CS415 Compilers

Code Generation

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
Review - Generating Code for Expressions

The concept

- Use a simple treewalk evaluator
- Bury complexity in routines it calls
  - base(), offset(), & val()
- Implements expected behavior
  - Visits & evaluates children
  - Emits code for the op itself
  - Returns register with result
- Works for simple expressions
- Easily extended to other operators
- Does not handle control flow

```
expr(node) {
    int result, t1, t2;
    switch (type(node)) {
        case ×,÷,+,−:
            t1 ← expr(left child(node));
            t2 ← expr(right child(node));
            result ← NextRegister();
            emit (op(node), t1, t2, result);
            break;
        case IDENTIFIER:
            t1 ← base(node);
            t2 ← offset(node);
            result ← NextRegister();
            emit (loadAO, t1, t2, result);
            break;
        case NUMBER:
            result ← NextRegister();
            emit (loadI, val(node), none, result);
            break;
    }
    return result;
}
```

“Expr” returns virtual register number that will contain result of subtree evaluation at runtime.
Example:

```
+ -> loadI @x ⇒ r1
  x
```

```
+ -> loadAO r0, r1 ⇒ r2
  y
```

```
expr("x") →
loadI @x ⇒ r1
loadAO r0, r1 ⇒ r2
```

```
expr("y") →
loadI @y ⇒ r3
loadAO r0, r3 ⇒ r4
```

```
NextRegister() → r5
```

```
emit(add,r2,r4,r5) →
add r2, r4 ⇒ r5
```

```
expr(node) {
  int result, t1, t2;
  switch (type(node)) {
    case ×,÷,+,−:
      t1 ← expr(left child(node));
      t2 ← expr(right child(node));
      result ← NextRegister();
      emit(op(node), t1, t2, result);
      break;
    case IDENTIFIER:
      t1 ← base(node);
      t2 ← offset(node);
      result ← NextRegister();
      emit(loadAO, t1, t2, result);
      break;
    case NUMBER:
      result ← NextRegister();
      emit(loadI, val(node), none, result);
      break;
  }
  return result;
}
```
Review - Generating Code for Expressions

expr(node) {
    int result, t1, t2;
    switch (type(node)) {
        case ×, ÷, +, − :
            t1 ← expr(left child(node));
            t2 ← expr(right child(node));
            result ← NextRegister();
            emit (op(node), t1, t2, result);
            break;
        case IDENTIFIER:
            t1 ← base(node);
            t2 ← offset(node);
            result ← NextRegister();
            emit (loadAO, t1, t2, result);
            break;
        case NUMBER:
            result ← NextRegister();
            emit (loadI, val(node), none, result);
            break;
    }
    return result;
}

Example:

```
Expr: - x × 2 y
```

Generates:

```
loadl @x ⇒ r1
loadAO r0, r1 ⇒ r2
loadl 2 ⇒ r3
loadl @y ⇒ r4
loadAO r0, r4 ⇒ r5
mult r3, r5 ⇒ r6
sub r2, r6 ⇒ r7
```
More complex cases for IDENTIFIER

- What about values in registers?
  - Modify the IDENTIFIER case
  - Already in a register ⇒ return the register name
  - Not in a register ⇒ load it as before, but record the fact
  - Choose names to avoid creating false dependences ("fresh" virtual register)

- What about parameter values?
  - Many linkages pass the first several values in registers
  - Call-by-value ⇒ just a local variable with "funny" offset
  - Call-by-reference ⇒ needs an extra indirection

- What about function calls in expressions?
  - Generate the calling sequence & load the return value
  - Severely limits compiler’s ability to reorder operations
    (We will learn about this later)
Adding other operators (in addition to +, -, *, /)
- Evaluate the operands, then perform the operation
- Complex operations may turn into library calls
- Handle assignment as an operator

**Mixed-type expressions**
- Insert conversions as needed from conversion table
- Most languages have symmetric & rational conversion tables

