CSc 553 — Principles of Compilation

20 : Code Generation III

Christian Collberg
Department of Computer Science
University of Arizona
collberg@gmail.com

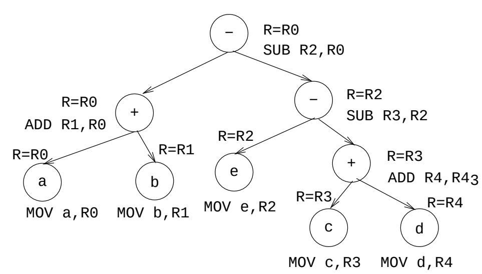
February 24, 2011

1

Trivial Code Generation

2 Generating Code From Trees

- To generate code from expression trees, traverse the tree and emit machine code instructions.
- For leaves (which represent operands), generate load instructions. For interior nodes, generate arithmetic instructions.
- Assume an infinite number of registers ⇒ easy algorithm!
- Each tree node N has an attribute `'R'`, the register into which the subtree rooted at N will be computed.

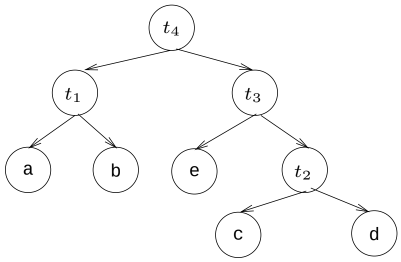


3

Generating Code From Labeled Trees

4 Optimal Ordering For Trees I

- We can generate 'optimal' code from a tree. 'Optimal' in the sense of 'smallest number of instructions generated'.
- The idea is to reorder the computations to minimize the need for register spilling.



First Order	Second Order
t_1 := a + b	t_2 := c + d
t_2 := c + d	t_3 := e - t_2
t_3 := e - t_2	t_1 := a + b
t_4 := t_1 - t_3	t_4 := t_1 - t_3

5 Optimal Ordering For Trees II

- Assume two registers available. The first ordering evaluates the left subtree first, and has to spill `R0` to have enough registers available for the right subtree.

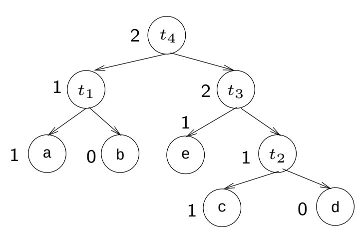
First Order	Second Order	First Order	Second Order
t_1 :=a+b	t_2 :=c+d	MOV a, R0	MOV c, R0
t_2 :=c+d	t_3 :=e-t_2	ADD b, R0	ADD d, R0
t_3 :=e-t_2	t_1 :=a+b	MOV c, R1	MOV e, R1
t_4 :=t_1-t_3	t_4 :=t_1-t_3	ADD d, R1	SUB R0, R1
		MOV R0, t_1	MOV a, R0
		MOV e, R0	ADD b, R0
		SUB R1, R0	SUB R0, R1
		MOV t_1, R1	MOV R0, t_4
		SUB R0, R1	
		MOV R1, t_4	

6 The Tree Labeling Phase I

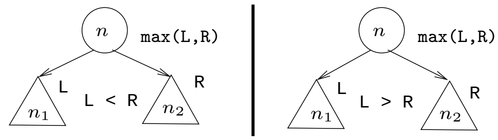
- The algorithm has two parts. First we label each sub-tree with the minimum number of registers needed to evaluate the subtree without any register spilling.

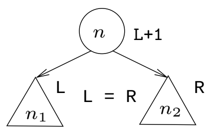
The Labeling Algorithm:

- n is a left leaf $\Rightarrow$ label(n) := 1;
- n is a right leaf $\Rightarrow$ label(n) := 0;
- n's children have labels l_L & l_R:
  - $l_L \neq l_R \Rightarrow$ label(n) := $\max(l_L, l_R)$
  - $l_L = l_R \Rightarrow$ label(n) := $l_L + 1$



7 The Tree Labeling Phase II



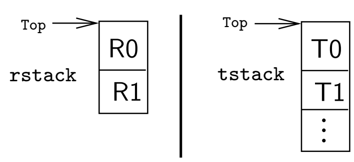
- If we have a node n with subtrees n_1 and n_2 with `L=label(`n_1`) & R=label(`n_2`) & L<R` then we can first evaluate n_2 into a register `Reg` using `R` registers. Then we use `R-1` registers to evaluate n_1.
- Similarly, if `L>R` then we can first evaluate n_1 into a register `Reg` and use the remaining `R-1` registers for n_2.
- However, if `L=R`  we'll need one extra register to hold the result of n_1 while we evaluate n_2.

8 The Generation Phase I

- `gencode(`n`)` generates machine code for a subtree n of a labeled tree T.

`MOV M, R`	Load variable `M` into register `R`.
`MOV R, M`	Store register `R` into variable `M`.
OP `M, R`	Compute `R := R` *OP* `M`. *OP* $\in$ `ADD, SUB, MUL, DIV`.
OP `R2, R1`	Compute `R1 := R1` *OP* `R2`.

