Code Generation Recap

- Create a `cgen(e)` function that emits instructions based on the type of expression `e`
  - emit labels
  - enforce stack discipline
  - emit instructions

- Using **stack machine code generation** means you don’t have to worry about register allocation; you use a fixed number for all code!
Remarks on Code Generation

- How does a symbol table fit into such a cgen function?
  - e.g., let bindings introduce new symbols...
  - Also have fields...

- What about handling intermediate values?
  - Just how bad are pushes and pops?
  - Is there an easier way to handle temporary intermediate values?

<table>
<thead>
<tr>
<th>Activation Record</th>
<th>Intermediates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activation Record</td>
<td>Intermediates</td>
</tr>
</tbody>
</table>
NumTemps Calculation

We can compute how much space is required to evaluate an expression recursively.

This allows us to have a specific slot in memory for each temporary or intermediate value created along the way during execution.

This is a simple optimization, but (more importantly) a way to help you think about how much space a stack machine generator requires to evaluate particular expressions.
NumTemps Calculation

- Let $NT(e)$ be the number of temporaries needed to evaluate $e$

Example: $NT(e_1 + e_2)$

cgen(e1)
mov temp <- r1
cgen(e2)
add r1 <- r1 temp

- Needs at least as many temporaries as $NT(e_1)$
- Needs at least as many temporaries as $NT(e_2) + 1$

- Space used for temporaries in $e_1$ can be reused for temporaries in $e_2$
How Many Temporaries?

\[ NT(e_1 + e_2) = \max(NT(e_1), 1 + NT(e_2)) \]
\[ NT(e_1 - e_2) = \max(NT(e_1), 1 + NT(e_2)) \]
\[ NT(\text{if } e_1 = e_2 \text{ then } e_3 \text{ else } e_4) = \max(NT(e_1), 1 + NT(e_2), NT(e_3), NT(e_4)) \]
\[ NT(\text{id}(e_1, \ldots, e_n)) = \max(NT(e_1), \ldots, NT(e_n)) \]
\[ NT(\text{int}) = 0 \]
\[ NT(\text{id}) = 0 \]

```haskell
1   fib ( x : Int ) : Int { 
2     if x = 1 then 0 else if x = 2 then 1 else 
3       fib(x-1) + fib(x-2) 
4   }; 
```
Notes on the Activation Record

You can compute the total stack space required for evaluating a function $f(x_1, \ldots, x_n)$ requires an AR with $2 + n + NT(e)$ elements

- Return address
- Frame pointer
- $n$ arguments
- $NT(e)$ locations for intermediate results
Stack Frame Picture

\[ f(x_1, \ldots, x_n) = e \]

```
<table>
<thead>
<tr>
<th>SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temp NT(e)</td>
</tr>
<tr>
<td>\ldots</td>
</tr>
<tr>
<td>Temp 1</td>
</tr>
<tr>
<td>RA</td>
</tr>
<tr>
<td>x_n</td>
</tr>
<tr>
<td>\ldots</td>
</tr>
<tr>
<td>x_1</td>
</tr>
<tr>
<td>Old FP</td>
</tr>
<tr>
<td>FP</td>
</tr>
</tbody>
</table>
```

high addresses
Object Layout Wrap up

An object represents an *abstraction*—we want to be able to generate code that *acts the same* over all objects. For example, all objects have *fields*. If we pick a consistent object representation, then we can get fields out of objects regardless of the particular type the object is.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Class Type Tag</td>
</tr>
<tr>
<td>1</td>
<td>Object Size</td>
</tr>
<tr>
<td>2</td>
<td>Dispatch / Vtable Ptr</td>
</tr>
<tr>
<td>3</td>
<td>Attribute 1</td>
</tr>
<tr>
<td>4</td>
<td>Attribute 2</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Cool Object Layout

- **Class tag**
  - An integer that identifies the class of the object (Int=1, Bool=2, ...)

