Code Generation

M.B.Chandak

Lecture notes on Language Processing

Code Generation

- It is final phase of compilation.
- Input from ICG and output in the form of machine code of target machine.
- Major issues
  - **Input to code generator**
  - **Target program**
  - **Memory management**
  - **Instruction selection**
  - **Allocation of registers**
  - **Choice of Evaluation order**

Major Issues: Input to Code Generator

- Output of Intermediate code generator and code optimizer.
- Entries of symbol tables used to generate run time addresses of data objects generated by ICG.
- Type of representation of ICG, generally TAC, Trees and DAG are suitable for most of code generators.
- Assumption: Code is free of errors.

Major Issues: Target Programs

- Target program may be in the form of Absolute machine code, re-locatable machine code or assembly language.
- Absolute machine code can be placed at any fixed location in memory and executed.
- Re-locatable machine code (object module) allows subprograms to be complied separately and linked and loaded in memory before execution. Advantage: Reusability
- Assembly language version makes process of code generation simple.
- **//Refer class material for remaining points**

Code Generation: Arrays

- **The assembly code is used for code generation to simplify the process of code generation. The instruction like: MOV, ADD, SUB are used for implementation purpose.**
- **The code generation process of array statements like: a = b[i] and a[i] = b is as follows:**
- **The process depends upon index variable "i" which can be stored at different places in memory.**
- **The possible memory representation of index variable are:**
  - Register Ri
  - Memory location Mi
  - On Stack at offset Si from top and register "A" holds the address of activation record containing details of stack.

Code Generation: Arrays

- The code generation for arrays will be:

Statement	i in Register Ri	i in memory Mi	i in stack
a = b[i]	MOV b[Ri], R1	MOV Mi, R MOV b(R), R1	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)

- Example: Generate code for : Assuming "i" is present in memory.

  sum = sum + a[i] + b[j]

- The TAC for above code will be:
  - **T1 = sum + a[i]**
  - **T2 = T1 + b[j]**
  - **Sum = T2**

Code Generation: Arrays

- The code generation for the TAC will be

Statement	Code Sequence	Register Descriptor	Address Descriptor

T1 = sum + a[i]	MOV X, R0
	MOV A[R0], R1
	ADD SUM, R1
		RO contains index “i“
		R1 contains A[i]
		R1 contains t1
			“i” in R0
			A[i] in R1
			T1 in R1

T2 = T1 + b[j]	MOV Y, R0
	MOV B[R0], R2
	ADD R1, R2
		RO contains index “j“
		R2 contains A[j]
		R2 contains t2
			“j” in R0
			B[j] in R2
			T2 in R2

SUM = T2	MOV R2, SUM		Memory location “sum” contains result.

Code Generation: Conditional Statements

- The conditions statements or loops in the program segments are implemented using “if” construct in TAC.
- For code generation phase, it is necessary to have implementation of “if” construct in assembly format.
- The two constructs: CMP and CJ are used for implementation.
- For example: if (x > y) goto z can be implemented as

```
CMP X, Y  //Compare X and Y
CJ>   Z    // Conditional jump to statement Z if “>” condition is satisfied.
```

Labelling Algorithm

- **Purpose: To determine number of registers required for executing the generated code.**
- **The algorithm makes use of DAG generated from TAC**
- **Algorithm: Operation starts with left most leaf node**

```
Algorithm label()
{
  if [node==leaf] and [node==left child]
     label(n) = 1
  else
     label(n) = 0
  else
    label (n) =      max label (n1, n2) → If label(n1) != label(n2)
                     L1+1 → Otherwise
}
```

