Intermediate Code Generation

**Code Generation**

- Create linear representation of program
- Result can be machine code, assembly code, code for an abstract machine (e.g. the JVM), threaded code, or anything in between.
- Common representation of intermediate code depend on target machine:
  - 0-address code for stack machines
  - 2-address code for machines with memory-register operations
  - 3-address code (quadruples) for RISC architectures
- in all cases, another (top down) tree traversal

**Intermediate Code for if_statements**

if cond then
  then_statement
else
  else_statements;
end if:

- Generate labels
- Generate quadruples for some descendant node
- Place label in code
- May need label for then_statements, for short-circuit conditions

**Intermediate Code for elsif parts**

if cond1 then S1
  t1 := cond1
  if not t1 goto else_label1
  quadruples for S1
  goto endif_label
else_label1:
  t2 := cond2
  if not t2 goto else_label2
  quadruples for S2
  goto endif_label
else_label2:
  quadruples for S3
endif_label;

elsif cond2 then S2
  else_label1:
  t2 := cond2
  if not t2 goto else_label2
  quadruples for S2
  goto endif_label
else_label2:
  quadruples for S3
endif_label;
More Details of Code Generation

Outline the part of the code generator that deals with (2-branch) conditional statement.

Function code_gen(node) : reg
var end_label, t1, else_label : integer;
end_label := newlabel;
else_label := newlabel;
case node.kind of
  if :
    t1 := code_gen(node.child1);
    issue(if ¬R(t1) then goto L(else_label));
    code_gen(node.child2);
    issue(goto L(end_label));
    issue(Label L(else_label) :);
    code_gen(node.child3);
    issue(Label L(end_label) :)
end
end

An Optimized Code Generation

A more efficient approach to the code generation accepts a parameter in_end which tells it whether the statement is embedded within an external conditional.

Function code_gen(node, in_end) : reg
var end_label, t1, else_label : integer;
end_label := if in_end = 0 then newlabel else in_end;
else_label := newlabel;
case node.kind of
  if :
    t1 := code_gen(node.child1, 0);
    issue(if ¬R(t1) then goto L(else_label));
    code_gen(node.child2, end_label);
    issue(goto L(end_label));
    issue(Label L(else_label) :);
    code_gen(node.child3, end_label);
    if in_end = 0 then issue(Label L(end_label) :)
    else issue(goto L(end_label))
end
end

Example of Generation

Consider the statement

\[
\text{if } C1 \text{ then } \{ \text{if } C2 \text{ then } x := 1 \text{ else } x := 2 \} \\
\text{else } \{ \text{if } C3 \text{ then } x := 3 \text{ else } x := 4 \}
\]

The generated code is given by

\[
\text{if } !C1 \text{ goto } L1 \\
\text{if } !C2 \text{ goto } L3 \\
x := 1 \\
goto L4 \\
L3: \\
x := 2 \\
goto L2 \\
L1: \\
\text{if } !C3 \text{ goto } L5 \\
x := 3 \\
goto L6 \\
L5: \\
x := 4 \\
goto L2 \\
L2: \\
\]

Example of an Improved Generation

Reconsider the statement

\[
\text{if } C1 \text{ then } \{ \text{if } C2 \text{ then } x := 1 \text{ else } x := 2 \} \\
\text{else } \{ \text{if } C3 \text{ then } x := 3 \text{ else } x := 4 \}
\]

The improved generated code is given by

\[
\text{if } !C1 \text{ goto } L1 \\
\text{if } !C2 \text{ goto } L3 \\
x := 1 \\
goto L4 \\
L3: \\
x := 2 \\
goto L2 \\
L1: \\
\text{if } !C3 \text{ goto } L5 \\
x := 3 \\
goto L6 \\
L5: \\
x := 4 \\
goto L2 \\
L2: \\
\]
Code Generation for While Loops

Generate two labels: `start_loop`, `end_loop`

```c
while (cond) {
    start_loop:
    if (!cond) goto end_loop
    S1;
    if (cond2) break;
    S2;
    if (cond3) continue;
    S3;
    goto start_loop
}
end_loop:
```

Intermediate Code for Numeric Loops

- **Semantics**: loop not executed if range is null, so must test before first pass.

```c
for J in expr1..expr2 loop
    J := expr1
    start_label:
    if J > expr2 goto end_label
    S1
end loop;
```

