IR Code Generation

Introduction to IR Code Generation

1. After syntax and semantics have been checked, the source code is translated from source language to an intermediate representation
   - IR is machine independent
   - Allows many analyses and optimizations to be performed

IR Quiz

1. Fill in the blanks with regards to Intermediate Representation.
   - IR is the representation of a program between the source and target languages
   - A good IR maximizes the goal of achieving an retargetable compiler
   - One reason to use an IR is to perform machine independent optimizations

IR Types

1. Three major categories:
   - Structural
     - Graph oriented
     - Heavily used in source-to-source translators
     - Tend to be large
     - Examples: Trees, DAGs
   - Linear
     - Pseudo-code for an abstract machine
     - Level of abstraction varies
     - Simple, compact data structures
     - Easier to rearrange
     - Examples: 3 address code, Stack machine code
   - Hybrid
     - Combination of graphs and linear code
     - Example: Control-flow graph

Three Address Code Part 1

1. Three addresses: operand 1, operand 2, result
   - Arithmetic operations
     - x := y op z (three addresses)
     - x := op y (two addresses)
   - Data movement
     - x := y[z]
     - x[z] := y
     - x := y
   - Control flow
     - if y op z goto x
     - goto x
   - Function call
     - param x
     - return y
     - call foo

Three Address Code Part 2

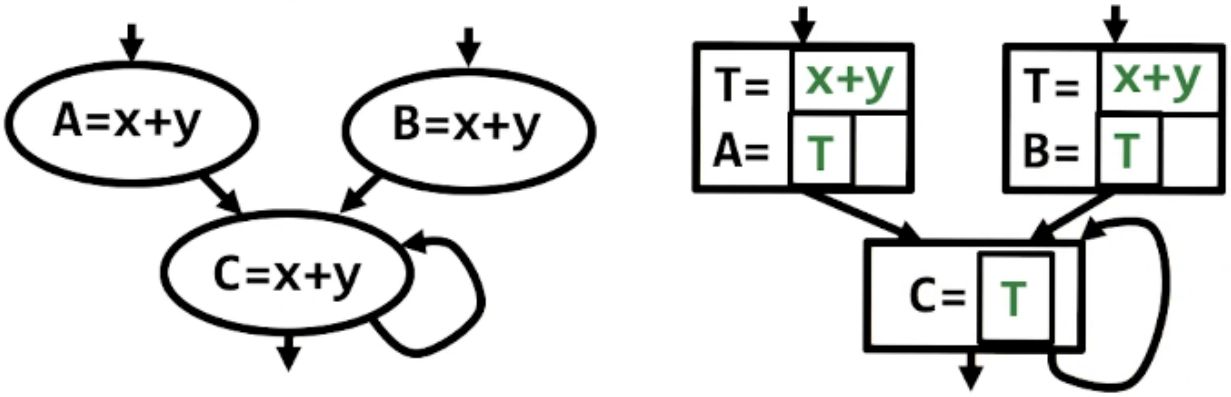
1. Complete three address code for x - 2 * y:

- t1 := 2
- t2 := y
- t3 := t1 * t2
- t4 := x
- t5 := t4 - t3

2. Compiler is trying to evaluate the subexpressions and hold intermediate results in temporaries
   - Main job of IR code generator is to manage these temporaries
   - Ensure the correct data flow is generated

Three Address Quiz

1. Given the following code, remove its redundancies
   - Let T = x + y
   - Reuse result of first subexpression instead of repeating computation



Code Redundancy Quiz

Code Generation

1. Source code:

```
if (c == 0) {
    while (c < 20) {
        c = c + 2;
    }
} else {
    c = n * n + 2;
}
```

2. Intermediate representation

```
t1 = c == 0
if_goto t1, lab1
t2 = n * n
t4 = t2 + 2
c = t4
goto end
lab1:
t3 = c >= 20
if_goto t3, end
t5 = c + 2
```

```
c = t5
goto lab1
end:
```

