Boolean and Relational Expressions
Comp 412

Copyright 2019, Keith D. Cooper & Linda Torczon, all rights reserved.
Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.
Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.
What is Left in Translation?

We have talked about the mechanism of translation. Next, we need to focus on the product, of translation.

Expressions
- Type-related issues in expressions
  - Mixed-type operations and value conversions.
- Assignment
- More operators in expressions
  - Boolean expressions, relational expressions, string operations

Control Flow
- Code shape and translation strategies

Procedure and Function Calls
- Functionality: what, precisely, does a procedure call do?
- Implementation: how does the compiler emit code to do that?

A little more on arrays today. Strings must wait until after control flow.
Scheme to Address a Structure Element

A structure aggregates a collection of fields into one named object

- Structures are a convenient way to organize data
- Structures are often used to specify a layout for some interface
  - e.g., an I/O control block for a file or a tuple for a return value

The access method for a structure element is a base and an offset

- Base address depends on declaration, scope, & lifetime
  - If it is a declared instance, then it is in a data area for the declaring scope
    - Local data area, static data area, object’s object record, some global area
  - If it is a heap allocated instance, then it’s base address is in some reference
    that the programmer designed and created
    - A named variable, the next or previous link in some list node, ...

- Offset is derived from the structure’s layout & stored in a symbol table
Scheme to Address a Structure Element

Structure Layout

• A structure’s layout is **usually** dictated by its declaration  \( (PL \, decision) \)
  – Programmer control of layout is critical for consistency in interfaces
• Compiler computes an offset from the declaration

```
struct OPERATION {
    int Opcode;
    int Line;
    int Ordinal;
    int LiveNow;
    struct ARG Operands[3];
    struct OPERATION* Next;
};
```

### Declaration from lab2_ref

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>int</td>
<td>0</td>
</tr>
<tr>
<td>Line</td>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>Ordinal</td>
<td>int</td>
<td>8</td>
</tr>
<tr>
<td>LiveNow</td>
<td>int</td>
<td>12</td>
</tr>
<tr>
<td>Operands</td>
<td>struct ARG *</td>
<td>16</td>
</tr>
<tr>
<td>Operation</td>
<td>struct OPERATION *</td>
<td>24</td>
</tr>
</tbody>
</table>

Assumes `int` is 4 bytes (word-aligned) and `*` is 8 bytes (double-word aligned)

Each structure needs a small symbol table (**YAST**)
Scheme to Address a Structure Element

But wait. That leads to wasted space!

- **Yes.** It does. Values must be padded to their alignment restrictions.
- Wasted space is the cost of providing the programmer with control over structure layout.
- What if alignment does not match my needs? *hardware control block?*
  - If the language does not support your needs, **change languages** or **cheat**.

```c
struct FEE {
    int a;
    long int b;
    char c;
    int *d;
};
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>int</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>long int</td>
<td>8</td>
</tr>
<tr>
<td>c</td>
<td>char</td>
<td>16</td>
</tr>
<tr>
<td>d</td>
<td>long int</td>
<td>24</td>
</tr>
</tbody>
</table>

Assumes **char** is 1 byte (byte-aligned), **int** is 4 bytes (word-aligned) and ***** is 8 bytes (double-word aligned).
Scheme to Address an Object

An object is just a structure \((to a \, 1^{st} \, approximation)\)

- Base address depends on declaration, scope, & lifetime
  - If it is a declared instance, then it is in a data area for the declaring scope
  - If it is an allocated instance, then it’s base address is in some reference that the programmer designed and created

- Offsets are computed from the class definition and hierarchy
  - Compiler is, typically, not bound to lay out object in declaration order
  - Compiler will use a consistent order for interoperability
    \(\rightarrow \text{Adopt rules regarding object layout from PL or from convention (more later)}\)

