block versions*** *of redundancy elimination, instruction scheduling, and register allocation, plus code modifications designed to improve the performance of loops. Both global redundancy elimination and loop improvement typically employ a* ***Control Flow Graph (CFG)*** *representation of the program. Both employ a family of algorithms known as data flow analysis to trace the flow of information across the boundaries between basic blocks.*

8.5.1 The DAG Representation of Basic Blocks

Many important techniques for local optimization begin by transforming a basic block into a DAG (directed acyclic graph).
In Section 6.1.1, we introduced the DAG as a representation for single expressions. The idea extends naturally to the collection of expressions that are created within one basic block.
We construct a DAG for a basic block as follows:

1. There is a node in the DAG for each of the initial values of the variables appearing in the basic block.
2. There is a node N associated with each statement s within the block. The children of N are those nodes corresponding to statements that are the last definitions, prior to s, of the operands used by s.
3. Node N is labeled by the operator applied at s, and also attached to N is the list of variables for which it is the last definition within the block.
4. Certain nodes are designated output nodes. These are the nodes whose variables are live on exit from the block; that is, their values may be used later, in another block of the flow graph. Calculation of these "live variables" is a matter for global flow analysis, discussed in Section 9.2.5.

The DAG representation of a basic block lets us perform several code-improving transformations on the code represented by the block.

a) We can eliminate local common subexpressions, that is, instructions that compute a value that has already been computed.
b) We can eliminate dead code, that is, instructions that compute a value that is never used.
c) We can reorder statements that do not depend on one another; such reordering may reduce the time a temporary value needs to be preserved in a register.
d) We can apply algebraic laws to reorder operands of three-address instructions, and sometimes thereby simplify the computation.

دو مثال از بهینه‌سازی محلی برگرفته از بخش ۸.۵ کتاب آهو

8.5.2 Finding Local Common Subexpressions

Common subexpressions can be detected by noticing, as a new node M is about to be added, whether there is an existing node N with the same children, in the same order, and with the same operator. If so, N computes the same value as M and may be used in its place. This technique was introduced as the "value-number" method of detecting common subexpressions in Section 6.1.1.

Example 8.10 : A DAG for the block

```
a = b + c
b = a - d
c = b + c
d = a - d
```

is shown in Fig. 8.12. When we construct the node for the third statement `c = b + c`, we know that the use of `b` in `b + c` refers to the node of Fig. 8.12 labeled −, because that is the most recent definition of `b`. Thus, we do not confuse the values computed at statements one and three.

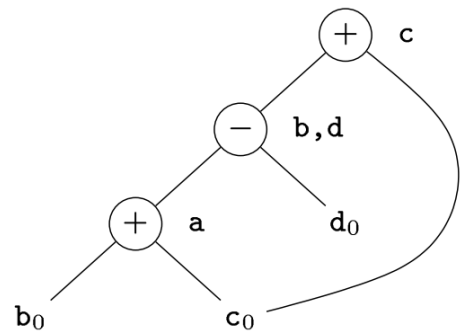


Figure 8.12: DAG for basic block in Example 8.10

However, the node corresponding to the fourth statement `d = a - d` has the operator − and the nodes with attached variables `a` and d_0 as children. Since the operator and the children are the same as those for the node corresponding to statement two, we do not create this node, but add `d` to the list of definitions for the node labeled −. □

It might appear that, since there are only three nonleaf nodes in the DAG of Fig. 8.12, the basic block in Example 8.10 can be replaced by a block with only three statements. In fact, if `b` *is not live on exit from the block, then we do not need to compute that variable, and can use* `d` *to receive the value represented by the node labeled* `−`*, in Fig. 8.12. The block then becomes*

```
a = b + c
d = a - d
c = d + c
```

However, if both `b` *and* `d` *are live on exit, then a fourth statement must be used to copy the value from one to the other.*

In general, we must be careful, when reconstructing code from DAG's, how we choose the names of variables. If a variable x is defined twice, or if it is assigned once and the initial value x_0 is also used, then we must make sure that we do not change the value of x until we have made all uses of the node whose value x previously held.