- A stack `rstack` initially contains all available registers. `gencode(`n`)` generates code for subtree n using the registers on `rstack`, computing its value into the register on the top of the stack.
- A stack `tstack` of temporary memory locations is used for register spilling.



9 The Generation Phase II

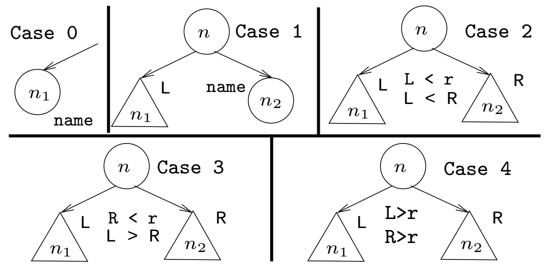
Case 0 A leaf n is the leftmost child of its parent.

Case 1 A leaf n_2 is the rightmost child of its parent.

Case 2 A right subtree n_2 requires more registers than the left subtree n_1.

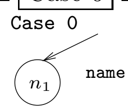
Case 3 A left subtree n_1 requires more registers than the right subtree n_2.

Case 4 Both subtrees require registers to be spilt.



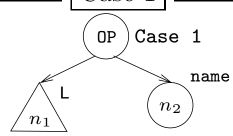
10 The Generation Phase III

Case 0



1. Generate a load instruction to load the variable into a register: `MOV name, top(rstack)`.

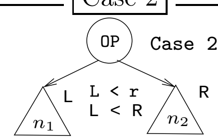
Case 1



1. Generate code for n_1 into register `top(rstack)`, i.e. call `gencode(`n_1`)`.
2. Generate `OP name, top(rstack)`.

11 The Generation Phase IV

Case 2

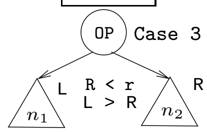


- n_1 can be evaluated without spilling, but n_2 requires more registers than n_1.
- We swap the two top registers on `rstack`, evaluate n_2 into `top(rstack)`, remove the top register, then evaluate n_1 into `top(rstack)`. Restore the stack.

1. `swap(rstack), gencode(`n_2`)`
2. `R := pop(rstack)`
3. `gencode(`n_1`)`
4. Generate `OP R, top(rstack)`
5. `push(rstack, R), swap(rstack)`

12 The Generation Phase V

Case 3

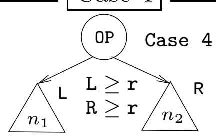


- n_2 can be evaluated without spilling, but n_1 requires more registers than n_1.
- We evaluate n_1 into `top(rstack)`, remove the top register, then evaluate n_2 into `top(rstack)`.

1. `gencode(`n_1`)`
2. `R := pop(rstack)`
3. `gencode(`n_1`)`
4. Generate `OP top(rstack), R`
5. `push(rstack, R)`

13 The Generation Phase VI

Case 4



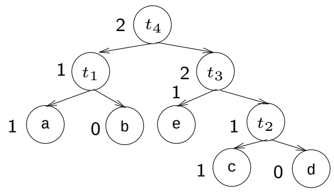
- Neither n_1 nor n_2 can be evaluated without spilling,
- We evaluate n_2 into a temporary memory location `top(tstack)`, and then we evaluate n_1 into `top(rstack)`.

1. `gencode(`n_2`)`
2. `T := pop(tstack)`
3. Generate `MOV top(rstack), T`
4. `gencode(`n_1`)`
5. `push(tstack, T)`
6. Generate `OP T, top(rstack)`

14

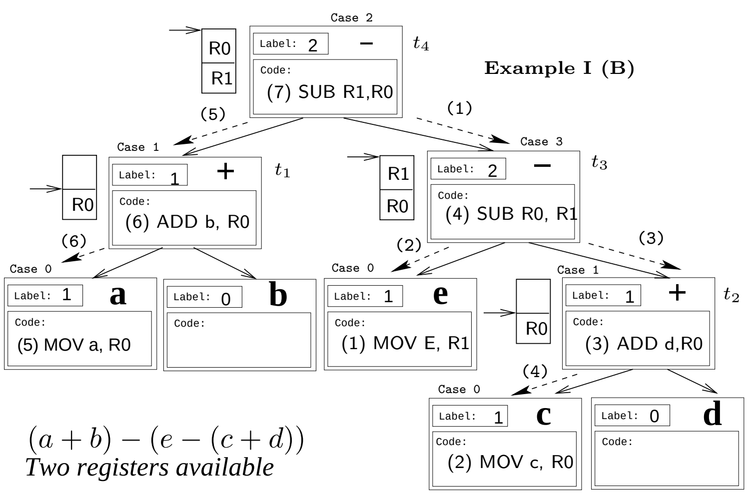
Examples

15 Example I (A)




```
gencode(t4)           [R1,R0]   case2
   gencode(t3)        [R0,R1]   case3
      gencode(e)      [R0,R1]   case0
         MOV e, R1
      gencode(t2)     [R0]      case1
         gencode(c)   [R0]      case0
            MOV c, R0
      SUB R0, R1
   gencode(t1)        [R0]      case1
      gencode(a)      [R0]      case0
         MOV a, R0
      ADD b, R0
   SUB R1, R0
```

