MODULE V - CODE GENERATION & MEMORY ALLOCATION

OBJECT CODE GENERATION:

The final phase in our compiler model is the **code generator**. It takes as input an intermediate representation of the source program and produces as output an equivalent target program.

The requirements traditionally imposed on a code generator are severe. The output code must be correct and of high quality, meaning that it should make effective use of the resources of the target machine. Moreover, the code generator itself should run efficiently.

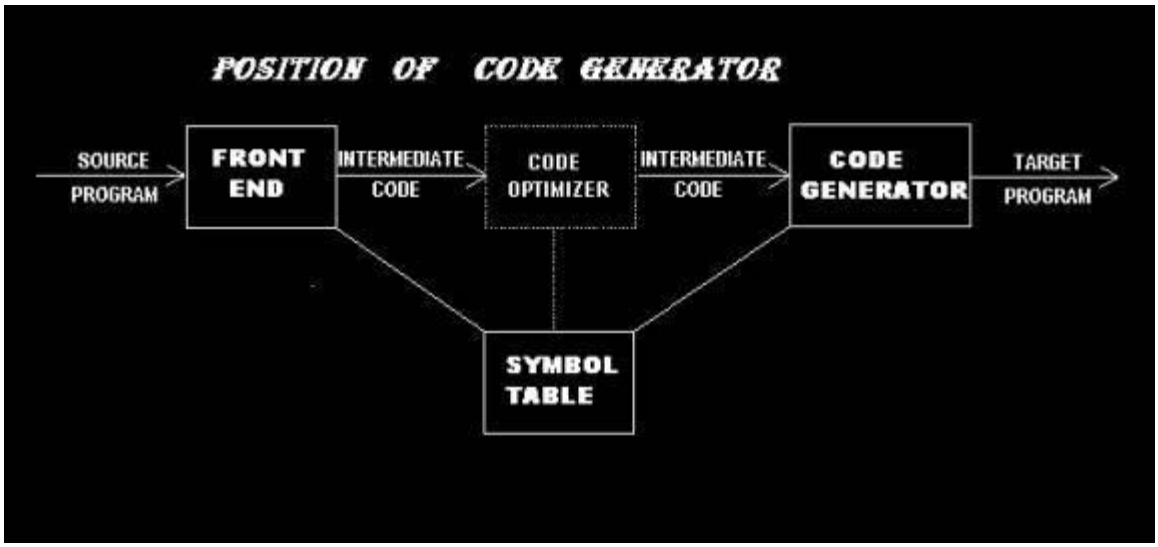


fig. 1

ISSUES IN THE DESIGN OF A CODE GENERATOR

While the details are dependent on the target language and the operating system, issues such as memory management, instruction selection, register allocation, and evaluation order are inherent in almost all code generation problems.

INPUT TO THE CODE GENERATOR

The input to the code generator consists of the intermediate representation of the source program produced by the front end, together 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.

There are several choices for the intermediate language, including: linear representations such as postfix notation, three address representations such as quadruples, virtual machine representations such as syntax trees and dags.

We assume that prior to code generation the front end has scanned, parsed, and translated the source program into a reasonably detailed intermediate representation, so the values of names appearing in the intermediate language can be represented by quantities that the target machine can directly manipulate (bits, integers, reals, pointers, etc.). We also assume that the necessary type checking has take place, so type conversion operators have been inserted wherever necessary and obvious semantic errors (e.g., attempting to index an array by a floating point number) have already been detected. The code generation phase can therefore proceed on the assumption that its input is free of errors. In some compilers, this kind of semantic checking is done together with codegeneration.

TARGET PROGRAMS

The output of the code generator is the target program. The output may take on a variety of forms: absolute machine language, relocatable machine language, or assembly language.

Producing an absolute machine language program as output has the advantage that it can be placed in a location in memory and immediately executed. A small program can be compiled and executed quickly. A number of "student-job" compilers, such as WATFIV and PL/C, produce absolute code.

Producing a relocatable machine language program 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 segments.

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.

Because producing assembly code does not duplicate the entire task of the assembler, this choice is another reasonable alternative, especially for a machine with a small memory, where a compiler must uses several passes.

MEMORY MANAGEMENT

Mapping names in the source program to addresses of data objects in run time memory is done cooperatively by the front end and the code generator. We assume that a name in a three-address statement refers to a symbol table entry for the name.

If machine code is being generated, labels in three address statements have to be converted to addresses of instructions. This process is analogous to the "back patching". Suppose that labels refer to quadruple numbers in a quadruple array. As we scan each quadruple in turn we can deduce the location of the first machine instruction generated for that quadruple, simply by maintaining a count of the number of words used for the instructions generated so far. This count can be kept in the quadruple array (in an extra field), so if a reference such as j: goto i is encountered, and i is less than j, the current quadruple number, we may simply generate a jump instruction with the target address equal to the machine location of the first instruction in the code for quadruple i. If, however, the jump is forward, so i exceeds j, we must store on a list for quadruple i the location of the first machine instruction generated for quadruple j. Then we process quadruple i, we fill in the proper machine location for all instructions that are forward jumps to i.

