# CSc 553

## Principles of Compilation

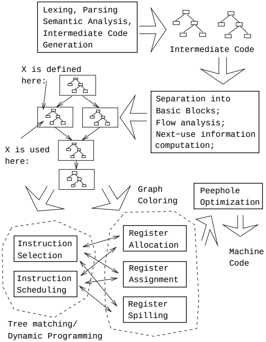
## 21 : Code Generation — Dynamic Programming

Department of Computer Science
University of Arizona

collberg@gmail.com



# Introduction

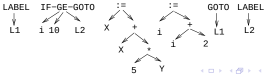



## Instruction Selection

- Starting with intermediate code in tree form, we generate the cheapest instruction sequence for each tree, using no more than $r$ registers $(R_0 \cdots R_{r-1})$.
- We will show an algorithm that integrates instruction selection and register allocation and generates optimal code for a large class of architectures.

**Intermediate Code Example**

```
WHILE i < 10 DO
   X := X + 5*Y;
   i := i + 2;
END
```



# Machine Model

## Machine Model

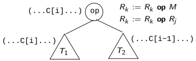
We will assume the existence of these types of instructions:

- $\boxed{R_i := E}$ $E$ is any expression containing operators, registers, and memory locations. $R_i$ must be one of the registers of $E$ (if any). I.e., we assume 2-address instructions:
  - 2-address $R_1 := R_1 + R_2$.
  - 3-address $R_1 := R_2 + R_3$.
- $\boxed{R_i := M}$ A load instruction.
- $\boxed{M := R_i}$ A store instruction.
- $\boxed{R_i := R_j}$ A register copy instruction.
- $\boxed{R_i := R_i + \texttt{ind } R_j}$ A register indirect instruction.

- All instructions have equal cost.

# Naive Algorithm

## Optimal Code Generation




- To generate optimal code for an expression $E \equiv E_1$ **op** $E_2$ we generate optimal code for $E_1$, optimal code for $E_2$, and then code for the operator.
- We have to consider every instruction that can evaluate op.
- If $E_1$ and $E_2$ can be computed in an arbitrary order, we have to consider both of them.
- We may not have enough registers available, so some temporary results may have to be stored in memory.

## Basic (Naïve) Algorithm I

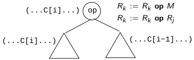
1. Compute the optimal cost for each node in the tree, assuming there are $1, 2, \cdots, r$ registers available. Also compute the optimal cost of computing the result into memory.
   - The cost of a node $n$ includes the cost of the code for $n$'s sub-trees and the cost of the operator at $n$.
2. Store the result for each node $n$ in a **cost vector** $C_n[i]$:
   - **C[1]** $=$ Cost of computing $n$ into a register, with 1 (one) register available.
   - **C[2]** $=$ As above, but with 2 available registers.
   - **C[3]** $= \cdots$
   - **C[0]** $=$ Cost of computing $n$ into memory.

## Basic (Naïve) Algorithm II

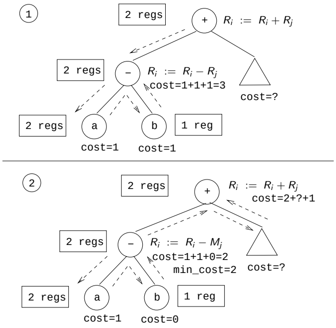
3. Traverse the tree and (using the cost vectors) decide which subtrees have to be computed into memory.
4. Traverse the tree and (again using the cost vectors) generate the final code:
   1. First code for subtrees that have to be computed into memory.
   2. Then code for other subtrees.
   3. Then code for the root.

- As we shall see, naïvely computing the costs recursively will result in us recomputing the same cost several times.

