4.2 Code Generator

asks for 5 marks in exam normally.

⊛Factors Affecting code generator/code generator design issues: [Imp]

The code generator mainly concern with:

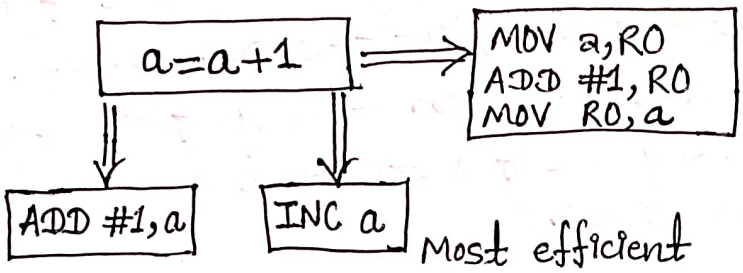
1) Input to the code generator: The input to the code generator is intermediate representation together with the information in the symbol table.

2) The Target Program: The output of the code generator is target code. Typically, the target code comes in three forms such as: absolute machine language, relocatable machine language and assembly language.

The advantage of producing target code in absolute machine form is that it can be placed directly at the fixed memory location and then can be executed immediately. The benefit of such target code is that small programs can be quickly compiled.

3) The target machine: Implementing code generation requires understanding of the target machine architecture and its instruction set.

4) Instruction Selection: Instruction selection is important to obtain efficient code. Suppose we translate three-address code,



5) Register Allocation: Since registers are the fastest memory in the computer, the ideal solution is to store all values in registers. We must choose which values are in the registers at any given time. Actually this problem has two parts:

i) Which values should be stored in registers?

ii) Which register should each select to store value?

The reason for the second problem is that often there are register requirements, e.g, floating-point values in floating-point registers and certain requirements for even-odd register pairs for multiplication/division.

6) Evaluation order: The order of the target code can be affected by the order in which the computations are performed.

⊛ Basic blocks:

A basic block is a sequence of consecutive instructions in which flow of control enters by one entry point and exit to another point without halt or branching except at the end.

Example:

```
    MOV 1, R0
    ADD n, R0
    MOV 2, R1
    MUL R0, R1
    JMP L2
L1: MUL 2, R0
    SUB 1, R1
L2: MUL 3, R1
    JMPNZ R1, L1
```

⟹

```
MOV 1, R0
ADD n, R0
MOV 2, R1
MUL R0, R1
JMP L2
```

```
L1: MUL 2, R0
    SUB 1, R1
```

```
L2: MUL 3, R1
    JMPNZ R1, L1
```

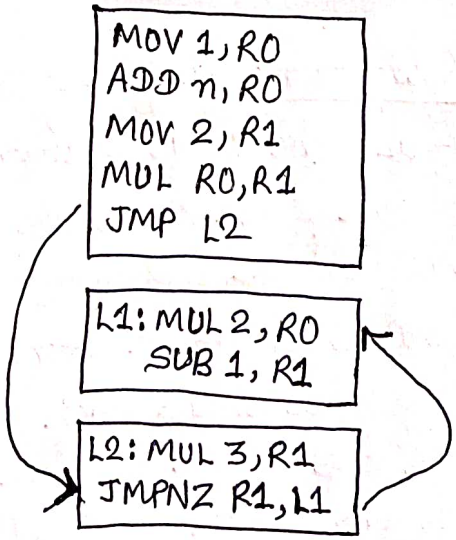
⊛ Flow Graphs:

A flow graph is a graphical representation of a sequence of instructions with control flow edges. A flow graph can be defined at the intermediate code level or target code level. The nodes of flow graphs are the basic blocks ~~of given sequence of instructions then such group of blocks~~ and flow-of-control to immediately follow node connected by directed arrow. Simply, if flow of control occurs in basic blocks of given sequence of instructions then such group of blocks is known as flow graphs.

