Unit 4

Chapter 6

Code Generation

Structure of the Unit

1. Issues in the design of Code Generator
2. The Target Machine
3. Run-time Storage Management
4. Basic blocks and Flow graphs
5. Next-use information
6. A Simple Code Generator
7. Register allocation and assignment
8. The dag representation of basic blocks
9. Generating code from dags.

6.1 Course Outcomes

After the successful completion of this unit, the student will be able to:

1. Identify and analyze common challenges in the design of a code generator.
2. Analyze how machine architecture influences code generation decisions.
3. Implement basic memory management techniques in generated code.
4. Analyze the control flow within a program and its impact on code generation.
5. Implement algorithms for register allocation and assignment.

6.2 Introduction

Code generation is the final phase of the compiler. It takes the intermediate representation of the source program as input and produces the equivalent target program as output. The target code must be correct. It should make effective use of resources of the target machine. Also, the code generator itself must run efficiently. It is very difficult to generate the optimal code. Some heuristic techniques are used to generate good code. The choice of heuristics is important. These techniques need not necessarily generate optimal code.

6.3 Issues in The Design of Code Generator

Even though the code generation is specific to a target machine and the operating system used, in this section we shall discuss the generic issues of code generation. They are;

1. **Input to the Code Generator:**

It consists of

- the intermediate representation of the source program and
- information in the symbol table to determine the run-time addresses of the data objects denoted by the names.

The intermediate representation may be

- Linear representations like postfix notations
- Three-address representations
- Virtual representations like stack machine code
- Syntax trees, directed acyclic graphs etc.,

2. **Target Programs:**

The target program may take various forms like *absolute machine language, relocatable machine language* or *assembly language*.

Absolute Machine Language:

It can be placed in a fixed location in memory and immediately executed.

Relocatable Machine Language:

The subprograms can be compiled separately. A set of relocatable modules can be linked together and loaded for execution. Linking and loading takes more time.

Assembly Language Programs:

Generate symbolic instructions and mnemonic operands. Use macro facilities of the assembler to generate the code.

3. **Memory Management:**

Data objects(names) in the source program are mapped to addresses in the run-time memory by both the front-end and code generator cooperatively. The name in a three-address statement refers to symbol table entries. The type of the symbol determines its width, which is obtained from the symbol table. From the symbol table information, a relative address can be obtained for a name in the data area of the procedure.

To generate the machine code, labels in the three-address statements(flow-of-control) need to be converted to addresses of instructions. This process is similar to *backpatching*. Suppose the labels refer to the indices in the quadruple array. As we scan the quadruple, we can determine the location of the first machine instruction generated for that quadruple. This is done by maintaining a count of the number of words used for the instructions generated so far.

If the instruction is j: `goto` i then

- If $i < j$, we simply generate a jump instruction with the target address equal to the address of the first machine instruction for *quadruple* i.
- If $i > j$, i.e., it is a forward jump. The quadruple i is not yet translated and we do not know the machine address for quadruple i. The location of the first machine instruction of quadruple j is stored in a list for quadruple i. When quadruple i is processed we know the address of the first machine instruction for quadruple i and it is filled in the proper machine location for all instructions in the list for quadruple i.

4. Instruction Selection:

The instruction set of the target machine is important in selecting the machine instructions for the quadruples. The selection of the machine instructions should be such that the target code generated must occupy less space and execute faster.

We can create a code skeleton(template) for every type of three-address statement, and use it as a basis for generating the target code. For example, three-address instruction of the form $x = y + z$, may be translated into the code sequence

```
MOV y, R0 /* load y into register R0 */
ADD z, R0  /* add z to R0 */
MOV R0, x /* store R0 into x */
```

In this way the target code may be produced for every three-address statement, using its template. But this may produce a poor-quality target code. For example, the sequence of statements

$a = b + c, \qquad d = a + e$

Would be translated into

```
MOV b, R0
ADD c, R0
MOV R0, a
MOV a, R0
ADD e, R0
MOV R0, d
```

In the above code the third and fourth statements are redundant.

Similarly, when the statement $a = a + 1$ is translated, the code generated is

```
MOV a, R0
ADD #1, R0
MOV R0, a
```

If the instruction set for the target machine has an instruction for increment, i.e., INC, the above code will be only one instruction, i.e., `INC a`.

