Code Generation: Integrated Instruction Selection and Register Allocation Algorithms

Amitabha Sanyal

(www.cse.iitb.ac.in/˜as)

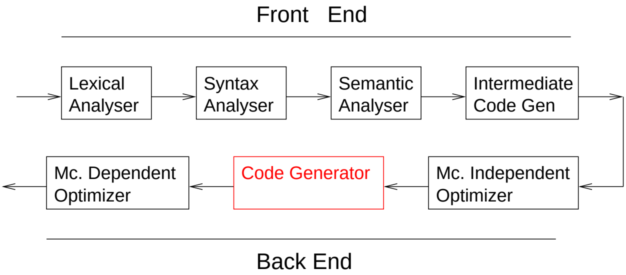
Department of Computer Science and Engineering,

Indian Institute of Technology, Bombay

February 2009

Place of Code Generator in a Compiler

Text book stuff ...



Code Generation - Issues

- Expressions and Assignments:

Code Generation - Issues

- Expressions and Assignments:
  - Instruction selection. Selection of the best instruction for the computation.

Code Generation - Issues

- Expressions and Assignments:
  - Instruction selection. Selection of the best instruction for the computation.
    - The instruction should be able to perform the computation.
    - It should be the fastest of possible choices.
    - It should combine well with the instructions of its surrounding computations?

Code Generation - Issues

- Expressions and Assignments:
  - Instruction selection. Selection of the best instruction for the computation.
  - Register allocation. To hold result of computations as long as possible in registers.

Code Generation - Issues

- Expressions and Assignments:
  - Instruction selection. Selection of the best instruction for the computation.
  - Register allocation. To hold result of computations as long as possible in registers.
    - What computations will be held in registers?
    - In which regions of the program?

Code Generation - Issues

- Expressions and Assignments:
  - Instruction selection. Selection of the best instruction for the computation.
  - Register allocation. To hold result of computations as long as possible in registers.
- Control Constructs:
  - Lazy evaluation of boolean expressions.
  - Avoiding jump statements to jump statements.

Code Generation - Issues

- Expressions and Assignments:
  - Instruction selection. Selection of the best instruction for the computation.
  - Register allocation. To hold result of computations as long as possible in registers.
- Control Constructs:
  - Lazy evaluation of boolean expressions.
  - Avoiding jump statements to jump statements.
- Procedure Calls:
  - Activation record building:
    - Division of work between caller and callee
    - Using special instruction for creation and destruction of activation records.
    - Saving and restoring of registers across procedure calls.

Outline of Lecture

- Unified algorithms for instruction selection and code generation.
  - Sethi-Ullman Algorithm
    - One of the earliest code generation algorithms.
  - Aho-Johnson Algorithm
    - Optimal code generation for realistic expression and machine models. Most code generators are variations of this.

Sethi-Ullman Algorithm – Introduction

- Generates code for expression trees (not dags).
- Target machine model is simple. Has
  - a load instruction,
  - a store instruction, and
  - binary operations involving either a register and a memory, or two registers.
- Does not use algebraic properties of operators.
  - If $e_1 * e_2$ has to be evaluated using $r \leftarrow r * m$, and
  - e_1 and e_2 are in m and r,

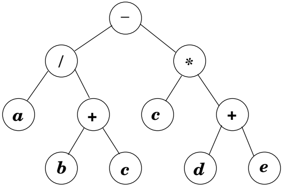
    then the code sequence has to be:

$$\begin{array}{lll} m_1 & \leftarrow & r \\ r & \leftarrow & m \\ r & \leftarrow & r * m_1 \end{array} \quad \text{and not simply:} \quad r \leftarrow r * m$$

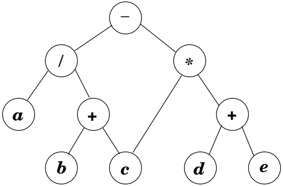
- Generates optimal code – i.e. code with an instruction sequence with least cost.
- Running time of the algorithm is linear in the size of the expression tree.

