

## Code Generation for Control Flow

Mooly Sagiv  
Ohad Shacham

<http://www.cs.tau.ac.il/~msagiv/courses/wcc10.html>

Chapter 6.4

---

---

---

---

---

---

---

---

## Outline

- Local flow of control
- Conditionals
- Switch
- Loops

---

---

---

---

---

---

---

---

## Machine Code Assumptions

Instruction	Meaning
GOTO Label	Jump to Label
GOTO label register	Indirect jump
IF condition register then GOTO Label	Conditional Jump
IF not condition register then GOTO Label	

---

---

---

---

---

---

---

---

## Boolean Expressions

- In principle behave like arithmetic expressions
- But are treated specially
  - Different machine instructions
  - Used in control flow instructions
  - Shortcut computations
  - Negations can be performed at compile-time

```
if (a < b) goto l
    Code for a < b yielding a condition value
    Conversion condition value into Boolean
    Conversion from Boolean in condition value
    Jump to l on condition value
```

---

---

---

---

---

---

---

---

## Shortcut computations

- Languages such as C define shortcut computation rules for Boolean
- Incorrect translation of `e1 && e2`

```
Code to compute e1 in loc1
Code to compute e2 in loc2
Code for && operator on loc1 and loc2
```

---

---

---

---

---

---

---

---

## Shortcut computations

```
if (ptr != null && ptr->val > 0) {
    ...
    ...
}
```

---

---

---

---

---

---

---

---

## Code for Booleans (Location Computation)

- Top-Down tree traversal
- Generate code sequences instructions
- Jump to a designated 'true' label when the Boolean expression evaluates to 1
- Jump to a designated 'false' label when the Boolean expression evaluates to 0
- The true and the false labels are passed as parameters

---

---

---

---

---

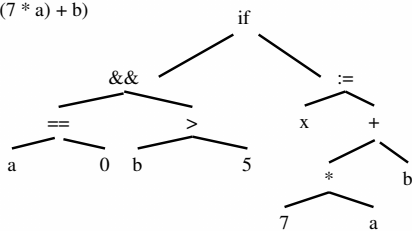
---

---

---

### Example

if ((a==0) && (b > 5))  
x = ((7 \* a) + b)




---

---

---

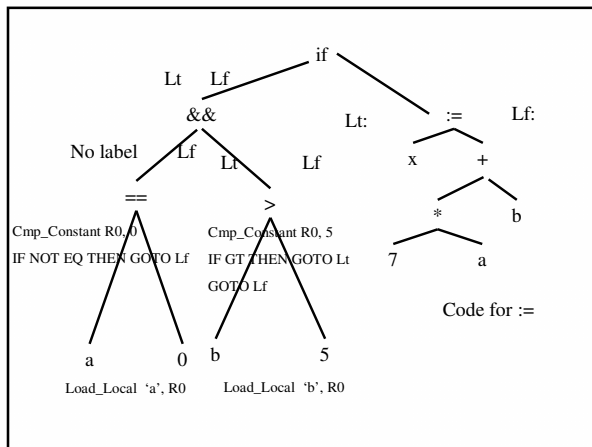
---

---

---

---

---




---

---

---

---

---

---

---

---

### Location Computation for Booleans

```

Procedure Generate Code for Boolean control expression (
Node, True label, False label) :
SELECT Node.type:
CASE Comparison type: // <, >, ==, etc. in C
Generate code for comparison expression (Node.expr);
// The comparison result is now in condition register
IF True label != No label:
Emit("If condition register THEN GOTO True label);
IF False label != No label:
Emit("GOTO" False label);
ELSE True label == No label:
IF False label != No label:
Emit("IF NOT" condition register THEN GOTO",
False label);
CASE Lazy and type: // && in C
Generate code for Boolean control expression
Node.left, No Label, False label);
Generate code for Boolean control expression
Node.right, True label, False label);
CASE Negation type: // ! in C
Generate code for Boolean control expression
Node.arg, False label, True label);
...
    
```

---

---

---

---

---

---

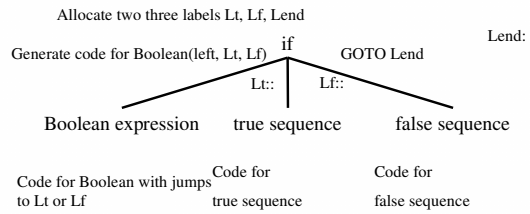
---

---

---

---

### Code generation for IF




---

---

---

---

---

---

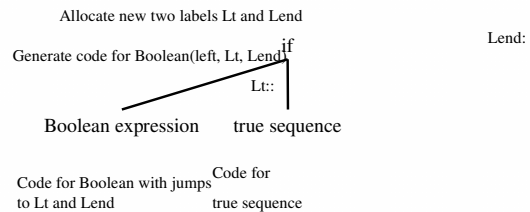
---

---

---

---

### Code generation for IF (no-else)




---

---

---

---

---

---

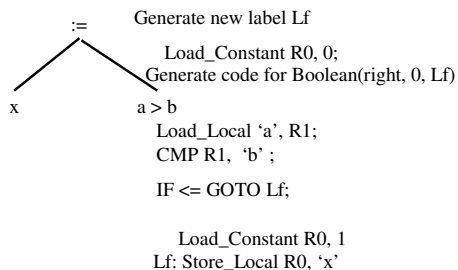
---

---

---

---

## Coercions into value computations



---

---

---

---

---

---

---

---

---

---

## Code for case statements

- Three possibilities
  - Sequence of IFs
    - O(n) comparisons
  - Jump table
    - O(1) comparisons
  - Balanced binary tree
    - O(log n) comparisons
- Performance depends on n
- Need to handle runtime errors