5. Register Allocation

Instructions involving register operands are usually shorter and faster. Hence efficient utilization of registers is important in the code generation phase. It is divided in to two subproblems:

i) *Register Allocation*: Selecting the set of variables whose value reside in the registers.

ii) *Register Assignment*: Pick the specific register for a variable.

Finding an optimal assignment of registers is a difficult task. The problem may become complicated if the hardware/operating system imposes some register usage conventions. Some machines require *register-pairs*(an even and the next odd numbered register) for some operations. For example, IBM system/370 uses register pairs for multiplication and division.

The *multiplication instruction* is of the form

```
M x,y
```

where `x`, the multiplicand, is the even register in the register pair and its value is taken from the odd numbered register. For example, if the instruction is `M R2,y` then, the multiplicand is taken from `R3`. The product is stored in the register pair(most significant bits in `R2` and least significant bits in `R3`).

The *division instruction* is of the form

```
D x,y
```

The 64-bit dividend occupies the register pair whose even register is `x`. After division the even register holds the remainder and the odd register holds the quotient.

Consider the three-address instructions in the fig 6.1 in which the only difference is the operator in the second instruction.

```
t = a + b
t = t * c
t = t / d
```

(a)

```
t = a + b
t = t + c
t = t / d
```

(b)

Fig 6.1 Two three-address code sequences

```
L R1, a
A R1, b
M R0, c
D R0, d
ST R1, t
```

(a)

```
L R0, a
A R0, b
A R0, c
SRDA R0, 32
D R0, d
ST R1, t
```

(b)

Fig 6.2 Optimal machine-code sequences

The first two instructions in Fig 6.2(a) place the multiplicand $a + b$ in the odd numbered register `R1` of the register pair. The multiplication instruction then uses the even numbered register `R0` as an operand. The first two instructions in Fig 6.2(b) place the dividend $a + b$ in the even numbered register `R0` of the register pair. The `SRDA` instruction shifts the contents of `R0` to `R1` and places `0` in `R0`, i.e., the 32-bit dividend in `R0`, i.e., $a + b + c$ is converted to a 64-bit number in the register pair `R0-R1`.

6. Choice of Evaluation Order

The efficiency of computations can be influenced by the order in which they are performed, even if the end result remains the same. Certain computation sequences may demand fewer registers to store intermediate results. Determining the optimal order poses a challenging task. Initially, the code is generated following the sequence of three-address statements.

7. Approaches to Code Generation

The most important criterion for code generator is to produce the correct code. Designing a code generator that can be easily implemented, tested and maintained is an important issue.

6.4 The target Machine

Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator. In this section we use a representative target machine. The code generation techniques presented in this section are applicable for many other classes of machines.

Our target machine is a byte-addressable machine. Each word is 4-bytes long. There are n general purpose registers `R0, R1, . . . , Rn-1`. The machine has two-address instructions of the form

```
op source, destination
```

`op` is an op-code, and `source, destination` are data fields. Few of the op-codes are shown below:

`MOV` (move the source to destination)

`ADD` (add source to destination)

`SUB` (subtract source from destination)

Instruction Costs

The cost of an instruction is directly proportional to its length. In other words, the longer the instruction, the more time it takes to fetch it from memory. Instructions that involve registers are shorter and, consequently, execute more quickly. On the other hand, instructions with memory operands are longer and require more time for execution. For example, the instruction `MOV R0, R1` occupies less space and executes faster, whereas the instruction `MOV R5,M` is longer and takes more time.

The challenge in code generation arises from determining the appropriate code for a three-address instruction. For example, a three-address instruction of the form $a = b + c$ may be implemented by different machine instruction sequences. Few sequences are illustrated below:

```
1. MOV b, R0
   ADD c, R0
   MOV R0, a
2. MOV b, a
   ADD c, a
```

6.5 Run-Time Storage Management

Information required during the execution of a procedure is stored in an activation record. In this section we explore what code to generate to manage the activation records at run-time. In section 4.5 the storage allocation strategies were presented, namely static allocation and stack allocation. In static allocation the position of an activation record is fixed at compile time. In stack allocation, a new activation record is pushed on the stack for each procedure call. The activation record is popped upon returning from the procedure call.