Expression Trees

- Here is the expression $a/(b+c) - c * (d+e)$ represented as a tree:



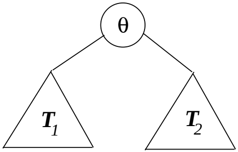
- We have not identified common sub-expressions; else we would have a directed acyclic graph (DAG):



Expression Trees

Let Σ be a countable set of variable names, and Θ be a finite set of binary operators. Then,

1. A single vertex labeled by a name from Σ is an expression tree.
2. If T_1 and T_2 are expression trees and θ is a operator in Θ, then



is an expression tree.

In the previous example
$\Sigma = \{a, b, c, d, e, \dots\}$, and $\Theta = \{+, -, *, /, \dots\}$

Target Machine Model

We assume a machine with finite set of registers r_0, r_1, ..., r_k, countable set of memory locations, and instructions of the form:

1. $m \leftarrow r$ (store instruction)
2. $r \leftarrow m$ (load instruction)
3. $r \leftarrow r\ op\ m$ (the result of $r\ op\ m$ is stored in r)
4. $r_2 \leftarrow r_2\ op\ r_1$ (the result of $r_2\ op\ r_1$ is stored in r_2)

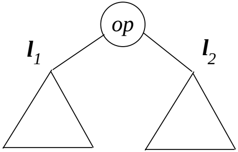
Note:

1. In instruction 3, the memory location is the right operand.
2. In instruction 4, the destination register is the same as the left operand register.

Order of Evaluation and Register Requirement

Consider evaluation of a tree without stores. Assume that the left and right subtrees require upto l_1, and l_2 ($l_1 < l_2$) registers.

In what order should we evaluate the subtrees to minimize register requirement?



Choice 1

1. Left subtree first, leaving result in a register. This requires upto l_1 registers.
2. Evaluate the right subtree. During this we require upto l_2 for evaluating the right subtree and one to hold value of the left subtree.

Register requirement — $max(l_1, l_2 + 1) = l_2 + 1$.

Key Idea

Choice 2

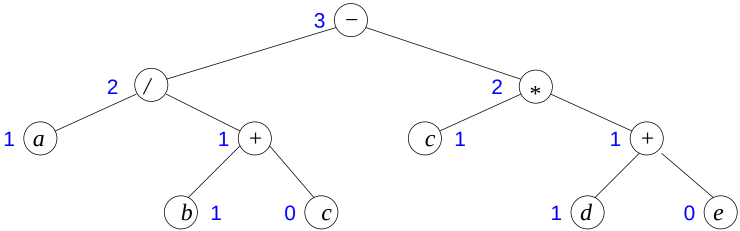
1. Evaluate the right subtree first, leaving the result in a register. During this evaluation we shall require up to l_2 registers.
2. Evaluate the left subtree. During this, we might require up to $l_1 + 1$ registers.

Register requirement — $max(l_1 + 1,\ l_2) = l_2$

Therefore the subtree requiring more registers should be evaluated first.

Labeling the Expression Tree

Label each node by the number of registers required to evaluate it in a store free manner.



Left and the right leaves are labeled 1 and 0 respectively, because the left leaf must necessarily be in a register, whereas the right leaf can reside in memory.

Labeling the Expression Tree

Visit the tree in post-order. For every node visited do:

1. Label each left leaf by 1 and each right leaf by 0.
2. If the labels of the children of a node n are l_1 and l_2 respectively, then

$$\begin{aligned} label(n) &= max(l_1, l_2), \text{ if } l_1 \neq l_2 \\ &= l_1 + 1, \text{ otherwise} \end{aligned}$$

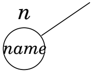
Assumptions and Notational Conventions

1. The code generation algorithm is represented as a function *gencode*(n), which produces code to evaluate the node labeled n.
2. Register allocation is done from a stack of register names *rstack*, initially containing $r_0, r_1, \ldots, r_k$ (with r_0 on top of the stack).
3. *gencode*(n) evaluates n in the register on the top of the stack.
4. Temporary allocation is done from a stack of temporary names *tstack*, initially containing $t_0, t_1, \ldots, t_k$ (with t_0 on top of the stack).
5. *swap*(*rstack*) swaps the top two registers on the stack.

