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
  quadruples for cond, result in t1
  if not t1 goto else_label
  then_statement
  quadruples for then_statement
else
  goto endif_label
else_label:
  else_statements;
  quadruples for else_statements
endif_label:

• 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**

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

```
if cond1 then S1  
  t1 := cond1
  if not t1 goto else_label1
  quadruples for S1
  goto endif_label

elsif cond2 then S2
  else_label1:
  t2 := cond2
  if not t2 goto else_label2
  quadruples for S2
  goto endif_label

else S3
  else_label2:
  quadruples for S3
  endif_label;
end if;
```
More Details of Code Generation

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

```plaintext
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
```
Example of Generation

Consider the statement

\[
\text{if } C_1 \text{ then } \begin{cases} 
\text{if } C_2 \text{ then } x := 1 \text{ else } x := 2 \\
\text{else } \begin{cases} 
\text{if } C_3 \text{ then } x := 3 \text{ else } x := 4 
\end{cases}
\end{cases}
\]

The generated code is given by

\[
\text{if } \neg C_1 \text{ goto L1} \\
\text{if } \neg C_2 \text{ goto L3} \\
\quad x := 1 \\
\quad \text{goto L4} \\
\text{L3:} \\
\quad x := 2 \\
\text{L4:} \\
\quad \text{goto L2} \\
\text{L1:} \\
\quad \text{if } \neg C_3 \text{ goto L5} \\
\quad x := 3 \\
\quad \text{goto L6} \\
\text{L5:} \\
\quad x := 4 \\
\text{L6:} \\
\text{L2:}
\]
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

  ... t1 := code_gen(node.child1, 0);
  issue(if \neg 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
```

Honors Compilers, NYU, Spring, 2007
Example of an Improved Generation

Reconsider the statement

\[
\text{if } C_1 \text{ then } \{ \text{if } C_2 \text{ then } x := 1 \text{ else } x := 2 \} \\
\text{else } \{ \text{if } C_3 \text{ then } x := 3 \text{ else } x := 4 \}
\]

The improved generated code is given by

\[
\begin{align*}
\text{if } !C_1 & \text{ goto L1} \\
\text{if } !C_2 & \text{ goto L3} \\
& x := 1 \\
& \text{goto L2} \\
\text{L3:} & \quad x := 2 \\
& \text{goto L2} \\
\text{L1:} & \quad \text{if } !C_3 \text{ goto L5} \\
& x := 3 \\
& \text{goto L2} \\
\text{L5:} & \quad x := 4 \\
& \text{goto L2} \\
\text{L2:}
\end{align*}
\]
**Code Generation for While Loops**

Generate two labels: `start_loop`, `end_loop`

```plaintext
while (cond) {
    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
```

```
end_loop:
```
Intermediate Code for Numeric Loops

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

```plaintext
for J in expr1..expr2 loop
    J := expr1
    start_label:
    if J > expr2 goto end_label
    quadruples for S1
    J := J+1
    goto start_label
end loop;
```

Honors Compilers, NYU, Spring, 2007

A. Pnueli
Place Test at End to Utilize Loop Instruction

for J in expr1..expr2 loop
  t1 := expr1
  t2 := expr2
  K := t1 - 1
  goto test_label
  start_label:
  quadruples for S1
  test_label:
  \[
  \begin{cases}
    K := K + 1 \\
    \text{if } K > t2 \text{ goto end_label} \\
    \text{goto start_label}
  \end{cases}
\]
end_label:
Code for Short-Circuit Expressions

- Short-circuit expressions are treated as control structures

- \texttt{if B1 or else B2 then S1 ...} — \texttt{if (B1 \parallel B2) \{S1...\}}
  
  - \texttt{if B1 goto then\_label}
  
  - \texttt{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.

```

table label1, label2

... 

case x is 
when up: y := 0;
label1:
y := 0
goto end_case

when down: y := 1
label2:
y := 1
goto end_case

end case;
```

Honors Compilers, NYU, Spring, 2007
Evaluating Expressions: Stack Machines

- Zero-address instructions: push, pop, arithmetic
- Binary operations remove two operands, push result.
- \( d := b^2 - 4ac \)
  
  load b      \quad \quad \text{-- load from memory}
  
  dupl      \quad \quad \text{-- duplicate value on top of stack}
  
  mult
  
  push a
  
  push c
  
  mult
  
  push\_const 4 \quad \quad \text{-- push explicit constant}
  
  mult
  
  sub
  
  store d
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: return 1
- For a variable: return 1
- For an expression \( \text{arg1 op arg2} \):
  - Let \( \text{min1} = \text{minimum for arg1} \)
  - Let \( \text{min2} = \text{minimum for arg2} \)
  - If \( \text{min1} \neq \text{min2} \) then return \( \max(\text{min1}, \text{min2}) \)
  - else return \( \text{min1} + 1 \)
- Optimal register use:
  - Compute weight of each node
  - At each step, compute subtree with largest weight
Example

- \( b^2 - 4ac \) needs 3 registers

**Naive 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
- 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
  - 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:
  \[ Y := x^{**}2; \quad Y = x \times x \]
  \[ Y := x^{**}4; \quad T1 = x \times x \]
  \[ Y = T1 \times T1 \]

- General case requires run-time support:
  \[ Y := x^{**}n; \quad Y = \text{exp\_float}(x,n) \]
  \[
  \text{exp\_float is part of runtime library linked with user program.}
  \]