Since run-time allocation and de-allocation of activation records occurs as a part of procedure call and return sequences, we focus on the following three-address statements.

1. `call`,
2. `return`,
3. `halt`, and
4. `action`, a placeholder for other statements.

Static Allocation

Consider the code needed to implement the static allocation. The call statement is implemented by two target-machine instructions. A `MOV` instruction saves the return address and a `GOTO` instruction transfers control to target code for the called procedure. The instructions are as follows:

`MOV` $\#here + 20, callee.static\_area$

`GOTO` $callee.code\_area$

The attributes $callee.static\_area$ and $callee.code\_area$ are constants.

$\boldsymbol{callee.static\_area}$: Address of the activation record

$\boldsymbol{callee.code\_area}$: Address of the first instruction for the called procedure.

The source $\#here + 20$ in the `MOV` instruction is the return address, i.e., the address of the instruction following the `GOTO` instruction. The code for a procedure ends with a return to the calling procedure. The return statement is implemented by

GOTO $* callee.static\_area$

which transfers control to the address saved at the beginning of the activation record.

Stack Allocation

Static allocation can become stack allocation by using relative addresses for storage in activation records. The position of the activation record for a procedure is not known until run-time. In stack allocation, this position is usually stored in a register.

The relative addresses in an activation record are taken as offsets from some known position, in the activation record. For convenience, we use positive offsets by maintaining in a register SP a pointer to the beginning of the activation record on the top of the stack. When a procedure is called, the calling procedure increments SP and transfers control to the called procedure. When the control returns to the calling procedure, it decrements SP, thereby deallocating the activation record of the called procedure.

The code for the first procedure initializes the stack by setting SP to the start of the stack area in the memory.

```
MOV #stackstart, SP        /* initialize the stack */
Code for the procedure
HALT                       /*terminate the execution */
```

A procedure call sequence increments SP, saves the return address, and transfer control to the called procedure.

```
ADD #caller.recordsize, SP
MOV #here + 16, *SP        /* save the return address */
GOTO callee.code_area
```

The attribute $caller.recordsize$ represents the size of the activation record. The ADD instruction makes SP to point to the beginning of the next activation record. The source $\#here + 16$ in the MOV instruction is the address of the instruction following GOTO. It is saved in the address pointed to by SP.

The return sequence consists of two parts. Firstly, the called procedure transfers control to the return address using

```
GOTO *0(SP)
```

The second part of the return sequence is in the caller, which decrements SP, restoring SP to its previous value:

```
SUB #caller.recordsize, SP
```

Run-Time Addresses for Names

The names in the three-address code must be replaced by code to access their storage locations. For example, consider the copy statement `x = 0`. Suppose the symbol-table entry for `x` contains a relative address `16`. First consider the case in which `x` is in statically allocated area beginning at the address $static$. Then the run-time address for `x` is $static + 16$. The assignment `x = 0` then translates into

```
static[16] = 0
```

If the static area starts at `100`, the target code for the above statement is

```
MOV #0, 116
```

6.6 Basic Blocks and Flow Graphs

Flow graph is a pictorial representation of three-address statements. Flow graphs are useful in understanding code generation algorithms. Nodes in the flow graph represent computations, and the edges represent the flow of control.

Basic Blocks

A *basic block* is a sequence of consecutive statements where control starts at the beginning and exits at the end without interruption or branching, except at the end. The following sequence of three-address statements forms a basic block:

```
t1 = a * a
t2 = a * b
t3 = 2 * t2
t4 = t1 + t3
t5 = b * b
t6 = t4 + t5
```

The following algorithm can be used to partition a sequence of three-address statements into basic blocks.

Input: A sequence of three-address statements.

Output: A list of basic blocks with each three-address statement in exactly one block.

Method:

1. Determine the set of *leaders*, i.e., the first statements of basic blocks. Following rules are used to determine the *leaders*.
   a) The first statement is a *leader*.
   b) Any statement that is the target of a conditional or unconditional branch is a *leader*.
   c) Any statement that immediately follows a conditional or unconditional branch is a *leader*.
2. For each *leader*, its basic block consists of the *leader* and all statements up to but not including the next *leader* or the end of the program.

