بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ـ دانشکدهٔ مهندسی برق و کامپیوتر
(نیم‌سال تحصیلی ۴۰۱۲)

کامپایلر

حسین فلسفین

یک توضیح مختصر دیگر دربارهٔ GCC و IFهای آن

Many readers will be familiar with the gcc compilers. Distributed as open source by the Free Software Foundation, gcc is used very widely in both academia and industry. The standard distribution includes front ends for C, C++, Objective-C, Ada, Fortran, Go, and Java. Front ends for additional languages, including Cobol, Modula-2 and 3, Pascal and PL/I, are separately available. The C compiler is the original, and the one most widely used (gcc originally stood for "GNU C compiler"). There are back ends for dozens of processor architectures, including all commercially significant options. There are also GNU implementations, not based on gcc, for some two dozen additional languages.

Gcc has three main IFs. Most of the (language-specific) front ends employ, internally, some variant of a high-level syntax tree form known as GENERIC. Early phases of machine-independent code improvement use a somewhat lower-level tree form known as GIMPLE (still a high-level IF). Later phases use a linear form known as RTL (register transfer language). RTL is a medium-level IF, but a bit higher level than most: it overlays a control flow graph on of a sequence of pseudoinstructions. RTL was, for many years, the principal IF for gcc. GIMPLE was introduced in 2005 as a more suitable form for machine-independent code improvement.

Bottom-up and top-down attribute grammars to build a syntax tree

Figures below contain bottom-up and top-down attribute grammars, respectively, to build a syntax tree for constant expressions. The attributes in these grammars hold neither numeric values nor target code fragments; instead they point to nodes of the syntax tree. Function `make_leaf` *returns a pointer to a newly allocated syntax tree node containing the value of a constant. Functions* `make_un_op` *and* `make_bin_op` *return pointers to newly allocated syntax tree nodes containing a unary or binary operator, respectively, and pointers to the supplied operand(s).*

$E_1 \longrightarrow E_2$ `+` T
✂ E_1.ptr := make_bin_op("+", E_2.ptr, T.ptr)

$E_1 \longrightarrow E_2$ `-` T
✂ E_1.ptr := make_bin_op("−", E_2.ptr, T.ptr)

$E \longrightarrow T$
✂ E.ptr := T.ptr

$T_1 \longrightarrow T_2$ `*` F
✂ T_1.ptr := make_bin_op("×", T_2.ptr, F.ptr)

$T_1 \longrightarrow T_2$ `/` F
✂ T_1.ptr := make_bin_op("÷", T_2.ptr, F.ptr)

$T \longrightarrow F$
✂ T.ptr := F.ptr

$F_1 \longrightarrow$ `-` F_2
✂ F_1.ptr := make_un_op("$^{+}/_{-}$", F_2.ptr)

$F \longrightarrow$ `(` E `)`
✂ F.ptr := E.ptr

$F \longrightarrow$ `const`
✂ F.ptr := make_leaf(const.val)

Bottom-up (S-attributed) attribute grammar to construct a syntax tree. The symbol $^{+}/_{-}$ is used (as it is on calculators) to indicate change of sign.

$E \longrightarrow T\ TT$
- ✄ TT.st := T.ptr
- ✄ E.ptr := TT.ptr

$TT_1 \longrightarrow +\ T\ TT_2$
- ✄ TT_2.st := make_bin_op("+", TT_1.st, T.ptr)
- ✄ TT_1.ptr := TT_2.ptr

$TT_1 \longrightarrow -\ T\ TT_2$
- ✄ TT_2.st := make_bin_op("−", TT_1.st, T.ptr)
- ✄ TT_1.ptr := TT_2.ptr

$TT \longrightarrow \epsilon$
- ✄ TT.ptr := TT.st

$T \longrightarrow F\ FT$
- ✄ FT.st := F.ptr
- ✄ T.ptr := FT.ptr

$FT_1 \longrightarrow *\ F\ FT_2$
- ✄ FT_2.st := make_bin_op("×", FT_1.st, F.ptr)
- ✄ FT_1.ptr := FT_2.ptr

$FT_1 \longrightarrow /\ F\ FT_2$
- ✄ FT_2.st := make_bin_op("÷", FT_1.st, F.ptr)
- ✄ FT_1.ptr := FT_2.ptr

