CS 473: COMPILER DESIGN
Multiple IRs

• Goal: get program closer to machine code without losing the information needed to do analysis and optimizations
• In practice, multiple intermediate representations might be used for multiple stages of translation
Multiple IRs

- Goal: get program closer to machine code without losing the information needed to do analysis and optimizations
- In practice, multiple intermediate representations might be used for multiple stages of translation
Tree IR: Loops

While
true  b

start:
Jmp

While
true  b

label-start  b
Jmp
label-start

start:
[b]
Jmp start
done:
IR 2: Block-Based IR

- Goal: linear sequence of statements
  - No sequences, no calls inside other expressions, \( \text{Cjmp fall-through} \)
- Same expressions and statements as tree IR, but certain combinations of nodes are ruled out
- Statements grouped into *basic blocks* with jumps at the end

\[
\text{if}(b) \ a[i] = 0;
\]

```
Cjmp
```

```
b
Move
Mem 0
Plus
fp 1
```

```
label2
```

```
Cjmp (not b) label2
Move (Mem (fp + 1)) 0
label2:
```
IR 2: Block-Based IR

- Translate in 3 steps:
  1. Linearize: get rid of $E_{Seq}$ nodes, convert to list of statements
  2. Group: group into blocks of straight-line code ("basic blocks")
  3. Schedule: set up fall-throughs for Cjmps

```
Cjmp (not b) label2
Move (Mem (fp + 1)) 0
label2:
```
Questions

Top
1. Linearize: get rid of \( E_{\text{Seq}} \) nodes, convert to list of statements
   - Goal: each statement does one simple operation
     (assignment/arithmetic/function call/etc.)

In C: \( y = (x = x + 1); \)

Exercise: How would you write the same program in a language where we can’t do assignments on the RHS?
1. Linearize: get rid of $ESeq$ nodes, convert to list of statements

In C: $y = (x = x + 1)$;
same as $x = x + 1; y = x$;

- Rewrite trees to move $ESeq$s to top
Block IR: Binary Operators

\[(x = x + 1) + y\]
\( y + (x = x + 1) \)
\[ x + (x = x + 1) \]
\{x = 3\}
Block IR: Evaluation Order

\[ x + (x = x + 1) \quad \iff \quad \text{temp} = x; \; x = x + 1; \; \text{temp} + x \]
{\(x = 3\)}
\[ x + (x = x + 1) \quad \iff \quad \text{temp} = x; \ x = x + 1; \ \text{temp} + x \]

\{x = 3\}
\[(s; e_1) + e_2 \implies (s; e_1 + e_2)\]

(assuming left-to-right evaluation)

\[e_1 + (s; e_2) \implies (\text{temp} = e_1; s; \text{temp} + e_2)\]

\[e_1 + (s; e_2) \implies (s; e_1 + e_2) \text{ only when } s \text{ and } e_1 \text{ commute}\]

- A statement *commutes* with an expression when running the statement doesn’t change the behavior of the expression.
- In general, we can’t always tell whether two things commute, so we need to be *conservative*. 
Questions

Top
f(x) + 5

f(x) + e  =>  (temp = f(x); temp + e)
Block IR: Function Calls

\[
f(x) + e \quad \Rightarrow \quad (\text{temp} = f(x); \text{temp} + e)
\]

Exercise: linearize \( y = (z = f(x) + g(y)); \)
(that is, turn it into a sequence of statements where there are no assignments or function calls inside other expressions)

\[
y = (z = f(x) + g(y)); \quad \Rightarrow
\]
\[
z = f(x) + g(y); \quad y = z; \quad \Rightarrow
\]
\[
temp1 = f(x); \quad z = temp1 + g(y); \quad y = z; \quad \Rightarrow
\]
\[
temp1 = f(x); \quad temp2 = g(y); \quad z = temp1 + temp2; \quad y = z;
\]
Block IR: Evaluation Order

\[ y = (z = f(x) + g(y)); \Rightarrow \]
\[
\text{temp1} = f(x); \text{temp2} = g(y); z = \text{temp1} + \text{temp2}; y = z;
\]

- This is assuming that when we write something like \( f(x) + g(y) \), the call to \( f \) always happens first
- What else could it be? (from cppreference.com)

**Order of evaluation**

Order of evaluation of any part of any expression, including order of evaluation of function arguments is *unspecified* (with some exceptions listed below). The compiler can evaluate operands and other subexpressions in any order, and may choose another order when the same expression is evaluated again.

