7CS5A COMPILER CONSTRUCTION (Code Generation)

Lecture 11: Oct. 16, 2018

Instructor: K.R. Chowdhary *: Professor of CS*

11.1 Introduction

The final phase in compiler is the code generation. It takes as input the intermediate representation produced by the front end of the compiler, along with relevant symbol table information, and produces as output a semantically equivalent target program (see Fig. 11.1).

Figure 11.1: Position of code generator.

There are too many requirements imposed on code generator. The target program (machine code) must preserve the semantics of the source program and be of high quality; that is, it must make effective use of the available resources of the target machine, like, CPU, memory, registers, etc. Moreover, the code generating program it self must run efficiently.

The challenge is that, mathematically, the problem of generating an optimal target program for a given source program is undecidable; many of the subproblems encountered in code generation such as register allocation are computationally intractable. In practice, we must be content with heuristic techniques that generate good, but not necessarily an optimal, code. Fortunately, heuristics have matured enough that a carefully designed code generator can produce code that is several times faster than code produced by a naive one.

Compilers that need to produce efficient target programs, include an optimization phase prior to code generation. The optimizer maps the intermediate representation into intermediate representation from which more efficient code can be generated. In general, the code-optimization and code-generation phases of a compiler, often referred to as the back end, may require multiple passes over the intermediate representation before generating the target program. The techniques presented in the following can be used irrespective of whether or not an optimization phase occurs before code generation.

A code generator has three primary tasks: instruction selection, register allocation and assignment, and instruction ordering. Instruction selection involves choosing appropriate target-machine instructions to implement the intermediate representation statements. Register allocation and assignment involves deciding what values to keep in which registers. Instruction ordering involves deciding in what order to schedule the execution of instructions.

11.2 Construction of Basic Blocks and Flow Graph

The graph representation of intermediate code is helpful for discussing code generation. To begin with, the graph is not constructed explicitly by a code-generation algorithm. Code generation is benefited from context. For example, we can do a better job of register allocation if we know how values are defined and used. We can select the instruction in a better way, by looking at sequences of three-address statements. The basic block and flow graph representation is done as follows:

1. Partition the intermediate code into basic blocks, which are maximal sequences of consecutive three-address instructions with the following properties
   (a) The flow of control can only enter the basic block through the first instruction in the block. That is, there are no direct jumps to enter into the middle of a block.
   (b) Control will leave the block without halting in the block or branching, from inside of a block exiting from the last instruction in the block.
2. The basic blocks become the nodes of a flow graph, whose edges indicate which blocks follow to what other blocks.

11.2.1 Basic Blocks Algorithm

Our first job is to partition a sequence of three-address instructions into *basic blocks*. We begin a new basic block with the first instruction and keep adding instructions until we meet either a jump, a conditional jump, or a label on the following instruction. In the absence of jumps and labels, control proceeds sequentially from one instruction to the next. This idea is formalized in the following algorithm.

Algorithm-1: Partitioning three-address instructions set into basic blocks.

INPUT: A sequence of three-address instructions.

OUTPUT: A list of the basic blocks, such that each instruction is assigned to exactly one basic block.

METHOD: First, we identify those instructions in the intermediate code that are *leader instructions*, that is, the first instructions in some basic block. The rules for finding leaders are:

1. The first three-address instruction in the intermediate code is a leader.
2. Any instruction that is the target of a conditional or unconditional jump is a leader.
3. Any instruction that immediately follows a conditional or unconditional jump is a leader.

Then, for each leader, its basic block consists of itself and all instructions up to but not including the next leader or the end of the intermediate program. □

Example 11.1 *Intermediate code to set a* 10×10 *matrix to an identity matrix.*

Solution. Consider the pseudocode as below, which turns a 10×10 matrix into an identity matrix.

```
for i from 1 to 10 do
  for j from 1 to 10 do
     a[i,j] = 0.0;
for i from 1 to 10 do
   a[i,i] = 1.0;
```

Corresponding to above pseudo-code, the intermediate code in Table 11.1 turns a 10×10 matrix a into an identity matrix. In generating the intermediate code, we have assumed that the real-valued array elements take 4 bytes each, and that the matrix a is stored in row-major form (first row is stored, element by element, then second row, and so on). Note that j is column counter and i is row counter.

