

CSE 401/M501 – Compilers

Code Shape I – Basic Constructs

Hal Perkins

Autumn 2025

Administrivia

- Midterm results out now; exam and solutions posted on the web.
 - Lots of great work! Congratulations!!
 - Regrade requests open at 5pm today, but please review your answers and sample solutions first
 - If you did not do as well as you'd hoped, take a step back and figure out why. Strong work the rest of the quarter will make a difference. Remember that course staff is here to help.
- Parser/AST feedback also out now – if any questions, please send email to cse401-staff and cc your partner. **Must** fix problems before doing semantics visitor(s)
- Static semantics/type checking due a week from Thursday
 - Symbol table & type ADT check-in during sections this Thursday
- CSE M 501 extra work info posted; please send mail to instructor at cse401-staff with your group's proposed plan
- Regrades for old assignments (hw1, hw2, scanner, ...) will close soon
 - should be handled by now

Agenda

- Mapping source code to x86-64
 - Mapping for other common architectures is similar
- This lecture: basic statements and expressions
 - We'll go quickly since this is review for many, fast orientation for others, and pretty straightforward
- Next: Object representation, method calls, and dynamic dispatch
- Later: specific details for project

Note: These slides include more than is specifically needed for the course project

Review: Variables

- For us, all data will be either:
 - In a stack frame (method local variables)
 - In an object (instance variables)
- Local variables accessed via `%rbp`
`movq -16(%rbp),%rax`
- Object instance variables accessed via an offset from an object address in a register
 - Details later

Conventions for Examples

- Examples show code snippets in isolation
 - Much the way we'll generate code for different parts of the AST in a compiler visitor pass
 - Different perspective from the 351 holistic view
- Register `%rax` used here as a generic example
 - Rename as needed for more complex code using multiple registers
- 64-bit data used everywhere
- A few peephole optimizations shown to suggest what's possible
 - Some might be fairly easy to do in our compiler project

What we're skipping for now

- Real code generator needs to deal with many other things like:
 - Which registers are busy at which point in the program
 - Which registers to spill into memory when a new register is needed and no free ones are available
 - Dealing with different sizes of data
 - Exploiting the full instruction set

Code Generation for Constants

- Source

17

- x86-64

`movq $17,%rax`

– Idea: realize constant value in a register

- Optimization: if constant is 0

`xorq %rax,%rax`

(but some processors do better with `movq $0,%rax` – and this has changed over time; also can be considerations about whether condition codes are set or not)

Assignment Statement

- Source

`var = exp;`

- x86-64

`<code to evaluate exp into, say, %rax>`

`movq %rax, offsetvar(%rbp)`

Unary Minus

- Source
 - exp
- x86-64
 - <code evaluating exp into %rax>
 - negq %rax
- Optimization
 - Collapse -(-exp) to exp
- Unary plus is a no-op

Binary +

- Source

$\text{exp}_1 + \text{exp}_2$

- x86-64

<code evaluating exp_1 into %rax>

<code evaluating exp_2 into %rdx>

addq %rdx,%rax

Binary +

- Some optimizations

- If exp_2 is a simple variable or constant, don't need to load it into another register first on x86-64. Instead:

`addq exp2,%rax`

- Change $\text{exp}_1 + (-\text{exp}_2)$ into $\text{exp}_1 - \text{exp}_2$

- If exp_2 is 1

`incq %rax`

- Somewhat surprising: whether this is better than `addq $1,%rax` depends on processor implementation and has changed over time

Binary -, *

- Same as +
 - Use `subq` for `-` (but *not* commutative!)
 - Use `imulq` for `*`
- Some optimizations
 - Use left shift to multiply by powers of 2
 - If your multiplier is slow or you've got free scalar units and the multiplier is busy or you don't want to power up the multiplier circuit, you can do $10 * x = (x \ll 3) + (x \ll 1)$
 - But might be slower depending on microarchitecture
 - Use `x+x` or shift instead of $2 * x$, etc. (often faster)
 - Can use `leaq (%rax,%rax,4),%rax` to compute $5 * x$, then `addq %rax,%rax` to get $10 * x$, etc. etc., but `leaq` doesn't set condition codes
 - Use `decq` for `x-1` (but check: `subq $1` might be faster)

Signed Integer Division

- Ghastly on x86-64
 - Only works for 128-bit int divided by 64-bit int
 - (similar instructions for 64-bit divided by 32-bit for 32-bit ints)
 - Requires use of specific registers
 - Very slow
- Source
$$\text{exp}_1 / \text{exp}_2$$
- x86-64
 - <code evaluating exp_1 into %rax **ONLY**>
 - <code evaluating exp_2 into, say, %rbx>
 - cqto # extend to %rdx:%rax, clobbers %rdx
 - idivq %rbx # quotient in %rax, remainder in %rdx

Control Flow

- Basic idea: decompose higher level operation into conditional and unconditional gotos
- In the following, j_{false} is used to mean jump when a condition is false
 - No such instruction on x86-64
 - Will have to realize with appropriate sequence of instructions to set condition codes followed by conditional jumps
 - Normally don't need to actually generate the value "true" or "false" in a register
 - But this can be a useful shortcut hack for our project

While

- Source

while (cond) stmt

- x86-64

test: <code evaluating cond>

 j_{false} done

 <code for stmt>

 jmp test

done:

- Note: In generated asm code we will need to have unique labels for each loop, conditional statement, etc.

