Code Generation

Dr. Avinash J. Agrawal
Department of CSE
Shri Ramdeobababa College of
Engineering and Management, Nagpur

Issues in Good Code Generation

Selection of instruction : It is the responsibility of code generator to choose the most efficient instructions to represent the computation specified by the three addressed statement. For ex: *INCR a*

Register allocation : A program has a number of values to be maintained during the execution. The target machine's architecture may not allow all of the values to be kept in the CPU memory or registers. Code generator decides what values to keep in the registers. Also, it decides the registers to be used to keep these values.

Ordering of instructions : The code generator decides the order in which the instruction will be executed. Execution order affects efficiency of the generated code. For ex: T1 = a+b

T2 = c+d

T3 = e-T2

T4 = T1-T3

Sample Machine Model

- Memory : Byte Addressable, 2B per word, size 2^{16} Bytes
- General purpose registers : 8 no., 16 bit value
- Instruction format : *Op source, destination* (with length 4,6,6 bits)

  ex: MOV R0, R1

  MOV R0, M

  MOV M1, M2

Data Structure needed for CG

Register descriptor : Register descriptor keeps track of values stored in each register. Whenever a new register is required during code generation, this descriptor is consulted for register availability.

Address descriptor : Values of the names *identifiers* used in the program might be stored at different locations while in execution. Address descriptors are used to keep track of memory locations where the values of identifiers are stored. These locations may include CPU registers, memory, stack or a combination of the mentioned locations.

Code generator keeps both the descriptor updated in real-time. For a load statement, LD R1, x

The code generator: updates the Register Descriptor R1 that has value of x and updates the Address Descriptor *x* to show that one instance of x is in R1.

Simple Code Generation

getReg(): Code generator uses *getReg* function to determine the status of available registers and the location of name values. It returns a location where computation is to be done.

getReg (x=y+z) works as follows:

If variable ‘y’ is already in register R, it uses that register.
Else If any register R is available, use that register.
Else If ‘x’ has future use, it chooses a register that requires min. cost to store value in memory.
Else it select a memory location and return it.

Simple Code Generation

For an instruction x = y OP z in the basic block

{

1. Call function getReg, to decide the location of L.
2. Determine the present location (*register or memory)* of **y** by consulting the Address Descriptor of **y**. If **y** is not presently in register **L**, then generate the following instruction to copy the value of **y** to **L**: **MOV y, L**
3. Determine the present location of **z** using the same method used for **y** and generate the instruction: **OP z, L**
   Now L contains the value of y OP z, that is intended to be assigned to **x**. So, if L is a register, update its descriptor to indicate that it contains the value of **x**. Update the descriptor of **x** to indicate that it is stored at location **L**.
4. If y and z has no further use and stored in register then free those registers