Table 11.1: Intermediate code to set the 10×10 matrix a to an identity matrix

1)	$i = 1$
2)	$j = 1$
3)	$t1 = 10 * i$
4)	$t2 = t1 + j$
5)	$t3 = 4 * t2$
6)	$t4 = t3 - 44$
7)	$a[t4] = 0.0$
8)	$j = j + 1$
9)	$if\ j <= 10\ goto\ (3)$
10)	$i = i + 1$
11)	$if\ i < 10\ goto\ (2)$
12)	$i = 1$
13)	$t5 = i - 1$
14)	$t6 = 44 * t5$
15)	$a[t6] = 1.0$
16)	$i = i + 1$
17)	$if\ i <= 10\ goto\ (13)$

First, instruction 1 is a *leader* by rule (1) of Algorithm 1. To find the other leaders, we first need to find the jumps. In this example, there are three jumps, all conditional, at instructions 9, 11, and 17. By rule (2), the targets of these jumps are leaders; they are instructions 3, 2, and 13, respectively. Then, by rule (3), each instruction following a jump is a leader; those are instructions 10 and 12. Note that no instruction follows 17 in this code, but if there were code following, the 18th instruction would also be a leader.

We conclude that the leaders are instructions $1, 2, 3, 10, 12$, and 13. The basic block of each leader contains all the instructions from itself until just before the next leader. Thus, the basic block of 1 is just 1, for leader 2 the block is just 2. Leader 3, however, has a basic block consisting of instructions 3 through 9, inclusive. Instruction 10's block is 10 and 11; instruction 12's block is just 12 and instruction 13's block is 13 through 17. □

11.2.2 Liveness and Next-Use

For generating good code, it is essential to know when the value of a variable will be used. If the value of a variable, currently in a register will never be referenced subsequently, then that register can be assigned to another variable.

The use of a name in a three-address statement is defined as follows: Let a three-address statement i assigns a value to variable x. If statement j has x as an operand, and control can flow from statement i to j along a path that has no intervening assignments to x, then we say statement j uses the value of x computed at statement i. We further say that x is *live* at statement i.

We wish to determine for each three-address statement $x = y + z$ what the next uses of x, y, and z are. For the present, we do not concern ourselves with uses outside the basic block containing this three-address statement.

Our algorithm to determine liveness and next-use information makes a back-ward pass over each basic block. We store the information in the symbol table. We can easily scan a stream of three-address statements to find the ends of basic blocks as in Algorithm 1. Since procedures can have arbitrary side effects, we assume for convenience that each procedure call starts a new basic block.

Algorithm 2. Determining the liveness and next-use information for each statement in a basic block.

Input. A basic block B of three-address statements. We assume that the symbol table initially shows all nontemporary variables in B as being live on exit.

Output. At each statement $i : x = y + z$ in B, we attach to i the liveness and next-use information of x, y, and z.

Method. We start at the last statement in B and scan backwards to the beginning of B. At each statement $i : x = y + z$ in B, we do the following:

1. Attach to statement i the information currently found in the symbol table regarding the next use and liveness of x, y, and z.
2. In the symbol table, set x to "not live" and "no next use."
3. In the symbol table, set y and z to "live" and the next uses of y and z to i.

Here we have used "+" as a symbol representing any operator. If the three-address statement i is of the form $x = +y$ or $x = y$, the steps are the same as above, ignoring z. Note that the order of steps (2) and (3) may not be interchanged because x may be y or x. □

11.2.3 Flow Graph

Once an intermediate-code program is partitioned into basic blocks, we represent the flow of control between them by a flow graph. The nodes of the flow graph are the basic blocks. There is an edge from block B to block C if and only if it is possible for the first instruction

in block C to immediately follow the last instruction in block B. There are two ways that such an edge could be justified:

- There is a conditional or unconditional jump from the end of B to the beginning of C.
- C immediately follows B in the original order of the three-address instructions, and B does not end in an unconditional jump.

We say that B is a predecessor of C, and C is a successor of B. Often we add two nodes, called the entry and exit in the flow graph, which are not executable intermediate instructions. There is an edge from the entry to the first executable node of the flow graph, i.e., to the basic block that comes from the first instruction. There is an edge to the exit from any basic block that contains an instruction. If the final instruction of the program is a conditional jump, then the block containing the final instruction of the program is one predecessor of the exit.

Example 11.2 *Construct a flow-graph from intermediate code in Table 11.1.*

Solution. The set of basic blocks constructed in Table 11.1 yields the flow graph of Fig. 11.2.