$FT \longrightarrow \epsilon$
- ✄ FT.ptr := FT.st

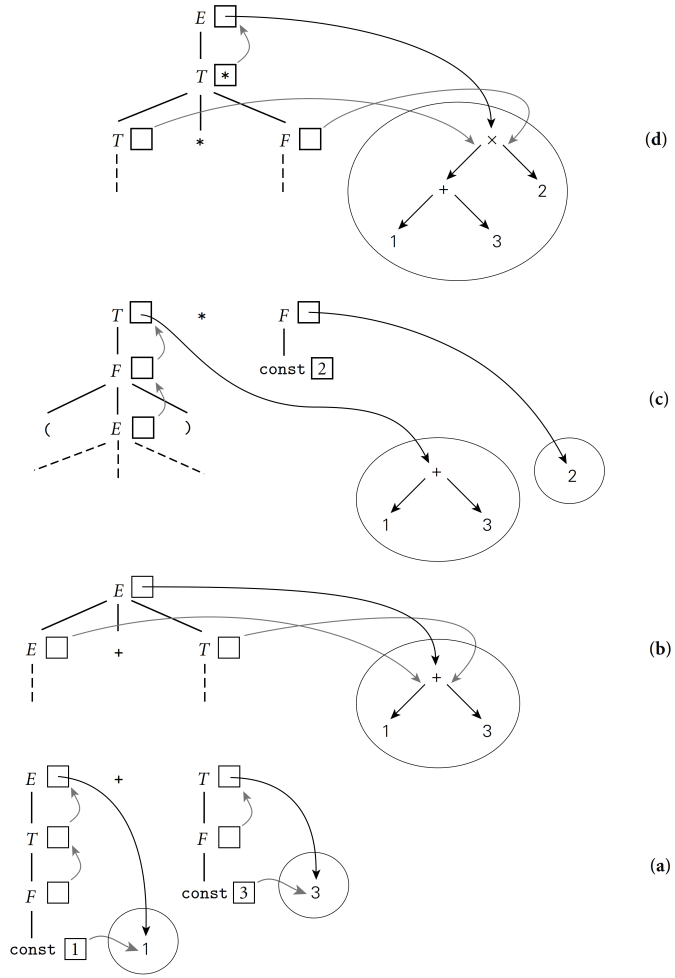
$F_1 \longrightarrow -\ F_2$
- ✄ F_1.ptr := make_un_op("+/−", F_2.ptr)

$F \longrightarrow (\ E\ )$
- ✄ F.ptr := E.ptr

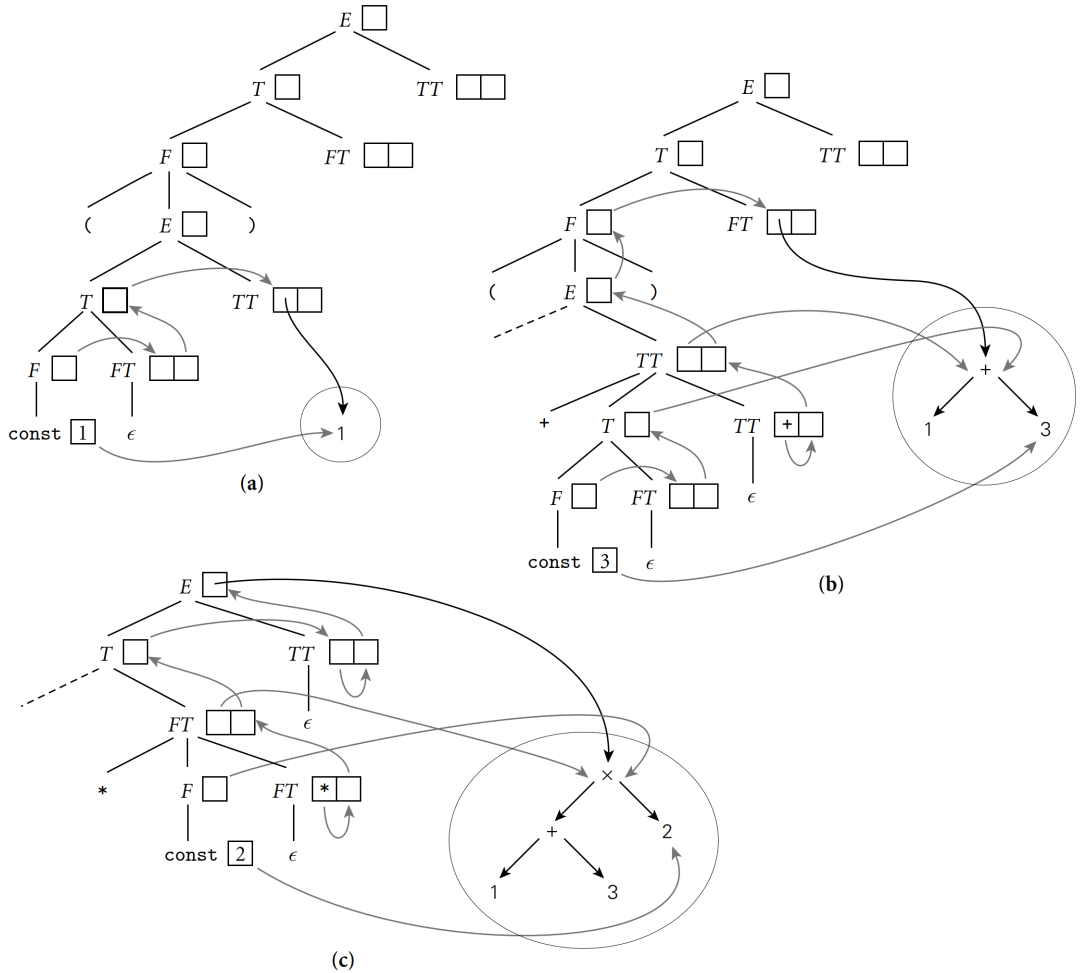
$F \longrightarrow$ `const`
- ✄ F.ptr := make_leaf(const.val)