<table>
<thead>
<tr>
<th>Typical Addition Table</th>
<th>+</th>
<th>Integer</th>
<th>Real</th>
<th>Double</th>
<th>Complex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Integer</td>
<td>Real</td>
<td>Double</td>
<td>Complex</td>
<td></td>
</tr>
<tr>
<td>Real</td>
<td>Real</td>
<td>Real</td>
<td>Double</td>
<td>Complex</td>
<td></td>
</tr>
<tr>
<td>Double</td>
<td>Double</td>
<td>Double</td>
<td>Double</td>
<td>Complex</td>
<td></td>
</tr>
<tr>
<td>Complex</td>
<td>Complex</td>
<td>Complex</td>
<td>Complex</td>
<td>Complex</td>
<td></td>
</tr>
</tbody>
</table>
What about evaluation order?
• Can use commutativity & associativity to improve code
• This problem is truly hard

What about order of evaluating operands?
• 1\textsuperscript{st} operand must be preserved while 2\textsuperscript{nd} is evaluated
• Takes an extra register for 2\textsuperscript{nd} operand
• Should evaluate more demanding operand expression first
  (Ershov in the 1950’s, Sethi in the 1970’s)

Taken to its logical conclusion, this creates Sethi-Ullman register allocation scheme (ASU p. 571)
Need to generate an initial IR form

- Might generate an AST, use it for some high-level, near-source work (type checking, optimization), then traverse it and emit a lower-level IR similar to ILOC

The big picture

- Recursive $expr(node)$ algorithm really works bottom-up
  - Actions on non-leaves occur after children are done
- Can encode same basic structure into ad-hoc SDT scheme
  - Identifiers load themselves & stack (live in parser stack) virtual register name
  - Operators emit appropriate code & stack resulting VR name
  - Assignment requires evaluation to an lvalue or an rvalue
expr(node) {
    int result, t1, t2;
    switch (type(node)) {
        case ×,÷,+,−:
            t1 ← expr(left child(node));
            t2 ← expr(right child(node));
            result ← NextRegister();
            emit(op(node), t1, t2, result);
            break;
        case IDENTIFIER:
            t1 ← base(node);
            t2 ← offset(node);
            result ← NextRegister();
            emit(loadAO, t1, t2, result);
            break;
        case NUMBER:
            result ← NextRegister();
            emit(loadI, val(node), none, result);
            break;
    }
    return result;
}

Goal :  Expr { $$ = $1; } ;
Expr:  Expr PLUS Term
    { t = NextRegister();
      emit(add,$1,$3,t); $$ = t; }
    | Expr MINUS Term { … } 
    | Term { $$ = $1; } ;
Term:  Term TIMES Factor
    { t = NextRegister();
      emit(mult,$1,$3,t); $$ = t; }
    | Term DIVIDES Factor { … } 
    | Factor { $$ = $1; } ;
Factor: NUMBER
    { t = NextRegister();
      emit(loadI, val(node), none, t );
      $$ = t; }
    | ID
    { t1 = base($1);
      t2 = offset($1);
      t = NextRegister();
      emit(loadAO,t1,t2,t);
      $$ = t; }
Handling Assignment  
(just another operator)

\[ lhs \leftarrow rhs \]

Strategy

- Evaluate \( rhs \) to a value \((an\ rvalue)\)
- Evaluate \( lhs \) to a location \((an\ lvalue)\)
  - \( lvalue \) is a register \(\Rightarrow\) move \( rhs \)
  - \( lvalue \) is an address \(\Rightarrow\) store \( rhs \)
- If \( rvalue \) & \( lvalue \) have different types
  - Evaluate \( rvalue \) to its “natural” type
  - Convert that value to the type of \(* lvalue\)

Unambiguous scalars may go into registers (no aliasing)
Ambiguous scalars or aggregates go into memory (possible aliasing)
Handling Assignment

What if the compiler cannot determine the rhs’s type?