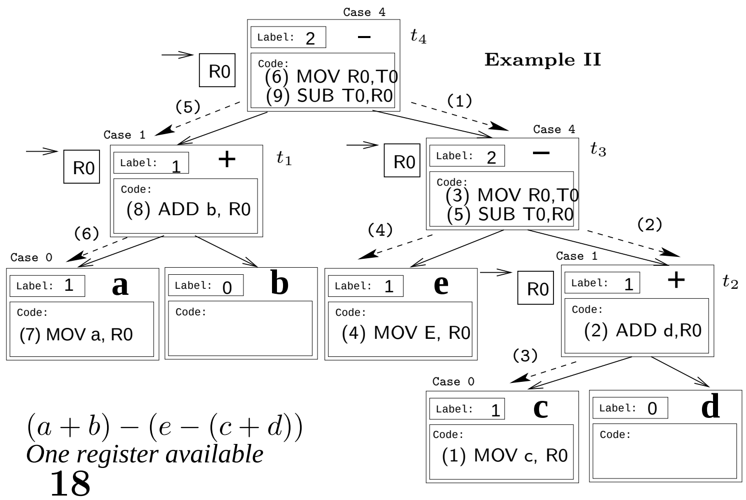
16



$(a + b) - (e - (c + d))$

Two registers available

17



18

Summary

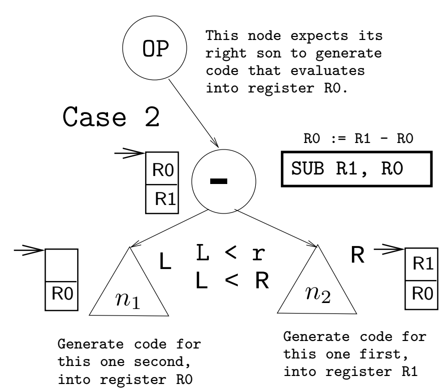
19 Readings and References

- This lecture is taken from the Dragon Book:

  Code Generation From Trees: 557–559, 561–566.

  Local Optimization: 530–532, 600–602.

20 Summary I



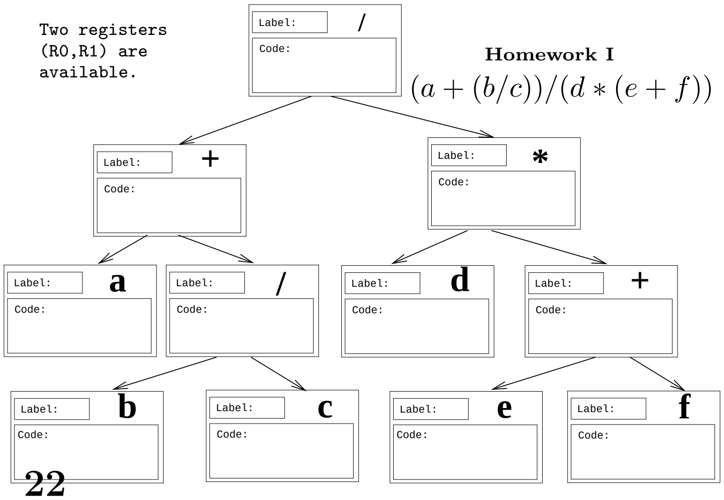
- Why do we swap registers in `Case 2`?

21

Two registers (R0,R1) are available.

Homework I

$(a + (b/c))/(d * (e + f))$

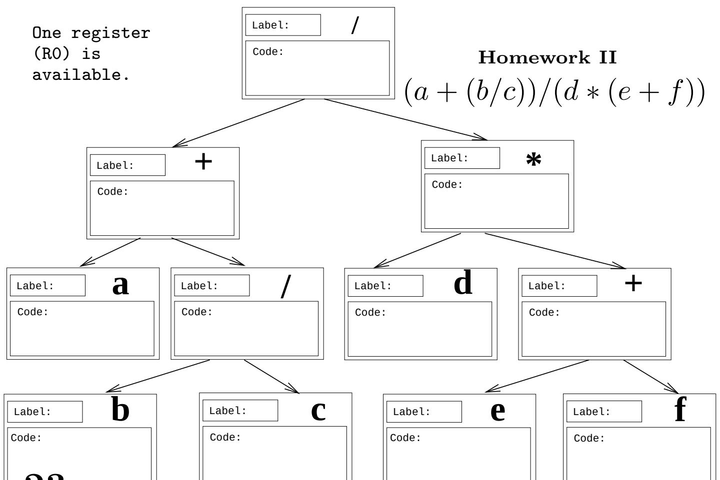


22

One register (R0) is available.

Homework II

$(a + (b/c))/(d * (e + f))$



23

The machine has two registers R0 and R1, and an infinite number of temporary memory locations T0,T1,...

Exam Question 07.330/96