}

Store All the Results

}

Examples

T1 = a + b

T2 = c + d

T3 = T2 - e

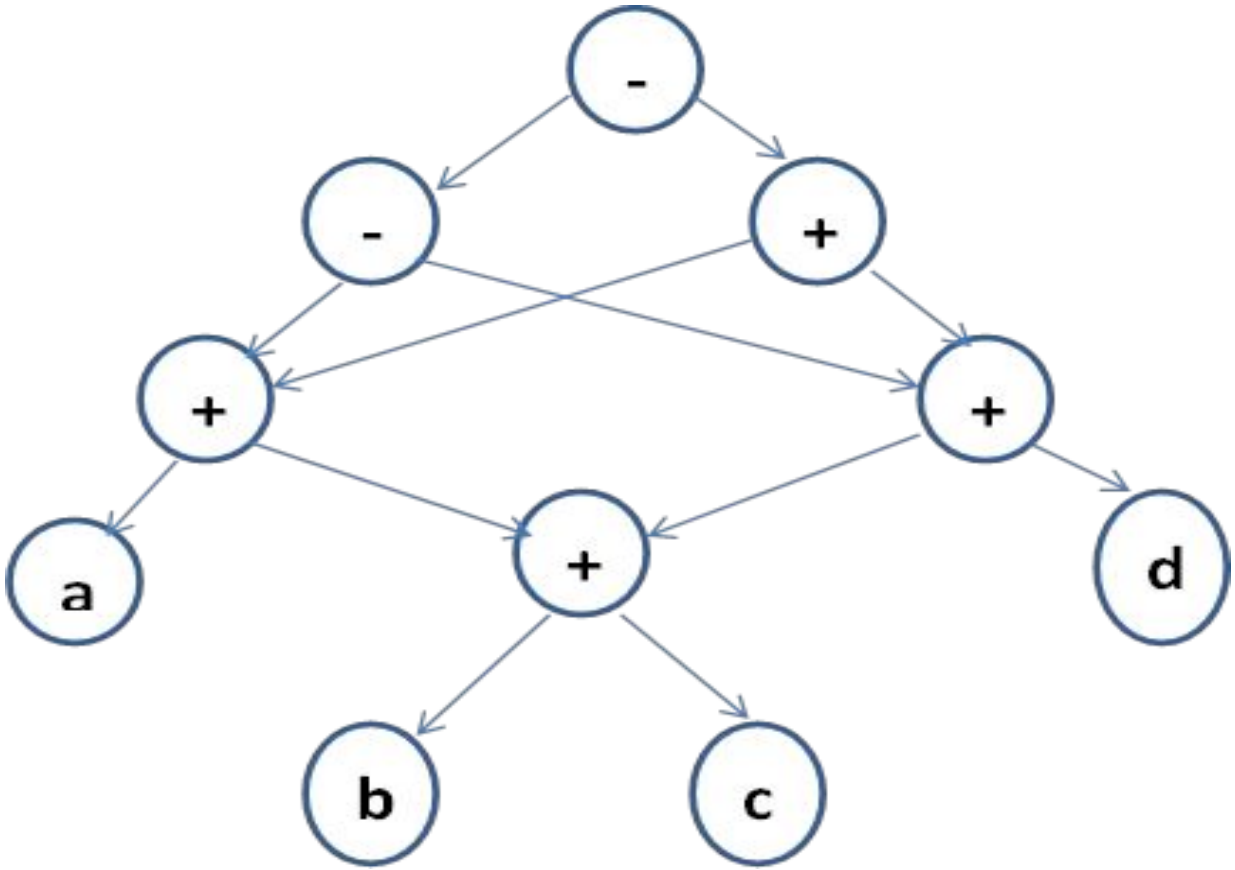
x = T1 - T3

Statement	**L**	**Instructions**	**Reg. Des.**	**Addr. Des.**
			All Reg. Empty	

Heuristic Algorithm

```
{
While there exist an unlisted interior node do
  {
        Select an unlisted node n whose parents have been listed
              List n
              While there exists a leftmost child m of n that has no unlisted
               parents and m is not leaf do
               {
                  list m
                  n = m
               }
    }
Order = reverse of the order of listing of nodes
}
```

Example



Labelling Algorithm Part 1

- Label each node of the tree with an integer: Minimum no. of registers required to evaluate the tree with no intermediate stores to memory
- Consider binary trees

```
{
  if n is a leaf node then
        if n is the leftmost child of its parent then
              label(n) := 1
        else
              label(n) := 0
  else
        label(n) = max (l1, l2), if l1<> l2
        label(n) = l1 + 1, if l1 = l2
}
```

Labelling Algorithm Part - 2

- Procedure GENCODE(n)
- RSTACK –stack of registers, R0,...,R(r-1)
- TSTACK –stack of temporaries, T0,T1,...

Gencode(n) is a recursive function.

It generates code to evaluate a tree T, rooted at node *n*.

It calculate value of *n* in the register top(RSTACK)

The rest of RSTACK remains same as the one before the call

Labelling Algorithm Part - 2

The Code Generation Algorithm

Procedure gencode(n);

{ /* case 0 */

if

n is a leaf representing operand n and is the leftmost child of its parent

then

generate(MOV name(n), top(RSTACK))

}

Labelling Algorithm Part - 2

The Code Generation Algorithm

/* case 1 */

else if

n is an interior node with operator OP, left child n1, and right child n2

then

if label(n2) == 0 ***then***

{

gencode(n1);

generate(OP name(n2), top(RSTACK));