Code Quiz

1. Given the following snippet, generate the three address code.

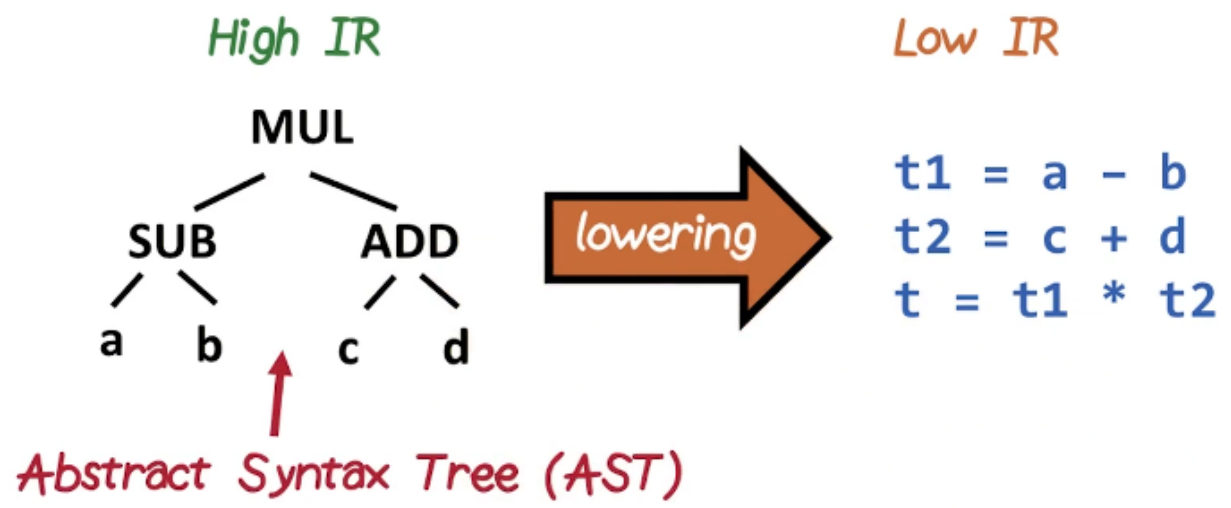
```
if (x + y * z > x * y + z)
    a = 0;
```

2. Three address code:

```
t1 = y * z;
t2 = x + t1;
t3 = x * y;
t4 = t3 + z;
if (t2 <= t4) goto L
A = 0
L:
```

IR Lowering Quiz

1. Match the definition to the term:
   - High IR captures high-level language constructs
   - Low IR captures low-level machine features



IR Lowering Quiz

Lowering

1. How do we translate from high-level IR to low-level IR?
   - HIR is complex, with nested structures
   - LIR is low-level, with everything explicit
   - Need a systematic algorithm
2. Idea:
   - Define translation for each AST node, assuming we have code for children
   - Come up with a scheme to stitch them together

- Recursively descend the AST

3. Define a generate function
   - For each kind of AST node
   - Produces code for that kind of node
   - May call generate for children nodes
4. How to stitch code together?
   - Generate function returns a temporary (or a register) holding the result
   - Emit code that combines the results

Lowering Expressions Part 1

1. Arithmetic operations:
   - expr1 op expr2
   - Generate code for left and right children, get the registers holding the results
     - t1 = generate(expr1)
     - t2 = generate(expr2)
     - r = new_temp()
     - Emit (r = t1 op t2)
       - Prints this particular code onto the output
     - return r
       - Return the register to generate call above

Lowering Expressions Part 2

1. Scheme works for:
   - Binary arithmetic
   - Unary operations
   - Logic operations
2. What about ?? and ||?
   - In C and Java, they are "short-circuiting"
     - For &&, don't evaluate op2 if op1 is false
     - For ||, don't evaluate op2 if op1 is true
   - Need control flow...

Short Circuiting

1. Code generation:

```
E = new_label()
r = new_temp()
t1 = generate(expr1)
emit(r = t1)
emit(if_goto t1, E)
t2 = generate(expr2)
emit(r = t2)
emit(E:)
return r
```

2. Generated code
   - Implements short circuiting by injecting the if_goto statement

```
t1 = expr1
r = t1
if_goto t1, E
t2 = expr2
```

```
r = t2
E:
```

Short Circuiting Quiz

1. Given the expression:

```
if (x < 100 || x > 200 && x != y)
    x = 0
```

2. Translate it to three address instructions.

```
if x < 100 goto L2
    ifFalse x > 200 goto L1
    ifFalse x != y goto L1
L2: x = 0
L1:
```

Helper Functions

1. emit()
   - The only function that generates instructions
   - Adds instructions to end of buffer
   - At the end, buffer contains code
   - Order of calls to emit is significant!
     - Generate all code for left child, then right child, then parents
2. new_label()
   - Generate a unique label name
   - Does not update code
3. new_temp()
   - Generate a unique temporary name
   - May require type information

Short Circuiting &&

1. Short circuit and is very similar concept, but if first expression is false, skip the rest
   - Can flip the predicate

```
E = new_label()
r = new_temp()
t1 = generate(expr1)
emit(r = t1)
emit(ifnot_goto t1, E)
t2 = generate(expr2)
emit(r = t2)
emit(E:)
return r
```

Array Access

1. Depends on abstraction
   - expr1[expr2]
   - Size type information needed from symbol table

```
r = new_temp()
a = generate(expr1)
o = generate(expr2)
```

```
emit(o = o * size)
emit(a = a + o)
emit(r = load a)
return r
```

Statements

1. Normal statements don't require anything fancy to do
    - statement1 = generate(statement1)
2. Conditionals

```
if (expr)
    statement;
```

```
E = new_label()
t = generate(expr)
emit(ifnot_goto t, E)
generate(statement)
emit(E:)
```