The Algorithm

gencode(n) described by case analysis on the type of the node n.

1. *n is a left leaf:*

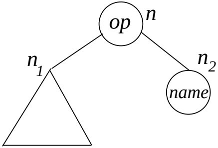


$$gen(\ top(rstack) \leftarrow name)$$

Comments: n is named by a variable say *name*. Code is generated to load *name* into a register.

The Algorithm

2. *n's right child is a leaf:*

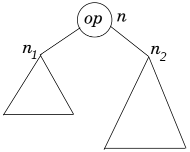


$gencode(n_1);$
$gen(top(rstack) \leftarrow top(rstack)\ op\ name)$

Comments: n_1 is first evaluated in the register on the top of the stack, followed by the operation *op* leaving the result in the same register.

The Algorithm

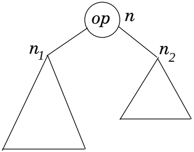
3. *The left child is the lighter subtree. This requirement is strictly less than the available number of registers*



$swap(rstack);$	
$gencode(n_2);$	Evaluate right child
$R := pop(rstack);$	
$gencode(n_1);$	Evaluate left child
$gen(top(rstack) \leftarrow top(rstack)\ op\ R);$	Issue *op*
$push(rstack, R);$	
$swap(rstack)$	Restore register stack

The Algorithm

4. *The right child of n is lighter or as heavy as the left child. Its requirement is strictly less than the available number of registers*

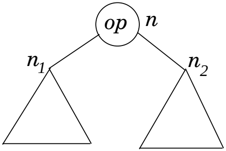


$gencode(n_1)$;
$R := pop(rstack)$;
$gencode(n_2)$;
$gen(top(rstack) \leftarrow top(rstack)\ op\ R)$;
$push(rstack, R)$

Comments: Same as case 3, except that the left sub-tree is evaluated first.

The Algorithm

5. *Both the children of n require registers greater or equal to the available number of registers.*

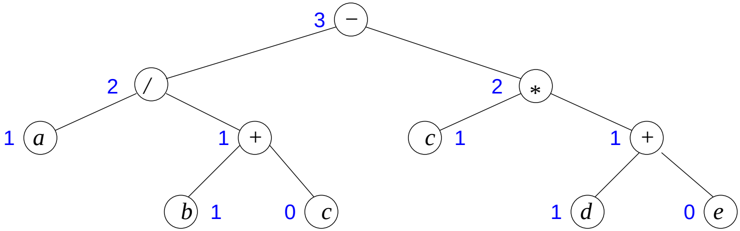


$gencode(n_2);$
$T := pop(tstack);$
$gen(T \leftarrow top(rstack));$
$gencode(n_1);$
$push(tstack, T);$
$gen(top(rstack) \leftarrow top(rstack)\ op\ T;$

Comments: Evaluate the right sub-tree into a temporary. Then evaluate the left sub-tree and n into the register on top of stack.

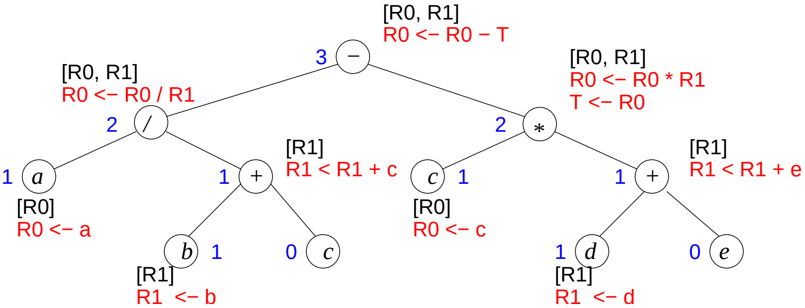
Example

For the example:



assuming two available registers r_0 and r_1, the calls to gencode and the generated code are shown below.