Example 8.11 : When we look for common subexpressions, we really are looking for expressions that are guaranteed to compute the same value, no matter how that value is computed. Thus, the DAG method will miss the fact that the expression computed by the first and fourth statements in the sequence

```
a = b + c
b = b - d
c = c + d
e = b + c
```

is the same, namely $\mathtt{b}_0 + \mathtt{c}_0$. That is, even though `b` and `c` both change between the first and last statements, their sum remains the same, because $b + c = (b - d) + (c + d)$. The DAG for this sequence is shown in Fig. 8.13, but does not exhibit any common subexpressions. However, algebraic identities applied to the DAG, as discussed in Section 8.5.4, may expose the equivalence. □

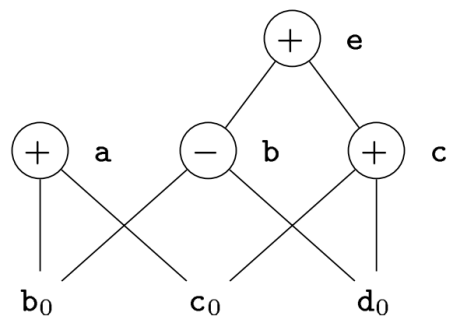


Figure 8.13: DAG for basic block in Example 8.11

شبه‌کد کتاب اپل ـ الگوریتم ۱۷.۷

$T \leftarrow empty$
$N \leftarrow 0$
for each quadruple $a \leftarrow b \oplus c$ in the block
  if $(b \mapsto k) \in T$ for some k
    $n_b \leftarrow k$
  else
    $N \leftarrow N + 1$
    $n_b \leftarrow N$
    put $b \mapsto n_b$ into T
  if $(c \mapsto k) \in T$ for some k
    $n_c \leftarrow k$
  else
    $N \leftarrow N + 1$
    $n_c \leftarrow N$
    put $c \mapsto n_c$ into T
  if $((n_b, \oplus, n_c) \mapsto m) \in T$ for some m
    put $a \mapsto m$ into T
    mark this quadruple $a \leftarrow b \oplus c$ as a common subexpression
  else
    $N \leftarrow N + 1$
    put $(n_b, \oplus, n_c) \mapsto N$ into T
    put $a \mapsto N$ into T

ALGORITHM 17.7. Value numbering.