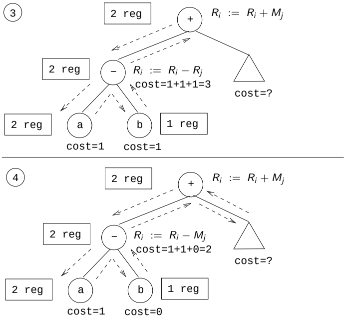
## (Naïvely) Computing the Costs




```
FOR EACH instruction I that matches op DO
  • If the instruction requires the left
    operand to be in a register, then
    (recursively) compute the optimal
    cost C_L[i] of evaluating the left
    subtree with i registers available.
  • If the instruction requires the
    right operand to be in a register,
    compute the cost C_R[i-1] of eval.
    the right subtree with i-1 regs.
  • Compute the cost of evaluating the
    subtree at n: C_L[i] + C_R[i-1] + 1.
ENDFOR
```




$R_i := M_j$
$R_i := R_i$ **op** $R_j$
$R_i := R_i$ **op** $M_j$
$R_i := R_j$
$M_i := R_j$




$R_i := M_j$
$R_i := R_l$ **op** $R_j$
$R_i := R_l$ **op** $M_j$
$R_i := R_j$
$M_i := R_j$

# Dynamic Programming

## Dynamic Programming I

- Some recursive algorithms are very inefficient, because they solve the same subproblem several times. That, for example, is the case with the Fibonacci function in the next slide.
- A rather obvious solution is to store the results in a table as they are computed, and then check the table before solving a subproblem to make sure that it's value hasn't already been computed. This is known as **memoization**.
- Even more efficient is to try to find a linear (topological) order in which the subproblems can be solved, and then solve them in that order, knowing that when we need the result of a specific subproblem, it has already been computed. This is **dynamic programming**.

## Dynamic Programming II

Recursive Fibonacci

```
function Fib (n)
   if n ≤ 1 then return 1
   else return Fib(n − 1) + Fib(n − 2)
```

Memoization Fibonacci

```
for i := 1 to n do A[i] := −1;
function Fib (n)
   if A[n] = −1 then
      if n ≤ 1 then A[n] := 1
      else A[n] := Fib(n − 1) + Fib(n − 2)
   return A[n]
```

## Dynamic Programming III

**Dynamic Programming Fibonacci**

```
function Fib (n)
    A[0] := A[1] := 1;
    for i := 2 to n do
        A[i] := A[i − 1] + A[i − 2]
```

# The Dynamic Programming Algorithm

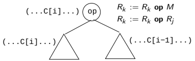
## Computing Costs

- There is a linear-time, dynamic programming, bottom-up algorithm for computing the costs.

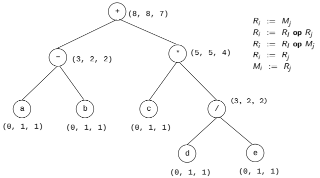
**Compute $C[i]$ at node $n$**

- Consider each instruction $R_k := E$ where $E$ matches the subtree, and choose the **minimum** $C[i]$, where $C[i]$=The sum of
  1. $C[i]$ of $n$'s left subtree
  2. $C[i-1]$ of $n$'s right subtree
  3. the cost of the instruction at $n$

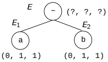
$$\begin{array}{l|l|l} R_i := R_i \textbf{ op } R_j & R_i := M_j & M_i := R_j \\ R_i := R_i \textbf{ op } M_j & R_i := R_j & \end{array}$$




## Computing Costs – Example I (a)



## Computing Costs – Example I (b)



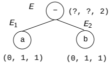

1. $R_0 := R_0 - R_1$
2. $R_0 := R_0 - M$

| | | |
|---|---|---|
| $R_i := R_i$ **op** $R_j$ | $R_i := M_j$ | $M_i := R_j$ |
| $R_i := R_i$ **op** $M_j$ | $R_i := R_j$ | |

1. $E_1$ into $R_0$ (2 regs avail); $E_2$ into $R_1$ (1 reg avail); Use $R_0 := R_0 - R_1$ at $E$; Cost$=E_1[2] + E_2[1] + 1 = 1 + 1 + 1 = 3$
2. $E_2$ into Memory (2 regs avail); $E_1$ into $R_0$ (2 regs avail); Use $R_0 := R_0 - M$ at $E$; Cost$=E_2[0] + E_1[2] + 1 = 0 + 1 + 1 = 2$

- $C[2]$ = `min(3,2)` = 2.