Example


[R0, R1]
R0 <− R0 − T
3
−
[R0, R1]
R0 <− R0 / R1
2
/
[R0, R1]
R0 <− R0 * R1
T <− R0
2
*
1
a
[R0]
R0 <− a
1
+
[R1]
R1 < R1 + c
b
1
[R1]
R1 <− b
0
c
c
1
[R0]
R0 <− c
1
+
[R1]
R1 < R1 + e
1
d
[R1]
R1 <− d
0
e

Optimality

The algorithm is optimal because

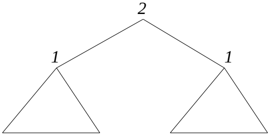
1. The number of load instructions generated is optimal.
2. Each binary operation specified in the expression tree is performed only once.
3. The number of stores is optimal.

1 and 2 are obvious. 3 is harder to prove.

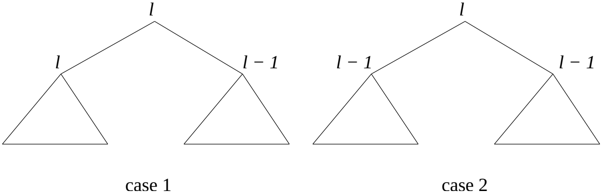
Optimality

If the label of a node is greater than the number of registers, then the tree under it cannot be evaluated (by any algorithm) without a store.

1. True for base case:



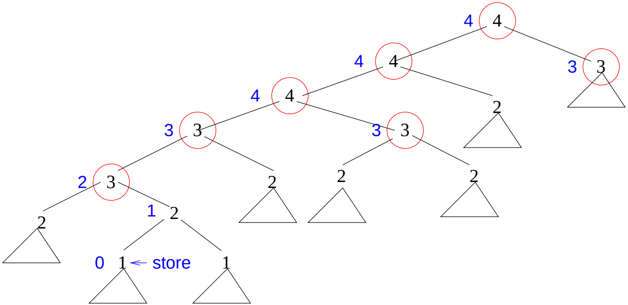
2. True for a larger tree, assuming true for subtrees.



Optimality

Define a major node as a node, both of whose children have a label greater than or equal to the number of registers. Then we have:

- Every store decreases the number of major nodes by at most one.



Optimality

If a tree has M major nodes, then any algorithm would need at least M stores to compute the tree

- Consider a subtree with a single major node at the root. Evaluating the tree would require at least one store (previous result).
- Replace the subtree by a memory node. The resulting tree has at least M-1 major nodes
- Repeating the argument, we see that at least M stores would be required.
- Since Sethi-Ullman issues a store for every major node, it is optimal.

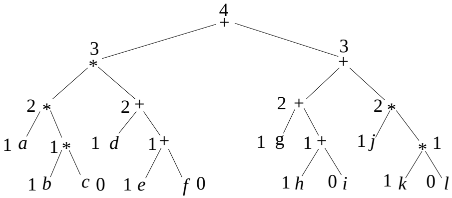
Complexity of the Algorithm

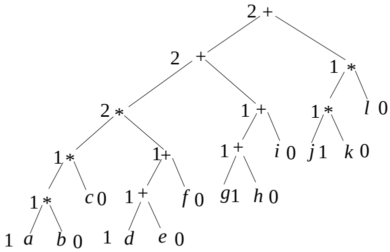
Since the algorithm visits every node of the expression tree twice – once during labeling, and once during code generation, the complexity of the algorithm is $O(n)$.