Top-down (L-attributed) attribute grammar to construct a syntax tree. Here the st attribute, like the ptr attribute, is a pointer to a syntax tree node.

Figures below show stages in the decoration of parse trees for `(1 + 3) * 2`***, using the the above grammars. Note that the final syntax tree is the same in each case.***



Construction of a syntax tree for `(1 + 3) * 2` via decoration of a bottom-up parse tree, using the grammar of Figure . This figure reads from bottom to top. In diagram (a), the values of the constants 1 and 3 have been placed in new syntax tree leaves. Pointers to these leaves propagate up into the attributes of E and T. In (b), the pointers to these leaves become child pointers of a new internal + node. In (c) the pointer to this node propagates up into the attributes of T, and a new leaf is created for 2. Finally, in (d), the pointers from T and F become child pointers of a new internal × node, and a pointer to this node propagates up into the attributes of E.



Construction of a syntax tree via decoration of a top-down parse tree, using the grammar of Figure . In the top diagram, (a), the value of the constant 1 has been placed in a new syntax tree leaf. A pointer to this leaf then propagates to the st attribute of *TT*. In (b), a second leaf has been created to hold the constant 3. Pointers to the two leaves then become child pointers of a new internal + node, a pointer to which propagates from the st attribute of the bottom-most *TT*, where it was created, all the way up and over to the st attribute of the top-most *FT*. In (c), a third leaf has been created for the constant 2. Pointers to this leaf and to the + node then become the children of a new × node, a pointer to which propagates from the st of the lower *FT*, where it was created, all the way to the root of the tree.

ساخت DAGها نیز می‌تواند به‌کمک SDDها صورت پذیرد

The SDD of Fig. 6.4 can construct either syntax trees or DAG's. It was used to construct syntax trees in Example 5.11, where functions *Leaf* and *Node* created a fresh node each time they were called. It will construct a DAG if, before creating a new node, these functions first check whether an identical node already exists. If a previously created identical node exists, the existing node is returned. For instance, before constructing a new node, *Node*(*op*, *left*, *right*), we check whether there is already a node with label *op*, and children *left* and *right*, in that order. If so, *Node* returns the existing node; otherwise, it creates a new node.

Example 6.2: The sequence of steps shown in Fig. 6.5 constructs the DAG in Fig. 6.3, provided *Node* and *Leaf* return an existing node, if possible, as discussed above. We assume that *entry-a* points to the symbol-table entry for `a`, and similarly for the other identifiers.