Example: Consider the following three-address code

```
(1)   prod = 0
(2)   i = 1
(3)   t1 = 4 * i
(4)   t2 = a [ t1 ]
(5)   t3 = 4 * i
(6)   t4 = b [ t3 ]
(7)   t5 = t2 * t4
(8)   t6 = prod + t5
(9)   prod = t6
(10)  t7 = i + 1
(11)  i = t7
(12)  If i <= 20 goto (3)
```

- In the above three-address code statement (1) is a *leader* as it is the first statement(by rule 1(a)).
- Statement (3) is a *leader*, since it is the destination of the branch statement, i.e., statement (12)(by rule 1(b).
- By rule 1(c) the statement following statement (12) is a leader.

Transformations on Basic Blocks

A basic block computes a set of expressions. These expressions are values of the names live on exit from the block. Two basic blocks are said to be *equivalent* if they compute the same set of expressions.

A number of transformations can be applied to a basic block without changing the set of expressions computed by the block. Many of these transformations are useful for improving the quality of the target code. There are two important classes of transformations that can be applied to basic blocks. They are:

- Structure preserving transformations.
- Algebraic transformations.

Structure Preserving Transformations

The main structure preserving transformations are:

1. Common subexpression elimination
2. Dead code elimination
3. Renaming of temporary variables
4. Interchange of two independent adjacent statements

1. Common Subexpression Elimination

Consider the basic block

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

In the above sequence of instructions, the expressions in the second and fourth statements compute the same values. Hence, the above sequence of statements may be rewritten as

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

In the modified code above the computation `a - d` is done only once. The target code generated from this sequence of three-address instructions occupies less space and also executes faster. Also, the first and third statements have the same expression `b + c` on the right-hand side. But the second statement redefines the value of `b`. Hence, the value of `b` in the third statement is different from value of `b` in the first statement, and the first and third statements do not produce the same result.

2. Dead-Code Elimination

Suppose a basic block contains the statement `a = b - c` and the value of the variable `a` is never used in the subsequent statements. In this case the statement `a = b - c` can be safely removed without changing the value of the basic block. The variable `a` is said to be dead and the statement `a = b - c` is the dead-code.

3. Renaming Temporary Variables

Suppose there is a statement `t = a + b`, where `t` is temporary variable. If we change this statement to `u = a + b`, where `u` is another temporary, and replace all references to `t` by `u`, then the value of the basic block is not changed. The new block with this transformation is called a *normal-form* block.

4. Interchange of Statements

Suppose a basic block contains the following adjacent statements

$t_1 = a + b$

$t_2 = x + y$

These two statements can be interchanged without affecting the value of the basic block, if and only if t_1 is neither `x` nor `y` and if t_2 is neither `a` nor `b`.

5. Algebraic Transformation

To simplify the expressions or replace expensive(time consuming) operations by cheaper ones, many algebraic transformations can be done without changing the value of the basic block. For example, the statements

```
a = a + 0
```

or

```
b = b * 1
```

can be eliminated from a basic block.

The statement of the form

`a = b ** 2 ( a = b ^ 2)` i.e., exponential operator, requires a function call, requiring more time. It can be replaced by a cheaper statement

```
a = b * b
```

Flow Graphs

We can construct a directed graph called *flow graph* to depict the flow of control between the basic blocks. The nodes are the basic blocks. The basic block whose *leader* is the first statement is identified as the *initial node*. There is a directed edge from block B_1 to block B_2, if the control flows from block B_1 to B_2. We say the control flows from B_1 to B_2 if;

1. There is a conditional or unconditional jump from the last statement of B_1 to the first statement of B_2, or
2. B_2 immediately follows B_1 in the order of the program and B_1 does not end in an unconditional jump.

We say that B_1 is a *predecessor* of B_2 and B_2 is a *successor* of B_1.

Representation of Basic Blocks

Basic blocks can be represented by various data structures. One of the representations is a record. Each record represents a basic block and consists of a count of the number of quadruples in the block. The count is followed by a pointer to the leader and by the list of predecessors and successors of the block. An alternative is to make a linked list of the quadruples in each block.