Problems

1. Consider the expression $((a*(b*c))*(d+(e+f)))+((g+(h+i))+(j*(k*l)))$. Assuming a machine with instructions:
   $R_i \leftarrow R_i \ op \ R_j$
   $R_i \leftarrow R_i \ op \ m$
   $R_i \leftarrow m$
   $m \leftarrow R_i$
   1.1 Draw the expression as a tree. Calculate the label at each node of the tree.
   1.2 Using algebraic properties of the operators rearrange the tree so that the label at the root is minimized. Once again label the tree.
   1.3 Assuming that the machine has 2 general purpose registers R_1 and R_2, generate optimal code for the tree.
2. If the code produced by Sethi-Ullman is storeless, is it necessarily strongly contiguous? If it contains stores, is it necessarily in strong normal form?
3. Let N be the total number of registers, l be the label of a node n, and r be the available number of registers while invoking $gencode(n)$. Then complete the following sentence: $l \geq N \Rightarrow r =$___, and $l < N \Rightarrow$ ___ $\leq r \leq$ ___.

Solutions

1. 

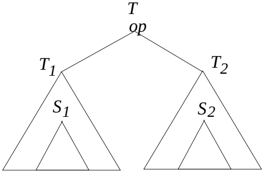
2. 

Solutions

1. $$\begin{aligned}
&R_1 \leftarrow a\\
&R_1 \leftarrow R_1 * b\\
&R_1 \leftarrow R_1 * c\\
&R_2 \leftarrow d\\
&R_2 \leftarrow R_2 + e\\
&R_2 \leftarrow R_2 + f\\
&R_1 \leftarrow R_1 + R_2\\
&R_2 \leftarrow g\\
&R_2 \leftarrow R_2 + h\\
&R_2 \leftarrow R_2 + i\\
&R_1 \leftarrow R_1 + R_2\\
&R_2 \leftarrow j\\
&R_2 \leftarrow R_2 * k\\
&R_1 \leftarrow R_1 * l\\
&R_1 \leftarrow R_1 + R_2
\end{aligned}$$

Solutions

1. Yes. Yes. In the figure assume that stores are required at the roots of the subtrees S_1 and S_2. Also assume that the label of T_1 is the same as label of T_2.



For this example, the root will be a major node and therefore a store is needed after evaluation of T_2.
T will be evaluated as:
S_2; *store*; $T_2 - S_2$; *store*; S_1; *store*; $T_1 - S_1$; *op*

2. Let N be the total number of registers, l be the label of a node n, and r be the available number of registers while invoking $gencode(n)$. Then complete the following sentence:
$l \geq N \Rightarrow r = N$, and $l < N \Rightarrow l \leq r \leq N$.

Aho-Johnson Algorithm – Introduction

- Considers expression trees.
- The target machine model is general enough to generate code for a large class of machines. Represented as a tree, an instruction
  - can have a root of any arity.
  - can have as leaves registers or memory locations appearing in any order.
  - can be of of any height
- Does not use algebraic properties of operators.
  - If $e_1 * e_2$ has to be evaluated using $r \leftarrow r * m$, and
  - e_1 and e_2 are in m and r,

    then the code sequence has to be:

$$\begin{array}{lcl} m_1 & \leftarrow & r \\ r & \leftarrow & m \\ r & \leftarrow & r * m_1 \end{array}$$

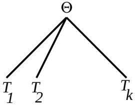
and not simply: $r \leftarrow r * m$

- Generates optimal code, where, once again, the cost measure is the number of instructions in the code. This can be modified.
- Complexity is linear in the size of the expression tree.

Aho-Johnson Algorithm

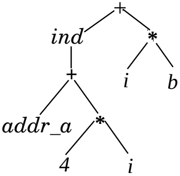
Let Θ be a finite set of operators. Then,

1. A single vertex labeled by a variable name or constant is an expression tree.
2. If T_1, T_2, . . . , T_k are expression and θ is a k-ary operator in Θ, then



is an expression tree.

