Uses of Boolean Expressions

Used to drive **conditional execution** of program sections, e.g.

```plaintext
IF (a < 17) OR (b = 12) THEN ... ELSE ...;

WHILE NOT ((x+1) > 39) DO ... END;
```

(In some languages) may be assigned to **boolean variables** or passed as parameters, e.g.:

```plaintext
VAR b : BOOLEAN := (a < 17) OR (b = 12);
...
IF b THEN ... ELSE ...
...
myproc(b); (* procedure call *)
...
Two representations may be useful:

- **Value** Representation.

  Encode `true` and `false` numerically, e.g., as 1 and 0, and treat boolean expressions like arithmetic expressions.

  Pro: Language may support boolean values.
  Con: Often a bad match to hardware.

- **Flow-of-control** Representation.

  Position in generated code represents boolean value.

  Pro: Good when "short-circuit" evaluation is allowed (or required), e.g., in C expression `e1 || e2`, `e2` should be evaluated only if `e1` is false.

  Reminder: Some languages mandate short-circuit evaluation; others prohibit it; still others leave it up to the compiler writer.

  Pro: Convenient for control statements.

- For fab, we’ll use flow-of-control approach, and convert to values when necessary.
SAMPLE PRODUCTIONS FOR VALUE-BASED BOOLEANS

B := E1 'LESS' E2
   B.place = newtemp()
   B.code =
     let true = newlabel()
     after = newlabel()
     in E1.code @
         E2.code @
         [gen(true, if <, E1.place, E2.place),
          gen(B.place, :=, 0, _),
          gen(after, goto, _, _),
          gen(true, :, _, _),
          gen(B.place, :=, 1, _),
          gen(after, :, _, _)]

Generates:

    IF E1 < E2 GOTO L1
    T := 0
    GOTO L2
L1: T := 1
L2: ...
MORE SAMPLE VALUE-BASED PRODUCTIONS

B := B1 OR B2  
    B.place = newtemp()  
    B.code = B1.code @ B2.code @  
    [gen(B.place,|,B1.place,B2.place)]  

S := IF B THEN S1 ELSE S2  
    S.code = let false = newlabel()  
    after = newlabel()  
    in B.code @  
    [gen(false,if=,B.place,0)] @  
    S1.code @ [gen(after,goto,_,_)] @  
    [gen(false,:,_,_)] @  
    S2.code @  
    [gen(after,:,_,_)]

Generates:

    IF B = 0 GOTO L1
    S1
    GOTO L2
    L1: S2
    L2: ...
**Example Value-based Code**

IF (a > 7) OR (b = 5) THEN x = 7 ELSE y = 2;

```
t1 := addr a
...
t2 := *t1
...
t3 := const 7
...
if t2 > t3 goto L1
...
t4 := const 0
...
goto L2
L1:
...
t4 := const 1
L2:
...
t5 := addr b
...
t6 := *t5
...
t7 := const 5
...
if t6 = t7 goto L3
...
t8 := const 0
...
goto L4
...
L3:
...
t8 := const 1
L4:
...
t9 := t4 | t8
...
if t9 = 0 goto L5
...
t10 := const 7
...
t11 := addr x
...
*t11 := t1 0
...
goto L6
L5:
...
t12 := const 2
...
t13 := addr y
...
*t13 := t12
L6:
...```

PSU CS322 Spr'12 Lecture 7  © 1992–2012 Andrew Tolmach
Idea: Code generated for boolean and relational expressions has **true and false “exits”**, i.e., code evaluates expression and then jumps to one place if true and another place if false.

- Relational expressions perform test and jump to true or false exit accordingly.
- Boolean variables and constants jump directly to appropriate true or false exit.
- Boolean expressions simply adjust/combine true/false exits of their sub-expressions.
- Conditional statements define true and false exits of boolean sub-expression to point to appropriate code blocks, e.g., THEN and ELSE branches.
- If boolean-typed expression must deliver a value, true and false exits are defined to point to code that loads the value.
EXAMPLE (ASSUMING SHORT-CIRCUITING)

IF (a > 7) OR (b = 5) THEN x = 7 ELSE y = 2;

\[
t1 := \text{addr } a \\
t2 := *t1 \\
t3 := \text{const 7} \\
\text{if } t2 > t3 \text{ goto L1} \\
goto L4 \\
\]

\[
\text{L1:} \\
t7 := \text{const 7} \\
t8 := \text{addr } x \\
*t8 := t7 \\
goto L3 \\
\]

\[
\text{L4:} \\
t4 := \text{addr } b \\
t5 := *t4 \\
t6 := \text{const 5} \\
\text{if } t5 = t6 \text{ goto L1} \\
goto L2 \\
\]

\[
\text{L2:} \\
t9 := \text{const 2} \\
t10 := \text{addr } y \\
*t10 := t9 \\
\]

\[
\text{L3:} \\
\]
CONDITIONAL STATEMENTS (SOMETHING NAIVE APPROACH)