## Computing Costs – Example I (c)



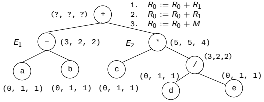

1. $R_0 := R_0 - M$

| | | |
|---|---|---|
| $R_i := R_i$ **op** $R_j$ | $R_i := M_j$ | $M_i := R_j$ |
| $R_i := R_i$ **op** $M_j$ | $R_i := R_j$ | |

1. $E_2$ into Memory (1 reg available); $E_1$ into $R_0$ (1 reg available); Use $R_0 := R_0 - M$ at $E$; Cost$=E_2[0] + E_1[1] + 1 = 0 + 1 + 1 = 2$

- Only one instruction to choose from.
- $C[1]$ = 2.
- The min cost of computing $E$ into memory is the min cost of computing $E$ into a register (= `min(2,2)`) plus 1 (=3).

## Computing Costs – Example I (d)

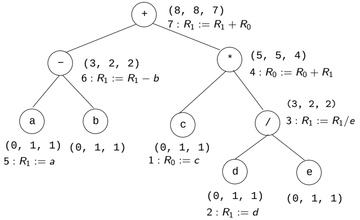



1. $R_0 := R_0 + R_1$
2. $R_0 := R_0 + R_1$
3. $R_0 := R_0 + M$

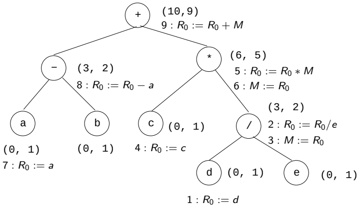
1. $E_1$ into $R_0$ (2 regs avail); $E_2$ into $R_1$ (1 reg avail); Use $R_0 := R_0 + R_1$ at $E$; Cost$=E_1[2] + E_2[1] + 1 = 2 + 5 + 1 = 8$
2. $E_2$ into $R_1$ (2 regs); $E_1$ into $R_0$ (1 reg); Use $R_0 := R_0 + R_1$ at $E$; Cost$=E_2[2] + E_1[1] + 1 = 4 + 2 + 1 = 7$
3. $E_2$ into Memory (2 regs); $E_1$ into $R_0$ (2 regs); Use $R_0 := R_0 + M$ at $E$; Cost$=E_2[0] + E_1[2] + 1 = 5 + 2 + 1 = 8$

- $C[2]$ = `min(8,7,8)` = 7.

## Generating Code – Example I (e)



## Dynamic Programming – Example II




# Summary

## Readings and References

- This lecture is taken from the Dragon book: 567–580.
- Read "Emmelmann, Schröer, Landwehr: *BEG – A generator for Efficient Back Ends*", PLDI '89.
- Additional material: "Aho,Ganapathi,Tjiang: *Code Generation Using Tree Matching and Dynamic Programming*, TOPLAS, Vol 11, No. 4, Oct. 1989, pp 491-516.
- For information on Dynamic Programming: see "*Algorithms*", by **Cormen, Leiserson, Rivest**, p. 310.

# Summary

- Use the dynamic programming algorithm to generate optimal code for the assignment

$$g := a * (b + c) + d * (e - f).$$

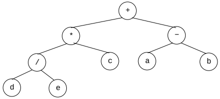
- Assume that two registers (`R0, R1`) are available.

Machine Model

$R_i$ := $M_j$
$R_i$ := $R_i$ **op** $R_j$
$R_i$ := $R_i$ **op** $M_j$
$R_i$ := $R_j$
$M_i$ := $R_j$



- Use the dynamic programming algorithm to generate code for the expression tree below using (a) 1 and (b) 2 registers. For each node show the cost vector and the instruction(s) generated.




$R_i$ := $M_j$
$R_i$ := $R_i$ **op** $R_j$
$R_i$ := $R_i$ **op** $M_j$
$R_i$ := $R_j$
$M_i$ := $R_j$