- **Object size**
  - Number of *words* occupied by the object

- **Dispatch pointer**
  - Address to a table of methods

- **Attributes**
  - Each attribute laid out in contiguous memory
Object Layout Example

```java
1 class A {
2     a : Int <- 0;
3     d : Int <- 1;
4     f () : Int { a <- a + d };  // override
5 }
6 class C inherits A {
7     c : Int <- 3;
8     h () : Int { a <- a * c };  // override
9 }
10 class B inherits A {
11     b : Int <- 2;
12     f () : Int { a };  -- override
13     g () : Int { a <- a - b };  // override
14 }
```
Attributes a and d are inherits by classes B and C

All methods in all classes refer to a

For A methods to work correctly in A, B, and C objects, attribute a must be in the same “place” in each object

- The same offset from the start of the object in memory!
- Facilitates polymorphism
Observation: Given a layout for class A, a layout for subclass B can be defined by extending the layout of A with additional slots for the additional attributes of B

(i.e., append new fields at the bottom)

This leaves the layout of A unchanged (and B is an extension)
class A {
    a: Int <- 0;
    d: Int <- 1;
    f(): Int {a <- a + d};
}

class C inherits A {
    c: Int <- 3;
    h(): Int {a <- a * c};
}

class B inherits A {
    b: Int <- 2;
    f(): Int {a}; -- override
    g(): Int {a <- a - b};
}
Subclasses

- The offset for an attribute is the same in a class and all of its subclasses

  - This choice allows any method for $A_1$ to be used on a subclass $A_2$

- Consider layout for $A_n \leq \ldots \leq A_3 \leq A_2 \leq A_1$

<table>
<thead>
<tr>
<th>Header</th>
<th>$A_1$ object</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1$ attrs</td>
<td></td>
</tr>
<tr>
<td>$A_2 - A_1$ attrs</td>
<td>$A_2$ object</td>
</tr>
<tr>
<td>$A_3 - A_2$ attrs</td>
<td>$A_3$ object</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Dynamic Dispatch

- `e.g()`
  - `g` refers to method in `B` if `e` is a `B`

- `e.f()`
  - `f` refers to method in `A` if `f` is a `A` or `C` (inherited in the case of `C`)
  - `f` refers to method in `B` for a `B` object

- The implementation of methods and dynamic dispatch strongly resembles the implementation of attributes!
Every class has a fixed set of methods (including inherited methods)

A dispatch table (or virtual function table or vtable) indexes these methods

- A vtable is an array of method entry points (array of function pointers)
- A method \( f \) lives at a fixed offset in the dispatch table for a class and all of its subclasses
Dispatch Table Example

```java
1 class A {
2   f () : Int {...} ...
3 }
4 class C inherits A {
5   f () : Int {...};
6   h () : Int {...};
7 }
8 class B inherits A {
9   f () : Int {...};
10  g () : Int {...};
11 }
```

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>f_A</td>
<td>f_B</td>
<td>f_</td>
</tr>
<tr>
<td>1</td>
<td>g</td>
<td></td>
<td>h</td>
</tr>
</tbody>
</table>

- The dispatch table for class A has only 1 method
- The tables for B and C extend the table for A with more methods
- Because methods can be overridden, the method for f is not the same in every class, but is always at the same offset (i.e., offset 0 here)
Using Dispatch Tables

- The dispatch pointer in an object of class $X$ points to the dispatch table for class $X$
  - i.e., all objects of class $X$ share one table

- Every method $f$ of class $X$ is assigned an offset $O_f$ in the dispatch table at compile time
  - (that’s you in PA5)
Dispatch Codegen

- `cgen(objexp . mname(arg1))`
- `push self`
- `push fp`
- `cgen(arg1)`
- `push r1 ; push arg1`
- `cgen(objexp)`
- `?`
- `push r1 ; “self” for callee`
- `ld temp <- r1[2] ; get vtable`
- `ld temp <- temp[X] ; X is offset of mname in vtables`
- `call temp`
- `pop fp`
- `pop self`
Much like the AST shows us structure of the program, it is also useful to think about where execution can flow at runtime. We can build a Control Flow Graph that represents the particular paths through the program that CPU control can flow.

A Control Flow Graph (CFG) is a directed graph whose nodes represent logically atomic sequences of instructions called basic blocks.
Control Flow Graphs

- **Basic blocks are nodes**
- **Edge** from block A to block B exists if the execution can (potentially) flow from the last instruction in A to the first instruction in B
  - e.g., the last instruction in A is a `jmp L_B`
  - e.g., if the execution can fall-through from block A to block B
CREATING AN AIM PROFILE:

HAVE FRIENDS?

