Handling Assignment

Comp 412

Chapter 7 in EaC2e
This material is likely to be on the final exam.
Review (from Lec. 18)

Example — Emitting ILOC

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
| 1 | Goal → Expr      | $$ = \text{NextRegister}();
|   | Expr → Expr +Term | \text{Emit}(\text{add}, $1, $3, $$);
| 3 |     | $$ = \text{NextRegister}();
|   | Expr - Term      | \text{Emit}(\text{sub}, $1, $3, $$);
| 4 |     | $$ = $1;
| 5 | Term → Term * Factor | $$ = \text{NextRegister}();
|   |     | \text{Emit}(%math{\text{mult}}, $1, $3, $$);
| 6 |     | $$ = \text{NextRegister}();
|   | Term / Factor    | \text{Emit}(%math{\text{div}}, $1, $3, $$);
| 7 |     | $$ = $1;
| 8 | Factor → (Expr) | $$ = $2;
| 9 |     | $$ = \text{NextRegister}();
|   | number          | \text{Emit}(%math{\text{loadI}}, \text{Value}(\text{lexeme}), $$);
| 10|     | $$ = \text{NextRegister}();
|    | ident           | \text{EmitLoad}(\text{ident}, $$);

Assumptions

- \text{NextRegister()} returns a virtual register name
- \text{Emit()} can format assembly code
- \text{EmitLoad()} handles addressability & gets a value into a register

We hide a lot of detail inside \text{EmitLoad}().
What is Left in Translation?

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

Expressions
• More operators in expressions
  – Boolean expressions, relational expressions, string operations
• Type-related issues in expressions
  – Mixed-type operations and value conversions.
• Assignment (including addressability)

Control Flow
• Code shape and SDT strategies

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

Identifiers have some subtlety, which we hid in EmitLoad().

For a scalar identifier \( x \) at some point \( p \):

- The compiler needs to know
  - Which declaration of \( x \) pertains at \( p \) ?
  - And what properties does it declare?
- The various symbol tables resolve this issue
- After storage assignment \( x \) has either
  - A virtual register number, or
  - A **base + offset** pair to compute an address

We will continue to assume that addressability can be handled. Ensuing lectures will fill in the pieces of this somewhat complex story.

```c
static int x;
int foo( int y ) {
    int x;
    x = y * 9;
    if(x > 1017)
        x = foo(x) + 17;
    return x;
}
```

\[ x = \text{foo}(12) \]
Boolean & Relational Expressions

We need to add more operators to the expression grammar. Here is one way to do that.

```
Boolean → Boolean ∨ AndTerm     Expr → Expr + Term
         | AndTerm

AndTerm → AndTerm ∧ RelExpr     Term → Term × Value
         | RelExpr

RelExpr → RelExpr < Expr       Value → ! Factor
         | RelExpr ≤ Expr
         | RelExpr = Expr
         | RelExpr ≠ Expr
         | RelExpr ≥ Expr
         | RelExpr > Expr
         | Expr

... where Reference derives a name, a subscripted name, a structure reference, a string reference, a function call, ...
```
Translation can be pretty simple and formulaic
• Assign values to true and false
• Use logical operators and comparison operators

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

(x = 0)
Mixed-Type Expressions

What if the operands to an operation have different types?

• **Key observation:** the language must define the behavior
• Might simply be an error, detected during semantic elaboration
  – Require the programmer to “fix” it *(insert a cast?)*
• Might require the compiler to insert an implicit conversion

**Implicit conversions**

• Compiler uses a conversion table to determine type of result
  – Convert arguments to result type & perform operation
• Most languages have symmetric & rational conversion tables

<table>
<thead>
<tr>
<th>Typical Conversion Table for Addition</th>
<th>Integer</th>
<th>Real</th>
<th>Double</th>
<th>Complex</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Integer</strong></td>
<td>Integer</td>
<td>Real</td>
<td>Double</td>
<td>Complex</td>
</tr>
<tr>
<td><strong>Real</strong></td>
<td>Real</td>
<td>Real</td>
<td>Double</td>
<td>Complex</td>
</tr>
<tr>
<td><strong>Double</strong></td>
<td>Double</td>
<td>Double</td>
<td>Double</td>
<td>Complex</td>
</tr>
<tr>
<td><strong>Complex</strong></td>
<td>Complex</td>
<td>Complex</td>
<td>Complex</td>
<td>Complex</td>
</tr>
</tbody>
</table>

*Note the symmetry*
Handling Assignment

Some languages treat assignment as a distinct statement.
Some languages treat assignment as an expression operator.

\[ lhs \leftarrow rhs \]

- **Evaluate** \( rhs \) to a **value** \((an\ rvalue)\)
- **Evaluate** \( lhs \) to a **location** \((an\ lvalue)\)
  - \( lvalue \) is a register \(\Rightarrow\) move \( rhs \) into the register
  - \( lvalue \) is an address \(\Rightarrow\) store \( rhs \) into the memory location