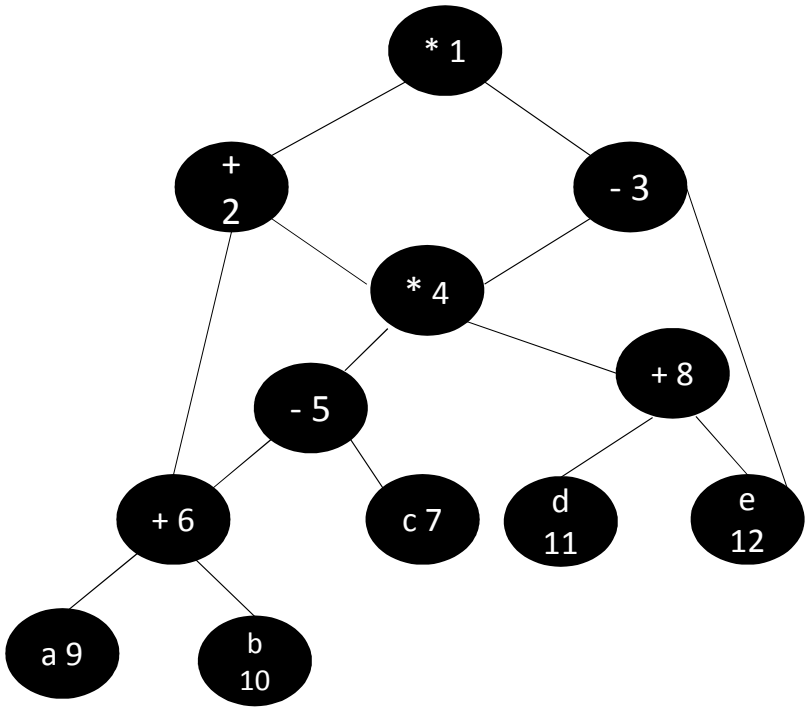
Code Generation: Using DAG

- This algorithm is also called as Heuristic Ordering Algorithm for code generation.
- In nodes of DAG are ordered and listed to generate the code.
- The root node represents operator and leaf nodes represents operands in the DAG.

```
Algorithm H_DAG()
{
   While (unlisted interior node remains) do
   begin
      select (an unlisted node "n" all of whose parents are listed)
       list node "n" //root node will have no parents, hence will be always listed//
            While (left child "m" of node "n" has no unlisted parent) and (is
            not a leaf node) do
              begin
                list m
                n = m
              end
    End
}
```

Code Generation: Using DAG

- Example – always move left first



- **Execution: Nodes number is already provided, if not given then assume suitable sequence**
- **Step 1: Select the parent node: as it is root, list the node 1.**
- **Left child of node 1 is node 2 and parent of node 2 is node 1. Since node 1 is already listed, list node 2.**
- **Left child of node 2 is node 6. The parent of node 6 are node 2 and node 5, since node 5 is not listed, it is necessary to find new opening. This opening should be a node with listed parent.**
- **This node is node 3, hence list node 3 and continue with node 4.**
- **Node 4, both the parents node 2 and 3 are listed, hence list node 4 and move left.**
- **Node 5, parent node 4 is listed, hence list node 5.**
- **Node 6 parent node 2 and node 5 are listed, hence list node 6.**
- **Node 6, left child is node 9, hence list node 9 – but it is leaf node so ignore and check another node.**
- **In the similar fashion, control reaches to node 4 and its right child node 8, so list node 8.**
- **All the nodes are now listed.**
- **Select the last listed node for execution.**
- **1,2,3,4,5,6,8 ➔ 8, 6, 5, 4, 3, 2, 1**

Peephole Optimization

- The theory contents of the topic are available in the study material with like "Code Optimization" at www.mbchandak.com (* available)
- It is an approach to improve the efficiency of target code. It is performed on output of code optimization phase, in order to generate efficient target code.
- It is considered as process carried out by code generator before finagling the target code.
- The "peephole" is small moving window, applied on code before generating target code. Peephole optimization is performed using various techniques like:
  - Elimination of redundant instructions
  - Flow of control optimization (Elimination of un-reachable code)*
  - Algebraic simplifications*
  - Use of machine Idioms.

Peephole Optimization

- Elimination of Redundant Instructions:

Consider the following sequence:

MOV R0, a

MOV a, R0

Instruction 2, can be deleted for further optimization, if both the instructions are present in same block in PFG.

- Elimination of Unreachable code: (Refer CO notes)
- Flow of control optimization
- This process involves optimization using controlling the flow across the code. The intermediate code generator frequently produces jumps to jumps. These jumps can be eliminated in peephole optimization process

Peephole Optimization

- Flow of control optimization
- This process involves optimization using controlling the flow across the code. The intermediate code generator frequently produces jumps to jumps. These jumps can be eliminated in peephole optimization process
- Example 1:

  Goto L1

  L1: Goto L2
- Above code requires statement L2 to be executed. It can be replaced by

  Goto L2
- Example 2:

  if (a<b) goto L1

  L1: goto L2
- Above code can be reduced to **if(a<b) goto L2**

Peephole Optimization

- **Flow of control optimization**
- Example 3

  Goto L1

  L1 if (a<b) goto L2

  goto L3
- Above code can be reduced to

  if(a<b) goto L2

  Goto L3

Algebraic Simplification → Code optimization notes

Reduction of Strength → Code optimization notes

- **Use of Machine Idioms:** This process allows to use standard machine architecture constructs instead of regular instructions. For example: i=i+1 can be replaced by i++ and i=i-1 can be replaced by i--.

Sample Code Generator: Register and Address Descriptors

- Refer Class notes

Sample Code Generator: getreg() algorithm

- Refer Class notes

Sample Code Generator: Final code generation algorithm

- Refer Class notes