Loops

Briefly a loop in a flow graph is a collection of nodes such that

1. All the bodes in the collection are strongly connected; between every pair of nodes there is a path.
2. The collection of nodes has a unique entry, that is, a node in the loop such that the only way to reach a node of the loop from a node outside the loop is to first go through the entry.

A loop that contains no other loops is called an *inner* loop.

6.7 Next-Use Information

Next-use information helps the compiler determine the point in the program execution where the value of a variable is no longer needed. This information is particularly useful for optimizing register allocation and minimizing the use of memory. If the name in a register is no longer needed, then the register can be assigned to some other name.

Computing Next-Uses

Suppose a three-address *statement i* assigns a value to `x`. If *statement j* has `x` as an operand, and control flows from *statement i* to *j*, along a path such that `x` is not assigned a new value on the path. We say *statement j* uses the value of `x` computed at *statement i*.

Given the three-address statement `x = y + z`, we wish to determine the next uses of `x`, `y` and `z` in the current block. The algorithm to determine the next uses makes a backward pass on each basic block, starting from the end to the beginning of the block. During the scan, the algorithm records in the symbol table whether each name `x` has a next use and if not whether it is live on exit from the block.

Suppose we reach the three-address *statement i* `x = y + z`, in the backward scan. Then we do the following:

1. Attach to *statement i* the information currently found in the symbol table, regarding the next use and liveliness 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*. the order of steps (2) and (3) may not be interchanged.

Storage for Temporary Names

Whenever the compiler creates new temporaries, space has to be allocated to hold the values of these temporaries. The size of the field for the temporaries in the activation record grows with the number of temporaries. If two temporaries are not *live* simultaneously, they can share the same location. Next use information helps the compiler to make these decisions. In many cases temporaries can be packed in to registers, if available. Consider the three-address code:

t_1 = a * a

t_2 = a * b

t_3 = 2 * t_2

t_4 = t_1 + t_3

t_5 = b * b

t_6 = t_4 + t_5

The six temporaries in the above basic block can be packed into two locations.

t_1 = a * a

t_2 = a * b

t_2 = 2 * t_2

t_2 = t_1 + t_2

t_1 = b * b

t_2 = t_2 + t_1

6.8 A Simple Code Generator

The code generator produces the target code for a sequence of three-address statements. It systematically processes each statement, keeping track of whether any operands are currently stored in registers and leveraging this information to its advantage. We assume for each operator in a three-address statement there is a corresponding target-language operator. We also assume that computed results can be left in the registers as long as possible. They are later stored in the memory

a) If the register is needed for another computation.

b) Just before a procedure call, return or a jump statement.

Condition (b) implies that everything must be stored in memory locations at the end of a basic block.

The code generator generates a single instruction `ADD Rj, Ri` leaving the result in the register `Ri` for the three-address statement `a = b + c`, with the following conditions:

a) Register `Ri` contains `b`

b) Register `Rj` contains `c` and

c) `b` is not live after the statement, i.e., `b` is not used after the statement.

If `Ri` contains `b` but `c` is in memory then the code sequence is

```
ADD c, Ri
```

or

```
MOV c, Rj
ADD Rj, Ri
```

provided `b` is subsequently not live.

Register and Address Descriptors

The code generation algorithm uses descriptors to keep track of register contents and addresses for names.

1. A register descriptor keeps track of what is currently in each register. It is consulted whenever a new register is needed. Initially the register descriptor shows all registers are empty. As the code generation for the block progresses, each register will hold the value of zero or more names at any given time.

2. An address descriptor keeps track of the location where the current value of the name can be found at run time. The location might be a register, a stack location, a memory address, or some set of these. This information can be stored in the symbol table, and is used to determine the access method for a name.

A Code-Generation Algorithm

The input to the code generation algorithm is a sequence of three-address instructions constituting a basic block. For each three-address instruction of the form `x = y op z` we perform the following actions.

1. Invoke a function *getreg* to determine the location `L` where the result of the computation `y op z` should be stored. `L` will usually be a register, but it could also be a memory location(in case registers are not available for allocation).

2. Consult the address descriptor for y to determine y', (one of)the current location(s) of y. Prefer the register for y' if the value of y is currently in both memory and a register. If the value of y is not already in L, generate the instruction MOV y',L to place a copy of y in L.