```java
Class Point {
    public int x, y;
    public void draw();
}
Class ColorPoint extends Point {
    Color c;
    public void draw() {...}
    public void test() { y = x; draw(); }
}
```

These are conceptual drawings. A real implementation would store code member pointers in the class’ OR.
What About an Array of Structures?

An array of structures is just an array with a non-standard element size

• Language must specify array layout
  – Row major, column major, indirection vectors

• Base address of array instance is found with the address polynomial
  – Many of the optimizations still apply (but, likely, not shift for multiply)

• Structure alignment restrictions may dictate internal padding between elements
  – The compiler cannot always win. See thermodynamics.
What About Strings?

String operators usually require control flow

• Iteration over the characters in a string
• Tests for string overflow, for nil characters, ...

Strings will have to wait until after loops and conditional control flow
What is Left in Translation?

We have talked about the mechanism of translation. Next, we need to focus on the product, of translation.

Expressions

• Type-related issues in expressions
  – Mixed-type operations and value conversions.

• Assignment

• More operators in expressions
  – Boolean expressions, relational expressions, string operations

Control Flow

• Code shape and translation strategies

Procedure and Function Calls

• Functionality: what, precisely, does a procedure call do?
• Implementation: how does the compiler emit code to do that?
Extending the Grammar

For control-flow operations, we need Boolean & relational expressions

• +, -, ×, and ÷ are not enough
• Need both operators in the grammar & a scheme to translate them

Defining the syntax

• Additional productions
• Additional levels of precedence

See, for example, Fig. 7.7 on p. 351 of EaC2e.
More operators, more levels of precedence

<table>
<thead>
<tr>
<th>Boolean</th>
<th>→</th>
<th>Boolean V AndTerm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>AndTerm</td>
</tr>
<tr>
<td>AndTerm</td>
<td>→</td>
<td>AndTerm ∧ RelExpr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RelExpr</td>
</tr>
<tr>
<td>RelExpr</td>
<td>→</td>
<td>RelExpr &lt; Expr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RelExpr ≤ Expr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RelExpr = Expr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RelExpr ≠ Expr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RelExpr ≥ Expr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RelExpr &gt; Expr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Expr</td>
</tr>
</tbody>
</table>

| Expr             | → | Expr + Term       |
|                  |   | Expr - Term       |
|                  |   | Term              |
| Term             | → | Term \( \times \) Value |
| Value            | → | ! Factor          |
| Factor           | → | ( Expr )          |
| number           |   | Reference         |

... where Reference derives a name, a subscripted name, a structure reference, a string reference, a function call, ...
Extending the Grammar

For control-flow operations, we need Boolean & relational expressions

• +, -, ×, and ÷ are not enough
• Need both operators in the grammar & a scheme to translate them

Defining the syntax

• Additional productions
• Additional levels of precedence

Code generation follows the scheme for existing arithmetic operators

• Evaluate the operands, then perform the operation
• Tricky question lies in how to represent the values
  – Some dependence on hardware mechanisms to create and use Boolean & relational values

See, for example, Fig. 7.7 on p. 351 of EaC2e.
Boolean and Relational Expressions

Translation can be pretty simple and formulaic

- Assign values to **true** and **false**
- Implement a translation scheme
  - *Binary operators, similar to arithmetic*

Numerical representation

- Assign values to **TRUE** and **FALSE** at design time
  - Common choices are 1 / 0, -1 / 0, or non-zero / 0
  - Not a lot of difference between the choices
- Use hardware **AND**, **OR**, and **NOT** operations
  ⇒ Make a choice and use it uniformly
Translation can be pretty simple and formulaic
• Assign values to true and false
• Implement a translation scheme

Significant optimization is possible, either in translation or post-translation
• Represent values implicitly with the PC
• Optimize evaluation
  – Short-circuit optimization, Boolean simplification, propagate inequalities and inequalities

Best choice depends on context, hardware, and available optimization
• At design time, you cannot make a choice that is always optimal
• The compiler cannot always win. See thermodynamics

Compiler writers tend to believe they can predict the future.
Relational Values

Different instruction sets (ISAs) provide different kinds of support

<table>
<thead>
<tr>
<th>Comparison Produces</th>
<th>Relational Values</th>
<th>Impact on Code Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean condition code</td>
<td>multiple compare ops</td>
<td>⇒ complexity into expressions</td>
</tr>
<tr>
<td></td>
<td>multiple branch ops</td>
<td>⇒ complexity into branches</td>
</tr>
</tbody>
</table>

Branch consumes what compare produces

- Fundamental design issue: operations intended to work together