When the call to *Leaf* (**id**, *entry-a*) is repeated at step 2, the node created by the previous call is returned, so $p_2 = p_1$. Similarly, the nodes returned at steps 8 and 9 are the same as those returned at steps 3 and 4 (i.e., $p_8 = p_3$ and $p_9 = p_4$). Hence the node returned at step 10 must be the same at that returned at step 5; i.e., $p_{10} = p_5$. □

	PRODUCTION	SEMANTIC RULES
1)	$E \to E_1 + T$	$E.node = \textbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \to E_1 - T$	$E.node = \textbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \to T$	$E.node = T.node$
4)	$T \to (\ E\ )$	$T.node = E.node$
5)	$T \to \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \to \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

Figure 6.4: Syntax-directed definition to produce syntax trees or DAG's

$$
\begin{aligned}
1)\quad & p_1 = \mathit{Leaf}\,(\mathbf{id}, \mathit{entry}\text{-}a) \\
2)\quad & p_2 = \mathit{Leaf}\,(\mathbf{id}, \mathit{entry}\text{-}a) = p_1 \\
3)\quad & p_3 = \mathit{Leaf}\,(\mathbf{id}, \mathit{entry}\text{-}b) \\
4)\quad & p_4 = \mathit{Leaf}\,(\mathbf{id}, \mathit{entry}\text{-}c) \\
5)\quad & p_5 = \mathit{Node}('-', p_3, p_4) \\
6)\quad & p_6 = \mathit{Node}('*', p_1, p_5) \\
7)\quad & p_7 = \mathit{Node}('+', p_1, p_6) \\
8)\quad & p_8 = \mathit{Leaf}\,(\mathbf{id}, \mathit{entry}\text{-}b) = p_3 \\
9)\quad & p_9 = \mathit{Leaf}\,(\mathbf{id}, \mathit{entry}\text{-}c) = p_4 \\
10)\quad & p_{10} = \mathit{Node}('-', p_3, p_4) = p_5 \\
11)\quad & p_{11} = \mathit{Leaf}\,(\mathbf{id}, \mathit{entry}\text{-}d) \\
12)\quad & p_{12} = \mathit{Node}('*', p_5, p_{11}) \\
13)\quad & p_{13} = \mathit{Node}('+', p_7, p_{12})
\end{aligned}
$$

Figure 6.5: Steps for constructing the DAG of Fig. 6.3

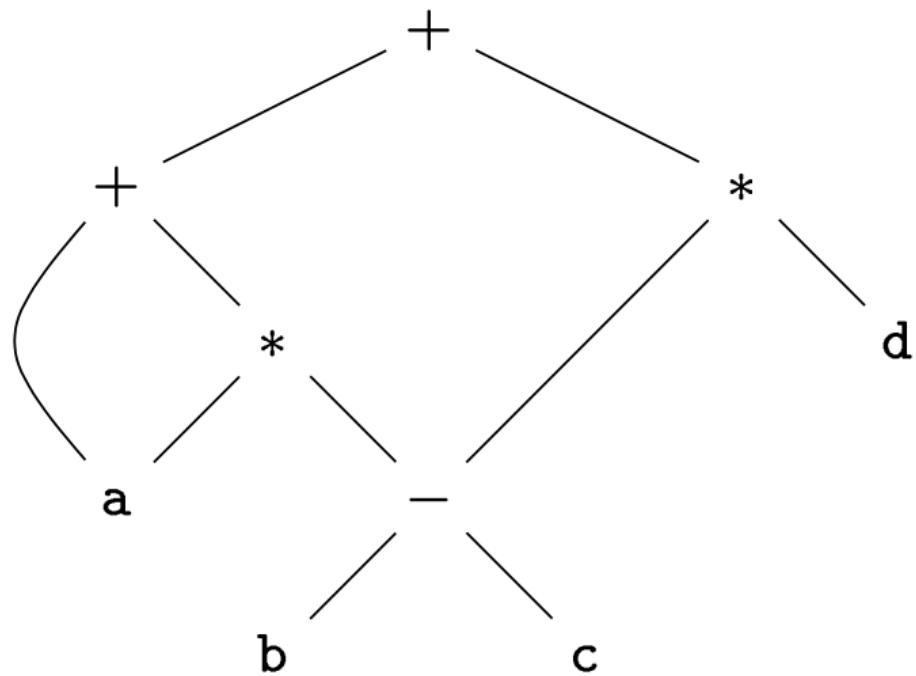


Figure 6.3: Dag for the expression `a + a * (b - c) + (b - c) * d`

3AC for expressions

PRODUCTION	SEMANTIC RULES
$S \rightarrow \mathbf{id} = E\ ;$	$S.code = E.code \mid\mid gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \mid\mid E_2.code \mid\mid gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \mid\mid gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \mathbf{id}$	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$

Figure 6.19: Three-address code for expressions

The syntax-directed definition in Fig. 6.19 builds up the three-address code for an assignment statement S using attribute *code* for S and attributes *addr* and *code* for an expression E. Attributes $S.code$ and $E.code$ denote the three-address code for S and E, respectively. Attribute $E.addr$ denotes the address that will hold the value of E. Recall from Section 6.2.1 that an address can be a name, a constant, or a compiler-generated temporary.