3. Generate the instruction op z',L, where z' is a current location of z. Prefer a register to a memory location if z is in both. Update the address descriptor for x to indicate that x is in location L. if L is a register, update its descriptor to indicate that it contains the value of x, and remove x from all other register descriptors.

4. If the current value of y and/or z have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after the execution of x = y *op* z, those registers no longer will contain y and/or z, respectively.

If the current three-address statement has a unary operator, the steps are similar to those above.

The function *getreg*

The function *getreg* returns the location L to hold the value of x for the assignment x = y *op* z.

1. If the name y is in a register and y is not live has no next-use after execution of x = y *op* z, then return the register of y for L. update the address descriptor of y to indicate that y is no longer in L.

2. If (1) fails, return an empty register for L if there is one.

3. Failing (2), if x has a next use in the block, or *op* is an operator, such as indexing, that requires a register, find an occupied register R. Store the value of register R into a memory location(by MOV R, M) if it is not already in the proper memory location M. Update the address descriptor for M, and return R. If R holds the value of several variables, a MOV instruction must be generated for each variable that needs to be stored.

4. If x is not used in the block, no suitable occupied register can be found, select memory location of x as L.

Generating Code for Other Types of Statements

Indexing and pointer operations in three-address statements are handled in the same manner as binary operations. The table 6.1 shows the code sequences generated for the assignment statements a = b[i] and a[i] = b, assuming b is statically allocated.

Statement	i in Register Ri	i in Memory Mi	i in stack
a = b[i]	MOV b(Ri), R	MOV Mi, R MOV b(R), R	MOV Si(A), R MOV b(R), R
A[i] = b	MOV b, a(Ri)	MOV Mi, R MOV b, a(R)	MOV Si(A), R MOV b, a(R)

Table 6.1 Code sequences for indexed assignments

The register R is the register returned when the function *getreg* is invoked.

Table 6.2 shows the code sequences generated for the pointer assignment statements a = *p and *p = a. in these examples the

Statement	*p* in Register Rp	*p* in Memory Mp	*p* in stack
a = *p	MOV *Rp, a	MOV Mp, R MOV *R, R	MOV Sp(A), R MOV *R, R
*p = a	MOV a, *Rp	MOV Mp, R MOV a, *R	MOV a, R MOV R, *Sp(A)

Table 6.2 Code sequences for pointer assignments

Conditional Statements

Conditional jumps are implemented in one of the two ways. One way is to branch if the value of a designated register meets one of the following conditions: negative, zero, positive, nonnegative, nonzero, and nonpositive. The three-address statement if x < y goto z can be implemented by subtracting y from x in register R, then jumping to z if the value in register R is negative.

A second approach is to use *condition codes* to indicate whether the last quantity computed or loaded into a register is negative, zero or positive. Often the compare(CMP) instruction sets the *condition code* without actually computing a value. The CMP x, y sets the condition code to positive if x > y, to zero if x = y and so on. The instruction if x < y goto z is implemented by

```
CMP x, y
CJ< z
```

6.9 Register Allocation and Assignment

Instructions involving only register operands are shorter and faster than those involving memory operands. But the number of registers is very limited on the target machine. Therefore, efficient utilization of registers is important in generating good code. This section presents strategies for which values in a program reside in registers(register allocation) and in which register a particular value should reside(register assignment). One approach is to assign specific values to particular registers. For example, base addresses to one group of registers, arithmetic computations to another group and so on. The advantage is simpler design of the compiler. The disadvantage is, when applied strictly, the register usage may be inefficient.

Global Register Allocation

In this approach, registers are assigned to frequently used variables, such as loop indices, especially in innermost loops. These registers are maintained consistently across block boundaries on a global scale. One effective strategy for this allocation is to designate a fixed number of registers to store the most actively used values within the inner loops. The method is simple to implement, but the fixed number of registers may not always be right number.

In languages like 'C' and 'Bliss', the programmer may do register allocation through declaration statements.

Usage Counts

To determine the savings by keeping the value of a variable x in a register is to count the number of times x is referred.

Register Assignment for Outer Loops

After assigning registers and generating code for the inner loop, the same idea may be applied progressively to the outer loops. Let the loop L_1 contain the loop L_2, i.e., L1 is an outer loop and L_2 is the inner loop. The names allocated registers in L_2 may not be allocated registers in L_1 - L_2.

