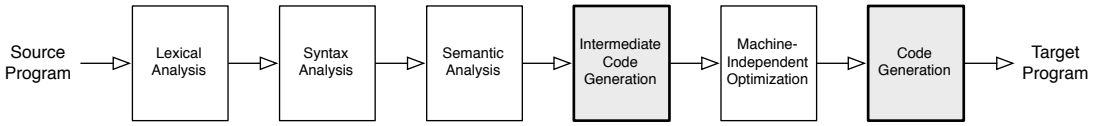
CS 301 Spring 2016
Meetings March 28

Code Generation

Plan



These exercises and readings serve as a quick intro to code generation, a quick review of instruction set architectures, call stacks, and some specifics of x86-64 assembly language. (a.k.a. CS 240, upgraded to 64 bits)

Readings

- Dragon 7 – 7.2 (*Note:* stack addresses and offsets are done "upside down" in Dragon, but this should not have significant effect on this section.)
- Dragon 8.1 – 8.3
- Back End resources on web site

Exercises

1. Here is a small IC program:

```
class A {
   int x;
   int y;
   void m(int w, int z) {
      int r = w / z + this.x * this.x;
      {
         int k = w + 1;
      }
   }
   void main(string[] args) { }
}

class B extends A {
   int z;
   void n(string s) { }
   void m(int g, int h) { }
}

class C extends B {
   void m(int g, int h) { }
   void p() { }
}

class D extends A {
   string k;
   void p() { }
}
```

(a) (Review) Show the object and method vector (or vtable) layouts for the four classes shown here. Include the object offsets for fields and dispatch vector indexes for methods. Assume all data is stored as 64-bit values.

(b) Lay out the stack frame for method `A.m`, assuming that the TAC for this method is:

```
t1 = w / z
t2 = this.x
t3 = this.x
t3 = t2 * t3
t0 = t1 + t2
r = t0
t4 = w + 1
k = t4
```

Include in your diagram:

- The stack pointer (%rsp).
- (Optional) The frame pointer (%rbp), ("Control Link" in Dragon)
- Locations of parameters, assuming all are stored in the stack. Including `this`, which is treated as an implicit first parameter to every method.
- Local variables and TAC temporary variables, assuming all are stored in the stack.
- Addresses of all stack frame contents given as offsets from `%rbp` (if using a frame/base pointer) or `%rsp` if not.

If you are following Dragon (*e.g.*, Figure 7.5), do not keep an "Access link". Your only "Saved machine status" is the return address.

2. The `gcc` compiler (when configured to pass all arguments on the stack) has generated the following x86-64 code for a method in an object-oriented language:

```
_f:
  pushq %rbp
  movq %rsp, %rbp
  subq $8, %rsp
  movq 32(%rbp), %rax
  addq 24(%rbp), %rax
  movq %rax, -8(%rbp)
.L2:
  movq 16(%rbp), %rax
  movq 16(%rbp), %rdx
  movq 8(%rax), %rcx
  movq 16(%rdx), %rax
  cmpq %rax, %rcx
  jge .L5
  movq 16(%rbp), %rcx
  movq 16(%rbp), %rdx
  movq -8(%rbp), %rax
  addq 8(%rdx), %rax
  movq %rax, 8(%rcx)
  jmp .L2
.L5:
  movq 16(%rbp), %rax
  movq (%rax), %rdx
  addq $64, %rdx
  pushq %rax
  movq (%rdx), %rax
  call *%rax
  addq $8, %rsp
  movq 16(%rbp), %rax
  movq %rbp, %rsp
  popq %rbp
  ret
```

   (a) Assuming that the stack grows downwards, draw the memory layout during the execution of this method. The layout must contain all of the pieces of memory that the execution of this method accesses.

   (b) Show a possible input program from which this code may have been generated.

   (c) Would it be safe to remove the instruction "`addq $8, %rsp`" at the end of the program? Explain.

   (d) Convert this x86 code to eliminate the use of `%rbp`. Use offsets relative to `%rsp` instead.

3. **Think this through, but do not write the whole solution.** The translation from TAC to assembly code can be expressed as a function $CG[\ \ ]$ in much the same was as the TAC translation function $T[\ ]$ was described last week. This function converts one TAC instruction into one or more assembly instructions that implement the TAC operation. In order to properly generate the assembly code, the code generation function will take an "augmented TAC" in which each variable name has been annotated with its offset from the frame pointer, each field access has been annotated with the offset at which the field is located in the object, and so on. (You may wish to think about how your TAC instruction objects will access this information when it is needed in your compiler.) Given these annotations, here is the translation of the TAC add instruction:

$$CG[\ \mathtt{x^a = s^b + t^c}\ ] \quad \equiv \quad \begin{array}{l}\texttt{movq b(\%rbp), \%rax}\\ \texttt{addq c(\%rbp), \%rax}\\ \texttt{movq \%rax, a(\%rbp)}\end{array}$$

Of course, we would translate differently if one or more operands to `+` were constants instead of variables:

$$CG[\ \mathtt{x^a = J + t^c}\ ] \quad \equiv \quad \begin{array}{l}\texttt{movl \$J, \%rax}\\ \texttt{addl c(\%rbp), \%rax}\\ \texttt{movl \%rax, a(\%rbp)}\end{array}$$

$$CG[\ \mathtt{x^a = J + K}\ ] \quad \equiv \quad \begin{array}{l}\texttt{movl \$J, \%rax}\\ \texttt{addl \$K, \%rax}\\ \texttt{movl \%rax, a(\%rbp)}\end{array}$$

Converting a sequence of TAC instructions $s_1; \ldots; s_n$ is defined in the obvious way: $CG[\ s_1; \ldots; s_n\ ] \equiv CG[\ s_1\ ]; \ldots; CG[\ s_n\ ]$.

(a) Write the compilation function for the following cases. Be sure to use proper x86 instructions and addressing modes!

- $CG[\ \mathtt{s^a = t^b}\ ]$
- $CG[\ \mathtt{x^a = s^b / t^c}\ ]$

  Hint: compile

```
int main() {
  long x,y,z;
  x = y/z;
}
```

  with `gcc -m64 -S` and look at the resulting assembly code.
- $CG[\ \mathtt{x^a = (y^b.f)^c}\ ]$, where c is the object offset of field `f`.
- $CG[\ \mathtt{x^a[y^c] = z^b}\ ]$

(b) Augment the TAC for `A.m` with the offset information and show the compilation of those TAC instructions using $CG[\ \ ]$. (Note: This is tedious to do by hand — just do the first couple of instructions so that you can think about and answer the next part of the question.)

(c) There will be some obvious inefficiencies in your translation. Identify some of them and discuss how your code generator might avoid them.

4. Get started on the IC compiler Back End by designing the `tac` package. Please write up the basic design of your TAC package, with enough detail to identify the following items:

- The TAC instruction set for your compiler (The TAC description we have used previously is a good start, but there will be some additions and subtractions).
- The top-level design of the `tac` package:
  - What are the main classes, what will they do?
  - How do you represent a TAC instruction?
  - How do you represent TAC operands?