A little computer architecture – Instruction execution

- Actual execution of an instruction has multiple steps/phases inside a processor. Fairly typical steps for a simple processor:
 - IF: instruction fetch. Load instruction from memory/cache into internal processor register(s)
 - ID: instruction decode / read operand registers
 - EX: execute or calculate memory addresses
 - MEM: access memory (not all instructions)
 - WB: write back – store result in registers, etc.
- (x86-64 is waaaaay more complex, but basic ideas are the same)
- See 351 textbook, sec. 4.4, 4.5, etc. for more details

Pipelining (on 1 slide, oversimplified)

- If instructions are independent, we can execute them on an assembly line – start processing the next one while previous one is in some later stage. Ideally we could overlap like this:

1.	IF	ID	EX	MEM	WB			
2.		IF	ID	EX	MEM	WB		
3.			IF	ID	EX	MEM	WB	
4.				IF	ID	EX	MEM	WB
5.					IF	ID	...	
- Modern processors have multiple function units and buffers to support this

Pipelining bottlenecks

- This strategy works great – *if* the instructions are independent. Things that cause problems:
 - Output of one instruction needed for next one: next one can't proceed until data is available from earlier one
 - Jumps: If there's a conditional jump, the processor has to either stall the pipeline until we decide whether to jump, or make a guess and be prepared to “undo” if it guesses wrong
- Processors have lots of hardware to try to “guess right” and avoid delays caused by these dependencies, but ...
- Compilers can help the processor by generating code to minimize these issues

Optimization for While

- Put the test at the end:

```
        jmp    test
loop:   <code for stmt>
test:   <code evaluating cond>
        jtrue  loop
```

- Why bother?
 - Pulls one instruction (jmp) out of the loop
 - Avoids a potential pipeline stall on jmp on each iteration
 - Although modern processors will often predict control flow and avoid the stall – x86-64 does this particularly well
- Easy to do from AST or other IR; not so easy if generating code on the fly (e.g., recursive descent 1-pass compiler)

Do-While

- Source

do stmt while(cond)

- x86-64

loop: <code for stmt>

<code evaluating cond>

j_{true} loop

If

- Source
if (cond) stmt
- x86-64
 <code evaluating cond>
 j_{false} skip
 <code for stmt>
skip:

If-Else

- Source
if (cond) stmt₁ else stmt₂
- x86-64
 <code evaluating cond>
 j_{false} else
 <code for stmt₁>
 jmp done
else: <code for stmt₂>
done:

Jump Chaining

- Observation: naïve implementation can produce jumps to jumps (if ... elseif ... else; or nested loops and conditionals, ...)
- Optimization: if a jump has as its target an unconditional jump, change the target of the first jump to the target of the second
 - Repeat until no further changes
 - Often done in peephole optimization pass after initial code generation

Boolean Expressions

- What do we do with this?

$x > y$

- It is an expression that evaluates to true or false
 - Could generate the value (1|0 or whatever the local convention is)
 - But normally we don't want/need the value – we're only trying to decide whether to jump
 - (Although for our project we might simplify and always produce the value)

Code for $\text{exp1} > \text{exp2}$

- Basic idea: Generated code depends on context:
 - What is the jump target?
 - Jump if the condition is true or if false?
- Example: evaluate $\text{exp1} > \text{exp2}$, jump on false, target if jump taken is L123

<evaluate exp1 into %rax>

<evaluate exp2 into %rdx>

cmpq %rdx,%rax # dst-src = exp1-exp2

jng L123

Boolean Operators: !

