CHAPTER EIGHT
8. CODE GENERATION

8.1. Introduction

The final phase in our compiler model is the code generator. 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, as shown in Fig. 8.1.

The requirements imposed on a code generator are severe. The target program must preserve the semantic meaning of the source program and be of high quality; that is, it must make effective use of the available resources of the target machine. Moreover, the code generator itself must run efficiently.

Compilers that need to produce efficient target programs, include an optimization phase prior to code generation. The optimizer maps the IR into IR 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 make multiple passes over the IR before generating the target program. Code optimization will be discussed in detail in the next chapter (Chapter 9).

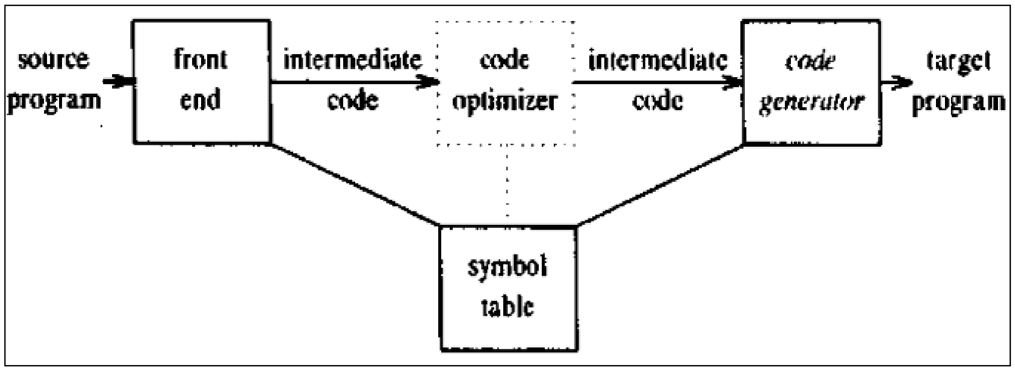


Figure 8.1: Position of code generator

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 IR 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.

8.2. Issues in the Design of a Code Generator

While the details are dependent on the specifics of the intermediate representation, the target language, and the run-time system, tasks such as instruction selection, register allocation and assignment, and instruction ordering are encountered in the design of almost all code generators.

The most important criterion for a code generator is that it produce correct code. Correctness takes on special significance because of the number of special cases that a code generator might face. Given the premium on correctness, designing a code generator so it can be easily implemented, tested, and maintained is an important design goal.

❖ Input to the Code Generator

The input to the code generator is the intermediate representation of the source program produced by the front end, along with information in the symbol table that is used to determine the run-time addresses of the data objects denoted by the names in the intermediate representation (IR).

The many choices for the IR include three-address representations such as quadruples, triples, indirect triples; virtual machine representations such as byte codes and stack-machine code; linear representations such as postfix notation; and graphical representations such as syntax trees and DAG's.

We assume that prior to code generation, the front end has scanned, parsed, and translated the source program into a relatively low-level IR, so that the values of the names appearing in the IR can be represented by quantities that the target machine can directly manipulate, such as integers and floating-point numbers.

We also assume that all syntactic and static semantic errors have been detected, that the necessary type checking has taken place, and that type conversion operators have been inserted wherever necessary. The code generator can therefore proceed on the assumption that its input is free of these kinds of errors.

❖ The Target Program

The instruction-set architecture of the target machine has a significant impact on the difficulty of constructing a good code generator that produces high-quality machine code. The most common target-machine architectures are RISC (reduced instruction set computer), CISC (complex instruction set computer), and stack based.

A RISC machine typically has many registers, three-address instructions, simple addressing modes, and a relatively simple instruction-set architecture. In contrast, a CISC machine typically has few registers, two-address instructions, a variety of addressing modes, several register classes, variable-length instructions, and instructions with side effects.

In a stack-based machine, operations are done by pushing operands onto a stack and then performing the operations on the operands at the top of the stack. To achieve high performance the top of the stack is typically kept in registers. Stack-based machines almost disappeared because it was felt that the stack organization was too limiting and required too many swap and copy operations.

The output of the code generator is the target program. The target program may take a variety of forms i.e., Absolute Machine language, Relocatable Machine code or Assembly language.