Use control flow representation for boolean-typed expressions; define labels on per-statement basis.

\[
S := \text{IF } B \text{ THEN } S1 \text{ ELSE } S2
\]

\[
B.\text{true} = \text{newlabel();}
\]

\[
B.\text{false} = \text{newlabel();}
\]

\[
S.\text{code} =
\]

\[
\text{let after = newlabel();}
\]

\[
\text{in } B.\text{code @}\]

\[
[\text{gen}(B.\text{true,:},_,_)] @ S1.\text{code @} [\text{gen}(\text{after,goto,:},_,_)] @\]

\[
[\text{gen}(B.\text{false,:},_,_)] @ S2.\text{code @}\]

\[
[\text{gen}(\text{after,:},_,_)]\]

Generates:

\[
\text{IF } B \text{ GOTO } L1
\]

\[
\text{GOTO } L2
\]

\[
L1: S1
\]

\[
\text{GOTO } L3
\]

\[
L2: S2
\]

\[
L3:
\]
Inherit true and false label attributes.

Synthesize code to perform appropriate test and jump to appropriate label.

Code doesn’t build a value, so no place attribute.

\[
\begin{align*}
B & := E1 \ '={} \ E2 \\
B.\text{code} & = E1.\text{code} \ @ \\
& \quad E2.\text{code} \ @ \\
& \quad \text{[gen}(B.\text{true},\text{if}=,E1.\text{place},E2.\text{place}), \\
& \quad \quad \text{gen}(B.\text{false},\text{goto,}_,\_)]
\end{align*}
\]

\[
\begin{align*}
B & := E1 \ '<' \ E2 \\
B.\text{code} & = E1.\text{code} \ @ \\
& \quad E2.\text{code} \ @ \\
& \quad \text{[gen}(B.\text{true},\text{if}<,E1.\text{place},E2.\text{place}), \\
& \quad \quad \text{gen}(B.\text{false},\text{goto,}_,\_)]
\end{align*}
\]

...
Inherit true and false label attributes.

Pass them down to subexpressions, after suitable manipulation; synthesize code attribute.

Again, no place attribute.

...
B := B1 OR B2

    B1.true = B.true
    B1.false = newlabel()
    B2.true = B.true
    B2.false = B.false
    B.code = B1.code @
          [gen(B1.false,:,:,_,_)] @
         B2.code

B := B1 AND B2

    B1.true = newlabel()
    B1.false = B.false
    B2.true = B.true
    B2.false = B.false
    B.code = B1.code @
          [gen(B1.true,:,:,_,_)] @
         B2.code

B := NOT B1

    B1.true = B.false
    B1.false = B.true
    B.code = B1.code
CONVERSION FROM VALUE FORM

Boolean-typed identifiers (variables, true and false constants) must be “converted” to control-flow form when tested.

\[
B := V \\
B.code = V.code @ \\
[\text{gen}(B.false, if=, V\.place, 0), \\
\text{gen}(B.true, goto, _, _)]
\]

(Assuming 0 = false, non-0 = true)
CONVERSION TO VALUE FORM

Similarly, must convert other way when a value is needed, generating code to build a value into a place.

\[
E := B \\
B.true = newlabel() \\
B.false = newlabel() \\
E.place = newtemp() \\
E.code = \\
\text{let after} = \text{newlabel()} \\
\text{in B.code @} \\
[\text{gen(B.true, : , , )}, \\
\text{gen(E.place, :=, 1, _)}, \\
\text{gen(after, goto, _, _)}, \\
\text{gen(B.false, : , , )}, \\
\text{gen(E.place, :=, 0, _)}, \\
\text{gen(after, :=, _ , _)}]
\]
Many processors implement conditional jumps in two parts:

- a comparison instruction sets internal **condition codes**
- a conditional branch instruction tests the condition codes to decide whether or not to branch

Some processors allow the condition codes to be used to drive instructions other than conditional branches, e.g., the X86 supports

- **set** instructions that place a 1 or 0 value directly in a register based on the condition codes
- **cmov** instructions that conditionally move data (or not) based on the condition codes

Either of these can be used to generate much more efficient code when the value form of a relational expression is needed. (To express these, we would need to expand our IR, of course.)
HANDLING LOOP EXITS

Same label-passing approach can be used to implement break or exit statements that can cause jumps out of loops. We simply add a .break inherited attribute to statements!

\[
S := \text{BREAK} \\
S.\text{code} = \text{gen}(S.\text{break}, \text{goto}, _, _)
\]

\[
S := \text{LOOP } S \text{ END} \\
S.\text{break} = \text{newlabel}(); \\
S.\text{code} = \\
\text{let } top = \text{newlabel}() \\
\text{in } [\text{gen}(\text{top}, :, _, _)] @ \\
S.\text{code} @ \\
[\text{gen}(\text{top}, \text{goto}, _, _), \\
\text{gen}(S.\text{break}, :, _, _)]
\]

Other loop statements (like WHILE) must define and pass a similar appropriate label to their child statement.