Register Allocation by Graph Coloring

When a register is needed for some computation, but no register is available for use, then the contents of one of the registers must be stored(spilled) in a memory location. Graph coloring is a simple technique for allocating registers and managing their storage in the memory.

A two-pass method is used. In the first pass target machine instructions are selected as though the number of registers(symbolic) is infinite. The names used in the three-address instructions become the register names and the three-address instructions become machine-language instructions. Access to stack pointers, display pointers, base registers, or other quantities is done through registers reserved for this purpose.

In the second pass, for each procedure a *register-interference graph* is constructed. The nodes in this graph are symbolic registers. An edge connects the two nodes if one is live at a point where the other is defined. An attempt is made to color the graph using *k* colors, where *k* is the number of registers that can be assigned.

6.10 The DAG representation of Basic Blocks

Directed Acyclic Graphs(DAGs) give a picture of how the value computed by a statement in a block is used in subsequent statements in the block. Construction of DAG helps in determining common sub expressions. A *dag* for a *basic block* has the following labels on the nodes.

1. Leaves are labeled by unique identifiers, either variable names or constants. The leaves represent initial values of names, and we subscript them with `0` to avoid confusion with the labels denoting the current values of names.
2. Interior nodes are labeled by an operator symbol.
3. Nodes are optionally given a sequence of identifiers for labels.

DAG Construction

A DAG is constructed for each basic block. Each statement of the block is processed in turn. When a statement of the form `x = y + z` is seen, look for the nodes that represent the "current" values of `y` and `z`. These could be leaves or interior nodes if `y` and/or `z` have been evaluated by the previous statements. Now a node labeled `+` is created and `y` is made its left child and `z` the right child. This node is now labeled as `x`. However, if there is already a node representing `y + z`, we do not create a new node, but give the existing node the additional label `x`.

Applications of DAG

1. The common subexpressions are detected automatically.
2. Determine which identifiers have their values used in the block.
3. Determine which statements compute values that can be used outside the block.

6.11 Generating Code from DAGs

In this section we shall present the code generation method for a basic block from its dag representation. The advantage is we can easily see how to rearrange the sequence of the machine instructions.

Rearranging the Order

The order in which computations are done affect the cost of the resulting object code. Consider the following block

$t_1 = a + b$

$t_2 = c + d$

$t_3 = e - t_2$

$t_4 = t_1 - t_3$

The code generated for the above three-address statements is shown below

```
MOV a, R0
ADD b, R0
MOV c, R1
ADD d, R1
MOV R0, t1
MOV e, R0
SUB R1, R0
MOV t1, R1
SUB R0, R1
MOV R1, t4
```

On the other hand, suppose the order of statements is rearranged as below

$t_2 = c + d$

$t_3 = e - t_2$

$t_1 = a + b$

$t_4 = t_1 - t_3$

We get the code sequence shown below

```
MOV c, R0
ADD d, R0
MOV e, R1
SUB R0, R1
MOV a, R0
ADD b, R0
```

```
SUB R1, R0
MOV R0, t4
```

In the above code we have been able to save two instructions.

A Heuristic Ordering for DAGs

In the above example the code is improved as the value of t_1 is made available to t_4 immediately. The nodes of the DAG are reordered preserving the edge relationships.

Optimal Ordering for Trees

Optimal order means the order yielding the shortest instruction sequence. This is done by avoiding storage of intermediate results and using minimal number of registers.

6.12 Multiple Choice Questions

Q1. The primary goal of code generation in a compiler is to:

a. Recognize tokens
b. Construct parse trees
c. Type checking
d. Translate the intermediate code into target machine code

Q2. The purpose of register allocation during code generation is:

a. Assigning memory addresses to variables
b. Mapping variables to CPU registers
c. Translating high-level code to assembly language
d. Allocating memory for the code segment

Q3. The purpose of the final code generation phase in a compiler is:

a. Generating assembly code from the intermediate code
b. Generating machine code from the assembly code
c. Converting source code to intermediate code
d. Optimizing code during execution

Q4. Which of the following is a major consideration in code generator design for a target machine?

