Translating Control Flow and boolean expressions

(on conventional machines)

• Condition codes and conditional branches

• Control flow constructs: if, while

• Short-circuit evaluation
If and While

Assume we have

• function for generating labels

• function for generating “if c goto l”

We can translate control flow constructs in a recursive tree walk over a boolean expression.

“if c goto l” is the basic “conditional branch” of conventional machines.
Translating “while”

AST: (while \( \langle \text{cond} \rangle \langle \text{stmts} \rangle \))

jmp .test
.body
    \textit{generate} \langle \text{stmts} \rangle \textit{here}
@test
    \textit{generate} \langle \text{cond} \rangle \textit{here}
    \textit{using} eval\_bool

Q: Why prefer test at end? What are other possibilities?
Translating “while” — details

\[
\text{node} = (\text{while} \ \langle \text{cond} \rangle \ \langle \text{stmts} \rangle)
\]

\[
\text{body} := \text{new\_label}(); \ \text{test} := \text{new\_label}(); \ \text{after} := \text{new\_label}();
\]

\[
\text{emit\_jump(test, unconditional)};
\]

\[
\text{emit\_label(body)};
\]

\[
\text{codegen( node.stmts )};
\]

\[
\text{emit\_label(test)};
\]

\[
\text{eval\_bool(node.cond, body, after)};
\]

\[
\text{emit\_label(after)};
\]
Condition codes

Conventional architectures contain special “flags” register

Example: 80x86 EFLAGS (16 bits)

Individual bits set implicitly by arithmetic and comparisons, used by branch instructions

    cmp $0,-4(%ebp)
    je .L15

...  

.L15

CMPL sets OF, SF, ZF, AF, PF, CF;

JE branches if ZF=1
Evaluating boolean expressions

Short circuit (test-and-jump) evaluation

Evaluate by control flow, *not* by boolean logic.

\[ A \text{ and } B \Rightarrow \text{if } A \text{ then } B \text{ else false} \]

\[ A \text{ or } B \Rightarrow \text{if } A \text{ then True else } B \]
Short-circuit evaluation: utility routines

function new_label return label;
     /* labels can be integers */

procedure emit_jump (l: label; c: condition);
     /* emits conditional jump */

procedure eval_bool(b: boolean_expr, tbranch, fbranch: label);
     /* generate if/then/else goto sequence */
Evaluating boolean expressions
basis cases

• If condition is a relational operator:
  emit comparison operator, e.g., CMPL
  emit_jump(tb, condition);
  emit_jump(fb, condition);

• If condition is a boolean variable
  CMPL with zero
  generate jumps as above

• If condition is a constant
  generate unconditional jump
Decomposing boolean expressions

Always two target branches, one of which is “fall through”

```c
eval_bool( x AND y , tb, fb ) =>
  eval_bool(x, L, fb);
  emit label(L);
  eval_bool(y, tb, fb);

eval_bool( x OR y, tb, fb ) =>
  eval_bool(x, tb, L);
  emit label(L);
  eval_bool(y, tb, fb);

eval_bool(NOT x, tb, fb) =>
  eval_bool(x, fb, tb)
```
Optimizer clean-up

Note that we produce “jumps to jumps” and “jumps to here.”

Avoiding them is not easy (try it). Cleaning them up is.

Another example where “optimizing” sloppy code is easier than generating good code.