Last Time

• Improving generated code using optimization
• Three methods: global, local and peephole optimization
• Today we begin on code generation itself, applying local optimization techniques as we go
Code Generation

- Code generation is designed in three steps:

  1. Design a set of *code templates* showing the code we'd like to be generating for a language feature

  2. Identify the *operand conditions* under which each template can be used

  3. Organize the conditions into a *decision tree* to choose the best template to use for each particular use of the language feature
Code Templates

- Code templates are examples of the code we'd like to generate - normally we begin by designing a general default template that can handle the particular language feature no matter what the operands are.

- Example, for the statement form a:=b+c on the PDP-11:

  \[
  \begin{align*}
  a & := b + c \\
  \text{mov} & \ b, R1 \\
  \text{add} & \ c, R1 \\
  \text{mov} & \ R1, a \\
  \end{align*}
  \]

- It isn't always possible to have just one general template that always works - on some machines, several are needed just to cover the general case.
Code Templates

• Each language feature typically needs several templates, some of which take advantage of special cases in the code to be translated in order to use better (optimized) code sequences

• The conditions under which a special case can be used are called operand conditions. They ask compile time questions about the operands such as:

  - is the operand the constant 1?
    (i.e. can we use an increment?)

  - are two of the operands the same variable?
    (i.e., can we coalesce?)

  - is the operand a constant that fits in 13 bits?
    (i.e., can we use the Sparc immediate mode?)

• Some operand conditions, like first two above, are important for many different target machines, whereas others are for a particular machine (such as the third condition above, which is Sparc specific)
### Code Templates

- **Example**: complete set of templates for \( a := b + c \) on the PDP-11

<table>
<thead>
<tr>
<th>Operand Condition</th>
<th>Code Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>(none)</td>
<td>(\text{mov}\ b,\text{reg}) (\text{add}\ c,\text{reg}) (\text{mov}\ \text{reg},a)</td>
</tr>
<tr>
<td>(c = 1)</td>
<td>(a := b+1) (\text{mov}\ b,a) (\text{inc}\ a) ((\text{similar for } b = 1))</td>
</tr>
<tr>
<td>(c = 0)</td>
<td>(a := b+0) (\text{mov}\ b,a) ((\text{similar for } b = 0))</td>
</tr>
<tr>
<td>(b = a)</td>
<td>(a := a+c) (\text{add}\ c,a) ((\text{similar if } c = a))</td>
</tr>
<tr>
<td>(b = a) &amp; (c = 1)</td>
<td>(a := a+1) (\text{inc}\ a) ((\text{similar if } c = a) &amp; (b = 1))</td>
</tr>
<tr>
<td>(b = a) &amp; (c = 0)</td>
<td>(a := a+0) (\text{no code}) ((\text{similar if } c = a) &amp; (b = 0))</td>
</tr>
</tbody>
</table>
Code Templates

- Once we have all of the templates for a particular language feature, we have a **specification** for that part of the code generator.

- We can encode the decision sequence for choosing the appropriate template to use for each instance of the feature using an **implementation strategy tree (IST)**.

- The IST encodes all the **templates** for the feature and the operand **conditions** under which they apply.
When compiling a particular instance of the language feature, we start at the top of the IST and follow all possible paths to the lowest node we can reach in the tree.

A path is possible if all of the conditions for each edge in the path are satisfied.

Example:
\[
x := x + 2
\]

Start at the top - the first edge has no condition, so we can traverse it - this gets us to the default template:
- `mov x, reg`
- `add 2, reg`
- `mov reg, x`

From there only one edge is possible, the `b = a` edge (since `x = x`)
- this gives us the better solution:
  - `add 2, x`

Since no other edges are satisfied, we choose this last solution.
Code Templates

• So far we have shown only templates for one complete statement form \((a := b+c)\) - but we cannot predict all possible statement forms in advance

• So in practice we have templates for parts of statements and parts of expressions, which when combined can handle all possible statements

• These code templates commonly interact with one another, with one template invoking others to generate code for its subparts, rather than encoding all possibilities in one IST

• For example, while we may have an IST for the simple \(a:=b+c\) assignment we have already seen, we will also have ISTs for each general operator, and arbitrary general assignments - these are used when the assignment statement is more complicated than those handled by our simple statement ISTs
Code Templates

• The templates that are used most, and that have the greatest impact on the quality of code for most architectures are those that handle array subscripts, record (or object) field selection, pointer dereferencing and other direct operations on variables.

• These operations are the most common (subscripts are used all over the place in most programs, and multiple subscripts in a single statement are quite common) and many machine instruction sets have special addressing modes for subscripting, selection and dereferencing.
Subscripting Templates

• Since the subscripting operation normally takes a memory reference for the array and generates code to combine it with the subscript expression to yield a memory reference for the element, we use data descriptors in subscripting templates.

• The subscripting problem then becomes:
  – given a data descriptor for the array and a subscript expression, generate code to yield a data descriptor for the subscripted element.

