Lecture 21

CS 131: COMPILERS

Announcements

- HW5: Oat v. 2.0
  - records, function pointers, type checking, array-bounds checks, etc.
  - typechecker & safety
  - Due: Friday, December 13th
- HW6: Analysis & Optimizations
  - Alias analysis, constant propagation, dead code elimination, register allocation
  - Available soon
  - Due: December 30th

Compiling lambda calculus to straight-line code.

Representing evaluation environments at runtime.

CLOSURE CONVERSION REVISITED

Compiling First-class Functions

- To implement first-class functions on a processor, there are two problems:
  - First: we must implement substitution of free variables
  - Second: we must separate 'code' from 'data'

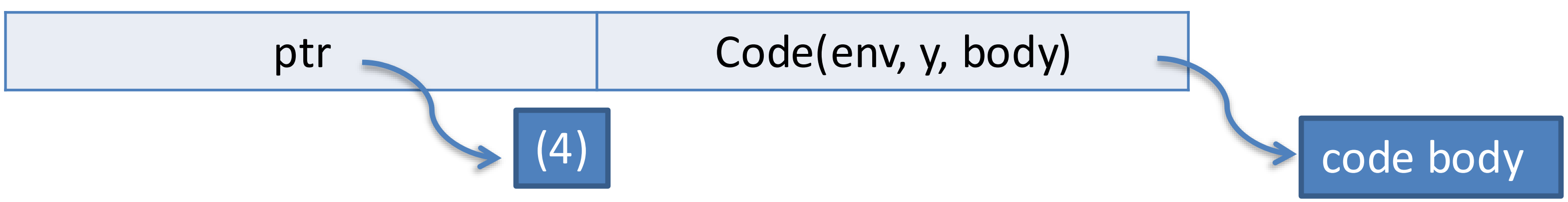
- Reify the substitution:
  - Move substitution from the meta language to the object language by making the data structure & lookup operation explicit
  - The environment-based interpreter is one step in this direction
- Closure Conversion:
  - Eliminates free variables by packaging up the needed environment in the data structure.
- Hoisting:
  - Separates code from data, pulling closed code to the top level.

Example of closure creation

- Recall the "add" function:
  let add = fun x -> fun y -> x + y

- Consider the inner function: fun y -> x + y

- When run the function application: add 4
  the program builds a closure and returns it.
  - The closure is a pair of the environment and a code pointer.



- The code pointer takes a pair of parameters: env and y
  - The function code is (essentially):
    fun (env, y) -> let x = nth env 0 in x + y