Loops Quiz

1. Emit label fo rtop of loop.
    - Generate condition and loop body
    - Complete the statements for the while loop

```
while (expr)
    statement;
```

```
E = new_label()
T = new_label()
emit(T:)
t = generate(expr)
emit(ifnot_goto t, E)
statement
emit(goto T)
emit(E:)
```

Function Calls

1. Different calling conventions

```
x = f(expr1, expr2, ...);
```

```
a = generate(f)
foreach expr - i
    ti = generate(expri)
    emit(push ti)
emit(call_jump a)
emit(x = get_result)
```

For Loop

1. How do the semantics of the for loop result in different code generation from the while loop?
    - The resulting code is essentially the same

```
for (expr1; expr2; expr3)
    statement
```

Translation Quiz

1. Translate the following into three address code:

```
while a < b do
    if c < d then
        x := y + z
    else
        x := y - z
```

```
L1: if a < b goto L2
goto Lnext
L2: if c < d goto L3
goto L4
L3: t1 := y + z
x := t1
goto L1
L4: t2 := y - z
x := t2
goto L1
Lnext:
```

Assignment

1. Problem: Difference between right-side and left-side
   - Right-side: a value (r-value)
   - Left-side: a location (l-value)
2. Example: array assignment
   - A[i] = B[j]
   - Store to A[i], load from B[j]
3. Define generate for l-values
   - lgenerate returns register containing address
4. r-value case

```
r = new_tmp()
a = generate(expr1)
o = generate(expr2)
emit(o = o * size)
emit(a = a + o)
emit(r = load a)
return r
```

5. l-value case

```
a = generate(expr1)
o = generate(expr2)
emit(o = o * size)
emit(a = a + o)
return a
```

6. Assignment
   - Use lgenerate for the left-side
   - Return r-value for nested assignment

```
//expr1 = expr2;
r = generate(expr2)
l = lgenerate(expr1)
```

```
emit(store *l = r)
return r
```

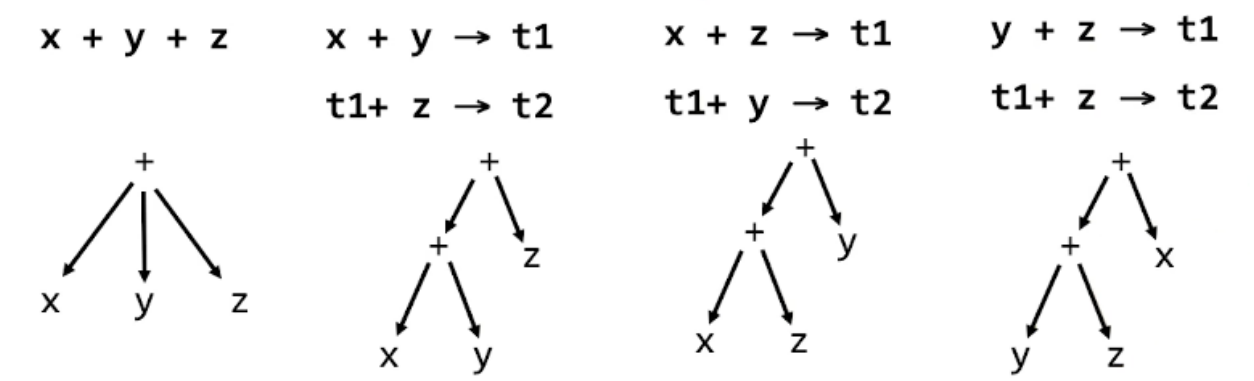
At Leaves

1. Depends on level of abstraction
   - generate(v) for variables
     - All virtual registers: return v
       - Virtual register is one which is given to a particular variable assuming infinite number of available registers
       - Key idea is to unify all references to register names
     - Strict register machine: emit(r = load v)
     - Lower level: emit(r = load base + offset)
     - base is stack pointer, offset from symbol table
   - generate(c) for constants
     - May return special object to avoid r = #

```
Reg generate(ASTNode node)
{
 Reg r;
 switch (node.getKind()) {
 case BIN:
  t1 = generate(node.getLeft());
  t2 = generate(node.getRight());
  r = new_temp();
  emit(r = t1 op t2);
  break;
 case NUM:
  r = new_temp();
  emit(r = node.getValue());
  break;
 case ID:
  r = new_temp();
  o = symtab.getOffset(node.getID());
  emit(r = load sp + o);
  break;
 }
 return r;
}
```

Code Shape

1. All code generation is taking place at the AST level
   - Should capture x+y in t1 if the value is used elsewhere
     - Should use x+z if that's what's used elsewhere
   - The "best" shape for x+y+z depends on context
     - There may be several conflicting options



Code Shape

Code Shape: Another Example

1. Another example: the switch statement
   - Implement it as a jump table
     - Lookup address in a table and jump to it
     - Uniform (constant) cost
   - Implement it as cascaded if-then-else statements
     - Cost depends on where your case actually occurs
     - O(number of cases)
   - Implement it as a binary search
     - Uniform log(n) cost
   - Compiler must choose best implementation strategy
     - No way to convert one into another