Example:

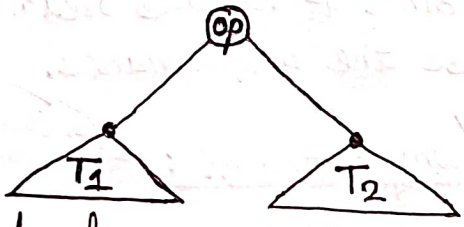
```
    MOV 1, R0
    ADD n, R0
    MOV 2, R1
    MUL R0, R1
    JMP L2
L1: MUL 2, R0
    SUB 1, R1
L2: MUL 3, R1
    JMPNZ R1, L1
```

⟹



⊛ Dynamic programming code-generation algorithm:

The dynamic programming algorithm can be used to generate code for any machine with r interchangable registers $R_0, R_1, \ldots, R_{r-1}$ and load, store, and add instructions. For simplicity, we assume every instruction costs one unit, although the dynamic programming algorithm can easily be modified to work even if each instruction has its own cost.



Contiguous evaluation: Compute the evaluation of T_1, T_2, then evaluate root.
Non-contiguous evaluation: First evaluate part of T_1 leaving the value in a register, next evaluate T_2, then return to evaluate the rest of T_1.

The dynamic programming algorithm uses contiguous evaluation and proceeds in three phases:

1). Compute bottom-up for each node n of the expression tree T, an array C of costs, in which the ith component C[i] is the optimal cost of computing the subtree S rooted at n into a register assuming i registers are available for the computation, for $1 <= i <= r$.

2). Traverse T, using the cost vectors to determine which subtrees of T must be computed in memory.

3). Traverse each tree using the cost vectors and associated instructions to generate the final target code. The code for the subtrees computed into memory locations is generated first.

4.3 Code Optimization

The code optimization in the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources (i.e, CPU, Memory) so that faster-running machine code will result.

Optimization of the code is often performed at the end of the development stage since it reduces readability and adds code that is used to increase the performance.

Code optimization defn and need or importance is imp.

⊛. Need of code optimization / why optimize? :

Code optimization is needed because optimization helps to:

→ Reduce the space consumed and increases the speed of compilation.

→ Manually analyzing datasets involves a lot of time. Hence we make use of software like Tableau for data analysis. Similarly manually performing the optimization is also tedious and is better done using a code optimizer.

→ An optimized code often promotes re-usability.

⊛. Criteria of code optimization:

→ The optimization must be correct, it must not, in any way, change the meaning of program.

→ Optimization should increase the speed and performance of the program.

→ The compilation time must be kept reasonable.

→ The optimization process should not delay the overall compiling process.

⊛Basic optimization techniques: [Imp]

Optimizations techniques are classified into two categories:
→Machine Independent optimization techniques
→ Machine dependent optimization techniques.

1) Machine Independent optimization techniques:

Machine independent optimization techniques are program transformations that improve the target code without taking into consideration any properties of the target machine.

a) Constant Folding: As its name suggests, it involves folding the constants. The expressions that contain the operands having constant values at compile time are evaluated. Those expressions are then replaced with their respective results.

Example: Circumference of Circle = (22/7) × Diameter
Here,
→This technique evaluates the expression 22/7 at compile time.
→The expression is then replaced with its result 3.14.
→This saves time at run time.

b) Constant Propagation: In this technique, if some variable has been assigned some constant value, then it replaces that variable with its constant value in the further program during compilation. The condition is that the value of variable must not get alter in between.

Example: Pi = 3.14, Radius = 10, Area of circle = Pi × Radius × Radius.
Here,
→This technique substitutes the variables 'pi' and 'radius' at compile time.
→It then evaluates the expression 3.14×10×10
→The expression is then replaced with its ~~resu~~ result 314.
→This saves the time at run time.

c) Redundant Code Elimination: In computer programming, redundant code is source code or compiled code in a computer program that is unnecessary such as: code that is never executed (unreachable code).

Example: x = y+z
a = 2+y+b+z
That is reduced by,
a = 2+x+b.

d) **Variable Propagation:** Variable propagation means use of one variable instead of another.

Example: $x = r$

$A = pi \times x^2$

That is reduced by, $A = pi \times r^2$

e) **Strength reduction:** It involves reducing the strength of expressions. This technique replaces the expensive and costly operators with the simple and cheaper ones.

Example:

Code before optimization	Code after optimization
B = A × 2	B = A + A

→ This is because the cost of multiplication operator is higher than that of addition operator.

f) **Loop Optimization:** ← Defn: Loop optimization is the process of increasing execution speed and reducing the overheads associated with loops. Following are loop optimization techniques.

i) **Code Motion / Frequency Reduction:** It is a technique which moves the code outside the loop.