NO

HAVE BOYFRIEND/GIRLFRIEND?

YES

A PROFILE TRIBUTE IS THE GREATEST POSSIBLE EXPRESSION OF LOVE.

NO

LINK TO YOUR LIVEJOURNAL

YES

ANGSTY ABOUT IT?

YES

YOU ARE

NO

INSIDE JOKES!

YES, AND WANT TO ALIENATE EVERYONE ELSE
The body of a method can be represented as a CFG

- One initial start node
- One or more terminal nodes with “return”
Constructing a CFG

1. `a <- 5;`
2. `b <- 6;`
3. `if a < b then`
   4. `a <- a + b;`
   5. `else`
   6. `a <- b;`
4. `fi`
7. `return t0`
class Main inherits IO {
    main(): Object {
        let a : Int,
        b : Int,
        c : Int in {
            a <- 1;
            b <- in_int();
            c <- a + 3;
            if (a < b) then {
                out_int(c + 6);
            } else {
                out_int(a);
            } fi;
            out_int(b);
        }
    }
}

label main_0
t$2 ← call in_int
t$7 ← int 1
t$8 ← int 3
t$3 ← + t$7 t$8
t$10 ← < t$7 t$2
t$18 ← not t$10
bt t$18 main_4
bt t$10 main_3
label main_3
t$14 ← t$3
t$15 ← int 6
t$13 ← + t$14 t$15
t$9 ← call out_int t$13
jmp main_5
label main_4
t$16 ← t$1
t$9 ← call out_int t$16
jmp main_5
label main_5
t$0 ← call out_int t$2
return t$0
class Main inherits IO {
    main(): Object {
        let a : Int,
        b : Int,
        c : Int in {
            a <- 1;
            b <- in_int();
            c <- a + 3;
            if (a < b) then {
                out_int(c + 6);
            } else {
                out_int(a);
            } fi;
            out_int(b);
        }
    }
}
label main_0
  t$2 ← call in int
  t$7 ← int 1
  t$8 ← int 3
  t$3 ← + t$7 t$8
  t$10 ← < t$7 t$2
  t$18 ← not t$10
  bt t$18 main_4
  bt t$10 main_3

label main_3
  t$14 ← t$3
  t$15 ← int 6
  t$13 ← + t$14 t$15
  t$9 ← call out int t$13
  jmp main_5

label main_4
  t$16 ← t$1
  t$9 ← call out_int t$16
  jmp main_5

label main_5
  t$0 ← call out_int t$2
  return t$0
label main_0
  t$2 ← call in_int
  t$7 ← int 1
  t$8 ← int 3
  t$3 ← + t$7 t$8
  t$10 ← < t$7 t$2
  t$18 ← not t$10
  bt t$18 main_4
  bt t$10 main_3

bt t$10 taken

label main_3
  t$14 ← t$3
  t$15 ← int 6
  t$13 ← + t$14 t$15
  t$9 ← call out_int t$13
  jmp main_5

bt t$18 taken

label main_4
  t$16 ← t$1
  t$9 ← call out_int t$16
  jmp main_5

label main_5
  t$0 ← call out_int t$2
  return t$0
Control Flow Graphs

Ways to generate CFG:

- You can make a CFG in which every instruction is a separate block!
  - Analyses become slower because $V + E$ increases in CFG
- You can create basic blocks while generating code from AST
  - If you know you’ll need 4 labels to generate an if expression, you can create BBs on the fly
- You can lift your straight-line generated code into basic blocks
  - Common for optimizations and binary analyses
So what’s the deal with CFGs?
Dataflow Analysis

- CFGs are the basis for performing a number of program analyses

- The first rule of compilers club:
  - Don’t break the build

- **Dataflow analysis** is a principled technique to assess how data about a program properties flows through the program

  - Optimizations: register allocation, loop invariant code motion, Common subexpression elimination, dead code elimination
  - Security: Taint analysis, alias analysis
Analysis Foundations