}

Labelling Algorithm Part - 2

```
/* case 2 */
else if (1<=(label(n1) < label(n2)) and( label(n1) < r))
then {
        swap(RSTACK);
        gencode(n2);
        R := pop(RSTACK);
        gencode(n1);
       /* R holds the result of n2 */
       generate(OP R, top(RSTACK));
       push (RSTACK,R);
       swap(RSTACK);
}
```

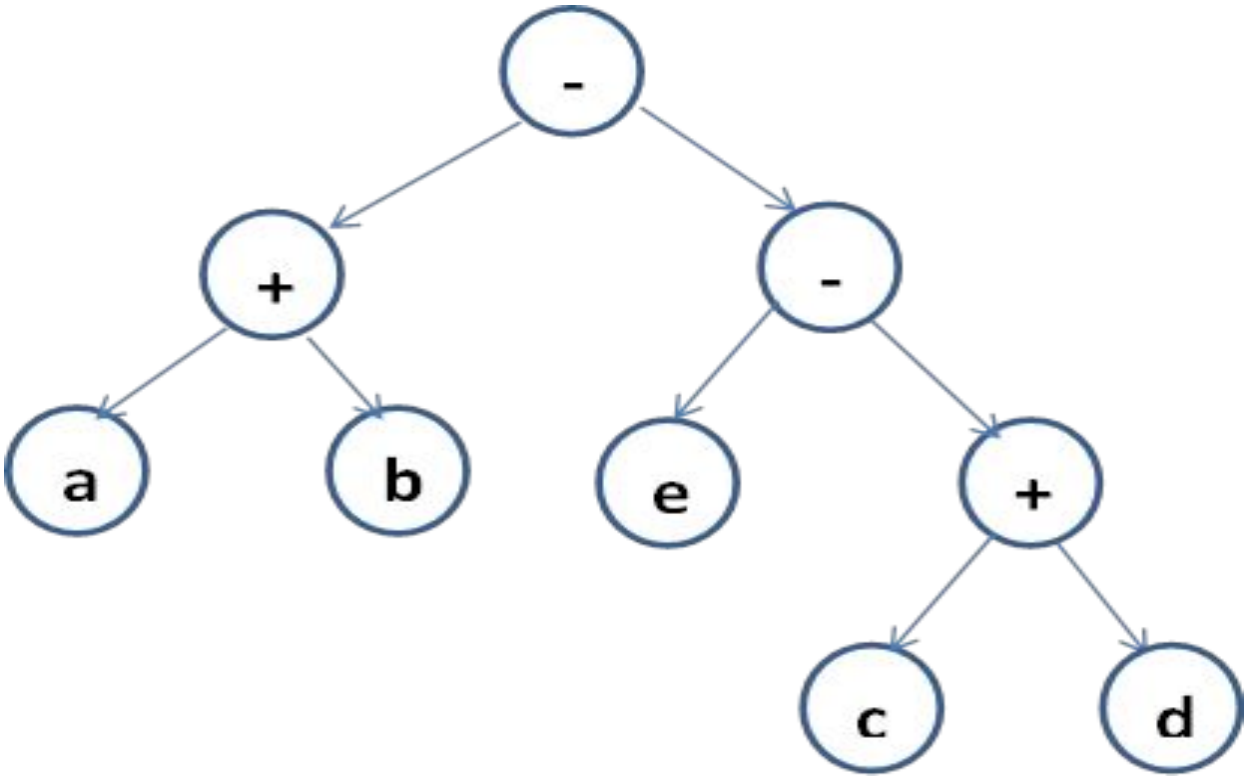
Labelling Algorithm Part - 2

```
/* case 3 */
else if (1<=(label(n2) <= label(n1)) and ( label(n2) < r))
then {
        gencode(n1);
        R := pop(RSTACK);
        gencode(n2);
        /* R holds the result of n1 */
       generate(OP top(RSTACK), R);
       push (RSTACK,R);
}
```

Labelling Algorithm Part - 2

```
/* case 4, both labels n1,n2 are > r */
else {
        gencode(n2);
        T:= pop(TSTACK);
        generate(MOV top(RSTACK), T);
        gencode(n1);
        generate(OP T, top(RSTACK));
        push(TSTACK, T);
}
}
```

Example



Call to gencode() Action Rstack Code Generated

Thank You….