---

---

---

---

---

---

---

---

---

---

## Simple Translation

```
tmp_case_value := case expression;
IF tmp_case_value = i1 THEN GOTO label1;
IF tmp_case_value = i2 THEN GOTO label2;
...
IF tmp_case_value = in THEN GOTO labeln;
GOTO label_else; // or insert the code at label else
label 1:
  Code for statement sequence1
  GOTO label_next;
label 2:
  Code for statement sequence2
  GOTO label_next;
...
label n:
  Code for statement sequencen
  GOTO label_next;
label else:
  Code for else-statement sequence
label label_next:
```

---

---

---

---

---

---

---

---

---

---

## Jump Table

- Generate a table of  $L_{high} - L_{low} + 1$  entries
  - Filled at ? time
- Each entry contains the start location of the corresponding case or a special label
- Generated code

```
tmp_case_value := case expression;
if tmp_case_value < L_low GOTO label_else;
if tmp_case_value > L_high GOTO label_else;
GOTO table[tmp_case_value - L_low];
```

---

---

---

---

---

---

---

---

## Balanced trees

- The jump table may be inefficient
  - Space consumption ( $L_{high} - L_{low} > n$ )
  - Cache performance
- Organize the case labels in a balanced tree
  - Left subtrees smaller labels
  - Right subtrees larger labels
- Code generated for node\_k

```
label k: IF tmp_case_value < l THEN
```

---

---

---

---

---

---

---

---

## Repetition Statements (loops)

- Similar to IFs
- Preserve language semantics
- Performance can be affected by different instruction orderings
- Some work can be shifted to compile-time
  - Loop invariant
  - Strength reduction
  - Loop unrolling

---

---

---

---

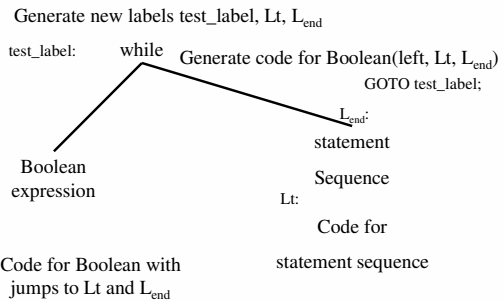
---

---

---

---

## while statements



---

---

---

---

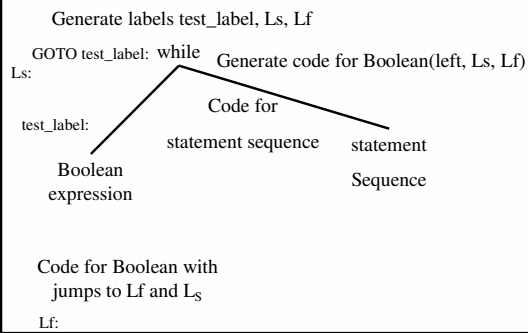
---

---

---

---

## while statements(2)



---

---

---

---

---

---

---

---

## For-Statements

- Special case of while
- Tricky semantics
  - Number of executions
  - Effect on induction variables
  - Overflow

---

---

---

---

---

---

---

---

## Simple-minded translation

```
FOR i in lower bound .. upper bound DO
  statement sequence
END for
```

⇓

```
i := lower_bound;
tmp_ub := upper_bound;
WHILE i <= tmp_ub DO
  code for statement sequence
  i := i + 1;
END WHILE
```

---

---

---

---

---

---

---

---

## Correct translation

```
FOR i in lower bound .. upper bound DO
  statement sequence
END for
```

⇓

```
  i := lower_bound;
  tmp_ub := upper_bound;
  IF i > tmp_ub THEN GOTO end_label;
loop_label:
  code for statement sequence
  if (i==tmp_ub) GOTO end_label;
  i := i + 1;
  GOTO loop_label;
end_label:
```

---

---

---

---

---

---

---

---

## Tricky question

```
for (exp1; exp2; exp3) {
  body;
}
exp1;
while (exp2) {
  body;
  exp3;
}
```

---

---

---

---

---

---

---

---



## Loop unrolling

```
FOR i := 1 TO n DO
  sum := sum + a[i];
END FOR;
```

---

---

---

---

---

---

---

---

## Loop unrolling

```
FOR i := 1 TO n-3 STEP 4 DO
  sum := sum + a[i];
  sum := sum + a[i+1];
  sum := sum + a[i+2];
  sum := sum + a[i+3];
END FOR;

FOR i := (n div 4) * 4 + 1 TO n DO
  sum := sum + a[i];
END FOR;
```

---

---

---

---

---

---

---

---

## Summary

- Handling control flow statements is usually simple
- Complicated aspects
  - Routine invocation
  - Non local gotos

---

---

---

---

---

---

---

---