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 . . .

Front End

Lexical Analyser → Syntax Analyser → Semantic Analyser → Intermediate Code Gen

Mc. Dependent Optimizer → Code Generator

Back End

Mc. Independent Optimizer
Code Generation - Issues

- Expressions and Assignments:
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.
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 \( e_1 \times e_2 \) has to be evaluated using \( r \leftarrow r \times m \), and
  - \( e_1 \) and \( e_2 \) are in \( m \) and \( r \),
    then the code sequence has to be:

\[
\begin{align*}
  m_1 & \leftarrow r \\
  r & \leftarrow m \\
  r & \leftarrow r \times m_1
\end{align*}
\]

- 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 \( a/(b + c) - c \times (d + e) \) represented as a tree:

```
          -
         /  *
     /    +  +
   a       c  d  e
  +   +
 b  c  d  e
```

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

```
          -
         /  *
     /    +  +
   a       c  d  e
  +   +
 b  c  d  e
```
Expression Trees

Let $\Sigma$ be a countable set of variable names, and $\Theta$ be a finite set of binary operators. Then,

1. A single vertex labeled by a name from $\Sigma$ is an expression tree.
2. If $T_1$ and $T_2$ are expression trees and $\theta$ is a operator in $\Theta$, then

$$T \theta T_1 T_2$$

is an expression tree.

In the previous example

$\Sigma = \{a, b, c, d, e, \ldots\}$, and $\Theta = \{+, -, *, /, \ldots\}$
Target Machine Model

We assume a machine with finite set of registers \( r_0, r_1, \ldots, r_k \), 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 up to \( l_1 \), and \( l_2 \) \((l_1 < l_2)\) registers.

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

\[ \text{op} \]

\[ l_1 \quad l_2 \]

**Choice 1**

1. Left subtree first, leaving result in a register. This requires up to \( l_1 \) registers.

2. Evaluate the right subtree. During this we require up to \( l_2 \) 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

   $$\text{label}(n) = \max(l_1, l_2), \text{ if } l_1 \neq l_2$$
   $$= l_1 + 1, \text{ otherwise}$$
1. The code generation algorithm is represented as a function $\text{gencode}(n)$, 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. $\text{gencode}(n)$ 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. $\text{swap}(rstack)$ swaps the top two registers on the stack.
The Algorithm

\( \text{gencode}(n) \) described by case analysis on the type of the node \( n \).

1. \( n \) is a left leaf:

\[
\text{gen}( \text{top}(\text{rstack}) \leftarrow \text{name})
\]

Comments: \( n \) is named by a variable say \( \text{name} \). Code is generated to load \( \text{name} \) into a register.
2. *n’s right child is a leaf:*

\[
\begin{align*}
\text{gen}(\text{top}(\text{rstack}) & \leftarrow \text{top}(\text{rstack}) \text{ op } \text{name}) \\
gencode(n_1);
\end{align*}
\]

*Comments: \(n_1\) is first evaluated in the register on the top of the stack, followed by the operation \(\text{op}\) 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(n2);           Evaluate right child
R := pop(rstack);
gencode(n1);           Evaluate left child
gen(top(rstack) ← 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

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

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.

\[
\begin{align*}
\text{gencode}(n_2); \\
T &:= \text{pop}(tstack); \\
\text{gen}(T \leftarrow \text{top}(rstack)); \\
\text{gencode}(n_1); \\
\text{push}(tstack, T); \\
\text{gen}(\text{top}(rstack) \leftarrow \text{top}(rstack) \text{ op } T);
\end{align*}
\]

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

```
+ 1 0
c
[R1] R1 <- b

+ 1
 1
[R0] R0 <- a

/ 2
1
[R0, R1] R0 <- R0 / R1

- 3
3
[R0, R1] R0 <- R0 - T

* 2
2
[R0, R1] R0 <- R0 * R1
T <- R0

+ 1
1
[R0] R0 <- c

+ 1
1
[R1] R1 <- d

+ 1
0
[R1] R1 <- e
```

Amitabha Sanyal
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
   /\  /\  \
  1 1 1
   /\  /\  \
  l  l l

2. True for a larger tree, assuming true for subtrees.
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.
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 \((a \times (b \times c)) \times (d + (e + f)) + (g + (h + i)) + (j \times (k \times l)))\). Assuming a machine with instructions:
   \[R_i \leftarrow R_i \text{ op } R_j\]
   \[R_i \leftarrow R_i \text{ op } m\]
   \[R_i \leftarrow m\]
   \[m \leftarrow R_i\]

   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.