Consider the last production, $E \rightarrow \mathbf{id}$, in the syntax-directed definition in Fig. 6.19. When an expression is a single identifier, say x, then x itself holds the value of the expression. The semantic rules for this production define $E.addr$ to point to the symbol-table entry for this instance of **id**. Let *top* denote the current symbol table. Function *top.get* retrieves the entry when it is applied to the string representation **id**.*lexeme* of this instance of **id**. $E.code$ is set to the empty string.

When $E \rightarrow (\, E_1 \,)$, the translation of E is the same as that of the subexpression E_1. Hence, $E.addr$ equals $E_1.addr$, and $E.code$ equals $E_1.code$.

The operators + and unary - in Fig. 6.19 are representative of the operators in a typical language. The semantic rules for $E \rightarrow E_1 + E_2$, generate code to compute the value of E from the values of E_1 and E_2. Values are computed into newly generated temporary names. If E_1 is computed into $E_1.addr$ and E_2 into $E_2.addr$, then $E_1 + E_2$ translates into $t = E_1.addr + E_2.addr$, where t is a new temporary name. $E.addr$ is set to t. A sequence of distinct temporary names $t_1, t_2, \ldots$ is created by successively executing **new** $Temp()$.

For convenience, we use the notation $gen(x\ '='\ y\ '+'\ z)$ to represent the three-address instruction $x = y + z$. Expressions appearing in place of variables like x, y, and z are evaluated when passed to *gen*, and quoted strings like $'='$ are taken literally. Other three-address instructions will be built up similarly by applying *gen* to a combination of expressions and strings.

When we translate the production $E \rightarrow E_1 + E_2$, the semantic rules in Fig. 6.19 build up $E.code$ by concatenating $E_1.code$, $E_2.code$, and an instruction that adds the values of E_1 and E_2. The instruction puts the result of the addition into a new temporary name for E, denoted by $E.addr$.

The translation of $E \rightarrow -E_1$ is similar. The rules create a new temporary for E and generate an instruction to perform the unary minus operation.

Finally, the production $S \rightarrow \mathbf{id} = E;$ generates instructions that assign the value of expression E to the identifier **id**. The semantic rule for this production uses function *top.get* to determine the address of the identifier represented by **id**, as in the rules for $E \rightarrow \mathbf{id}$. $S.code$ consists of the instructions to compute the value of E into an address given by $E.addr$, followed by an assignment to the address $top.get(\mathbf{id}.lexeme)$ for this instance of **id**.

Example 6.11 : The syntax-directed definition in Fig. 6.19 translates the assignment statement `a = b + - c ;` into the three-address code sequence

$$
\begin{aligned}
t_1 &= \texttt{minus c} \\
t_2 &= \texttt{b} + t_1 \\
\texttt{a} &= t_2
\end{aligned}
$$

□

8.4 Basic Blocks and Flow Graphs

This section introduces a graph representation of intermediate code that is helpful for discussing code generation even if the graph is not constructed explicitly by a code-generation algorithm. Code generation benefits from context. We can do a better job of register allocation if we know how values are defined and used, as we shall see in Section 8.8. We can do a better job of instruction selection by looking at sequences of three-address statements, as we shall see in Section 8.9.

The representation is constructed as follows:

1. Partition the intermediate code into ***basic blocks****, which are maximal sequences of consecutive three-address instructions with the properties that*

(a) The flow of control can only enter the basic block through the first instruction in the block. That is, there are no jumps into the middle of the block.

(b) Control will leave the block without halting or branching, except possibly at the last instruction in the block.

2. The basic blocks become the nodes of a ***flow graph****, whose edges indicate which blocks can follow which other blocks.*

Starting in Chapter 9, we discuss transformations on flow graphs that turn the original intermediate code into "optimized" intermediate code from which better target code can be generated. The "optimized" intermediate code is turned into machine code using the code-generation techniques in Chapter 8.

8.4.1 Basic Blocks

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 8.5 : Partitioning three-address instructions into basic blocks.

INPUT: A sequence of three-address instructions.

OUTPUT: A list of the basic blocks for that sequence in which each instruction is assigned to exactly one basic block.