مثال کتاب اپل ـ شکل ۱۷.۸

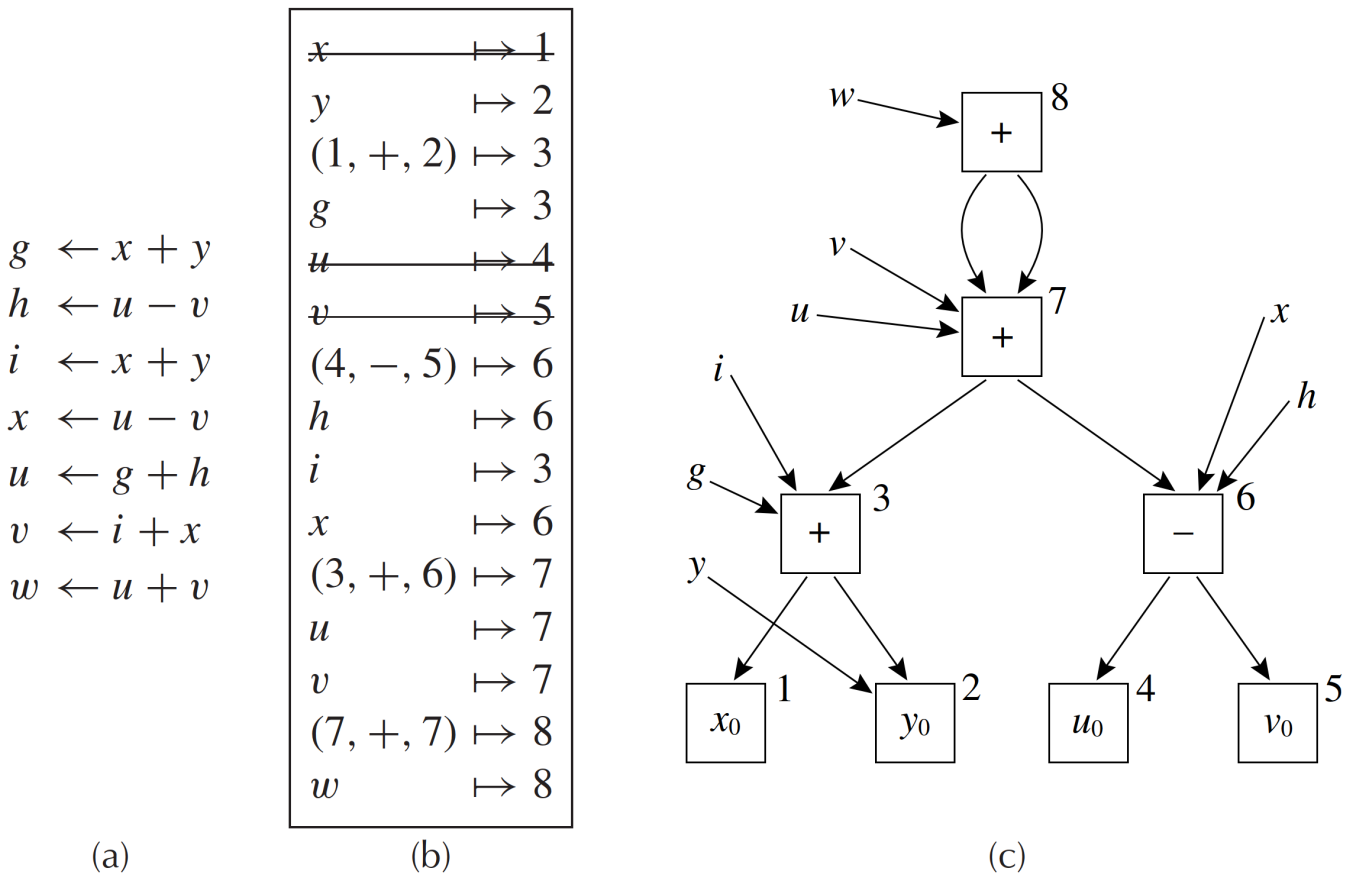


FIGURE 17.8. An illustration of value numbering. (a) A basic block; (b) the table created by the value-numbering algorithm, with hidden bindings shown crossed out; (c) a view of the table as a DAG.

8.5.3 Dead Code Elimination

The operation on DAG's that corresponds to dead-code elimination can be implemented as follows. We delete from a DAG any root (node with no ancestors) that has no live variables attached. Repeated application of this transformation will remove all nodes from the DAG that correspond to dead code.

Example 8.12: If, in Fig. 8.13, `a` *and* `b` *are live but* `c` *and* `e` *are not, we can immediately remove the root labeled* `e`*. Then, the node labeled* `c` *becomes a root and can be removed. The roots labeled* `a` *and* `b` *remain, since they each have live variables attached.*

بحث کوتاهی دربارهٔ بهینه‌سازی حلقه

8.4.5 Loops

Programming-language constructs like while-statements, do-while-statements, and for-statements naturally give rise to loops in programs. Since virtually every program spends most of its time in executing its loops, it is especially important for a compiler to generate good code for loops. Many code transformations depend upon the identification of "loops" in a flow graph. We say that a set of nodes L in a flow graph is a loop if L contains a node e called the loop entry, such that:

1. ***e is not ENTRY, the entry of the entire flow graph.***
2. ***No node in L besides e has a predecessor outside L. That is, every path from ENTRY to any node in L goes through e.***
3. ***Every node in L has a nonempty path, completely within L, to e.***

Example 8.9 : The flow graph of Fig. 8.9 has three loops:

1. B_3 by itself.
2. B_6 by itself.
3. $\{B_2, B_3, B_4\}$.

The first two are single nodes with an edge to the node itself. For instance, B_3 forms a loop with B_3 as its entry. Note that the last requirement for a loop is that there be a nonempty path from B_3 to itself. Thus, a single node like B_2, which does not have an edge $B_2 \to B_2$, is not a loop, since there is no nonempty path from B_2 to itself within $\{B_2\}$.