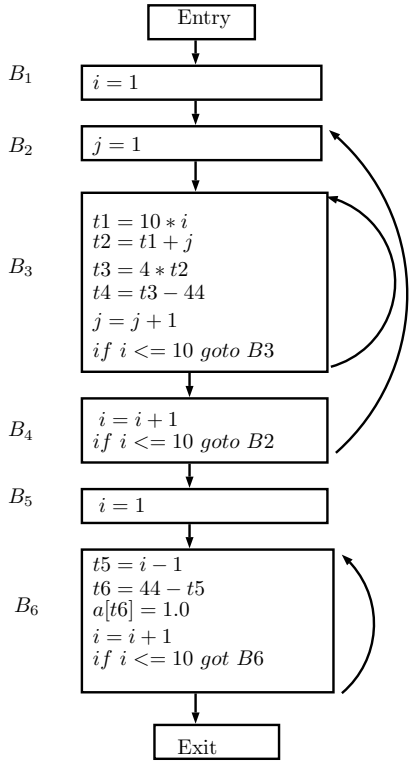

Figure 11.2: Flow graph from Table 11.1.

The entry points to the basic block B_1, as B_1 contains the first instruction of the program. The only successor of B_1 is B_2, because B_1 does not end in an unconditional jump, and the leader of B_2 immediately follows the end of B_1.

Block B_3 has two successors. One is itself, because the leader of B_3, instruction 3, is the target of the conditional jump at the end of B_3, instruction 9. The other successor is B_4, because control can fall through the conditional jump at the end of B_3 and next enter the leader of B_4. Only B_6 points to the exit of the flow graph, since the only way to get to code that follows the program from which we constructed the flow graph is to fall through the conditional jump that ends B_6. □

11.2.4 Representation of Flow Graphs

First, note from Fig. 11.2 that in the flow graph, it is normal to replace the jumps to instruction numbers or labels by jumps to basic blocks. Recall that every conditional or unconditional jump is directed to the leader instruction of some basic block. The reason for this change is that after constructing the flow graph, it is common to make substantial changes to the instructions in the various basic blocks. If jumps were to instructions, we would have to fix the targets of the jumps every time one of the target instructions was changed.

Flow graphs, being quite ordinary graphs, can be represented by any of the data structures appropriate for graphs. The content of nodes (basic blocks) need their own representation. We might represent the content of a node by pointer to the leader, together with a count of the number of instructions or a second pointer to the last instruction. However, since we may be changing the number of instructions in a basic block frequently, it is likely to be more efficient to create a linked list of instructions for each basic block.

11.2.5 Loops

The loops in programs are due to 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

1. There is a node in L called the loop entry with the property that no other node in L has a predecessor outside L. That is, every path from the entry of the entire flow graph to any node in L goes through the loop entry.
2. Every node in L has a nonempty path, completely within L, to the entry of L.

Example 11.3 *Analyse the loops in flow-graph in Figure 11.2.*

Solution. The flow graph of Fig. 11.2 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 second 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 $B_2 + B_3 + B_4 + B_2$. □

11.3 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.

11.3.1 The DAG Representation of Basic Blocks

Many important techniques for *local optimization* begin by transforming a *basic block* into a DAG (directed acyclic graph). The idea of DAG 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.

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

1. We can eliminate local common subexpressions, that is, instructions that compute a value that has already been computed.
2. We can eliminate dead code, that is, instructions that compute a value that is never used.
3. 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.

4. We can apply algebraic laws to reorder operands of three-address instructions, and sometimes it simplify the computation.

11.3.2 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 earlier.

Example 11.4 *Construct the DAG for the block and find the common subexpressions.*

$$\begin{aligned} a &= b + c \\ b &= a - d \\ c &= b + c \\ d &= a - d \end{aligned} \tag{11.1}$$

Solution. The DAG is shown in Fig. 11.3. 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. 11.3 labeled $-$, because that is the most recent definition of b. Thus, we do not confuse the values computed at statements one and three.

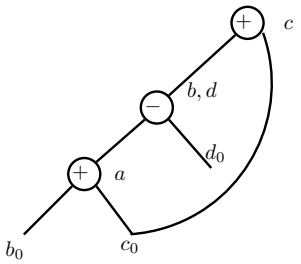

Figure 11.3: DAG for basic block in equation 11.1.

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. 11.3, the basic block in Example 11.4 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. 11.3. The block then becomes

$$\begin{aligned} a &= b + c \\ d &= a - d \\ c &= d + c \end{aligned} \tag{11.2}$$

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.

