Trivial Code Generation

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.
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
\[
\begin{align*}
t_1 &:= a + b \\
t_2 &:= c + d \\
t_3 &:= e - t_2 \\
t_4 &:= t_1 - t_3
\end{align*}
\]

Second Order
\[
\begin{align*}
t_1 &:= a + b \\
t_2 &:= c + d \\
t_3 &:= e - t_2 \\
t_4 &:= t_1 - t_3
\end{align*}
\]

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
\[
\begin{align*}
t_1 &:= a + b \\
t_2 &:= c + d \\
t_3 &:= e - t_2 \\
t_4 &:= t_1 - t_3
\end{align*}
\]

Second Order
\[
\begin{align*}
t_1 &:= a + b \\
t_2 &:= c + d \\
t_3 &:= e - t_2 \\
t_4 &:= t_1 - t_3
\end{align*}
\]

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

If we have a node \( n \) with subtrees \( n_1 \) and \( n_2 \) with \( L=\text{label}(n_1) \) & \( R=\text{label}(n_2) \) & \( L<R \) then we can first evaluate \( n_2 \) into a register \( \text{Reg} \) 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 \( \text{Reg} \) 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 \( n_2 \).
The Generation Phase I
\[ \text{gencode}(n) \] generates machine code for a subtree \( n \) of a labeled tree \( T \).

\[ \text{MOV} \ M, R \quad \text{Load variable} \ M \quad \text{into register} \ R. \]
\[ \text{MOV} \ R, M \quad \text{Store register} \ R \quad \text{into variable} \ M. \]
\[ \text{OP} \ M, R \quad \text{Compute} \ R := R \ \text{OP} \ M. \]
\[ \text{OP} \in \{\text{ADD, SUB, MUL, DIV}\}. \]
\[ \text{OP} \ R2, R1 \quad \text{Compute} \ R1 := R1 \ \text{OP} \ R2. \]

A stack \( rstack \) initially contains all available registers.
\[ \text{gencode}(n) \] 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.

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.

The Generation Phase III

Case 0

1. Generate a load instruction to load the variable into a register:
\[ \text{MOV} \ \text{name}, \ \text{top}(rstack) \]

Case 1

1. \( n_1 \) can be evaluated without spilling, but \( n_2 \) requires more registers than \( n_1 \).
2. We swap the two top registers on \( rstack \), evaluate \( n_2 \) into \( \text{top}(rstack) \), remove the top register, then evaluate \( n_1 \) into \( \text{top}(rstack) \). Restore the stack.

The Generation Phase IV

Case 2

1. \( n_1 \) can be evaluated without spilling, but \( n_2 \) requires more registers than \( n_1 \).
2. We swap the two top registers on \( rstack \), evaluate \( n_2 \) into \( \text{top}(rstack) \), remove the top register, then evaluate \( n_1 \) into \( \text{top}(rstack) \). Restore the stack.
3. \( \text{swap}(rstack), \ gencode(n_2) \)
4. \( R := \text{pop}(rstack) \)
5. \( \text{gencode}(n_1) \)
6. Generate \( \text{OP} \ R, \ \text{top}(rstack) \)
7. \( \text{push}(rstack, R), \ \text{swap}(rstack) \)
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 $\text{top(rstack)}$, remove the top register, then evaluate $n_2$ into $\text{top(rstack)}$.

1. $\text{gencode}(n_1)$
2. $R := \text{pop(rstack)}$
3. $\text{gencode}(n_1)$
4. Generate $\text{OP top(rstack)}, R$
5. $\text{push(rstack, R)}$

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 $\text{top(tstack)}$, and then we evaluate $n_1$ into $\text{top(rstack)}$.

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

Examples

Example I (A)

```
Example I (A)
gencode(t_4) [R1,R0] case2
gencode(t_3) [R0,R1] case3
gencode(e) [R0,R1] case0
MOV e, R1
gencode(t_2) [R0] case1
gencode(c) [R0] case0
MOV c, R0
SUB R0, R1
gencode(t_1) [R0] case1
gencode(a) [R0] case0
MOV a, R0
```
Summary

- This lecture is taken from the Dragon Book:
Why do we swap registers in Case 2?

This node expects its right son to generate code that evaluates into register R0.

Case 2

\[ R0 := R1 - R0 \]

Generate code for this one first, into register R1

Generate code for this one second, into register R0

Two registers (R0,R1) are available.

\[ (a + (b/c))/(d \times (e + f)) \]

One register (R0) is available.

\[ (a + (b/c))/(d \times (e + f)) \]

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