Amitabha Sanyal
Solutions

1.  

\[
\begin{align*}
R_1 & \leftarrow a \\
R_1 & \leftarrow R_1 \times b \\
R_1 & \leftarrow R_1 \times c \\
R_2 & \leftarrow d \\
R_2 & \leftarrow R_2 + e \\
R_2 & \leftarrow R_2 + f \\
R_1 & \leftarrow R_1 + R_2 \\
R_2 & \leftarrow g \\
R_2 & \leftarrow R_2 + h \\
R_2 & \leftarrow R_2 + i \\
R_1 & \leftarrow R_1 + R_2 \\
R_2 & \leftarrow j \\
R_2 & \leftarrow R_2 \times k \\
R_1 & \leftarrow R_1 \times l \\
R_1 & \leftarrow R_1 + R_2
\end{align*}
\]
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:

$S_2; \text{store}; T_2 – S_2; \text{store}; S_1; \text{store}; T_1 – S_1; \text{op}$

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 $\text{gencode}(n)$. 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 any height
- Does not use algebraic properties of operators.
  - If $e_1 \times e_2$ has to be evaluated using $r \leftarrow r \times m$, and
  - $e_1$ and $e_2$ are in $m$ and $r$,
    then the code sequence has to be:

\[
\begin{align*}
  m_1 & \leftarrow r \\
  r & \leftarrow m \\
  r & \leftarrow r \times m_1
\end{align*}
\]

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

- 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 $\Theta$ be a finite set of operators. Then,

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

$$\Theta(T_1, T_2, \ldots, T_k)$$

is an expression tree.

An example of an expression tree is

$$\text{ind} + \\text{addr}_a \ast i b$$
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 $\Theta$ 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.
Here is an example of a machine.

\[
\begin{align*}
  r &\leftarrow c & \{\text{MOV } \#c, r\} \\
  r &\leftarrow m & \{\text{MOV } m, r\} \\
  m &\leftarrow r & \{\text{MOV } r, m\} \\
  r &\leftarrow \text{ind} & \{\text{MOV } m(r), r\} \\
  r &\leftarrow \text{op} & \{\text{op } r_2, r_1\}
\end{align*}
\]
A machine program consists of a finite sequence of instructions $P = I_1 I_2 \ldots I_q$.

The machine program below evaluates $a[i] + i * b$

$$
\begin{align*}
    r_1 &\leftarrow 4 \\
    r_1 &\leftarrow r_1 * i \\
    r_2 &\leftarrow addr_a \\
    r_2 &\leftarrow r_2 + r_1 \\
    r_2 &\leftarrow ind(r_2) \\
    r_3 &\leftarrow i \\
    r_3 &\leftarrow r_3 * b \\
    r_2 &\leftarrow r_2 + r_3
\end{align*}
$$

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.

\[
\begin{align*}
    r_1 &\leftarrow a \\
    r_2 &\leftarrow b \\
    r_1 &\leftarrow r_1 + r_2 \\
    r_3 &\leftarrow c \\
    r_3 &\leftarrow r_3 + d \\
    r_1 &\leftarrow r_1 \ast r_3
\end{align*}
\]

\[
\begin{align*}
    r_1 &\leftarrow a \\
    r_2 &\leftarrow b \\
    r_1 &\leftarrow r_1 + r_2 \\
    r_2 &\leftarrow c \\
    r_2 &\leftarrow r_2 + d \\
    r_1 &\leftarrow r_1 \ast r_2
\end{align*}
\]
Contiguity and Strong Contiguity

Can one decrease the width of a program?

\[
\begin{align*}
P_1 & \quad \begin{align*}
r_1 & \leftarrow a \\
r_2 & \leftarrow b \\
r_3 & \leftarrow c \\
r_4 & \leftarrow d \\
r_5 & \leftarrow e \\
r_6 & \leftarrow f \\
r_5 & \leftarrow r_5/r_6 \\
r_3 & \leftarrow r_3 \times r_4 \\
r_1 & \leftarrow r_1 + r_2 \\
r_1 & \leftarrow r_1 + r_3 \\
r_1 & \leftarrow r_1 \times r_5
\end{align*}
\end{align*}
\]