Some values go into registers, others go into memory

- Storage allocation makes that decision, based on available knowledge
- Lifetime, storage class, knowledge all factor into that decision
Handling Assignment

Type information factors into the implementation of assignment

\( \text{lhs} \leftarrow \text{rhs} \)

- What if \( \text{lhs} \) and \( \text{rhs} \) are distinct types?
- The programming language must specify the correct behavior.
- Typical rule is the same as an arithmetic operator
  - Evaluate \( \text{rhs} \) to its natural type
  - Convert \( \text{rhs} \) to the type of \( \text{lhs} \)
- Alternative: The PL might disallow automatic or implicit conversions
  - Require the programmer to get the type information correct
    - \textit{Programmer inserts a cast, which is an explicit conversion}
  - In many ways, this alternative is safer than an implicit conversion
Handling Assignment

Type information factors into the implementation of assignment

\[ \text{lhs} \leftarrow \text{rhs} \]

- What if the compiler cannot determine the type of \text{lhs} and/or \text{rhs}?
  - Must store a runtime type tag with the value
  - Must emit code to compute types alongside operations
  - Must emit code to check compatibility and to perform conversions

- Wow. That sounds expensive.
  - It is. That is why PLs should have sound & complete type systems

evaluate \text{rhs}

if \text{type}(\text{lhs}) \neq \text{rhs}\.tag
then
    convert \text{rhs} to \text{type}(\text{lhs}) or
throw an exception
\text{lhs} \leftarrow \text{rhs}

Choice between conversion & a runtime exception depends on details of language & type system

Much more complex than static checking, plus costs occur at runtime rather than compile time

Alternative is to allow unsafe programs to execute. (Bad idea)
Handling Assignment

**Compile-time type-checking**
- Goal is to eliminate the need for both tags & runtime checks
- Determine, at compile time, the type of each subexpression
- Use runtime check only if compiler cannot determine types

**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

Can and should *design* the language so all checks are static
Handling Assignment

The compiler must emit code to compute an address for any \textit{lhs} that is not kept in a register.

Many interesting values cannot be kept in a register
- Strings, arrays, structures, objects
- Global and static values
- Known values that are too large

Aggregate Value (arrays, structures/records, strings, objects)
- Compiler needs to know a starting address
- Compiler needs to compute an internal layout to derive an offset
- Compiler needs to understand size & type

\textit{From this information, it emits code to compute the runtime address}

\begin{center}
\begin{tabular}{c|c|c}
Factor & ( Expr ) & number \\
& & Reference \\
\end{tabular}
\end{center}
Scheme to Generate Code For A[i,j]?

Compiler must generate the runtime address of the element A[i,j]

• The compiler needs to know where A begins
  – Tie between compile-time knowledge and runtime behavior
  – We will defer this discussion until we discuss procedure calls ...
    → assume that @A is the address of A’s first element

• The compiler must have a plan for the internal layout of A
  – Programming language usually dictates array element layout
  – Three common choices
    → Row-major order
    → Column-major order
    → Indirection vectors

• And a formula for calculating the address of an array element
  → General scheme: compute address, then issue load (rvalue) or store (lvalue)
Array Layout

Row-Major Order
• 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]

Storage Layout

<table>
<thead>
<tr>
<th>1,1</th>
<th>1,2</th>
<th>1,3</th>
<th>1,4</th>
<th>2,1</th>
<th>2,2</th>
<th>2,3</th>
<th>2,4</th>
</tr>
</thead>
</table>

Stride One Access

for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        A[i][j] = 0;

Declared arrays in C (and most languages)

The Concept

<table>
<thead>
<tr>
<th>1,1</th>
<th>1,2</th>
<th>1,3</th>
<th>1,4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,1</td>
<td>2,2</td>
<td>2,3</td>
<td>2,4</td>
</tr>
</tbody>
</table>
Array Layout

Column-Major Order
- Lay out as a sequence of columns
- Leftmost subscript varies fastest
- \( A[1,1], A[2,1], A[1,2], A[2,2], A[1,3], A[2,3] \)

Storage Layout

<table>
<thead>
<tr>
<th></th>
<th>1,1</th>
<th>2,1</th>
<th>1,2</th>
<th>2,2</th>
<th>1,3</th>
<th>2,3</th>
<th>1,4</th>
<th>2,4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Stride One Access

```c
for ( j = 0; j < n; j++)
for ( i = 0; i < n; i++)
    \( A[i][j] = 0; \)
```

All arrays in FORTRAN

The Concept

<table>
<thead>
<tr>
<th></th>
<th>1,1</th>
<th>1,2</th>
<th>1,3</th>
<th>1,4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,1</td>
<td>2,2</td>
<td>2,3</td>
<td>2,4</td>
<td></td>
</tr>
</tbody>
</table>

COMP 412, Fall 2018