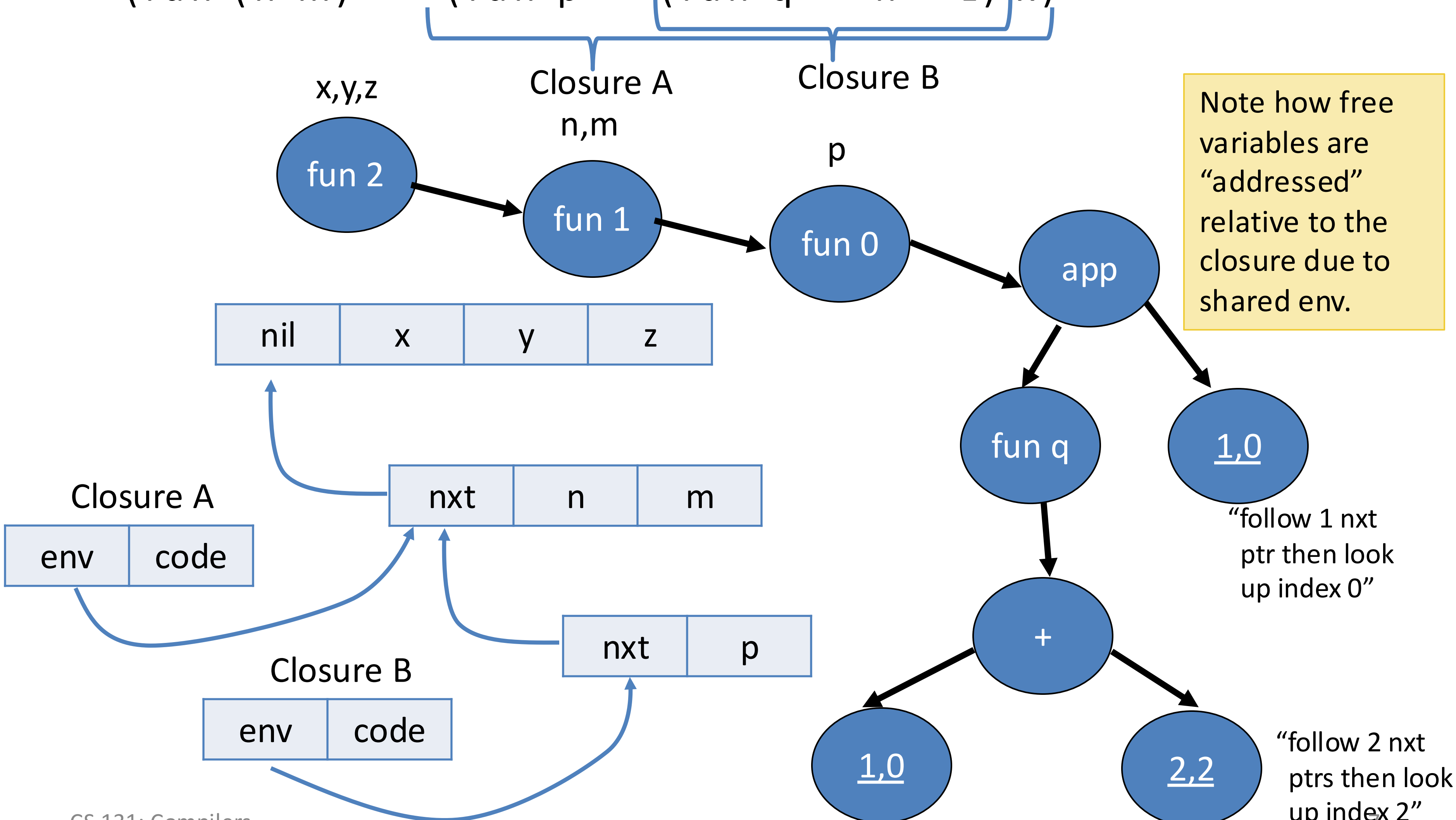
Representing Closures

- As we saw, the simple closure conversion algorithm doesn't generate very efficient code.
  - It stores all the values for variables in the environment, even if they aren't needed by the function body.
  - It copies the environment values each time a nested closure is created.
  - It uses a linked-list datastructure for tuples.

- There are many options:
  - Store only the values for free variables in the body of the closure.
  - Share subcomponents of the environment to avoid copying
  - Use vectors or arrays rather than linked structures

Array-based Closures with N-ary Functions

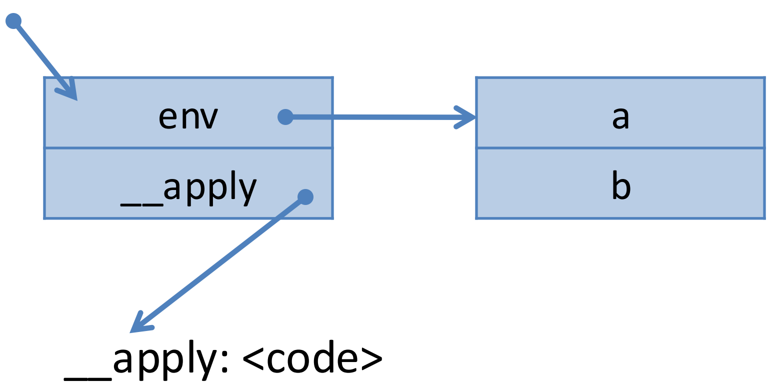
```
(fun (x y z) ->
   (fun (n m) -> (fun p -> (fun q -> n + z) x)
```



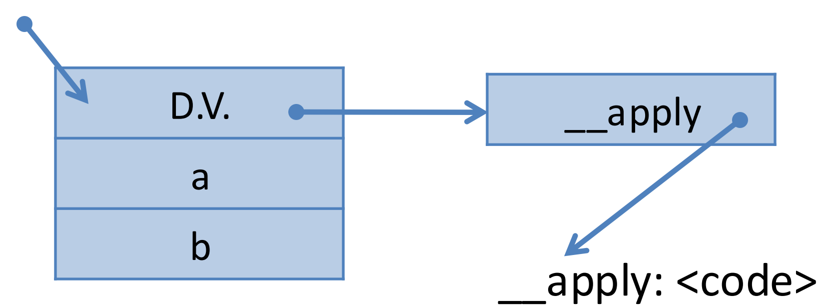
Observe: Closure ≈ Single-method Object

- Free variables ≈ Fields
- Environment pointer ≈ "this" parameter
- Closure for function: ≈ Instance of this class:

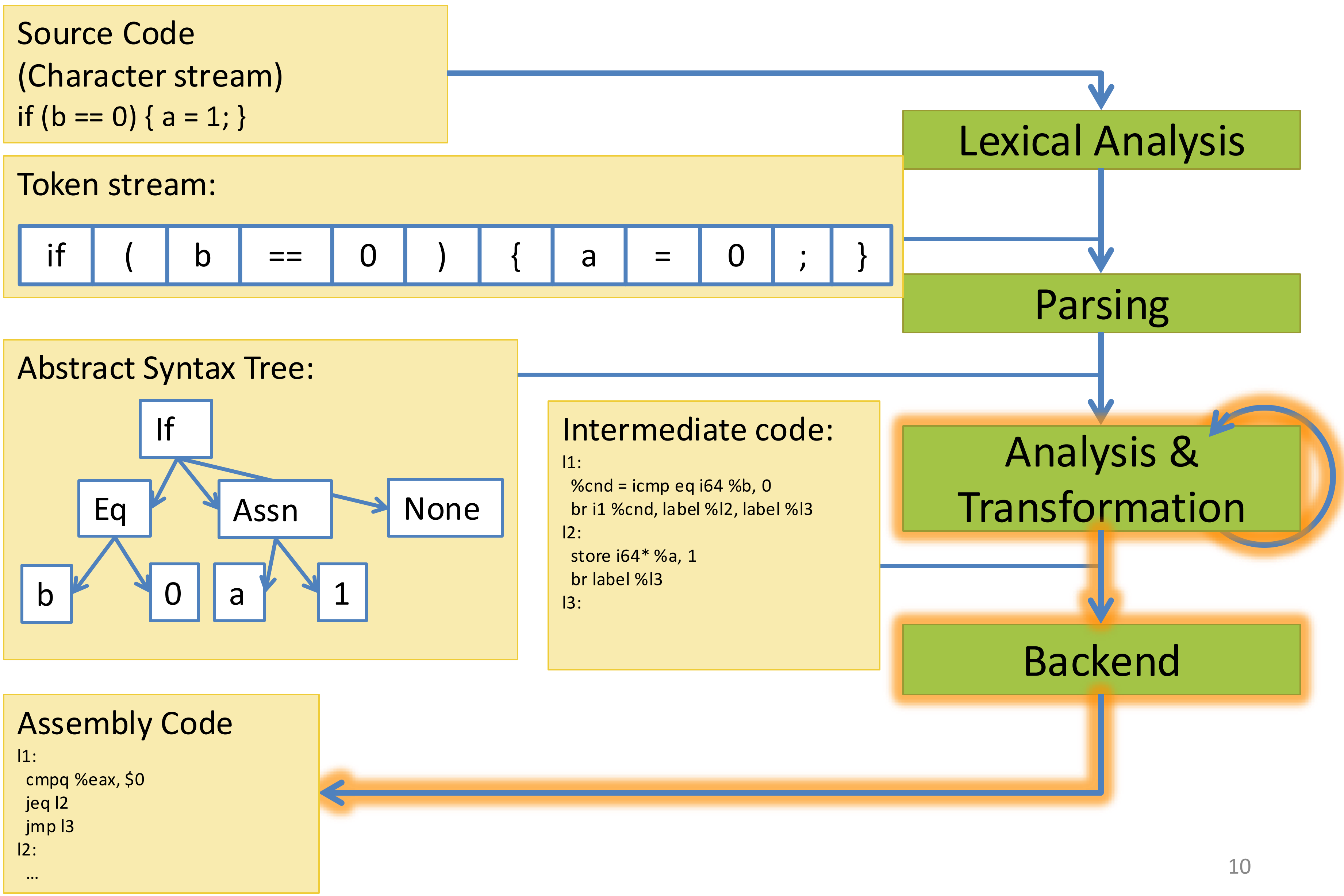
```
fun (x,y) ->
    x + y + a + b
```




```
class C {
  int a, b;
  int apply(x,y) {
    x + y + a + b
  }
}
```



Optimizations



Why optimize?

OPTIMIZATIONS, GENERALLY

Optimizations

- The code generated by our OAT compiler so far is pretty inefficient.
  - Lots of redundant moves.
  - Lots of unnecessary arithmetic instructions.