An example of an expression tree is



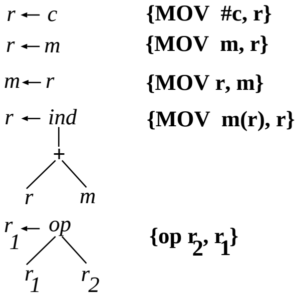
The Machine Model

Considers machines which have

1. n general purpose registers (no special registers).
2. sequence of memory locations,
3. instructions of the form
   a. $r \leftarrow E$, r is a register and E is an expression tree whose operators are from Θ and operands are registers, memory locations or constants. Further, *r should be one the register names occurring (if any) in E*.
   b. $m \leftarrow r$, a store instruction.

The Machine Model

- Here is an example of a machine.



Machine Program

- A machine program consists of a finite sequence of instructions $P = I_1 I_2 \ldots I_q$.
- The machine program below evaluates $a[i] + i * b$

$$
\begin{aligned}
r_1 &\leftarrow 4 \\
r_1 &\leftarrow r_1 * i \\
r_2 &\leftarrow addr\_a \\
r_2 &\leftarrow r_2 + r_1 \\
r_2 &\leftarrow ind(r_2) \\
r_3 &\leftarrow i \\
r_3 &\leftarrow r_3 * b \\
r_2 &\leftarrow r_2 + r_3
\end{aligned}
$$

- A machine program computing an expression tree will have at most one use for each definition.

Rearrangability of Programs

- We shall show that any program can be rearranged to obtain an equivalent program of the same length in strong normal form.
- Aho-Johnson's algorithm searches for the optimal only amongst strong normal form programs.
- The above result assures us that by doing so, we shall not miss out an optimal solution.

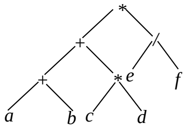
Width

- The width of a program is the maximum number of registers live at any instruction.
- A program of width w (but possibly using more than w registers) can always be rearranged into an equivalent program using exactly w registers.
- In the example below, the first program has width 2 but uses 3 registers. By suitable renaming, the number of registers in the second program has been brought down to 2.

$r_1 \leftarrow a$	$r_1 \leftarrow a$
$r_2 \leftarrow b$	$r_2 \leftarrow b$
$r_1 \leftarrow r_1 + r_2$	$r_1 \leftarrow r_1 + r_2$
$r_3 \leftarrow c$	$r_2 \leftarrow c$
$r_3 \leftarrow r_3 + d$	$r_2 \leftarrow r_2 + d$
$r_1 \leftarrow r_1 * r_3$	$r_1 \leftarrow r_1 * r_2$

Contiguity and Strong Contiguity

Can one decrease the width of a program?



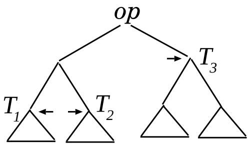
P_1	P_2	P_3
$r_1 \leftarrow a$	$r_1 \leftarrow a$	$r_1 \leftarrow a$
$r_2 \leftarrow b$	$r_2 \leftarrow b$	$r_2 \leftarrow b$
$r_3 \leftarrow c$	$r_3 \leftarrow c$	$r_1 \leftarrow r_1 + r_2$
$r_4 \leftarrow d$	$r_4 \leftarrow d$	$r_2 \leftarrow c$
$r_5 \leftarrow e$	$r_1 \leftarrow r_1 + r_2$	$r_3 \leftarrow d$
$r_6 \leftarrow f$	$r_3 \leftarrow r_3 * r_4$	$r_2 \leftarrow r_2 * r_3$
$r_5 \leftarrow r_5 / r_6$	$r_1 \leftarrow r_1 + r_3$	$r_1 \leftarrow r_1 + r_2$
$r_3 \leftarrow r_3 * r_4$	$r_2 \leftarrow e$	$r_2 \leftarrow e$
$r_1 \leftarrow r_1 + r_2$	$r_3 \leftarrow f$	$r_3 \leftarrow f$
$r_1 \leftarrow r_1 + r_3$	$r_2 \leftarrow r_2 / r_3$	$r_2 \leftarrow r_2 / r_3$
$r_1 \leftarrow r_1 * r_5$	$r_1 \leftarrow r_1 * r_2$	$r_1 \leftarrow r_1 * r_2$