- ✓ Producing an absolute machine-language program as output has the advantage that it can be placed in a fixed location in memory and immediately executed. Programs can be compiled and executed quickly.
- ✓ Producing a relocatable machine-language program (often called an *object module)* as output allows subprograms to be compiled separately. A set of relocatable object modules can be linked together and loaded for execution by a linking loader. Although we must pay the added expense of linking and loading if we produce relocatable object modules, we gain a great deal of flexibility in being able to compile subroutines separately and to call other

previously compiled programs from an object module. If the target machine does not handle relocation automatically, the compiler must provide explicit relocation information to the loader to link the separately compiled program modules.

- ✓ Producing an assembly-language program as output makes the process of code generation somewhat easier. We can generate symbolic instructions and use the macro facilities of the assembler to help generate code. The price paid is the assembly step after code generation.

❖ **Instruction Selection**

The code generator must map the IR program into a code sequence that can be executed by the target machine. The complexity of performing this mapping is determined by a factors such as

- ✓ the level of the IR
- ✓ the nature of the instruction-set architecture
- ✓ the desired quality of the generated code

If the IR is high level, the code generator may translate each IR statement into a sequence of machine instructions using code templates. Such statement-by-statement code generation, however, often produces poor code that needs further optimization. If the IR reflects some of the low-level details of the underlying machine, then the code generator can use this information to generate more efficient code sequences.

The nature of the instruction set of the target machine has a strong effect on the difficulty of instruction selection. For example, the uniformity and completeness of the instruction set are important factors. If the target machine does not support each data type in a uniform manner, then each exception to the general rule requires special handling. On some machines, for example, floating-point operations are done using separate registers.

Instruction speeds and machine idioms are other important factors. If we do not care about the efficiency of the target program, instruction selection is straightforward. For each type of three-address statement, we can design a code skeleton that defines the target code to be generated for that construct. For example, every three-address statement of the form x = y + z, where x, y, and z are statically allocated, can be translated into the code sequence

```
LD    R0, y          // R0 = y          (load y into register R0)
ADD   R0, R0, z      // R0 = R0 + z     (add z to R0)
ST    x, R0          // x = R0          (store R0 into x)
```

This strategy often produces redundant loads and stores. For example, the sequence of three-address statements

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

would be translated into

```
LD     R0, b            // R0 = b
ADD    R0, R0, c        // R0 = R0 + c
ST     a, R0            // a = R0
LD     R0, a            // R0 = a
ADD    R0, R0, e        // R0 = R0 + e
ST     d, R0            // d = R0
```

Here, the fourth statement is redundant since it loads a value that has just been stored, and so is the third if *a* is not subsequently used.

The quality of the generated code is usually determined by its speed and size. On most machines, a given IR program can be implemented by many different code sequences, with significant cost differences between the different implementations. A naive translation of the intermediate code may therefore lead to correct but unacceptably inefficient target code.

For example, if the target machine has an "increment" instruction (INC), then the three-address statement **a = a + 1** may be implemented more efficiently by the single instruction *INC a*, rather than by a more obvious sequence that loads *a* into a register, adds one to the register, and then stores the result back into *a*:

```
LD     R0, a            // R0 = a
ADD    R0, R0, #1       // R0 = R0 + 1
ST     a, R0            // a = R0
```

We need to know instruction costs in order to design good code sequences but, unfortunately, accurate cost information is often difficult to obtain. Deciding which machine-code sequence is best for a given three-address construct may also require knowledge about the context in which that construct appears.

❖ Register Allocation

A key problem in code generation is deciding what values to hold in what registers. Registers are the fastest computational unit on the target machine, but we usually do not have enough of them to hold all values. Values not held in registers need to reside in memory. Instructions involving register operands are invariably shorter and faster than those involving operands in memory, so efficient utilization of registers is particularly important. The use of registers is often subdivided into two sub-problems:

i. *Register allocation*, during which we select the set of variables that will reside in registers at each point in the program.

ii. *Register assignment*, during which we pick the specific register that a variable will reside in.

Finding an optimal assignment of registers to variables is difficult, even with single-register machines. Mathematically, the problem is NP-complete. The problem is further complicated because the hardware and/or the operating system of the target machine may require that certain register-usage conventions be observed.

Example: Certain machines require register-pairs (an even and next odd numbered register) for some operands and results. For example, on some machines, integer multiplication and integer division involve register pairs. The multiplication instruction is of the form

M x, y

where x, the multiplicand, is the even register of an even/odd register pair and y, the multiplier, is the odd register. The product occupies the entire even/odd register pair. The division instruction is of the form

D x, y

where the dividend occupies an even/odd register pair whose even register is x; the divisor is y. After division, the even register holds the remainder and the odd register the quotient.

