IR modeling, application issues

Variety of issues we'll consider briefly:

- Modeling control: combining linear codes and control flow graphs
- Examining impact of variable naming choices/methods
- Examining impact of assumed memory models
- Examining use of symbol tables as part of our IR (and alternatives to use of symbol tables)

Modeling flow of control

- Typical program needs to support iteration, selection, subroutines, etc: blocks of sequential statements with (possibly conditional) jumps into/out of them
- One idea is to divide code into blocks of linear code, where each block has exactly one entry point (the first statement) and exactly one exit point (the last statement)
- Then use a control flow graph where each node corresponds to a block, and each directed edge corresponds to a potential jump/transition from the end of one block to the start of another

Finding entry/exit points in blocks

- Call the entry point, first statement in block, the leader
- Block ends as soon as a jump/branch statement reached
- Make two passes through linear code to identify blocks:
  - First pass we identify all the leaders (each statement immediately following a code label)
  - Second pass we identify block ending statements by going through each statement, we are on an ending statement when the *next* statement is the leader of a block
- Each sequence of statements from a leader to the subsequent end point becomes one block

Naming choices

- Base our names on original source (shows clear relationship to source) vs create new ones (increasing possibility of identifying optimizations/rewrites)
- Suppose we move all variables into temp variables
- We can safely reuse those temp variables whenever the source code used the original variable
  - Keeps number of temp variables low, but IR code closely mimics original source
- Alternative is to generate new temp variable for each internal node: more variables, but more independence

Example: naming choices

```
// original
w = x * y
x = w + z
y = x – y
z = w / y
```

```
// keeping same name structure
t1 = w
t2 = x
t3 = y
t4 = z
t1 = t2 * t3
t2 = t1 + t4
t3 = t2 – t3
t4 = t1 / t3
w = t1
x = t2
y = t3
z = t4
```

```
// new names for internal nodes
t1 = w
t2 = x
t3 = y
t4 = z
t5 = t2 * t3
t6 = t5 + t4
t7 = t6 – t3
t8 = t5 / t7
w = t5
x = t6
y = t7
z = t8
```

Choice of memory models

- Do we assume our basic operations work strictly off registers, or directly with memory?
- Register-to-register model assumes we keep as much data as possible in registers, for as long as possible
- Memory-to-memory model assumes we leave things in memory as much as possible/as long as possible

Register-to-register model

- Ignores actual size of physical register sets in target, assigns each item in a block a virtual register
- Assumes that, if possible, each virtual register will be mapped to a physical register
- (Assumes that if there are insufficient physical registers then most-used virtual registers will be mapped to physical, remainder will spill to memory, to be loaded into temp register when needed)
- Simple, provides good opportunities for later optimization

Memory-to-memory model

- Assumes values kept in memory until needed
- Move to register just before needed
- If value changes then copy back to memory as soon as possible (once changes complete)
- Uses fewer registers
- Might be chosen model if our IR operations assume a memory-to-memory to give reduced num of IR op types

Symbol tables in IR

- Often want to look up information about a variable, constant, function, etc, at point in the IR where the declaration information isn't right at hand
- Can either search the IR to find the relevant declaration, or maintain a separate data structure (e.g. Symbol table) to allow cross-referencing/look-ups
- Often refer to “the symbol table”, but in fact is often implemented as multiple separate tables for different types of entity (variables, functions, labels, etc)

Types of data stored

- Variables/constants: name, scope, type, storage type/form, memory address
- Functions: name, return type, parameter types/number
- Labels: name, address, scope
- Arrays: as per variables, but also dimensions
- Structures: field types, names, sizes

Implementation of symbol table

- Array of records is simple, but O(n) searching
- Binary search tree gives O(log n)
- Hash table is common implementation approach, with suitable hash function and table size chosen to limit probable size of individual buckets

Handling of nested scopes

- In labs we considered use of unique identifiers for each scope, and maintaining a stack of active scopes: search scopes from top-of-stack down, looking for most recent
- An alternative is to generate a unique symbol table for each scope, with a linked-list of active tables
- Search current symbol table, then follow link to the one for the enclosing scope and search that, etc until reach global scope and find/fail