Contiguity and Strong Contiguity

- Can one decrease the width of a program? For storeless programs, there is an arrangement which has minimum width.
- All the three programs P_1, P_2, and P_3 compute the expression tree shown below:
- The program P_2 has a width less than P_1, whereas P_3 has the least width of all three programs. P_2 is a contiguous program whereas P_3 is a strongly contiguous (SC) program.
- Every program without stores can be transformed into SC form.

Strong Normal Form Programs

- Programs requiring stores can also be cast in a certain form called strong normal form.

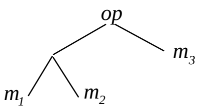


The marked nodes, T_1, T_2 and T_3 require stores.

- Compute T_1 using a SC program P_1. Store the result in m_1.
- Compute T_2 using a SC program P_2. Store the result in m_2.
- Compute T_3 using a SC program P_3. Store the result in m_3.
- Compute the resulting tree using a SC program P_4.

The resultant program has the form $P_1 J_1 P_2 J_2 P_3 J_3 P_4$.
The J_is are stores.

Strong Normal Form Programs



- A program in such a form is called a normal form program. A normal form program looks like $P_1 J_1 P_2 J_2 \ldots P_{s-1} J_{s-1} P_s$.
- Further, P is in strong normal form, if each P_i is strongly contiguous.

- THEOREM: Let P be a program of width w. We can transform P into an equivalent program Q such that:
  - P and Q have the same cost.
  - Q has width at most w, and
  - Q is in strong normal form.

The Algorithm

- The algorithm makes three passes over the expression tree.

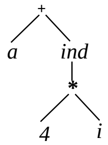
Pass 1 Computes an array of costs for each node. This helps to select an instruction to evaluate the node, and the evaluation order to evaluate the subtrees of the node.

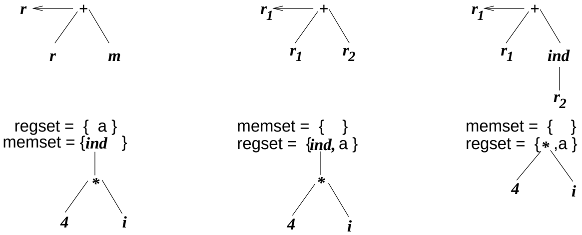
Pass 2 Identifies the subtrees which must be evaluated in memory locations.

Pass 3 Actually generates code.

Cover

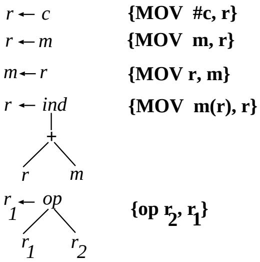
- An instruction covers a node in an expression tree, if it can be used to evaluate the node.





The Algorithm – Pass 1

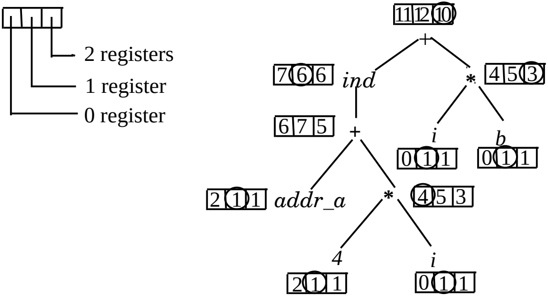
- **Pass 1:** Calculates an array of costs $C_j(s)$ for every subtree S of T, whose meaning is to be interpreted as follows:
  - $C_j(S), j \neq 0$: is the minimum cost of evaluating S with a strong normal form program using j registers.
  - $C_0(S)$: cost of evaluating S strong normal form program in a memory location.
- Consider a machine with the instructions shown below.