❖ Evaluation order

The order in which computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others. However, picking a best order in the general case is a difficult NP-complete problem. Initially, we shall avoid the problem by generating code for the three-address statements in the order in which they have been produced by the intermediate code generator.

8.3. The Target Machine

Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator. Unfortunately, in a general discussion of code generation it is not possible to describe any target machine in sufficient detail to generate good code for a complete language on that machine.

Implementing code generation requires thorough understanding of the target machine architecture and its instruction set. We shall use as a target language assembly code for a simple computer that is representative of many register machines.

❖ A Simple Target Machine Model

Our target computer models a three-address machine with load and store operations, computation operations, jump operations, and conditional jumps. The underlying computer is a byte-addressable machine with *n* general-purpose registers, R_0, R_1, . . ,R_{n-1}.

The following are instructions available.

Load operations: The instruction *LD dst, addr* loads the value in location *addr* into location *dst*. This instruction denotes the assignment *dst = addr*. The most common form of this instruction is *LD r, x* which loads the value in location *x* into register *r*. An instruction of the form *LD* r_l, r_2 is a *register-to-register copy* in which the contents of register r_2 are copied into register r_l.

Store operations: The instruction *ST x, r* stores the value in register *r* into the location *x*. This instruction denotes the assignment *x = r*.

Computation operations of the form *OP dst*, src_l, src_2, where *OP* is an operator like ADD or SUB, and *dst*, src_l , and src_2 are locations, not necessarily distinct. The effect of this machine instruction is to apply the operation represented by *OP* to the values in locations src_l and src_2, and place the result of this operation in location *dst*. For example, *SUB* r_l, r_2, r_3 computes $r_l = r_2 - r_3$.

Any value formerly stored in r_l is lost, but if r_l is r_2 or r_3, the old value is read first. Unary operators that take only one operand do not have a src_2.

Unconditional jumps: The instruction *BR L* causes control to branch to the machine instruction with label *L*. (*BR* stands for branch.)

Conditional jumps of the form *Bcond r, L*, where *r* is a register, *L* is a label, and *cond* stands for any of the common tests on values in the register *r*. For example, *BLTZ r, L* causes a jump to label *L* if the value in register *r* is less than zero, and allows control to pass to the next machine instruction if not.

We assume our target machine has a variety of addressing modes, listed below:

- In instructions, a location can be a **variable name** *x* referring to the memory location that is reserved for *x* (that is, the l-value of x).
- A location can also be an **indexed address** of the form **a(r)**, where *a* is a variable and *r* is a register. The memory location denoted by *a(r)* is computed by taking the 1-value of *a* and adding to it the value in register r. For example, the instruction *LD* R_1, *a*(R_2) has the effect of setting R_l = *contents (a + contents* (R_2)), where *contents(x)* denotes the contents of the register or memory location represented by *x*. This addressing mode is useful for accessing arrays, where *a* is the base address of the array (that is, the address of the first element), and *r* holds the number of bytes past that address we wish to go to reach one of the elements of array *a*.
- A memory location can be an **integer indexed by a register**. For example, *LD* R_1, *100*(R_2) has the effect of setting R_1 = *contents(100 + contents*(R_2)), that is, of loading into R_1 the value in the memory location obtained by adding 100 to the contents of register R_2. This feature is useful for following pointers (the next example, below).
- We also allow two **indirect addressing** modes: ***r** means the memory location found in the location represented by the contents of register *r* and ***100(r)** means the memory location found in the location obtained by adding 100 to the contents of *r*. For example, *LD* R_1, **100*(R_2) has the effect of setting R_1 = *contents(contents(l00 + contents*(R_2))), that is, of loading into R_1 the value in the memory location stored in the memory location obtained by adding 100 to the contents of register R_2.
- Finally, we allow an **immediate constant addressing** mode. The constant is prefixed by **#.** The instruction *LD* R_1, *#100* loads the integer 100 into register R_1, and *ADD* R_1, R_1, *#100* adds the integer 100 into register R_1.