INSTRUCTION SELECTION

The nature of the instruction set of the target machine determines the difficulty of instruction selection. 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.

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 outlines 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

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

Unfortunately, this kind of statement – by - statement code generation often produces poor code. For example, the sequence of statements

```
a := b +c
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
```

Here the fourth statement is redundant, and so is the third if „a“ is not subsequently used.

The quality of the generated code is determined by its speed and size.

A target machine with a rich instruction set may provide several ways of implementing a given operation. Since the cost differences between different implementations may be significant, a naive translation of the intermediate code may 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, add one to the register, and then stores the result back into a.

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

Instruction speeds are needed to design good code sequence but unfortunately, accurate timing 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 constructappears.

REGISTER ALLOCATION

Instructions involving register operands are usually shorter and faster than those involving operands in memory. Therefore, efficient utilization of register is particularly important in generating good code. The use of registers is often subdivided into two sub problems:

1. During **register allocation,** we select the set of variables that will reside in registers at a point in theprogram.

2. During a subsequent **register assignment** phase, we pick the specific register that a variable will residein.

Finding an optimal assignment of registers to variables is difficult, even with single register values. 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.

Certain machines require **register pairs** (an even and next odd numbered register) for some operands and results. For example, in the IBM System/370 machines integer multiplication and integer division involve register pairs. The multiplication instruction is of the form

```
M    x,y
```

where x, is the multiplicand, is the even register of an even/odd register pair.

The multiplicand value is taken from the odd register pair. The multiplier y is a single register. The product occupies the entire even/odd register pair.

The division instruction is of the form

D x,y

where the 64-bit dividend occupies an even/odd register pair whose even register is x; y represents the divisor. After division, the even register holds the remainder and the odd register the quotient.

Now consider the two three address code sequences (a) and (b) in which the only difference is the operator in the second statement. The shortest assembly sequence for (a) and (b) are given in(c).

Ri stands for register i. L, ST and A stand for load, store and add respectively. The optimal choicefortheregisterintowhich„a‟istobeloadeddependsonwhatwillultimatelyhappentoe.

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

(b) fig. 2 Two three address codesequences

L R1, a	L R0, a
A R1, b	A R0, b
M R0, c	A R0, c
D R0, d	SRDA R0, 32
ST R1, t	D R0, d
	ST R1, t
(a)	**(b)**

fig.3 Optimal machine code sequence

CHOICE OF 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. Picking a best order is another 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.

APPROCHES TO CODE GENERATION

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 code generator must face. Given the premium on correctness, designing a code generator so it can be easily implemented, tested, and maintained is an important design goal.

BASIC BLOCKS AND FLOW GRAPHS

A graph representation of three-address statements, called a **flow graph**, is useful for understanding code-generation algorithms, even if the graph is not explicitly constructed by a code-generation algorithm. Nodes in the flow graph represent computations, and the edges represent the flow of control. Flow graph of a program can be used as a vehicle to collect information about the intermediate program. Some register-assignment algorithms use flow graphs to find the inner loops where a program is expected to spend most of its time.

BASIC BLOCKS

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. 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

A three-address statement x :=y+z is said to define x and to use y or z. A name in a basic block is said to live at a given point if its value is used after that point in the program, perhaps in another basicblock.

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

Algorithm 1: Partition 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. We first determine the set of **leaders**, the first statements of basicblocks.

The rules we use are the following:

I) The first statement is aleader.

II) Any statement that is the target of a conditional or unconditional goto is aleader.

III) Any statement that immediately follows a goto or conditional goto statement is aleader.

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 theprogram.

Example: Consider the fragment of source code shown in fig. 7; it computes the dot product of two vectors a and b of length 20. A list of three-address statements performing this computation on our target machine is shown in fig. 8.

```
begin
    prod := 0;
    i := 1;
    do begin
```

```
        prod := prod + a[i] * b[i];

        i := i+1;

      end

       while i<= 20

    end
```

fig 7: program to compute dot product

Let us apply Algorithm 1 to the three-address code in fig 8 to determine its basic blocks. Statement (1) is a leader by rule (I) and statement (3) is a leader by rule (II), since the last statement can jump to it. By rule (III) the statement following (12) is a leader. Therefore, statements (1) and (2) form a basic block. The remainder of the program beginning with statement (3) forms a second basicblock.