Example 11.5 *Basic Block optimization for*

$$\begin{aligned} a &= b + c \\ b &= b - d \\ c &= c + d \\ e &= b + c \end{aligned} \tag{11.3}$$

Solution. 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 in equation 11.3 is the same, namely $b_0 + c_0$. 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. 11.4, but does not exhibit any common subexpressions. However, *algebraic identities* applied to the DAG, are discussed later in next parts, may expose the equivalence. These entities are basically, $x + 0 = 0 + x = x$, $x/1 = x$, etc.

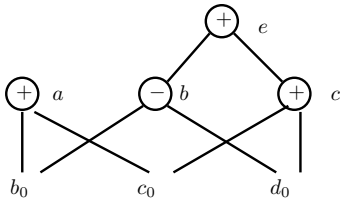

Figure 11.4: DAG for basic block in equation 11.3.

□

11.3.3 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 11.6 *Dead Code Elimination.*

Solution. If, in Fig. 11.4, 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. □

11.3.4 Representation of Array References

At first glance, it might appear that the array-indexing instructions can be treated like any other operator. Consider for instance the sequence of three- address statements:

$$x = a[i]$$
$$a[j] = y$$
$$z = a[i]$$

If we think of $a[i]$ as an operation involving a and i, similar to $a + i$, then it might appear as if the two uses of $a[i]$ were a common subexpression. In that case, we might be tempted to "optimize" by replacing the third instruction $z = a[i]$ by the simpler $z = x$. However, since j could equal i, the middle statement may in fact change the value of $a[i]$; thus, it is not legal to make this change.

The proper way to represent array accesses in a DAG is as follows.

1. An assignment from an array, like $x = a[i]$, is represented by creating a node with operator $= []$ and two children representing the initial value of the array, a_0 in this case, and the index i. Variable x becomes a label of this new node.
2. An assignment to an array, like $a[j] = y$, is represented by a new node with operator and three children representing a_0, j and y. There is no variable labeling this node. What is different is that the creation of this node *kills* all currently constructed nodes whose value depends on a_0. A node that has been killed cannot receive any more labels; that is, it cannot become a common subexpression.

Example 11.7 *Construct a DAG for a sequence of array assignments, for basic block:*

$$x = a[i]$$
$$a[j] = y$$
$$z = a[i]$$

Solution. The DAG is is shown in Fig. 11.5.

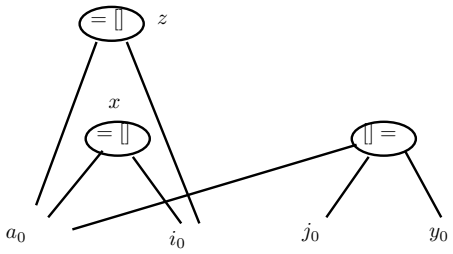

Figure 11.5: DAG for sequence of array assignments.

The node N for x is created first, but when the node labeled $[] =$ is created, N is killed. Thus, when the node for x is created, it cannot be identified with N, and a new node with the same operands a_0 and i_0 must be created. □

11.4 Exercises

1. What are the challenges in code generation phase of compiler, in respect of following:

   (a) Code generator it self
   (b) Language to be compiled
   (c) The hardware for which it is supposed to generate the code

2. In the code generation phase, what are the major subareas for applying the heuristics?
3. What are the primary tasks of the code generator?
4. Why it is important to segment the intermediate code into blocks?
5. Define block, and flow graph.
6. Below given is simple matrix addition program.

```
for(i=0; i < 10; i++}
   c[i][j] = a[i][j] + b[i][j];
```

7. Translate the program into three-address statements of the type. Assume the matrix entries are numbers that require 4 bytes, and that matrices are stored in column-major order.
8. Write the algorithm in pseudo code to determine liveness of variables in a block.
9. Construct the flow graph for your code from above.
10. Identify the loops in your flow graph from above
11. What are the advantages of representing blocks of code using DAGs?
12. What kind of optimizations are possible in intermediate code using DAGs?
13. Write your own version of algorithm for dead-code elimination?
14. How you will determine whether a value is live or not on exit of a block?
15. Write the algorithm in pseudo code to construct the basic blocks out of a given program in intermediate representation.
16. Modify the code in Table 11.1 so that it works for 8-byte data values, consider the array of 5×5 of real numbers.
17. Modify the Table 11.1 so that it represents intermediate code for addition of two matrices of 10×10, with each element of 4-byte size.
18. Construct flow graph for above exercise.
19. Explain the value number method to determine common sub-expressions.