• In all the subscript templates that follow, we assume that the array is declared as:

  var a: array [lower .. upper] of T

  where T is the (arbitrary) element type of the array.
Subscripting Templates

\textbf{var a: array \{lower..upper\} of char}

- Simplest case: \textit{a} is a global (static) array of characters
  
  The data descriptor for \textit{a} is then of the form: \texttt{@null.d}

- The template for \textit{a\{e\}}, where \textit{e} is an arbitrary integer expression, involves (at least) two code templates, one for evaluating the integer expression, and the other to do the actual array reference

- For the PDP-11 (and most other computers) for this case we have the template:
  
  \begin{align*}
  \text{(code for \textit{e} giving \textit{eresult})} \\
  \text{mov eresult,R}
  \end{align*}

- Result data descriptor \texttt{@R.(d-lower)}
  
  \begin{align*}
  \text{i.e., } \text{addr(a\{e\})} &= \text{addr(a) + e - lower} \\
  &= d + R - lower \\
  &= R + d - lower
  \end{align*}

- If \textit{eresult} is already a register, then no code is generated
Subscripting Templates

\textbf{var a: array [lower..upper] of char}

- Local case: \textit{a} is a local (automatic) array of characters
  
  The data descriptor for \textit{a} is then of the form: \texttt{@r.d}
  
  where \textit{r} is the register representing the display entry

- For this case the code template for \texttt{a[e]} for the PDP-11 (and many other computers) is:
  
  \begin{align*}
  \text{(code for e giving eresult)} \\
  \text{mov eresult,R} \\
  \text{add r,R}
  \end{align*}

- Result data descriptor \texttt{@R.(d-lower)}
  
  i.e., \texttt{addr(a[e])} = \texttt{addr(a) + e - lower} = \texttt{r + d + R - lower}
  
  \[= \texttt{R + r + d - lower}\]

- If \textit{eresult} is already a register, then no \texttt{mov} is generated

- If the target machine has an indexing mode (e.g., IBM 360), then no \texttt{add} is needed - the result data desc is: \texttt{@r.(d-lower).R}
Subscripting Templates

procedure P (var a: array [lower..upper] of char)

- Parameter case: a is a reference parameter array of characters
  
The data descriptor for a is then of the form: @@r.d
  where r is register representing a display entry

- For this case the code template for a[e] for the PDP-11 (and many
  other computers) is:
    (code for e giving eresult)
    mov eresult,R
    add   @r.d,R

- Result data descriptor @R.(-lower)
  
i.e., addr(a[e]) = addr(a) + e - lower = Mem[r+d] + R - lower
     =  R + Mem[r+d] - lower

- If eresult is already a register, then no mov is generated
Scaling of Subscripts

• Thus far we have assumed that the array elements are characters (each with a size of 1)

• When array elements are not unit size (e.g., when they are integers, reals, strings, records, etc.), then the subscript and lower bound must both be scaled by the size of the elements

• The same general templates apply, with the addition of scaling code for the subscript and constant scaling of the lower bound

• Example (static array of integers, PDP-11):
  
  \[
  \begin{align*}
  \text{mov e result, R} \\
  \text{asl R} \\
  \end{align*}
  \]

  \(i.e., R := R \times 2\)

• Result data descriptor

  \[\text{@R.(d-lower*2)}\]

• In general, the index and the lower bound must be multiplied by the size of the elements (in the most efficient way possible)
  - similar changes apply to each of the other subscripting templates
Record Field Selection

- Record field selection templates are easy - simply add the offset of the field to the address of the record.

- **Example** (PDP-11):

  
  ```
  var x:
  
  record
      a: char   (offset 0)
      b: integer (offset 2)
  end record
  ```

- Global record: 
  
  \[
  \text{@null.d} \rightarrow \text{@null.}(d + \text{offset})
  \]

  In the example, \(x.b = \text{@null.}(d+2)\).

- Local record:
  
  \[
  \text{@r.d} \rightarrow \text{@r.}(d + \text{offset})
  \]

- Reference parameter record:
  
  \[
  \text{@@r.d} \rightarrow \text{(@@r.d).offset}
  \]

  So generate:
  
  \[
  \text{mov @r.d,R}
  \]

  and result data descriptor is:
  
  \[
  \text{@R.offset}
  \]
Subscript Folding

• When an array subscript is known at compile time, e.g, \(a[5]\), then the subscript value can be folded into the displacement of the data descriptor for the array

• Example \(a[5]\):
  
  Global case: \(@null.d \rightarrow @null.(d+5)\)
  
  Local case: \(@r.d \rightarrow @r.(d+5)\)

  (Unfortunately this optimization does not apply to reference parameter arrays)

• Note that this optimization is just a special case of constant folding
Addressability

• Although we may be able to make data descriptors for the results of templates, some data descriptors may not have direct implementations as addressing modes on the target machine.

• We say that these data descriptors are *not addressable* on the machine.

• If a result data descriptor is not addressable, we must force it to be addressable on the machine by emitting the code to change it to a data descriptor that does have an addressing mode on the machine ("addressing code").

• For example, many computers (including the Sparc) do not have an addressing mode for data descriptors of the form `@@r` (i.e., a variable referenced by a pointer in a register).

• To solve this, we need a special template to make the mode `@@r` addressable - for example:

  ```
  mov @r.0,R
  ```

  Result data descriptor `@R.0`
Summary

• Code generation designed in three steps: code templates, operand conditions, decision tree

• Implementation strategy trees (ISTs) encode the code selection process in this way

• The most important code templates are those for variable access: subscripting, field selection and pointer dereferencing

• Data descriptors help us understand these

• Next: Code generation in S/SL