There is no concept of left-to-right or right-to-left evaluation in C++. This is not to be confused with left-to-right and right-to-left associativity of operators: the expression \( a() + b() + c() \) is parsed as \((a() + b()) + c()\) due to left-to-right associativity of operator+, but the function call to \( c \) may be evaluated first, last, or between \( a() \) or \( b() \) at run time:

```c
#include <cstdio>
int a() { return std::puts("a"); }
int b() { return std::puts("b"); }
int c() { return std::puts("c"); }
void z(int, int, int) {}
int main() {
    z(a(), b(), c()); // all 6 permutations of output are allowed
    return a() + b() + c(); // all 6 permutations of output are allowed
}
```
Block IR: Evaluation Order

\[
y = (z = f(x) + g(y)); \Rightarrow \\
temp1 = f(x); \ temp2 = g(y); \ z = temp1 + temp2; \ y = z;
\]

• This is assuming that when we write something like \( f(x) + g(y) \), the call to \( f \) always happens first
• What else could it be? right-to-left, but also \textit{unspecified}
• If it’s unspecified, then \( f(x) + g(y) \) could turn into
  \[
  \begin{align*}
  \text{temp1} &= f(x); \text{temp2} = g(y); \text{temp1 + temp2} \quad \text{or} \\
  \text{temp2} &= g(y); \text{temp1} = f(x); \text{temp1 + temp2}
  \end{align*}
  \]
• The compiler gets to choose when it does IR translation!
  – For instance, gcc evaluates function arguments left-to-right, but clang does them right-to-left
  – Could be bad news for a programmer who thought it was left-to-right!
  – If you don’t want the compiler to decide, do it yourself:
    \[
    \text{int a = f(x); z = a + g(y);}\]
Block IR: Linearize

\[ x = (s; e) \rightarrow (s; x = e) \]

\[ (s; e_1) + e_2 \rightarrow (s; e_1 + e_2) \]

\[ e_1 + (s; e_2) \rightarrow (temp = e_1; s; temp + e_2) \]

\[ e_1 + (s; e_2) \rightarrow (s; e_1 + e_2) \text{ only when } s \text{ and } e_1 \text{ commute} \]

\[ f(x) + e \rightarrow (temp = f(x); temp + e) \]

• Move `ESeq`s up to the top
• When we don’t have any sequences or calls in expressions, we have an assembly-style list of statements
Questions

Top
2. Group: group into blocks of straight-line code ("basic blocks")

- A basic block has the form:
  
  ```
  label:
  ...
  // straight-line code, no jumps or labels
  ...
  Jump/Cjump
  ```

- This lets us deal with code in sections instead of individual statements, and is useful for control flow analysis
2. Group: group into blocks of straight-line code ("basic blocks")

- A *basic block* has the form:
  label:
  ...
  // straight-line code, no jumps or labels
  ...
  Jump/Cjump

  stmt1
  stmt2
  ...
  stmtn
2. Group: group into blocks of straight-line code ("basic blocks")

- A **basic block** has the form:
  
  ```
  label:
  ...
  // straight-line code, no jumps or labels
  ...
  Jump/Cjump
  ```

  ```
  new_label1:
  stmt1
  stmt2
  ...
  stmtn
  ```
2. Group: group into blocks of straight-line code ("basic blocks")

- A **basic block** has the form:

```plaintext
label:
... // straight-line code, no jumps or labels
...
Jump/Cjump

new_label1:
stmt1
stmt2
Jump/Cjump
...
```
2. Group: group into blocks of straight-line code ("basic blocks")

- A *basic block* has the form:
  
  ```
  label:
  ...
  // straight-line code, no jumps or labels
  ...
  Jump/Cjump
  ```

  ```
  new_label1:
  stmt1
  stmt2
  Jump/Cjump
  ```

  ```
  ...
  ```
2. Group: group into blocks of straight-line code ("basic blocks")

- A basic block has the form:
  
  ```
  label:
  ... // straight-line code, no jumps or labels
  ...
  Jump/Cjump
  
  new_label1:
  stmt1
  stmt2
  start:    
  ...
  ```
2. Group: group into blocks of straight-line code ("basic blocks")

- A basic block has the form:
  label:
  ...
  // straight-line code, no jumps or labels
  ...
  Jump/Cjump

  new_label1:
  stmt1
  stmt2
  Jump start

  start: ←
  ...