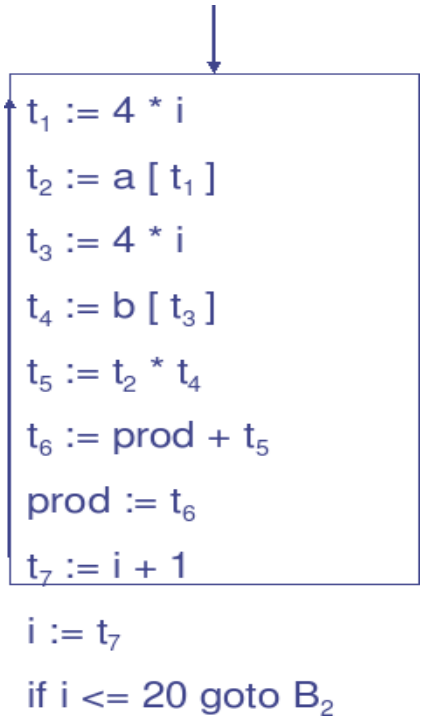
```
(1)  prod := 0
(2)  i :=1
(3)  t1 := 4*i
(4)  t2 := a [ t1 ]
(5)  t3 := 4*i
(6)  t4 :=b [ t3 ]
(7)  t5 :=t2*t4
(8)  t6 := prod+t5
(9)  prod := t6
(10) t7 :=i+1
(11) i := t7
(12) if i<=20 goto (3)
```

fig 8. Three-address code computing dot product

```
prod :=0

  i :=1
```

TRANSFORMATIONS ON BASIC BLOCKS



A basic block computes a set of expressions. These expressions are the values of the names live on exit from 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 code that will be ultimately generated from a basic block. There are two important classes of local transformations that can be applied to basic blocks; these are the structure-preserving transformations and the algebraictransformations.

STRUCTURE-PRESERVING TRANSFORMATIONS

The primary structure-preserving transformations on basic blocks are:

1. Common sub-expressionelimination
2. dead-codeelimination
3. Renaming of temporaryvariables
4. Interchange of two independent adjacentstatements

We assume basic blocks have no arrays, pointers, or procedure calls.

1. Common sub-expressionelimination

Consider the basic block

a:=b+c

b:= a-d

c:= b+c

d:= a-d

The second and fourth statements compute the same expression,

namelyb+c-d, and hence this basic block may be transformed into the equivalent block

a:= b+c

b:= a-d

c:= b+c

d:= b

Although the 1st and 3rd statements in both cases appear to have the same expression on the right, the second statement redefines b. Therefore, the value of b in the 3rd statement is different from the value of b in the 1st, and the 1st and 3rd statements do not compute the same expression.

2. Dead-codeelimination

Suppose x is dead, that is, never subsequently used, at the point where the statement x:=y+z appears in a basic block. Then this statement may be safely removed without changing the value of the basic block.

3. Renaming temporaryvariables

Suppose we have a statement t:=b+c, where t is a temporary. If we change this statement to u:= b+c, where u is a new temporary variable, and change all uses of this instance of t to u, then the value of the basic block is not changed. In fact, we can always transform a basic block into an equivalent block in which each statement that defines a temporary defines a new temporary. We call such a basic block a normal-form block.

4. Interchange ofstatements

Suppose we have a block with the two adjacent statements

t1:= b+c

t2:= x+y

Then we can interchange the two statements without affecting the value of the block if and only if neither x nor y is t1 and neither b nor c is t2. A normal-form basic block permits all statement interchanges that arepossible.

The target machine characteristics are

- Byte-addressable, 4 bytes/word, n registers
- Two operand instructions of theform
- Op source,destination
- Example opcodes: MOV, ADD, SUB,MULT
- Several addressingmodes
- An instruction has an associated cost
- Cost corresponds to length ofinstruction

Addressing Modes & Extra Costs

Mode	Form	Address	Extra cost
Absolute	M	M	1
Register	R	R	0
Indexed	*k*(R)	*k* + *contents*(R)	1
Indirect register	*R	*contents*(R)	0
Indirect indexed	**k*(R)	*contents*(k + *contents*(R))	1

1) Generate target code for the source language statement

"(a-b) + (a-c) + (a-c);"

The 3AC for this can be written as

t := a –b

u := a – c

v := t +u

d := v +u //d live at theend

Show the code sequence generated by the simple code generation algorithm

What is its cost? Can it be improved?

Statements	Generated Code	*RegDes*	*AdDes*
		Registers empty	
t := a - b	MOV a, R0 SUB b, R0	R0 contains t	t in R0
u := a - c	MOV a, R1 SUB c, R1	R0 contains t R1 contains u	t in R0 u in R1
v := t + u	ADD R1, R0	R0 contains v R1 contains u	u in R1 v in R0
d := v + u	ADD R1, R0 MOV R0, d	R0 contains d	d in R0 d in R0 and memory

Total cost=12