- Source
 ! exp
- Context: evaluate exp and jump to L123 if false (or true)
- To compile !, just reverse the sense of the test: evaluate exp and jump to L123 if true (or false)

Boolean Operators: && and ||

- In C/C++/Java/C#/many others, these are short-circuit operators
 - Right operand is evaluated only if needed
- Basically, generate the if statements that jump appropriately and only evaluate operands when needed

Example: Code for &&

- Source

if (exp_1 && exp_2) stmt

- x86-64

<code for exp_1 >

j_{false} skip

<code for exp_2 >

j_{false} skip

<code for stmt>

skip:

Example: Code for `||`

- Source

if ($\text{exp}_1 \ || \ \text{exp}_2$) stmt

- x86-64

<code for exp_1 >

j_{true} doit

<code for exp_2 >

j_{false} skip

doit: <code for stmt>

skip:

Realizing Boolean Values

- If a boolean value needs to be stored in a variable or method call parameter, generate code needed to actually produce it
- Typical representations: 0 for false, +1 or -1 for true
 - C specifies 0 and 1 if stored; we'll use that
 - Best choice can depend on machine instructions & language; normally some convention is picked during the primeval history of the architecture

Boolean Values: Example

- Source

var = bexp;

- x86-64

<code for bexp>

j_{false} genFalse

movq \$1,%rax

jmp store

genFalse:

movq \$0,%rax # or xorq

store:

movq %rax,offset_{var}(%rbp) # generated by asg stmt

Better, If Enough Registers

- Source

var = bexp;

- x86-64

xorq %rax,%rax # or movq \$0,%rax

<code for bexp>

j_{false} store

incq %rax # or movq \$1,%rax

store:

movq %rax,offset_{var}(%rbp) # generated by asg

- Better: use movecc instruction to avoid conditional jump
- Can also use conditional move instruction for sequences like
x = y < z ? y : z

Better yet: setcc

- Source

var = x < y;

- x86-64

```
movq    offset_x(%rbp),%rax    # load x
cmpq    offset_y(%rbp),%rax    # compare to y
setl     %al                   # set low byte %rax to 0/1
movzbq   %al,%rax              # zero-extend to 64 bits
movq     %rax,offset_var(%rbp) # gen. by asg stmt
```

Other Control Flow: switch

- Naïve: generate a chain of nested if-else if statements
- Better: switch statement is intended to allow $O(1)$ selection, provided the set of switch values is reasonably compact
- Idea: create a 1-D array of jumps or labels and use the switch expression to select the right one
 - Need to generate the equivalent of an if statement to ensure that expression value is within bounds

Switch

- Source

```
switch (exp) {  
    case 0: stmts0;  
    case 1: stmts1;  
    case 2: stmts2;  
}
```

“break” is an unconditional
jump to the end of switch

- x86-64:

```
<put exp in %rax>  
“if (%rax < 0 || %rax > 2)  
    jmp defaultLabel”  
movq    swtab(,%rax,8),%rax  
jmp     *%rax  
    .data  
swtab:  
    .quad L0  
    .quad L1  
    .quad L2  
    .text  
L0:    <stmts0>  
L1:    <stmts1>  
L2:    <stmts2>
```

Arrays

- Several variations
- C/C++/Java
 - 0-origin: an array with n elements contains variables $a[0] \dots a[n-1]$
 - 1 dimension (Java); 1 or more dimensions using row major order (C/C++)
- Key step is evaluate subscript expression, then calculate the location of the corresponding array element

0-Origin 1-D Integer Arrays

- Source

$\text{exp}_1[\text{exp}_2]$

- x86-64

<evaluate exp_1 (array address) into %rax>

<evaluate exp_2 into %rdx>

address is (%rax,%rdx,8) # if 8 byte elements

- For our project, we'll likely add $\text{exp}_1 + 8 * \text{exp}_2$ to get the address of (ptr to) the array element in a register. Use either shift/addq or leaq. Maybe simpler that way....

2-D Arrays

- Subscripts start with 0 (default)
- C/C++, etc. use row-major order
 - E.g., an array with 3 rows and 2 columns is stored in sequence: $a(0,0)$, $a(0,1)$, $a(1,0)$, $a(1,1)$, $a(2,0)$, $a(2,1)$
- Fortran uses column-major order
 - Exercises: What is the layout? How do you calculate location of $a[i][j]$? What happens when you pass array references between Fortran and C/C++ code?
- Java does not have “real” 2-D arrays. A Java 2-D array is a pointer to a list of pointers to the rows
 - And rows may have different lengths (ragged arrays)

$a[i][j]$ in C/C++/etc.

- If a is a “real” 0-origin, 2-D array, to find $a[i][j]$, we need to know:
 - Values of i and j
 - How many columns (but not rows!) the array has
- Location of $a[i][j]$ is:
 - Location of a + $(i * (\text{\#of columns}) + j) * \text{sizeof(elt)}$
- Can factor to pull out allocation-time constant part and evaluate that once – no recalculating at runtime; only calculate part depending on i, j
 - Details in most compiler books

Coming Attractions

- Code Generation for Objects
 - Representation
 - Method calls
 - Inheritance and overriding
- Strategies for implementing code generators
- Code improvement – “optimization”