- Consider this OAT program:

```
int foo(int w) {
  var x = 3 + 5;
  var y = x * w;
  var z = y - 0;
  return z * 4;
}
```

frontend.ml

```
define i64 @foo(i64 %_w1) {
  %_w2 = alloca i64
  %_x5 = alloca i64
  %_y10 = alloca i64
  %_z14 = alloca i64
  store i64 %_w1, i64* %_w2
  %_bop4 = add i64 3, 5
  store i64 %_bop4, i64* %_x5
  %_x7 = load i64, i64* %_x5
  %_w8 = load i64, i64* %_w2
  %_bop9 = mul i64 %_x7, %_w8
  store i64 %_bop9, i64* %_y10
  %_y12 = load i64, i64* %_y10
  %_bop13 = sub i64 %_y12, 0
  store i64 %_bop13, i64* %_z14
  %_z16 = load i64, i64* %_z14
  %_bop17 = mul i64 %_z16, 4
  ret i64 %_bop17
}
```

- opt-example.c, opt-example.oat

Unoptimized vs. Optimized Output

```
define i64 @foo(i64 %_w1) {
  %_w2 = alloca i64
  %_x5 = alloca i64
  %_y10 = alloca i64
  %_z14 = alloca i64
  store i64 %_w1, i64* %_w2
  %_bop4 = add i64 3, 5
  store i64 %_bop4, i64* %_x5
  %_x7 = load i64, i64* %_x5
  %_w8 = load i64, i64* %_w2
  %_bop9 = mul i64 %_x7, %_w8
  store i64 %_bop9, i64* %_y10
  %_y12 = load i64, i64* %_y10
  %_bop13 = sub i64 %_y12, 0
  store i64 %_bop13, i64* %_z14
  %_z16 = load i64, i64* %_z14
  %_bop17 = mul i64 %_z16, 4
  ret i64 %_bop17
}
```

backend.ml

```
        .text
        .globl  _foo
_foo:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $136, %rsp
        movq    %rdi, %rax
        movq    %rax, -8(%rbp)
        pushq   $0
        movq    %rsp, -16(%rbp)
        pushq   $0
        movq    %rsp, -24(%rbp)
        pushq   $0
        movq    %rsp, -32(%rbp)
        pushq   $0
        movq    %rsp, -40(%rbp)
        movq    -8(%rbp), %rcx
        movq    -16(%rbp), %rax
        movq    %rcx, (%rax)
        movq    $3, %rax
        movq    $5, %rcx
        addq    %rcx, %rax
        movq    %rax, -56(%rbp)
        movq    -56(%rbp), %rcx
        movq    -24(%rbp), %rax
        movq    %rcx, (%rax)
        movq    -24(%rbp), %rax
        movq    (%rax), %rcx
        movq    %rcx, -72(%rbp)
        movq    -16(%rbp), %rax
        movq    (%rax), %rcx
        movq    %rcx, -80(%rbp)
        movq    -72(%rbp), %rax
        movq    -80(%rbp), %rcx
        imulq   %rcx, %rax
        movq    %rax, -88(%rbp)
        movq    -88(%rbp), %rcx
        movq    -32(%rbp), %rax
        movq    %rcx, (%rax)
        movq    -32(%rbp), %rax
        movq    (%rax), %rcx
        movq    %rcx, -104(%rbp)
        movq    -104(%rbp), %rax
        movq    $0, %rcx
        subq    %rcx, %rax
        movq    %rax, -112(%rbp)
        movq    -112(%rbp), %rcx
        movq    -40(%rbp), %rax
        movq    %rcx, (%rax)
        movq    -40(%rbp), %rax
        movq    (%rax), %rcx
        movq    %rcx, -128(%rbp)
        movq    -128(%rbp), %rax
        movq    $4, %rcx
        imulq   %rcx, %rax
        movq    %rax, -136(%rbp)
        movq    -136(%rbp), %rax
        movq    %rbp, %rsp
        popq    %rbp
        retq
```

???

Optimized code:

```
_foo:
        pushq   %rbp
        movq    %rsp, %rbp
        movq    %rdi, %rax
        shlq    $5, %rax
        popq    %rbp
        retq
```

- Code above generated by clang –O3
- Function foo may be inlined by the compiler, so it can be implemented by just one instruction!

Why do we need optimizations?

- To help programmers…
  - They write modular, clean, high-level programs
  - Compiler generates efficient, high-performance assembly

- Programmers don't write optimal code
- High-level languages make avoiding redundant computation inconvenient or impossible
  - e.g. A[i][j] = A[i][j] + 1

In Oat/ Java it's not possible for the programmer to manually express the sharing of the two computations of A[i][j] because there is no concept of "interior pointer".

- Architectural independence
  - Optimal code depends on features not expressed to the programmer
  - Modern architectures *assume* optimization

- Different kinds of optimizations:
  - Time: improve execution speed
  - Space: reduce amount of memory needed
  - Power: lower power consumption (e.g. to extend battery life)

Some caveats

- Optimization are code transformations:
  - They can be applied at any stage of the compiler
  - They must be *sound* – they shouldn’t change the meaning of the program.
- In general, optimizations require some ***program analysis***:
  - To determine if the transformation really is safe
  - To determine whether the transformation is cost effective

> ***(static) program analysis:*** the process of (soundly) approximating the dynamic behavior of a program at compile time, usually by representing some facts about the state of the computation at each program point.