- This is a property of the language & the specific program
- If type-safety is desired, compiler must insert a run-time check
- Add a *tag field* to the data items to hold type information

**Code for assignment becomes more complex**

```plaintext
evaluate rhs
if type(lhs) ≠ rhs.tag
    then
        convert rhs to type(lhs) or signal a run-time error
    lhs ← rhs
```

This is much more complex than if it knew the types
Compile-time type-checking
• Goal is to eliminate both the check & the tag
• Determine, at compile time, the type of each subexpression
• Use compile-time types to determine if a run-time check is needed

Optimization strategy
• If compiler knows the type, move the check to compile-time
• Unless tags are needed for garbage collection, eliminate them
• If check is needed, try to overlap it with other computation (superscalar or multi-core architectures)
Handling Assignment (with reference counting)

The problem with reference counting

- Must adjust the count on each pointer assignment
- Overhead is significant, relative to assignment

Code for assignment becomes

```plaintext
evaluate rhs
lhs→count ← lhs→count - 1
lhs ← addr(rhs)
rhs→count ← rhs→count + 1
```

This adds 1+, 1-, 2 loads, & 2 stores

With extra functional units & large caches, this may become either cheap or free.
How does the compiler handle $A[i,j]$?

First, must agree on a storage scheme

**Row-major order**  
(largest languages)  
Lay out as a sequence of consecutive rows  
Rightmost subscript varies fastest  
$A[1,1], A[1,2], A[1,3], A[2,1], A[2,2], A[2,3]$  

**Column-major order**  
(Fortran)  
Lay out as a sequence of columns  
Leftmost subscript varies fastest  
$A[1,1], A[2,1], A[1,2], A[2,2], A[1,3], A[2,3]$  

**Indirection vectors**  
(Java)  
Vector of pointers to pointers to ... to values  
Takes much more space, trades indirection for arithmetic  
Not amenable to analysis
Laying Out Arrays

The Concept

Row-major order

Column-major order

Indirection vectors

These have distinct & different cache behavior
Computing an Array Address

Declaration: \( A[\text{low .. high}] \) of ...

\[ A[\ i \ ] \]
- \( @A + (i - \text{low}) \times \text{sizeof}(A[1]) \)
- In general: base(A) + (i - low) \times \text{sizeof}(A[1])
Computing an Array Address

Declaration: \( A[\text{low .. high}] \) of ...

\[ A[\ i\ ] \]
- \( @A + (i - \text{low}) \times \text{sizeof}(A[1]) \)
- In general: \( \text{base}(A) + (i - \text{low}) \times \text{sizeof}(A[1]) \)

\[ \text{int } A[1:10] \Rightarrow \text{low is } 1 \]
Make low 0 for faster access (saves a -)

Almost always a power of 2, known at compile-time \( \Rightarrow \) use a shift for speed
Computing an Array Address

Declaration: $A[\text{low1} .. \text{high1}, \text{low2} .. \text{high2}]$ of ...

$A[i]$
- $@A + (i - \text{low}) \times \text{sizeof}(A[1])$
- In general: base($A) + (i - \text{low}) \times \text{sizeof}(A[1])$

What about $A[i_1, i_2]$?

Row-major order, two dimensions
- $@A + ((i_1 - \text{low}_1) \times (\text{high}_2 - \text{low}_2 + 1) + i_2 - \text{low}_2) \times \text{sizeof}(A[1])$

Column-major order, two dimensions
- $@A + ((i_2 - \text{low}_2) \times (\text{high}_1 - \text{low}_1 + 1) + i_1 - \text{low}_1) \times \text{sizeof}(A[1])$

Indirection vectors, two dimensions
- $*(A[i_1])[i_2]$ — where $A[i_1]$ is, itself, a 1-d array reference

This stuff looks expensive!
Lots of implicit $+ , - , \times$ ops
In row-major order

\[ @A + (i - low_1) \times (high_2 - low_2 + 1) \times w + (j - low_2) \times w \]

where \( w = \text{sizeof}(A[1,1]) \)

Which can be factored into

\[ @A + i \times (high_2 - low_2 + 1) \times w + j \times w \]

\[ - (low_1 \times (high_2 - low_2 + 1) \times w) - (low_2 \times w) \]

If \( low_i, high_i, \) and \( w \) are known, the last term is a constant

Define \( @A_0 \) as

\[ @A - (low_1 \times (high_2 - low_2 + 1) \times w - low_2 \times w \]

And \( len_2 \) as \((high_2 - low_2 + 1)\)