  Jump start
2. Group: group into blocks of straight-line code ("basic blocks")

- A basic block has the form:
  
  label:
  ...
  // straight-line code, no jumps or labels
  ...
  Jump/Cjump

- Go through program statement by statement
- If first statement isn’t a label, insert a new label
- If we reach a jump, end block and start new block
- If we reach a label, add jump to that label and start new block
Top
3. Schedule: set up fall-throughs for Cjumps
   • Cjump instructions in the IR have two labels: a true case and a false case
   • But conditional jumps in assembly usually only have one label, and just move to the next instruction otherwise (fall-through)
   • Goal: remove false cases from Cjumps and set up fall-throughs instead

   • A basic block has the form:
     label:
     ... // straight-line code, no jumps or labels
     ... Jump/Cjump

   • So we can rearrange the order of the basic blocks without changing the program’s behavior at all!
Goal: remove false cases from Cjumps and set up fall-throughs instead

```c
label:
...
// straight-line code, no jumps or labels
...
Cjump LT a 0 ltrue lfalse

ltrue:
Move(x, 1)

lfalse:
Move(x, 0)
```
• Goal: remove false cases from Cjumps and set up fall-throughs instead

label:
...     // straight-line code, no jumps or labels
...
Cjump LT a 0 ltrue lfalse

lfalse:
Move(x, 0)

ltrue:
Move(x, 1)
• Goal: remove false cases from Cjumps and set up fall-throughs instead

label:
...
// straight-line code, no jumps or labels
...
Cjump LT a 0 ltrue

lfalse:
Move(x, 0)

ltrue:
Move(x, 1)
• While we’re at it, maybe we can get rid of some extra Jumps too

label:

...  // straight-line code, no jumps or labels

...  

Jump label2

label1:

codel

label2:

code2
• While we’re at it, maybe we can get rid of some extra Jumps too

```c
label:
...  // straight-line code, no jumps or labels
...
Jump label2

label2:
code2

label1:
code1
```
While we’re at it, maybe we can get rid of some extra Jumps too

```plaintext
label:
...
  // straight-line code, no jumps or labels
...

label2:
code2

label1:
code1
```
Questions

Top
3. Schedule: set up fall-throughs for \texttt{Cjumps}

- A \textit{basic block} has the form:

  \begin{verbatim}
  label:
  ...
  // straight-line code, no jumps or labels
  ...
  \end{verbatim}

- So we can rearrange the order of the basic blocks without changing the program’s behavior at all!

- First pass: move through the code starting from the beginning; when we hit a jump to a label we haven’t seen yet, choose the block with that label as the next block
• First pass: move through the code starting from the beginning; when we hit a jump to a label we haven’t seen yet, choose the block with that label as the next block

l1: \text{Cjump 13 14}
l2: 
l3: 
l4: 
l5: 
• First pass: move through the code starting from the beginning; when we hit a jump to a label we haven’t seen yet, choose the block with that label as the next block

l1: Cjump l3 l4

l4:
l2:
l3:
l5:
• First pass: move through the code starting from the beginning; when we hit a jump to a label we haven’t seen yet, choose the block with that label as the next block
• Second pass: fix any \texttt{Cjump}s that still aren’t followed by their false label
• If it’s followed by its true label instead:

\begin{align*}
\text{l1: } & \quad \text{Cjump LT a 0 l3 l4} \\
\text{l3: } & \quad \text{Cjump GE a 0 l4 l3}
\end{align*}
First pass: move through the code starting from the beginning; when we hit a jump to a label we haven’t seen yet, choose the block with that label as the next block.

Second pass: fix any C-jumps that still aren’t followed by their false label.

This might add extra code, but we need it in order to get rid of the false labels!
Block IR: Scheduling

• First pass: move through the code starting from the beginning; when we hit a jump to a label we haven’t seen yet, choose the block with that label as the next block
• Second pass: fix any \texttt{CJump}s that still aren’t followed by their false label
  – This might add extra code, but we need it in order to get rid of the false labels!
• Third pass: get rid of all the false labels of \texttt{CJump}s, and remove any \texttt{Jump}s that go straight to the next block

• Now we can just \textit{fall through} to the next block when a basic block ends with a non-jump statement or false \texttt{CJump}
Questions
**IR 2: Block-Based IR**

- Translate in 3 steps:
  1. **Linearize**: get rid of `ESeq` nodes, convert to list of statements
  2. **Group**: group into blocks of straight-line code ("basic blocks")
  3. **Schedule**: set up fall-throughs for `Cjmps`

- **Result**: something a lot like a generic assembly program!

```plaintext
Cjmp (not b) label2
Move (Mem (fp + 1)) 0
label2:
```