a. Source code optimization
b. Intermediate code representation
c. Register allocation
d. Intermediate code optimization

Q5. The primary goal of instruction selection in the code generation process is:

a. Generating compact intermediate code
b. Mapping intermediate code to target machine instructions
c. Performing loop optimization
d. Eliminating dead code

Q6. The purpose of run-time storage management in a compiler is:

a. Allocating memory for the compiler itself
b. Managing memory during program execution
c. Generating code for memory allocation
d. Optimizing code for storage efficiency

Q7. Which of the following activities is associated with dynamic memory allocation during program execution?

a. Allocating registers to names
b. Run-time storage management
c. Detecting errors in declaration statements
d. Allocating memory to names in declaration statements

Q8. What is a dynamic memory allocation technique that allows memory to be allocated and deallocated in any order?

a. Stack allocation
b. Heap allocation
c. Linked list allocation
d. Garbage collection

Q9. In the context of run-time storage management, what does "garbage collection" refer to?

a. Allocating memory for unused variables
b. Identifying and reclaiming memory occupied by unused objects
c. Generating code for collecting garbage
d. Managing memory leaks during program execution

Q10. What is the purpose of a stack frame in run-time storage management?

a. Storing global variables
b. Managing memory on the heap
c. Storing local variables and control information
d. Allocating memory for arrays

Q11. Which of the following is a benefit of garbage collection in a programming language?

a. Increased control over memory allocation
b. Prevention of memory fragmentation
c. Automatic reclamation of unused memory
d. Faster program execution

Q12. What is a memory leak in the context of run-time storage management?

a. Failing to release memory when it is no longer needed
b. Unintentional release of allocated memory
c. Allocating memory for large data structures
d. Allocating memory for temporary variables

Q13. A basic block a block of code with

a. Single-entry point and multiple exit points
b. Multiple entry points and a single exit point
c. A single-entry point and a single exit point
d. No entry or exit points

Q14. A flow graph is used for

a. Representing the program's control flow structure
b. Analyzing the data flow within a basic block
c. Managing memory allocation during execution
d. Generating machine code for the target architecture

Q15. In a flow graph, what do the nodes represent?

a. Basic blocks
b. Instructions
c. Variables
d. Registers

Q16. An edge in a flow graph represents

a. Data dependency between instructions
b. Control flow between basic blocks
c. Memory allocation information
d. Register allocation details

Q17. How is next-use information helpful in register allocation?

a. It helps identify variables that will not be used in the future, allowing for efficient register allocation.
b. It has no relevance to register allocation.
c. It ensures that all variables are allocated registers.
d. It is used to track the next use of registers in the program.

Q18. The purpose of register allocation in compiler design is to?

a. Allocate memory for variables
b. Assign registers to variables
c. Manage the flow of control in a program
d. Generate intermediate code

6.13 Summary

Code Generation in Compiler Design is a critical phase that converts intermediate code into machine code or assembly language for the target machine. Key topics covered include addressing efficiency and correctness issues in code generation, balancing code quality and compilation time, understanding the target machine's architecture, and adapting code generation strategies. Runtime storage management involves allocating and managing memory, considering variable lifetimes, and storage locations. Basic Blocks and Flow Graphs are optimized, identifying sequences of consecutive instructions and constructing control flow representations. Next-use information is analyzed for variable optimization, minimizing register spills and reloads. A Simple Code Generator is implemented to understand fundamental concepts while balancing simplicity and efficiency. Register Allocation and Assignment minimize memory accesses by allocating registers and assigning values based on next-use information. The DAG Representation of Basic Blocks involves using Directed Acyclic Graphs (DAGs) to represent expressions, simplifying and optimizing expressions. Generating Code from DAGs transforms representations into efficient machine code, applying optimization techniques. In summary, Code Generation involves translating high-level language constructs into efficient machine code, considering target machine architecture, runtime storage management, basic blocks, flow graphs, register allocation, and DAG usage for code representation and optimization.

6.14 Key Words

- Absolute Machine Language
- Relocatable Machine Language
- Instruction Selection
- Register Allocation
- Register Assignment
- Static Allocation
- Stack Allocation
- Basic Blocks
- Flow Graphs
- Dead Code
- Next Uses