- This course: most common and valuable performance optimizations
  - See Muchnick (optional text) for ~10 chapters about optimization

When to apply optimization

High level
- AST
- IR

Mid level
- Canonical IR

Low level
- Abstract assembly
- Assembly

- Inlining
- Function specialization
- Constant folding
- Constant propagation
- Value numbering
- Dead code elimination
- Loop-invariant code motion
- Common sub-expression elimination
- Strength Reduction
- Constant folding & propagation
- Branch prediction / optimization
- Register allocation
- Loop unrolling
- Instruction Selection
- Cache optimization

Where to Optimize?

- Usual goal: improve time performance
- Problem: many optimizations trade space for time
- Example: *Loop unrolling*
  - Idea: rewrite a loop like:
    ```
    for(int i=0; i<100; i=i+1) {
      s = s + a[i];
    }
    ```
  - Into a loop like:
    ```
    for(int i=0; i<99; i=i+2){
      s = s + a[i];
      s = s + a[i+1];
    }
    ```
- Tradeoffs:
  - Increasing code space slows down whole program a tiny bit (extra instructions to manage) but speeds up the loop a lot
  - For frequently executed code with long loops: generally a win
  - Interacts with instruction cache and branch prediction hardware
- Complex optimizations may never pay off!

Writing Fast Programs In Practice

- Pick the right algorithms and data structures.
  - These have a much bigger impact on performance than compiler optimizations.
  - Reduce # of operations
  - Reduce memory accesses
  - Minimize indirection – it breaks working-set coherence
- *Then* turn on compiler optimizations
- Profile to determine program hot spots
- Evaluate whether the algorithm/data structure design works
- …if so: “tweak” the source code until the optimizer does “the right thing” to the machine code

Soundness

- Whether an optimization is ***sound*** (i.e., correct) depends on the programming language semantics.
  – Languages that provide weaker guarantees to the programmer permit more optimizations but have more ambiguity in their behavior.
  – *e.g.*, In C, writing to unallocated memory is undefined behavior, so the compiler can do anything if a program writes to an array out of bounds.
  – *e.g.*, In Java, tail-call optimization (which turns recursive function calls into loops) is not valid because of “stack inspection”.
- Example: ***loop-invariant code motion***
  – Idea: hoist invariant code out of a loop

```
while (b) {
  z = y/x;
  …                       // y, x not updated
}
```

```
z = y/x;
while (b) {
  …                       // y, x not updated
}
```

- Is this more efficient?
- Is this safe?

A high-level tour of a variety of optimizations.

BASIC OPTIMIZATIONS

Constant Folding

- Idea: If operands are known at compile type, perform the operation statically.

  int x = (2 + 3) * y ➔ int x = 5 * y

  b & false ➔ false

- Performed at every stage of optimization…
- Why?
  - Constant expressions can be created by translation or earlier optimizations

    Example: A[2] might be compiled to:

    MEM[MEM[A] + 2 * 4] ➔ MEM[MEM[A] + 8]

Constant Folding Conditionals

if (true) S → S

if (false) S → ;

if (true) S else S' → S

if (false) S else S' → S'

while (false) S → ;

if (2 > 3) S →
    if (false) S → ;

Algebraic Simplification

- More general form of constant folding
  - Take advantage of mathematically sound simplification rules

- ***Mathematical identities***:
  - a * 1 ➔ a　　　　　　a * 0 ➔ 0
  - a + 0 ➔ a　　　　　　a – 0 ➔ a
  - b | false ➔ b　　　　b & true ➔ b
- ***Reassociation & commutativity***:
  - (a + 1) + 2 ➔ a + (1 + 2) ➔ a + 3
  - (2 + a) + 4 ➔ (a + 2) + 4 ➔ a + (2 + 4) ➔ a + 6
- ***Strength reduction***: (replace expensive op with cheaper op)
  - a * 4 ➔ a << 2
  - a * 7 ➔ (a << 3) – a
  - a / 32767 ➔ (a >> 15) + (a >> 30)

- *Note 1:* must be careful with floating point (due to rounding) and integer arithmetic (due to overflow/underflow)
- *Note 2:* iteration of these optimizations is useful… how much?
- *Note 3:* must be sure that rewrites terminate:
  - commutativity apply like: (x + y) ➔ (y + x) ➔ (x + y) ➔ (y + x) ➔ …