Consider a single basic block with instructions (variable names added for legibility):

\[
x <- li 3 \\
y <- * z w \\
q <- x + y
\]
Analysis Foundations

Consider a single basic block with instructions (variable names added for legibility):

\[
\begin{align*}
x & \leftarrow \text{li} \ 3 \\
y & \leftarrow \ * \ z \ w \\
q & \leftarrow \ x + y
\end{align*}
\]

Compiler Construction
Analysis Foundations

Consider a single basic block with instructions (variable names added for legibility):

\[
\begin{align*}
    x & \leftarrow \text{li} \ 3 \\
    y & \leftarrow \ast \ z \ w \\
    q & \leftarrow x + y
\end{align*}
\]

\[
\begin{align*}
    x & \leftarrow \text{li} \ 3 \\
    y & \leftarrow \ast \ z \ w \\
    q & \leftarrow + \ 3 \ y
\end{align*}
\]

\[
\begin{align*}
    y & \leftarrow \ast \ z \ w \\
    q & \leftarrow 3 + y
\end{align*}
\]
Analysis Foundations

Consider a single basic block with instructions (variable names added for legibility):

\[
\begin{align*}
x & \leftarrow \text{li} 3 \\
y & \leftarrow \ast z w \\
q & \leftarrow x + y
\end{align*}
\]

Two analyses: Constant propagation and Dead code elimination
Constant Propagation

It’s easy for a single block:
- Encounter a variable on lhs
- If that variable is a constant, replace subsequent rhs references to that variable

\[
x := 3 \\
\ldots \\
w := x
\]

\[
x := 3 \\
\ldots \\
w := 3
\]

What could go wrong with propagation?
Dead Code

Sometimes, code does not contribute to the program’s result

- If
  - x := rhs appears in a block
  - x does not appear anywhere else in the program

- Then
  - The statement x := rhs is dead and can be eliminated

```
x <- li 3
x <- li 4  x <- li 4
q <- + 1 x  q <- + 1 x
```
Remarks

Even within a single basic block, it is possible to change code while conservatively preserving semantics!

Dead code elimination and constant propagation can work in tandem to reduce code size and increase execution speed

Propagating constants can create dead code, all while preserving semantics
Remarks

Even within a single basic block, it is possible to change code while conservatively preserving semantics!

Dead code elimination and constant propagation can work in tandem to reduce code size and increase execution speed

- Propagating constants can create dead code, all while preserving semantics

But what about whole CFGs?
Drain the Swamp of Dead Code
Liveness Analysis

- We can use *liveness analysis* to remove *dead code*
  - An assignment instruction is dead when its assignee is not used in the future
- A variable is *live* when it is used
- A variable gets *killed* when it is written to
- Propagate liveness throughout CFG
Liveness Intuition

\[
x \leftarrow + y z
\]

\[
w \leftarrow + t x
\]

return \(x\)

all vars dead at end of program
Liveness Intuition

\[ x \leftarrow + y z \]

\[ w \leftarrow + t x \]

\( x \) must be live

\[ \text{return } x \]

all vars dead at end of program
Liveness Intuition

\[ x \leftarrow + y z \]

\( w \) gets killed, \( t \) and \( x \) live

\[ w \leftarrow + t x \]

\( x \) must be live

return \( x \)

all vars dead at end of program
Liveness Intuition

\[ t, y, \text{ and } z \text{ must be live; } x \text{ is killed} \]

\[ x \leftarrow + y \, z \]

\( w \) gets killed, \( t \) and \( x \) live

\[ w \leftarrow + t \, x \]

\( x \) must be live

\[ \text{return } x \]

all vars dead at end of program
Liveness Intuition

t, y, and z must be live; x is killed

\[ x \leftarrow + y \ z \]

w gets killed, t and x live

\[ w \leftarrow + t \ x \]

w is dead

x must be live

\[ \text{return } x \]

all vars dead at end of program
Liveness Intuition

```
x ← + y z
```

Now we recompute...

```
return x
```

all vars dead at end of program
Liveness Intuition

$y$ and $z$ are live, $x$ is killed

$x \leftarrow + y z$

$x$ remains live (no $w$ or $t$ anymore)

