COEN 175

Lecture 16: Code Generation for Expressions

Code Generation

- Functions consist of statements, and statements consist of expressions.
- So, we will start with code generation techniques for expressions.
- Note that some expressions affect the flow of control and will be discussed together with statements.
  - Short-circuiting logical operators in C and Simple C.
  - The inline conditional operator in C.

Using Registers

- Ideally, we would keep all data in registers.
- However, this is not always possible.
  - We may not have enough registers.
  - A variable may be too large to fit in a register.
  - A variable might be a global variable.
  - A variable might have its address taken.
- The process of assigning variables and expressions to registers is known as **register allocation**.
  - Optimal register allocation is NP-complete.

Intermediate Results

- As a compromise, we can keep variables in memory but intermediate results of expressions in registers.
- This approach eliminates many problems:
  - Intermediate results will always fit in a register;
  - They cannot have their addresses taken;
  - They cannot be referenced globally.
- We still have to deal with the problem of not having enough registers.
- This approach is used by many compilers such as GCC and Clang unless optimizations are enabled.

Stack-Based Temporaries

- A final, naïve approach would be to store everything on the stack.
- This approach would be very slow, but would not have any of the problems we discussed earlier.
- Effectively, we would need to assign the result of each operation to a temporary variable.
- Nothing would be kept in registers beyond the lifetime of a single operation.

Three-Address Code

- We would effectively be building an intermediate representation known as **three-address code**.
- A three-address code statement can do at most one operation: *x* `:=` *y op z* or *x* `:=` *op y*.
- Translate `a * b + c * d` into three-address code.

$$
\begin{aligned}
t_0 &:= a * b \\
t_1 &:= c * d \\
t_2 &:= t_0 + t_1
\end{aligned}
$$

- Here, t_0, t_1, and t_2 are stack-based temporaries.

Terminology

- Moving a value from memory to a register is a **load**.
- Moving a value from a register to a named memory location is a **store**.
- Moving a value from a register to a temporary on the stack is a **spill**.
- Why might we have to spill?
    - We may not have enough registers.
    - We may need a dedicated register for an operation.
    - We need to call a function and the register is caller-saved.

Intel Instruction Set

- ARM and MIPS are **load-store architectures**.
  - Dedicated instructions are used to move values between memory and registers.
  - All other instructions require register operands.
- Intel is a **register-memory architecture**.
  - Any instruction can specify a memory reference as either a source or destination operand.
  - However, at most one operand can be a memory reference.
- Intel has a **two-operand** instruction set.
  - One of the operands is both a source and destination.

A Simple Algorithm

- Let's develop a simple algorithm to generate code using registers to hold intermediate results.
- Consider a binary expression *left op right*:
  1. Generate code for the left and right operands.
  2. If *left* is not in a register, then allocate a register and load it into that register.
  3. Perform the operation by emitting *opcode right*, *left* where *opcode* is the appropriate instruction.
  4. If *right* is a register, then that register is now available.
- Assume that code is generated during a left-to-right parse of the program.

Example 1

- Assume the registers are `%eax`, `%ecx`, `%edx`, etc.
- Assume that all variables are 32-bit integers and are global variables so they can be referred to by name.
- Generate code for `a * b + c * d`.

```
movl    a, %eax          # load: %eax allocated
imull   b, %eax
movl    c, %ecx          # load: %ecx allocated
imull   d, %ecx
addl    %ecx, %eax       # %ecx deallocated
```

- Here "`imul`" means integer, or signed, multiplication.

Example 2

- Generate code for `a + b * c – (a + b + c)`.

```
movl    b, %eax         # load: %eax allocated
imull   c, %eax
movl    a, %ecx         # load: %ecx allocated
addl    %eax, %ecx      # %eax deallocated
movl    a, %eax         # load: %eax allocated
addl    b, %eax
addl    c, %eax
subl    %eax, %ecx      # %eax deallocated
```

Order is important

- Note that in evaluating `a + b * c`, we first evaluated `b * c` before applying our algorithm to the addition.

Example 3

- Generate code for `a * b - (c * d + (a – b * c))`.

```
movl    a, %eax        # load: %eax allocated
imull   b, %eax
movl    c, %ecx        # load: %ecx allocated
imull   d, %ecx
movl    b, %edx        # load: %edx allocated
imull   c, %edx
movl    a, %ebx        # load: %ebx allocated
subl    %edx, %ebx     # %edx deallocated
addl    %ebx, %ecx     # %ebx deallocated
subl    %ecx, %eax     # %ecx deallocated
```

- This example required four registers! Would you believe we can do it using only two registers?

Example 3: Optimal Code

- Generate code for `a * b - (c * d + (a – b * c))`.

```
movl   b, %eax       # load: %eax allocated
imull  c, %eax
movl   a, %ecx       # load: %ecx allocated
subl   %eax, %ecx    # %eax deallocated
movl   c, %eax       # load: %eax allocated
imull  d, %eax
addl   %ecx, %eax    # %ecx deallocated
movl   a, %ecx       # load: %ecx allocated
imull  b, %ecx
subl   %eax, %ecx    # %eax deallocated
```

- We used only two registers; however, we generated code in an order different from parsing to do so.