The third loop, $L = \{B_2, B_3, B_4\}$, has B_2 as its loop entry. Note that among these three nodes, only B_2 has a predecessor, B_1, that is not in L. Further, each of the three nodes has a nonempty path to B_2 staying within L. For instance, B_2 has the path $B_2 \to B_3 \to B_4 \to B_2$. $\square$

Loop Optimization

☞ *Inefficiencies are not limited to single blocks. Code in one block may provide the context for improving code in another block. Thus, many optimizations examine contexts larger than a single block.*

☞ *Regional techniques operate over subsets of the control-flow graph that include multiple blocks but do not, typically, extend to the entire procedure.*

☞ *Because regional methods examine more context than local methods, they can discover opportunities that local methods cannot see. Because they focus on a subset of the procedure's CFG, these methods can often be simpler than a global approach to the same problem.*

☞ *Loop optimizations fit this mold. Many loop optimizations require extensive analysis to prove safety. Because so many programs spend a large part of their execution time inside loops, compiler writers have found these techniques to be profitable.*

Loop Optimization

There are some techniques particularly related to loop optimization that can have significant effects on execution speed. The first step is of course to identify the loops. This can be done using the CFG and it involves finding a basic block (the loop header) through which all paths to basic blocks in the loop pass and also finding a path from one of the loop's blocks back to the loop header (the back edge). The header block is said to dominate all the nodes in the loop. It should be possible to follow a control path from any of the nodes in the loop to any of the other nodes in the loop without passing through a node not in the loop. Once the loops have been identified, each can be optimized in turn.

Removing Loop-Invariant Code

If a computation is performed in a loop such that the values used in the computation remain unchanged for each iteration of the loop, then the computation should be performed just once before the loop starts. The loop invariant code is hoisted out of the loop. For example in

```
i = 0;
while (i<n) {
    a[i] = a[i] + n*n;
    i = i + 1
}
```

the `n * n` *computation is done each time the loop executes but it only needs to be done once because* `n` *is a loop invariant—it does not change as the loop is executed. The code then becomes:*

```
i = 0;
t = n*n;
while (i<n) {
    a[i] = a[i] + t;
    i = i + 1
}
```

One optimization for loops is finding computations that are repeated in every iteration of the loop without changing the values involved, i.e., loop-invariant computations, and then lift these computations outside the loop, so they are performed only once before the loop is entered. This is called code hoisting.

Example: there may be good opportunities for loop-invariant code motion when using multi-dimensional arrays in loops. Consider the C program fragment:

```
for (k=0; k<n; k++)
    a[i][j][k] = a[i][j][k] + 1;
```

Finding the offset from the start of the array of element `a[i][j][k]` *does not need to be done from scratch on each iteration of the loop. The offset calculations involving the subscripts* `i` *and* `j` *can be done outside the loop. The IR has to be designed so that this type of optimization can be achieved. Finally, although it is unlikely to be a problem with this type of optimization, any movement of loop-invariant code must take into account the possibility that the loop is not executed at all.*

Induction Variables

Loops often contain one or more variables that are incremented or decremented by a constant value each time control passes through the loop. In other words, these variables take values which are linear functions of the loop counter, containing the number of times the loop has been executed since it was last entered. Consequently a variable whose value is a linear function of induction variables must also be an induction variable. They keep in step with each other as the loop executes. It may then be possible to reduce the number of induction variables in a loop by replacing induction variables by linear functions of other induction variables.

Some loops have a variable i that is incremented or decremented, and a variable j that is set (in the loop) to $i \cdot c + d$, where c and d are loop-invariant. Then we can calculate j's value without reference to i; whenever i is incremented by a we can increment j by $c \cdot a$.

Strength Reduction

It may be possible to save some processor cycles by replacing potentially costly operations on induction variables by cheaper operations. For example:

```
i = 0;
t = 0;
while (i<n) {
    t = i*3;
    a[i] = a[i] + t;
    i = i + 1
}
```

Here, `t` *is an induction variable. The code can be rewritten:*

```
i = 0;
t = 0;
while (i<n) {
    a[i] = a[i] + t;
    t = t + 3;
    i = i + 1
}
```

The $*$ operator has been replaced by a presumably cheaper + operator.

The transformation of replacing an expensive operation, such as multiplication, by a cheaper one, such as addition, is known as strength reduction.