Example:
```
while (i<5000)
    x = i*sin(A)*cos(A);
```

This can be optimized as;
```
t = sin(A)*cos(A);
while (i<5000)
    x = i*t;
```

ii) **Loop jamming / Loop Fusion:** In loop fusion method several loops are merged to one loop.

Example:
```
for i=1 to n do
    for j=1 to m do
        a[i,j]=10.
```

It can be written as;
```
for i=1 to n*m do
    a[i]=10
```

iii) **Loop Unrolling:** The number of jumps and tests can be reduced by writing code two times.

Example:
```
int i=1;
while (i<=100)
{ a[i]=b[i];
  i++;
}
```

It can be written as;
```
int i=1;
while (i<=100)
{ a[i]=b[i];
  i++;
  a[i]=b[i];
  i++;
}
```

2) Machine Dependent Optimization Techniques:

Machine dependent optimization techniques are based on register allocation and utilization of special machine-instruction sequences. It is done after the target code has been generated and when the code is transformed according to the target machine architecture.

Peephole Optimization: Peephole optimization is a simple and effective technique for locally improving target code. This technique is applied to improve performance of the target program by examining the short sequence of target instructions (called the peephole) and replace these instructions replacing by shorter or faster sequence whenever possible. Peephole is a small, moving window on the target program. Peephole optimization techniques are as follows:

a) Redundant Load and Store Elimination: In this technique the redundancy is eliminated.

Example:

Initial code:
```
y=x+5;
i=y;
z=i;
w=z*3;
```

Optimized code
```
y=x+5;
i=y;
w=y*3;
```

b) The Flow of Control Optimization:

i) Dead Code Elimination: It involves eliminating the dead code i.e, statements of code which never executes or are unreachable.

Example:

Code before optimization
```
i=0;
if(i==1)
{
  a=x+5;
}
```

Code after optimization
```
i=0;
```

ii) Avoid jump on jump: The unnecessary jumps can eliminate in either the intermediate code or the target code.

We can replace the jump sequence:
```
Goto L1
.....
L1: goto L2
```
By the sequence:
```
Goto L2
.....
L1: goto L2.
```

c) Algebraic Simplification: Peephole optimization is an effective technique for algebraic simplification. The statements such as $x = x+0$ or $x = x*1$ can be eliminated by peephole optimization.

d) Machine Idioms: The target instructions have equivalent machine instructions for performing some operations. Hence we can replace these target instructions by equivalent machine instructions in order to improve the efficiency.

Example: Some machines have auto-increment or auto-decrement addressing modes.

e) Strength reduction: It involves reducing the strength of expressions. This technique replaces the expensive and costly operators with the simple and cheaper ones.

Example: Code before optimization:

$B = A \times 2$

Code after optimization:

$B = A + A$

Compiler Design and Construction (Marking Scheme) with Imp Topics:

Unit1: (3 hrs) : 5 marks (Analysis phase, Synthesis phase, compiler vs interpreter, one pass vs. Multipass compiler)

Unit2: (22 hrs):

- **2.1 Lexical Analysis:** 10 marks (lexemes, patterns, tokens, thomsons construction, subset construction)
- **2.2 Syntax Analysis:** 20 marks (role of syntax analyzer, Left recursion, left factoring, LL(1) parsing table, handle, LR(0) item, cannonical LR(0) collection, SLR parsing tables, cannonical LR(1) collection, LR(1), LALR(1), Top-down vs bottom-up parsing)
- **2.3 Semantic Analysis:** 10 marks (Type checking of expressions, Static vs. dynamic type checking, STD, STDS)

Unit3: (4 hrs):

- Symbol Table Design: 5 marks
- Run-time storage management: 5 marks

(very short unit, everything imp, read all)

Unit4: (16 hrs):

- **4.1 Immediate Code Generator:** 10 marks (Role of immediate code generator, Syntax tree, DAG, Postfix notation, finding 3 address code of expressions and statements)
- **4.2 Code Generator:** 5 marks (code generator design issues)
- **4.3 Code Optimization:** 5 marks (defn of code optimization with its need, basic optimization techniques)

If my notes really helped you,then you can support me on esewa for my hardwork.

Esewa ID: 9806470952