Then, the address expression becomes

\[ @A_0 + (i \times len_2 + j) \times w \]

Compile-time constants
How should the compiler represent them?
• Answer depends on the target machine

Two classic approaches
• Numerical representation
• Positional (implicit) representation

Correct choice depends on both context and ISA
Boolean & Relational Values

Numerical representation

- Assign values to TRUE and FALSE
- Use hardware AND, OR, and NOT operations
- Use comparison to get a boolean from a relational expression

Examples

\[
\begin{align*}
  x < y & \quad \text{becomes} \quad \text{cmp\_LT} \ r_x,r_y \Rightarrow r_1 \\
\text{if} \ (x < y) \quad \text{then stmt}_1 & \quad \text{becomes} \quad \text{cmp\_LT} \ r_x,r_y \Rightarrow r_1 \\
\text{else stmt}_2 & \quad \text{becomes} \quad \text{cbr} \ r_1 \Rightarrow \text{stmt}_1,\text{stmt}_2
\end{align*}
\]
What if the ISA uses a condition code?

- Must use a conditional branch to interpret result of compare
- Necessitates branches in the evaluation

Example:

```
cmp r_x, r_y ⇒ cc_1
\text{cbr}_{LT} cc_1 \rightarrow L_T, L_F
\text{x < y becomes}
```

\begin{align*}
L_T: & \text{loadI 1} \Rightarrow r_2 \\
 & \text{br} \rightarrow L_E \\
L_F: & \text{loadI 0} \Rightarrow r_2 \\
L_E: & \text{…other stmts…}
\end{align*}

This “positional representation” is much more complex
What if the ISA uses a condition code?
• Must use a conditional branch to interpret result of compare
• Necessitates branches in the evaluation

Example:

\[
\text{x < y \ becomes \ } \begin{array}{rl}
\text{cmp} & r_x, r_y \rightarrow cc_1 \\
\text{cbr}_{LT} & cc_1 \rightarrow L_T, L_F \\
\text{L}_T: \text{loadI} & 1 \Rightarrow r_2 \\
\text{br} & \rightarrow L_E \\
\text{L}_F: \text{loadI} & 0 \Rightarrow r_2 \\
\text{L}_E: \text{...other stmts...} \\
\end{array}
\]

Condition codes
• are an architect’s hack
• allow ISA to avoid some comparisons
  \((x < y \lor y < z)\)
• complicates code for simple cases

This “positional representation” is much more complex
Boolean & Relational Values

The last example actually encodes result in the PC
If result is used to control an operation, this may be enough

<table>
<thead>
<tr>
<th>Variations on the ILOC Branch Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Straight Condition Codes</strong></td>
</tr>
<tr>
<td>comp</td>
</tr>
<tr>
<td>cbr_LT</td>
</tr>
<tr>
<td>$L_1$: add</td>
</tr>
<tr>
<td>br</td>
</tr>
<tr>
<td>$L_2$: add</td>
</tr>
<tr>
<td>br</td>
</tr>
<tr>
<td>$L_{OUT}$: nop</td>
</tr>
</tbody>
</table>

Condition code version does not directly produce $(x < y)$
Boolean version does
Still, there is no significant difference in the code produced
Conditional move & predication both simplify this code

<table>
<thead>
<tr>
<th>Example</th>
<th>Conditional Move</th>
<th>Predicated Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(x &lt; y)$</td>
<td>$\text{comp } r_x, r_y \Rightarrow cc_1$</td>
<td>$\text{cmp}_{LT} r_x, r_y \Rightarrow r_1$</td>
</tr>
<tr>
<td>then $a \leftarrow c + d$</td>
<td>$\text{add } r_c, r_d \Rightarrow r_1$</td>
<td>$(r_1)? \text{ add } r_c, r_d \Rightarrow r_a$</td>
</tr>
<tr>
<td>else $a \leftarrow e + f$</td>
<td>$\text{add } r_e, r_f \Rightarrow r_2$</td>
<td>$(\neg r_1)? \text{ add } r_e, r_f \Rightarrow r_a$</td>
</tr>
<tr>
<td>$i2i_&lt;$</td>
<td>$\text{cc}_1, r_1, r_2 \Rightarrow r_a$</td>
<td></td>
</tr>
</tbody>
</table>