Place Test at End to Utilize Loop Instruction

```c
for J in expr1..expr2 loop
    t1 := expr1
    t2 := expr2
    K := t1 - 1
    goto test_label
start_label:
    quadruples for S1
    test_label:
    K := K + 1
    if K > t2 goto end_label
    S1
end_label:
```

Code for Short-Circuit Expressions

- Short-circuit expressions are treated as control structures
  - if B1 or else B2 then S1 ... — if (B1 || B2) {S1...}
    - if B1 goto then_label
    - if not B2 goto else_label
    - then_label:
    - quadruples for S1
    - else_label:
    - Inherit target labels from enclosing control structure
    - Generate additional labels for composite short-circuit
Intermediate Code for Case Statements

- If range is small and most cases are defined, create jump table as array of code addresses, and generate indirect jump.

  \[
  \text{table label1, label2} \\
  \text{...} \\
  \text{case } x \text{ is} \\
  \text{jumpi table}[x] \\
  \text{when up: } y := 0; \\
  \text{label1:} \\
  \text{y := 0} \\
  \text{goto end_case} \\
  \text{when down: } y := 1 \\
  \text{label2:} \\
  \text{y := 1} \\
  \text{goto end_case} \\
  \text{end case;} \\
  \text{end_case:}
  \]

Evaluating Expressions: Stack Machines

- Zero-address instructions: push, pop, arithmetic
- Binary operations remove two operands, push result.

\[
\begin{align*}
  d & := b^2 - 4 \ a \ c \\
  \text{load } b & \quad \text{-- load from memory} \\
  \text{dupl} & \\
  \text{mult} & \\
  \text{push a} & \\
  \text{push c} & \\
  \text{mult} & \\
  \text{push_const 4} & \quad \text{-- push explicit constant} \\
  \text{mult} & \\
  \text{sub} & \\
  \text{store } d & \quad \text{-- duplicate value on top of stack}
\end{align*}
\]

Code Generation for Expressions on Stack Machines

- To evaluate a variable load its value.
- To evaluate a constant push its literal value.
- To evaluate an expression
  - Evaluate left operand
  - Evaluate right operand
  - Apply operator

Quadruples for Expressions

- Create new temporaries for each intermediate results: infinite number of virtual registers.
- Better model: assume finite number of registers
  - Select one register to hold result
  - Compute first operand into reserved register R1
  - Compute second operand using remaining registers
  - Compute result into R1
- To minimize number of registers needed, compute larger expression first.
- Simple implementation: use stack for available registers.
Aho-Sethi Algorithm for Minimal Registers Use

- For a constant: \texttt{return 1}
- For a variable: \texttt{return 1}
- For an expression \texttt{arg1 op arg2}:
  - Let \texttt{min1 = minimum for arg1}
  - Let \texttt{min2 = minimum for arg2}
  - If \texttt{min1 \neq min2 then return max(min1, min2)}
  - \texttt{else return min1 + 1}
- Optimal register use:
  - Compute weight of each node
  - At each step, compute subtree with largest weight

Example

- \texttt{b^2 - 4 a c} needs 3 registers
  - Naive left to right
    - \texttt{load b, R1 load b, R2 mul R1, R2, R1}
    - \texttt{load 4, R2}
    - \texttt{load a, R2 load c, R4 mul R2, R3, R2}
    - \texttt{mul R3, R4, R3 load 4, R3}
    - \texttt{mul R2, R3, R2 sub R1, R2 sub R1, R2}
  - Optimal
    - \texttt{load b, R1 load b, R2 mul R1, R2, R1}
    - \texttt{Load 4, R2 Load a, R2}
    - \texttt{Load c, R3 Load c, R4 mul R2, R3, R2}
    - \texttt{mul R3, R4, R3 Load 4, R3}
    - \texttt{mul R2, R3, R2 sub R1, R2 sub R1, R2}

Code Generation for More Complex Constructs

- Tree transformations
  - exponentiation
- Inline expansion
  - dispatching calls
- Calls to run-time routines
  - Storage management
  - 64-bit arithmetic
  - threads and tasks
  - calendar, time

Exponentiation

- Simple cases are computed efficiently:
  - \texttt{Y := x**2; Y = x * x}
  - \texttt{Y := x**4; T1 = x * x Y = T1 * T1}
- General case requires run-time support:
  - \texttt{Y := x**n; Y = exp_float(x, n)}
  - \texttt{exp_float} is part of runtime library linked with user program.