$x$ must be live

return $x$

all vars dead at end of program
Liveness in Basic Block

\[ LIVE_{in} = \{y, z\} \]

\[ LIVE_{out} = \{\} \]
Liveness in Basic Block

\[ \text{LIVE}_{in} = \{\} \]

\[
\begin{align*}
y & \leftarrow \text{int 3} \\
z & \leftarrow \text{int 1}
\end{align*}
\]

\[ \text{LIVE}_{out} = \{y, z\} \]

\[
\begin{align*}
x & \leftarrow + y \ z \\
w & \leftarrow + t \ x \\
\text{return } x
\end{align*}
\]

\[ \text{LIVE}_{out} = \{\} \]
Liveness in Basic Block

\[ \text{LIVE}_{in} = \{ \} \]

\[ \begin{align*}
\text{y} &\leftarrow \text{int } 3 \\
\text{z} &\leftarrow \text{int } 1
\end{align*} \]

\[ \text{LIVE}_{out} = \{ y, z \} \]

\[ \begin{align*}
\text{x} &\leftarrow + y \ z \\
\text{w} &\leftarrow + t \ x \\
\text{return } x
\end{align*} \]

\[ \text{LIVE}_{out} = \{ \} \]

\[ \text{y} \leftarrow \text{int } 1 \]
Liveness in Basic Block

\[ \text{LIVE}_\text{in} = \{\} \]

\[ \text{LIVE}_\text{out} = \{y, z\} \]

\[ y \leftarrow \text{int } 3 \]
\[ z \leftarrow \text{int } 1 \]

\[ \text{LIVE}_\text{in} = \{y, z\} \]

\[ \text{LIVE}_\text{out} = \{y, z\} \]

\[ x \leftarrow + y \ z \]
\[ w \leftarrow + t \ x \]
\[ \text{return } x \]

\[ \text{LIVE}_\text{out} = \{\} \]
Liveness in Basic Block

\[ \text{LIVE}_{in} = \{\} \]

\[ y \leftarrow \text{int 3} \]
\[ z \leftarrow \text{int 1} \]

\[ \text{LIVE}_{out} = \{y, z\} \]

\[ x \leftarrow + y z \]
\[ w \leftarrow + t x \]
\[ \text{return x} \]

\[ \text{LIVE}_{out} = \{\} \]

\[ \text{LIVE}_{in} = \{z\} \]

\[ y \leftarrow \text{int 1} \]

\[ \text{LIVE}_{out} = \{y, z\} \]
Summary for Liveness

- All basic blocks in CFG initialized with $\text{LIVE}^{\prime}_{in} = \emptyset$ and $\text{LIVE}^{\prime}_{out} = \emptyset$
- Repeat until $\text{LIVE}^{\prime}_{in} = \text{LIVE}^{\prime}_{in}$ for all blocks
- For each block $b$ in CFG
  - $\text{LIVE}^{\prime}_{out} = \bigcup_{b^{\prime} \text{children}} \text{LIVE}^{\prime}_{in} (b^{\prime} \text{children})$
- Starting with $\text{myset} = \text{LIVE}^{\prime}_{out}$, move backwards through instructions
- Remove assignee of instruction if present
- Add operands
- At top of block, $\text{LIVE}^{\prime}_{in} = \text{myset}$
Correctness in Liveness

x <- li 3
y <- li 3

z <- + y 1

y <- li 2
z <- li 3

return w

What could go wrong?
Correctness

If we want to remove an instruction $x \leftarrow \text{rhs}$, then we must consider a correctness condition:

On every path following an instruction $x \leftarrow \text{rhs}$, the variable $x$ is not alive

Remember: preserving semantics is key!
General Dataflow Analysis

- Dataflow analysis helps us reason about program properties
  - e.g., “variable $x$ is alive”, “variable $y$ is always 4”
- General dataflow analysis often involves several traits:
  - Depends on knowing a property $P$ at a particular point in program execution
  - Proving $P$ at any point requires knowledge of the entire method body
  - $P$ is typically undecidable
- Dataflow Analysis is conservative
Conservatism

- We must maintain correctness
  - Correctness vs. aggressiveness tradeoff
  - Either we **definitely know for sure** that the property is true
    - “$x$ is definitely 3”
  - Or we **don’t know for sure**
    - “can’t tell was $x$ is”

- Err on the side of correctness—if you can’t make a definite determination, it’s always OK to know you “don’t know”