Example-1: The three-address statement *x = y - z* can be implemented by the machine instructions:

```
LD    R1, y          // R1 = y
LD    R2, z          // R2 = z
SUB   R1, R1, R2     // R1 = R1 - R2
ST    x, R1          // x = R1
```

Example-2: Suppose *a* is an array whose elements are 8-byte values, perhaps real numbers. Also assume elements of *a* are indexed starting at 0. We may execute the three-address instruction *b = a[i]* by the machine instructions:

```
LD    R1, i              // R1 = i
MUL   R1, R1, 8          // R1 = Rl * 8
LD    R2, a(R1)          // R2 = contents(a + contents(R1))
ST    b, R2              // b = R2
```

That is, the second step computes *8i*, and the third step places in register *R_2* the value in the ith element of *a* - the one found in the location that is *8i* bytes past the base address of the array *a*.

Example-3: Similarly, the assignment into the array *a* represented by three-address instruction a[j] = c is implemented by the following machine instruction:

```
LD    R1, c              // R1 = c
LD    R2, j              // R2 = j
MUL   R2, R2, 8          // R2 = R2 * 8
ST    a(R2),  R1         // contents(a +  contents(R2)) = R1
```

Example-4: To implement a simple pointer indirection, such as the three-address statement *x = *p*, we can use machine instructions like:

```
LD   R1, p          // R1 = p
LD   R2, 0(R1)      // R2 = contents(0 + contents(R1))
ST   x, R2          // x = R2
```

Example-5: The assignment through a pointer **p = y* is similarly implemented in machine code by:

```
LD   R1, p          // R1 = p
LD   R2, y          // R2 = y
ST   0(R1), R2      // contents(0 + contents(R1)) = R2
```

Example-6: Finally, consider a conditional-jump three-address instruction like *if x < y goto M*. The machine-code equivalent would be something like:

```
LD     R1, x        // R1 = x
LD     R2, y        // R2 = y
SUB    R1, R1, R2   // R1 = R1 - R2
BLTZ  R1, M         // if R1 < 0 jump to M
```

Here, M is the label that represents the first machine instruction generated from the three-address instruction that has label L.

❖ Program and Instruction Costs

We often associate a cost with compiling and running a program. Depending on what aspect of a program we are interested in optimizing, some common cost measures are the length of compilation time and the size, running time and power consumption of the target program.

Determining the actual cost of compiling and running a program is a complex problem. Finding an optimal target program for a given source program is an undecidable problem in general, and many of the sub-problems involved are NP-hard.

We shall assume each target-language instruction has an associated cost. For simplicity, we take the cost of an instruction to be one plus the costs associated with the addressing modes of the operands. This cost corresponds to the length in words of the instruction. Addressing modes involving registers have zero additional cost, while those involving a memory location or constant in them have an additional cost of one, because such operands have to be stored in the words following the instruction.

If space is important, then we should clearly minimize instruction length. However, doing so has an important additional benefit. For most machines and for most instructions, the time taken to fetch an instruction from memory exceeds the time spent executing the instruction. Therefore, by minimizing the instruction length we also tend to minimize the time taken to perform the instruction as well.

Some examples:

- The instruction *LD R_0, R_1* copies the contents of register R_1 into register R_0. This instruction has a cost of one because no additional memory words are required.
- The instruction *LD R_0, M* loads the contents of memory location M into register R_0. The cost is two since the address of memory location M is in the word following the instruction.
- The instruction *LD R_1, *100(R_2)* loads into register R_1 the value given by contents(contents(l00 + contents(R2))). The cost is three because the constant 100 is stored in the word following the instruction.
- The instruct ion *ADD R_1, R_1, #1* adds the constant 1 to the contents of R_1, and has cost two, since the constant 1 must appear in the next word following the instruction.

Some of the difficulties in generating code for this machine can be seen by considering what code to generate for a three-address statement of the form *a = b + c* where b and c are simple variables in distinct memory locations denoted by these names. This statement can be implemented by many different instruction sequences. Here are a few examples:

```
1).    LD R0, b
       ADD R0, R0, c
       LD a, R0                Cost = 6
2).    LD a, b
       ADD a, a, c             Cost = 6
```

Assuming R_0, R_1, and R_2 contain the addresses of a, b, and c, respectively, we can use:

3). LD $*R_0$, $*R_1$

ADD $*R_0$, $*R_0$, $*R_2$ Cost = 2

Assuming R_1 and R_2 contain the values of b and c, respectively, and that the value of b is not needed after the assignment, we can use:

4) ADD R_1, R_1, R_2

LD a, R_1 Cost = 3

8.4. Basic Blocks and Flow graphs

This section introduces a graph representation of intermediate code, called *flow graph*, which is helpful for discussing code generation algorithms, 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 use. We can do a better job of instruction selection by looking at sequences of three-address statements.