**All other** (non-loop) statement translations must pass the .break attribute through (unchanged) to their children!
• Code for each statement always ends by “falling through” to next statement.

• There is no information flow between code generation for statements.

\[ S := S_1 ; S_2 \]
\[ S\text{.code} = S_1\text{.code} \oplus S_2\text{.code} \]

This can lead to bad code, e.g.,

\[
\text{WHILE } B_1 \text{ DO (WHILE } B_2 \text{ DO } S)\\
\text{L1: IF } B_1 \text{ GOTO L2}\\
\text{GOTO L3}\\
\text{L2: IF } B_2 \text{ GOTO L4}\\
\text{GOTO L5 “jump to jump”}\\
\text{L4: S}\\
\text{GOTO L2}\\
\text{L5: GOTO L1}\\
\text{L3:}\\
\]

We can eliminate problems like this during optimization, but it’s easy to avoid some of them in the first place.
IDEA: DEFER DEFINITION OF TARGET LABELS

- Give each statement an inherited attribute `.next`, which says where to transfer control after statement.
- Code generated for each statement guarantees either to transfer control to `.next` label or to “fall through.”

\[
\begin{align*}
S & := S1 \ ';;' \ S2 \\
S1.next & = \text{newlabel()} \\
S2.next & = S.next \\
S.code & = S1.code @ \\
& \quad [\text{gen}(S1.next,:,_,_)] @ \\
& \quad S2.code \\
S & := \text{WHILE} \ B \ \text{DO} \ S1 \\
B.true & = \text{newlabel()} \\
B.false & = S.next \\
S1.next & = \text{newlabel()} \\
S.code & = [\text{gen}(S1.next,:,_,_)] @ \\
& \quad B.code @ \\
& \quad [\text{gen}(B.true,:,_,_)] @ \\
& \quad S1.code @ \\
& \quad [\text{gen}(S1.next,goto,_,_)]
\end{align*}
\]
Now get better code, e.g.

```
WHILE B1 DO (WHILE B2 DO S)
```

now generates

```
L1: IF B1 GOTO L2
    GOTO L?
L2: IF B2 GOTO L3
    GOTO L1
L3: S
    GOTO L2
...
L?:
```
Target label attributes (true, false, break, etc.) are inherited, so won’t work with one-pass bottom-up code generation, e.g. when generating code while doing bottom-up parsing.

Solution: Instead, keep lists of locations of gotos that need to be filled in (“backpatched”) when final target is known. These backpatch lists are synthesized attributes.
Example (to fill in): \((a > 7) \text{ OR } (b = 5)\)

1. \(t1 := \text{addr } a\)
2. \(t2 := *t1\)
3. \(t3 := \text{const } 7\)
4. \(\text{if } t2 > t3 \text{ goto } _____\)
5. \(\text{goto } _____\)

6. \(t4 := \text{addr } b\)
7. \(t5 := *t4\)
8. \(t6 := \text{const } 5\)
9. \(\text{if } t5 = t6 \text{ goto } _____\)
10. \(\text{goto } _____\)

At reduction for \(B := B1 \text{ OR } B2\)

- **Backpatch** \(B1.\text{false}\) list with address of first instruction in \(B2\).
- **Merge** \(B1.\text{true}\) and \(B2.\text{true}\) to form \(B.\text{true}\).
- **Make** \(B2.\text{false}\) into \(B.\text{false}\).
At reduction for conditional statement, backpatch true and false lists for expression.

E.g.: On reducing if $B$ then $S_1$ else $S_2$, backpatch $B$.true to location of $S_1$ and $B$.false to location of $S_2$. 
Example (to fill in):

IF (a > 7) OR (b = 5) THEN x := 7 ELSE y := 2;

1. t1 := addr a
2. t2 := *t1
3. t3 := const 7
4. if t2 > t3 goto _____
5. goto __6__
6. t4 := addr b
7. t5 := *t4
8. t6 := const 5
9. if t5 = t6 goto _____
10. goto _____
11. t7 := const 7
12. t8 := addr x
13. *t8 := t7
14. goto _18__
15. t9 := const 2
16. t10 := addr y
17. *t10 := t9
18. ...
CASE STATEMENTS

```c
case e of
  v_1 : s_1
| v_2 : s_2
| ...  
| v_n : s_n
else s
end
```

Good code generation for case statement depends on analysis of the values on the case labels $v_i$.

Options include:

- List of conditional tests and jumps (linear search).
- Binary decision code (binary tree).
- Other search code (e.g., hash table).
- Jump table (constant time).
- Hybrid schemes.
Best option depends on range of values (min and max) and their “density,” i.e., what percentage of the values in the range are used as labels.

Jump tables work well for dense value sets (even if large), but waste lots of space for sparse sets. Linear search works well for small value sets.