Both versions avoid the branches
Both are shorter than $CC$s or Boolean-valued compare
Are they better?
Consider the assignment $x \leftarrow a < b \land c < d$

<table>
<thead>
<tr>
<th>Variations on the ILOC Branch Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Straight Condition Codes</strong></td>
</tr>
<tr>
<td>comp $r_a, r_b \Rightarrow cc_1$</td>
</tr>
<tr>
<td>cbr_LT $cc_1 \rightarrow L_1, L_2$</td>
</tr>
<tr>
<td>(L_1): comp $r_c, r_d \Rightarrow cc_2$</td>
</tr>
<tr>
<td>(L_1): cbr_LT $cc_2 \rightarrow L_3, L_2$</td>
</tr>
<tr>
<td>(L_2): loadI 0 \Rightarrow r_x \rightarrow L_{OUT}</td>
</tr>
<tr>
<td>(L_2): br \rightarrow L_{OUT}</td>
</tr>
<tr>
<td>(L_3): loadI 1 \Rightarrow r_x \rightarrow L_{OUT}</td>
</tr>
<tr>
<td>(L_3): br \rightarrow L_{OUT}</td>
</tr>
<tr>
<td>(L_{OUT}): nop</td>
</tr>
</tbody>
</table>

Here, the boolean compare produces much better code.
Consider the assignment $x \leftarrow a < b \land c < d$

**Variations on the ILOC Branch Structure**

<table>
<thead>
<tr>
<th>Straight Condition Codes</th>
<th>Boolean Compare</th>
</tr>
</thead>
</table>
| ```
comp $r_a, r_b \Rightarrow cc_1$
cbr_LT $cc_1 \rightarrow L_1, L_2$
L1: comp $r_c, r_d \Rightarrow cc_2$
cbr_LT $cc_2 \rightarrow L_3, L_2$
L2: loadI $0 \Rightarrow r_x$
br $\rightarrow L_{OUT}$
L3: loadI $1 \Rightarrow r_x$
br $\rightarrow L_{OUT}$
LOUT: nop |
| ```
cmp_LT $r_a, r_b \Rightarrow r_1$
cmp_LT $r_c, r_d \Rightarrow r_2$
and $r_1, r_2 \Rightarrow r_x$

Potential Problem: Optimized Boolean expression (short circuiting)

Here, the boolean compare produces much better code
If-then-else

- Follow model for evaluating relational & boolean with branches

Branching versus predication

- Frequency of execution
  - Uneven distribution $\Rightarrow$ do what it takes to speed common case
- Control flow inside the construct
  - Any branching activity within the case base complicates the predicates and makes branches attractive
Loops
- Evaluate condition before loop (if needed)
- Evaluate condition after loop
- Branch back to the top (if needed)
Merges test with last block of loop body

\textbf{while}, \textbf{for}, \textbf{do}, & \textbf{until} all fit this basic model
for (i = 1; i < 100; i++) {
  body
  next statement
}

Initialization

Pre-test

Post-test
Many modern programming languages include a break
• Exits from the innermost control-flow statement
  → Out of the innermost loop
  → Out of a case statement

Translates into a jump
• Targets statement outside control-flow construct
• Creates multiple-exit construct
• Skip/continue in loop goes to next iteration
Case Statements
1. Evaluate the controlling expression
2. Branch to the selected case
3. Execute the code for that case
4. Branch to the statement after the case
Parts 1, 3, & 4 are well understood, part 2 is the key
Control Flow

Case Statements
1. Evaluate the controlling expression
2. Branch to the selected case
3. Execute the code for that case
4. Branch to the statement after the case  \(\text{(use break)}\)

Parts 1, 3, & 4 are well understood, part 2 is the key

Strategies
- Linear search (nested if-then-else constructs)
- Build a table of case expressions & binary search it
- Directly compute an address (requires dense case set)

Surprisingly many compilers do this for all cases!
Optimizations
Read EaC: Chapter 7.8, 8.1 – 8.5