- Most ISAs settle on a single model

Processor architect makes these choices

Compiler writer is expected to make the architect look smart.
**Boolean & Relational Values**

**Boolean-valued comparisons**
- Code can use the result of a relational comparison directly
- Presume that Boolean ops (e.g., **AND**, **OR**, **NOT**) work on result
- Leads to straightforward translation

**Examples**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x &lt; y )</td>
<td>becomes ( \text{cmp}_{\text{LT}} \ r_x, r_y \Rightarrow r_1 )</td>
</tr>
<tr>
<td><code>if (x &lt; y) then stmt_1 else stmt_2</code></td>
<td>becomes ( \text{cmp}<em>{\text{LT}} \ r_x, r_y \Rightarrow r_1 ) ( \text{cbr} \ r_1 \rightarrow \text{L}</em>\text{stmt_1},\text{L}_\text{stmt_2} )</td>
</tr>
</tbody>
</table>
Boolean & Relational Values

When compare produces a condition code

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

Example

<table>
<thead>
<tr>
<th>$x &lt; y$</th>
<th>becomes</th>
<th>$\text{cmp}$</th>
<th>$r_x, r_y$</th>
<th>$\Rightarrow$</th>
<th>$CC_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$\text{cbr_LT}$</td>
<td>$CC_1$</td>
<td>$\rightarrow$</td>
<td>$L_T, L_F$</td>
</tr>
<tr>
<td>$L_T$:</td>
<td>$\text{loadI}$</td>
<td>$\text{TRUE}$</td>
<td>$\Rightarrow$</td>
<td>$r_2$</td>
<td>$\rightarrow$</td>
</tr>
<tr>
<td></td>
<td>$\text{jump}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$L_F$:</td>
<td>$\text{loadI}$</td>
<td>$\text{FALSE}$</td>
<td>$\Rightarrow$</td>
<td>$r_2$</td>
<td>$\rightarrow$</td>
</tr>
<tr>
<td></td>
<td>$\text{jump}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| $L_E$:  |                  |              |            |               |        | ...

If \textbf{cmp} sets a condition-code register, rather than returning a Boolean value, the implementation of $x < y$ becomes much more complex.
Boolean & Relational Values

When compare produces a condition code

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

Example

If \((x < y)\) then \(stmt_1\) else \(stmt_2\)

becomes

\[
\begin{align*}
\text{cmp} & \quad r_x, r_y \quad \Rightarrow \quad CC_1 \\
cbr_{LT} & \quad CC_1 \quad \rightarrow \quad L_T, L_F \\
L_T: & \quad stmt_1 \\
& \quad \text{jump} \quad \rightarrow \quad L_E \\
L_F: & \quad stmt_2 \\
& \quad \text{jump} \quad \rightarrow \quad L_E \\
L_E: & \quad \ldots \text{other statements} \ldots
\end{align*}
\]

If the sole use of the relational expression’s value is to control an if-then-else construct, then we can fold the statements into the evaluation.
Boolean & Relational Values

When compare produces a condition code:

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

Example

If the sole use of the relational expression’s value is to control an if-then-else construct, then we can fold the statements into the evaluation.

\[
\text{if } (x < y) \quad \text{then } stmt_1 \\
\text{else } stmt_2
\]

This code represents the result of the comparison implicitly, in the PC.

This idea works on most ISAs and in many circumstances.

Here, \( x < y \) is true

Here, it is false

Here, \( x < y \) is true

Here, it is false

\[
\begin{align*}
\text{if } (x < y) \quad \text{then } stmt_1 \\
\text{else } stmt_2
\end{align*}
\]

\[
\begin{align*}
\text{becomes } & \quad \text{cmp } r_x, r_y \Rightarrow \text{CC}_1 \\
& \quad \text{cbr}_{\text{LT}} \quad \text{CC}_1 \rightarrow \text{L}_T, \text{L}_F \\
\text{L}_T: & \quad \text{stmt}_1 \\
& \quad \text{jump} \rightarrow \text{L}_E \\
\text{L}_F: & \quad \text{stmt}_2 \\
& \quad \text{jump} \rightarrow \text{L}_E \\
\text{L}_E: & \quad \ldots \text{other statements} \ldots
\end{align*}
\]
Other Ops That Interpret A Comparison

ISAs sometimes include other ops that can interpret a comparison

Conditional Move Operation
• Chooses its argument based on result of the comparison
• Can envision either a Boolean or a cc-based version

Avoids a disruptive branch

Predicated Instructions
• Allows a Boolean value to control execution of a single operation
• Typically, controls whether result is assigned or discarded

Avoids a disruptive branch

Branch on a compare may be hard to predict
Implementing **IF-THEN-ELSE**

**Example**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>if (x &lt; y)</td>
<td>if (x &lt; y)</td>
</tr>
<tr>
<td>then a ← c + d</td>
<td>then a ← c + d</td>
</tr>
<tr>
<td>else a ← e + f</td>
<td>else a ← e + f</td>
</tr>
</tbody>
</table>

Both approaches avoid branches and lead to shorter code. Note, however, that the path lengths through the code are about the same.

**Bottom line:** your mileage may vary. The best implementation depends on specific details of the **ISA**.

<table>
<thead>
<tr>
<th><strong>Conditional Move</strong></th>
<th><strong>Predicated Execution</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>comp ( r_x, r_y )</td>
<td>cmp_LT ( r_x, r_y )</td>
</tr>
<tr>
<td>add ( r_c, r_d )</td>
<td>(( r_1 )) add ( r_c, r_d )</td>
</tr>
<tr>
<td>add ( r_e, r_f )</td>
<td>(!( r_1 )) add ( r_e, r_f )</td>
</tr>
<tr>
<td>i2i_LT ( CC_1, r_1, r_2 )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( r_a )</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Encodes** \( x < y \) **in** \( CC_1 \), but never creates a Boolean value
- **Creates a Boolean value** for \( x < y \)

COMP 412, Fall 2019