\[
\begin{align*}
P_2 & \quad \begin{align*}
r_1 & \leftarrow a \\
r_2 & \leftarrow b \\
r_3 & \leftarrow c \\
r_4 & \leftarrow d \\
r_1 & \leftarrow r_1 + r_2 \\
r_3 & \leftarrow r_3 \times r_4 \\
r_1 & \leftarrow r_1 + r_3 \\
r_2 & \leftarrow r_2/r_3 \\
r_1 & \leftarrow r_1 \times r_2
\end{align*}
\end{align*}
\]

\[
\begin{align*}
P_3 & \quad \begin{align*}
r_1 & \leftarrow a \\
r_2 & \leftarrow b \\
r_1 & \leftarrow r_1 + r_2 \\
r_2 & \leftarrow e \\
r_3 & \leftarrow f \\
r_2 & \leftarrow r_2/r_3 \\
r_1 & \leftarrow r_1 \times r_2
\end{align*}
\end{align*}
\]
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**.

![Diagram](op

\[ T_1 \quad \rightarrow \quad T_2 \quad \rightarrow \quad T_3 \]

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 \( J_i \)s are stores.
**Strong Normal Form Programs**

- A program in such a form is called a **normal form program**. A normal form program looks like $P_1J_1P_2J_2 \ldots P_{s-1}J_{s-1}P_s$.
- 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.
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.

\[
\begin{align*}
  & \quad \{\text{MOV } \#c, r\} \\
  r & \leftarrow c \\
  & \quad \{\text{MOV } m, r\} \\
  r & \leftarrow m \\
  m & \leftarrow r \\
  & \quad \{\text{MOV } r, m\} \\
  & \quad \{\text{MOV } m(r), r\} \\
  r & \leftarrow \text{ind} \\
  & \quad \{\text{MOV } m(r), r\} \\
  r & \leftarrow \text{ind} \\
  \quad \{\text{op } r_2, r_1\} \\
  r_1 & \leftarrow \text{op } r_2, r_1 \\
  r_1 & \leftarrow \text{op } r_2, r_1 \\
  r_2 & \leftarrow \text{op } r_2, r_1
\end{align*}
\]

Note that there are no instructions of the form \( r \leftarrow r \text{ op } m \).
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 \quad \text{(evaluate $4 \times i$ first, since it is to be stored)}
  r_2 \leftarrow i \\
r_1 \leftarrow r_1 \times r_2 \\
m_1 \leftarrow r_1 \\
r_1 \leftarrow i \quad \text{(evaluate $i \times b$ next, since it requires 2 registers)}
  r_2 \leftarrow b \\
r_1 \leftarrow r_1 \times r_2 \\
r_2 \leftarrow \#\text{addr}_a \\
r_2 \leftarrow m_1(r_2) \quad \text{(evaluate the ind node)}
  r_2 \leftarrow r_2 + r_1 \quad \text{(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.

\[
\begin{align*}
R_i &\leftarrow R_i \text{ op } R_j \quad \text{cost} - 2 \\
R &\leftarrow c \quad \text{cost} - 1 \\
R &\leftarrow m \quad \text{cost} - 1 \\
R &\leftarrow \text{ind}(R) \quad \text{cost} - 1 \\
R &\leftarrow \text{ind}(R + m) \quad \text{cost} - 4 \\
m &\leftarrow R \quad \text{cost} - 1
\end{align*}
\]

Now consider the expression

\[
((a \ast b) \ast (\text{ind}(1 + 2))) \ast (\text{ind}(c + d)).
\]
The Cost Array

\[
\begin{align*}
\text{store} & \quad \ast \\
\text{ind} & \quad \ast \\
\text{ind} & \quad \ast \\
\end{align*}
\]
The code generated is:

\[
\begin{align*}
R_1 &\leftarrow a &\text{code for the subtree to be stored} \\
R_2 &\leftarrow b \\
R_1 &\leftarrow R_1 \ast R_2 \\
m &\leftarrow R_1 \\
R_1 &\leftarrow 1 &\text{code for } ind(1 + 2) \\
R_2 &\leftarrow 2 \\
R_1 &\leftarrow R_1 + R_2 \\
R_1 &\leftarrow ind(R_1) \\
R_2 &\leftarrow m &\text{code for } (a \ast b) \ast ind(1 + 2) \\
R_2 &\leftarrow R_2 \ast R_1 \\
R_1 &\leftarrow c &\text{code for } ind(c + d) \\
R_1 &\leftarrow ind(R_1 + d) \\
R_2 &\leftarrow R_2 \ast R_1 &\text{code for the root}
\end{align*}
\]