METHOD: First, we determine those instructions in the intermediate code that are *leaders*, that is, the first instructions in some basic block. The instruction just past the end of the intermediate program is not included as a leader. 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 8.6 : The intermediate code in Fig. 8.7 turns a 10×10 matrix `a` into an identity matrix. Although it is not important where this code comes from, it might be the translation of the pseudocode in Fig. 8.8. In generating the intermediate code, we have assumed that the real-valued array elements take 8 bytes each, and that the matrix `a` is stored in row-major form.

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**
    $a[i, i] = 1.0;$

Figure 8.8: Source code for Fig. 8.7

First, instruction 1 is a leader by rule (1) of Algorithm 8.5. 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. □

```
1)   i = 1
2)   j = 1
3)   t1 = 10 * i
4)   t2 = t1 + j
5)   t3 = 8 * t2
6)   t4 = t3 - 88
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 = 88 * t5
15)  a[t6] = 1.0
16)  i = i + 1
17)  if i <= 10 goto (13)
```

Figure 8.7: Intermediate code to set a 10×10 matrix to an identity matrix

8.4.3 Flow Graphs

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, that do not correspond to executable intermediate instructions. There is an edge from the entry to the first executable node of the flow graph, that is, to the basic block that comes from the first instruction of the intermediate code. There is an edge to the exit from any basic block that contains an instruction that could be the last executed instruction of the program. If the final instruction of the program is not an unconditional jump, then the block containing the final instruction of the program is one predecessor of the exit, but so is any basic block that has a jump to code that is not part of the program.

Example 8.8: The set of basic blocks constructed in Example 8.6 yields the flow graph of Fig. 8.9. The entry points to basic block B_1, since 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. □

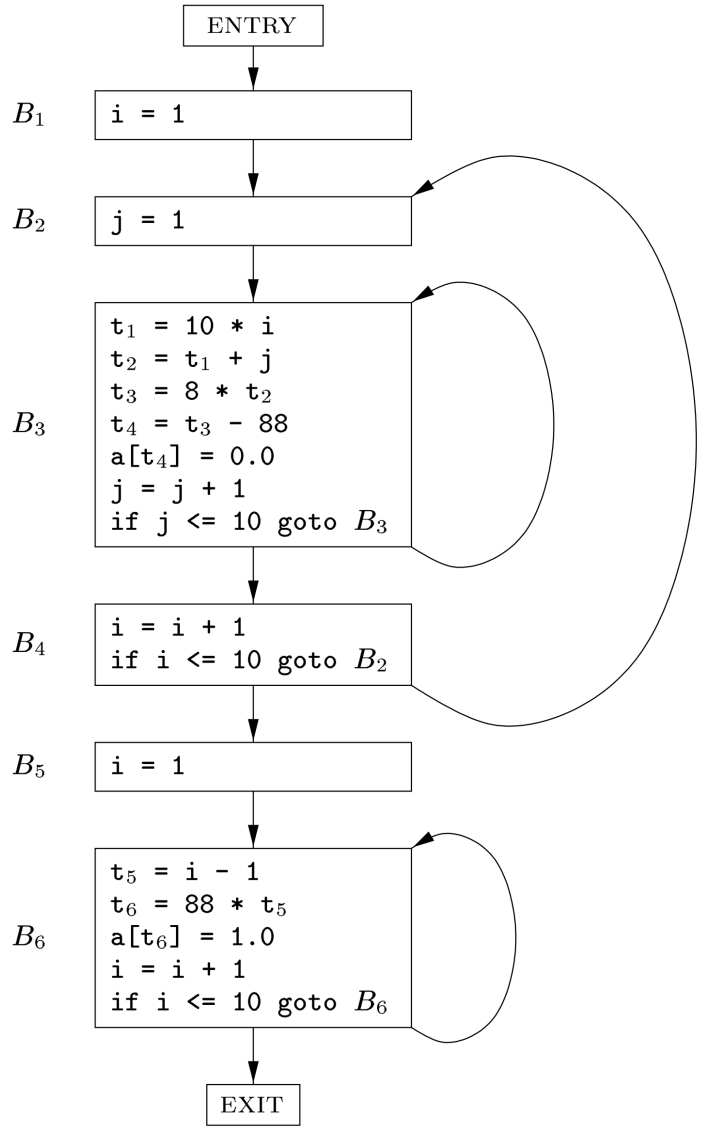


Figure 8.9: Flow graph from Fig. 8.7

8.4.4 Representation of Flow Graphs

First, note from Fig. 8.9 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 to the leader of some basic block, and it is to this block that the jump will now refer.

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.