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.

❖ Basic block

A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end. 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: Partitioning three-address instructions into basic blocks.

Input: A sequence of three-address instructions.

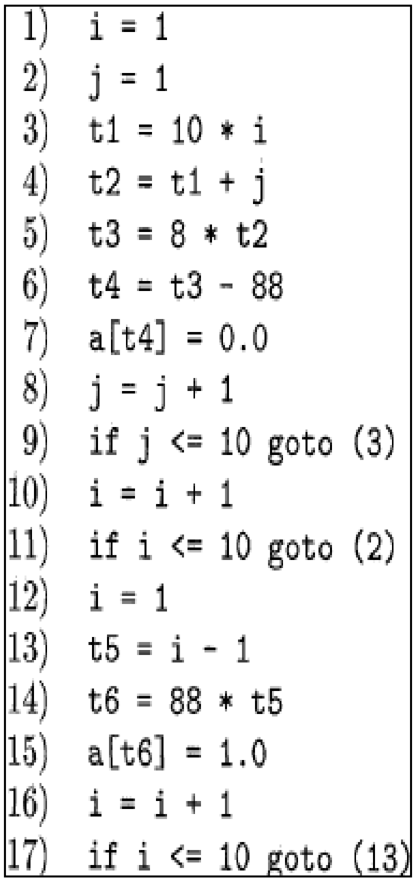
Output: A list of the basic blocks for that sequence in which each instruction is assigned to exactly one basic block.

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.

For example, let us take the following intermediate code which sets a 10 x 10 matrix to an identity matrix.



```
 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)
```

First, instruction 1 is a leader by rule (1) of the above Algorithm. 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.

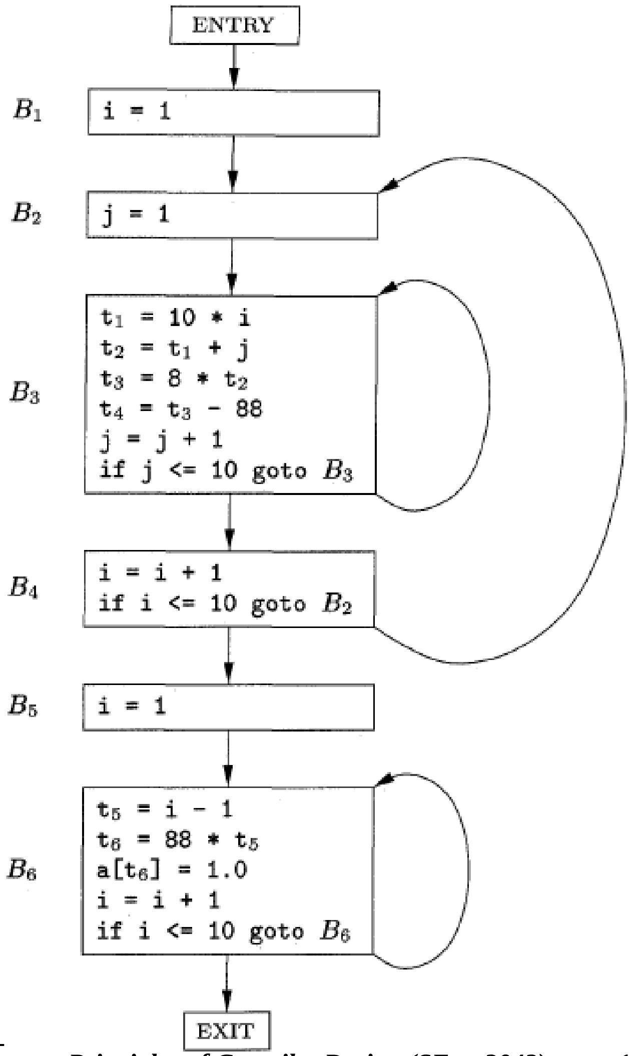
❖ **Flow Graphs**

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.



For example: The set of basic blocks constructed in previous example yields the following flow graph. 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.