CS302 Spr'99 Lecture Notes
Lecture 7

Uses of Boolean Expressions

Used to drive **conditional execution** of program sections.

Examples:

```
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.

Example:

```
VAR b : BOOLEAN := (a < 17) OR (b = 12);
...
IF b THEN ... ELSE ...
...
myproc(b);   (* procedure call *)
...
```


Boolean Expressions

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: 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 PCAT, we'll use flow-of-control approach, and convert to values when necessary.

Sample Productions for Value-based Conditional Evaluation

```
B := E1 '<' 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)]
```

Here | represents bit-wise OR. Note that this implements NON-short-circuiting form of OR.

```
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:
```


Basic Control-flow Representation

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 := addr a
    t2 := *t1
    t3 := const 7
    if t2 > t3 goto L1
    goto L4
L4:
    t4 := addr b
    t5 := *t4
    t6 := const 5
    if t5 = t6 goto L1
    goto L2
L1:
    t7 := const 7
    t8 := addr x
    *t8 := t7
    goto L3
L2:
    t9 := const 2
    t10 := addr y
    *t10 := t9
L3:
```


Conditional Statements – Semi-Naive Approach

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

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

Generates:

```
    IF B GOTO L1
    GOTO L2
L1: S1
    GOTO L3
L2: S2
L3:
```


Relational Expressions

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.

```
B := E1 '=' E2
       B.code =
          E1.code @
          E2.code @
          [gen(B.true,if=,E1.place,E2.place),
           gen(B.false,goto,_,_)]

B :=  E1 '<' E2
       B.code =
          E1.code @
          E2.code @
          [gen(B.true,if<,E1.place,E2.place),
           gen(B.false,goto,_,_)]

...
```


Boolean Expressions

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
```


Conversions to and 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 @
               [gen(B.false,if=,V.place,0),
                gen(B.true,goto,_,_)]
```

(Assuming 0 = false, non-0 = true)

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 =
         let after = newlabel()
         in  B.code @
             [gen(B.true,:,_,_),
              gen(E.place,:=,1,_),
              gen(after,goto,_,_),
              gen(B.false,:,_,_),
              gen(E.place,:=,0,_),
              gen(after,:,_,_)]
```


Disadvantages of Naive Approach

• Code for each statement always concludes by "falling through" to next statement.

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

```
S := S1 ';' S2
      S.code = S1.code @ S2.code
```

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

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

generates

```
L1: IF B1 GOTO L2
    GOTO L3
L2: IF B2 GOTO L4
    GOTO L5     ``jump to jump"
L4: S
    GOTO L2
L5: GOTO L1
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."

```
S := WHILE B DO S1
      B.true = newlabel()
      B.false = S.next
      S1.next = newlabel()
      S.code = [gen(S1.next,:,_,_)] @
               B.code @
               [gen(B.true,:,_,_)] @
               S1.code @
               [gen(S1.next,goto,_,_)]
```

Deferred Label Definition (continued)

```
S := IF B THEN S1 ELSE S2
      B.true = newlabel()
      B.false = newlabel()
      S1.next = S.next
      S2.next = S.next
      S.code = B.code @
               [gen(B.true,:,_,_)] @
               S1.code @
               [gen(S.next,goto,_,_),
                gen(B.false,:,_,_)] @
               S2.code

S := S1 ';' S2
      S1.next = newlabel()
      S2.next = S.next
      S.code = S1.code @
               [gen(S1.next,:,_,_)] @
               S2.code
```


Now Get Better Code

```
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?:
```

What about `break` or `exit` statements that can cause jumps out of loops? Add a `.break` inherited attribute!

```
S := BREAK
    S.code = gen(S.break,goto,_,_)

S := WHILE B DO S1
     ... as before, plus:
     S1.break = S.next
```

All other statement translations must pass the `.break` attribute through (unchanged) to their children!

Backpatching

Target label attributes (`true,false,next`) are **inherited**, so won't work with one-pass bottom-up code generation.

Solution: Instead, keep **lists** of locations of `goto`s that need to be filled in ("**backpatched**") when final target is known. These **backpatch lists** are **synthesized** attributes.

Example (to fill in): `(a > 7) OR (b = 5)`

```
 1. t1 := addr a
 2. t2 := *t1
 3. t3 := const 7
 4. if t2 > t3 goto _____
 5. goto _____

 6. t4 := addr b
 7. t5 := *t4
 8. t6 := const 5
 9. if t5 = t6 goto _____
10. goto _____
```

At reduction for `B := B1 OR B2`

• Backpatch `B1.false` list with address of first instruction in `B2`.

• Merge `B1.true` `B2.true` to form `B.true`.

• Make `B2.false` into `B.false`.

Backpatching (Continued)

At reduction for conditional statement, backpatch `true` and `false` lists for expression.

E.g.: On reducing `if B then S1 else S2`, backpatch `B.true` to location of `S1` and `B.false` to location of `S2`.

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

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.