Note that there are no instructions of the form $r \leftarrow r \; op \; m$.

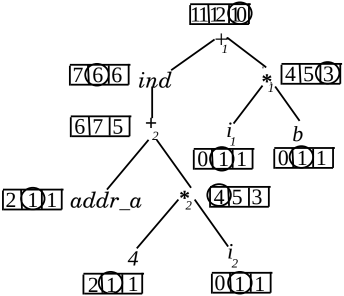
The Algorithm – Pass 1

- We show an example expression tree, along with the cost array at each node:



The Algorithm – Pass 2

- This pass marks the nodes which have to be evaluated into memory. It returns a sequence of nodes $x_1, \ldots, x_s$, where $x_1, \ldots, x_s$ represent the nodes to be evaluated in memory.



- The node $*_2$ has to be stored in memory.

The Algorithm – Pass 3

- The algorithm generates code for the subtrees rooted at $x_1, \ldots x_s$, in that order.
- After generating code for x_i, the algorithm replaces the node with a distinct memory location m_i.
- For the example, the code generated is:

$r_1 \leftarrow \#4$	(evaluate $4 * i$ first, since it is to be stored)
$r_2 \leftarrow i$	
$r_1 \leftarrow r_1 * r_2$	
$m_1 \leftarrow r_1$	
$r_1 \leftarrow i$	(evaluate $i * b$ next, since it requires 2 registers)
$r_2 \leftarrow b$	
$r_1 \leftarrow r_1 * r_2$	
$r_2 \leftarrow \#addr\_a$	
$r_2 \leftarrow m_1(r_2)$	(evaluate the *ind* node)
$r_2 \leftarrow r_2 + r_1$	(evaluate the root)

Complexity of the Algorithm

1. The time required by Pass 1 is an, where a is a constant depending
   - linearly on the size of the instruction set
   - exponentially on the arity of the machine, and
   - linearly on the number of registers in the machine

   and n is the number of nodes in the expression tree.
2. Time required by Passes 2 and 3 is proportional to n

Therefore the complexity of the algorithm is $O(n)$.

Example

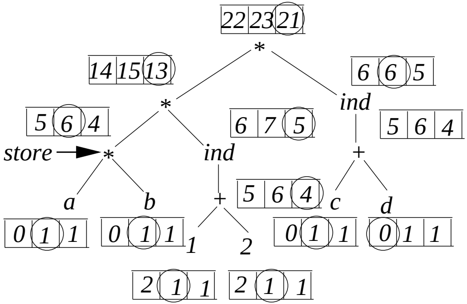
- Consider a machine model with 2 general purpose registers and instructions shown below with their costs.

$R_i \leftarrow R_i \ op \ R_j$	cost – 2
$R \leftarrow c$	cost – 1
$R \leftarrow m$	cost – 1
$R \leftarrow ind(R)$	cost – 1
$R \leftarrow ind(R + m)$	cost – 4
$m \leftarrow R$	cost – 1

Now consider the expression

$$((a * b) * (ind(1 + 2))) * (ind(c + d)).$$

The Cost Array



Generated code

The code generated is:

$R_1 \leftarrow a$	code for the subtree to be stored
$R_2 \leftarrow b$	
$R_1 \leftarrow R_1 * R_2$	
$m \leftarrow R_1$	
$R_1 \leftarrow 1$	code for $ind(1+2)$
$R_2 \leftarrow 2$	
$R_1 \leftarrow R_1 + R_2$	
$R_1 \leftarrow ind(R_1)$	
$R_2 \leftarrow m$	code for $(a * b) * ind(1+2)$
$R_2 \leftarrow R_2 * R_1$	
$R_1 \leftarrow c$	code for $ind(c+d)$
$R_1 \leftarrow ind(R_1 + d)$	
$R_2 \leftarrow R_2 * R_1$	code for the root