Intermediate code generation

Honors Compilers
April 2\textsuperscript{nd}, 2002
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_statements
else
  else_statements;
end if;

t1 = cond
if not t1 goto else_label
then_statements

goto endif_label
else_label:
  quadruples for
  else_statements
endif_label:

Generate labels
Generate quadruples for some descendant node
Place label in code
May need label for then_statements (short-circuit conditions)
Intermediate code for elsif parts

For each alternative, place in code the current else_label, and generate a new one. All alternatives inherit the end label from parent.

\[\text{cond then s1}\]
\[t1 = \text{cond1}\]
\[\text{if not } t1 \text{ goto else_label1}\]
\[\text{quadruples for s1}\]
\[\text{goto endif_label}\]
\[\text{elsif}\]
\[\text{cond2 then s2}\]
\[t2 = \text{cond2}\]
\[\text{if not } t2 \text{ goto else_label1}\]
\[\text{quadruples for s2}\]
\[\text{goto endif_label}\]
\[\text{else}\]
\[s4\]
\[\text{else_label2:}\]
\[\text{quadruples for s4}\]
\[\text{ endif_label:}\]
Code generation for while loops

Generate two labels: start_loop, end_loop

while (cond) {
    if (!cond) goto end_loop
    s1;
    if (cond2) break;
    s2;
    if (cond3) continue;
    s3;
}

start_loop:
    if (!cond) goto end_loop
    quadruples for s1
    if (cond2) goto end_loop
    quadruples for s2
    if (cond3) goto start_loop
    quadruples for s3
    goto start_loop
Intermediate code for numeric loops

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

\[ \text{for } j \text{ in expr1..expr2 loop } \]
\[ \text{start_label: } \]
\[ \text{if } j > \text{expr2 goto end_label} \]
\[ \text{end_label: } \]
\[ \text{gotogoto start_label start_label} \]
\[ S1 \]
\[ \text{end loop;} \]
\[ \text{quadruples for S1} \]
\[ j = j + 1 \]
\[ \text{goto start_label} \]
\[ \text{end_label: } \]
Place test at end to utilize loop instruction

```plaintext
K 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

end_label:
  goto start_label:
\}
```

Intermediate 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-circuits
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 table label1, label2

... jumpi x table
  label1:
    y = 0
    goto end_case
  label2:
    y = 1
    goto end_case
end_case:

case x is
  when up: y := 0;

  when down: y := 1;
end case;
Code generation for expressions: stack machines

Zero-address instructions: push, pop, arithmetic

Binary operations remove two entries, push one.

\[ d = b^2 - 4a \cdot c \]

load b
dupl
mult
push a
push c
mult
push_const 4
mult
sub
store d

-- load from memory
-- duplicate value on top of stack
-- push constant on stack
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 result:

infinite number of 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.
Computing the minimum number of registers (Aho-Sethi)

For a constant: return 1
For a variable: return 1
For a tree:

- min1 = minimum for left_operand
- min2 = minimum for right_operand
- if min1 /= min2 then return max (min1, min2)
- else return min1 + 1

Optimal register use:

- Compute weight of each node
- At each step, compute subtree with larger weight
example

\[ b^2 - 4ac \] needs 3 registers.

left-to-right
load \( b \), R1
load \( b \), R2
mul R1, R2, R1
load 4, R2
load a, R3
load c, R4
mul R3, R4, R3
mul R2, R3, R2
sub R1, R2

optimal
load \( b \), R1
load \( b \), R2
mul R1, R2, R1
load a, R2
load c, R3
mul R2, R3, R2
load 4, R3
mul R2, R3, R2
sub R1, R2
Code generation for more complex constructs

Tree transformations
  • aggregates
  • exponentiation

Inline expansion
  • dispatching calls

Calls to run-time routines
  • storage management
  • 64-bit arithmetic
  • threads and tasks
  • calendar, time,
Aggregates

A : array (1 .. N, 1 .. N) of integer;  --  N non-static
A := ( (1 => 0, others => 15), others => (others => -1))

Becomes:

A (1, 1) := 0;
for J in 2 .. N loop A (1, J) := 15;
  for J in 2 .. N
    for K in 1 .. N loop
      A (J, K) := -1;
    end loop;
  end loop;
end loop;
Exponentiation

Simple cases are computed efficiently:

\[
\begin{align*}
Y & := x ^{ \ast 2}; \\
Y & := x ^{ \ast 4}; \\
x & = x \times x
\end{align*}
\]

General case requires run-time support:

\[
\begin{align*}
Y & := x ^{ \ast n}; \\
Y & = \exp \_float \left(x, n\right)
\end{align*}
\]

\exp \_float is part of runtime library linked with user program
Dispatching calls

class Thing { virtual void modify (int x) ...

Thing* this_one ();
...

this_one => modify (42);

Modify has an implicit parameter (this)
Call depends on runtime class of designated object
Each object has a pointer to vtabl.
Each virtual method has a known offset in vtabl
Call becomes:

(this_one=>vtabl (7)) (this_one, 42);

Multiple inheritance requires more elaborate data structure
Interface calls in Java

Different classes can implement an interface method.

The method has the same name and signature, but a different position in the vtabl of each class.

Locating the method during dispatching must be done by sequential search and string comparison:

runtime code

Just-in-time compilation (JIT) can cache the result of such a search for subsequent use.