Phases of a compiler

Intermediate Representation (IR): specification and generation

Figure 1.6, page 5 of text
switch (E) {
    case C_1 : S_1
    case C_2 : S_2
    ...
    case C_{n-1} : S_{n-1}
    default : S_n
}

switch (E) {
    case C_1 : sblock_1
    case C_2 : sblock_2
    ...
    case C_{n-1} : sblock_{n-1}
    otherwise : sblock_n
}
switch (E) {
    case C₁ : sblock₁
    case C₂ : sblock₂
    ...
    case Cₙ₋₁ : sblockₙ₋₁
    otherwise : sblockₙ
}

Exercise: What intermediate code would you generate? Discuss with your team for ~5 minutes.
Switch (6.8.2)  
[read p.420-421]

```java
switch (E) {
    case C_1 : sblock_1
    case C_2 : sblock_2
    ...
    case C_{n-1} : sblock_{n-1}
    otherwise : sblock_n
}
```

<table>
<thead>
<tr>
<th>start</th>
<th>E.code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>goto test</td>
</tr>
<tr>
<td>L_1</td>
<td>sblock_1.code</td>
</tr>
<tr>
<td></td>
<td>goto next</td>
</tr>
<tr>
<td>L_2</td>
<td>sblock_2.code</td>
</tr>
<tr>
<td></td>
<td>goto next</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>L_{n-1}</td>
<td>sblock_{n-1}.code</td>
</tr>
<tr>
<td></td>
<td>goto next</td>
</tr>
<tr>
<td>L_n</td>
<td>sblock_n.code</td>
</tr>
<tr>
<td></td>
<td>goto next</td>
</tr>
<tr>
<td>test</td>
<td>if t=C_1 goto L_1</td>
</tr>
<tr>
<td></td>
<td>if t=C_2 goto L_2</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>if t=C_{n-1} goto L_{n-1}</td>
</tr>
<tr>
<td></td>
<td>if t=C_n goto L_n</td>
</tr>
<tr>
<td>next</td>
<td></td>
</tr>
</tbody>
</table>
Function calls

- Basic form: \( \text{id}(e_1, e_2, \ldots, e_k) \)
- General form: \( \text{assignable}(e_1, e_2, \ldots, e_k) \)

- If \( f \) is a function, \( g(4,5) \) returns a function, and \( r.h \) yields a function, then the following are legal:

  \[
  f(3) \quad g(4,5)(3) \quad r.h(3)
  \]
How is function call carried out?

1. evaluate each of the argument expressions
2. mark the resulting values as parameters
3. invoke the function
examples

\[ f(x+1) \]
Remember that the function call has structure.
examples

$g(x+1)$

$t1 = x + 1$

Generate code for the argument expression
examples

\[ f(x+1) \]

\[ t1 = x + 1 \]

param \( t1 \)

Mark the result as a parameter of the function call
examples

\[ f(x+1) \]

\[ t_1 = x + 1 \]

param \( t_1 \)

call(f,1)

Call the function. The second argument of the call indicates the arity of the function (i.e. how many parameters it has)
examples

\[ f(x+1) \quad f(x+1,2*y) \]
\[
\begin{align*}
t1 &= x + 1 \\
\text{param } t1 \\
\text{call}(f,1)
\end{align*}
\]
examples

\[ f(x+1) \]
\[ t1 = x + 1 \]
\[ \text{param } t1 \]
\[ \text{call}(f,1) \]
\[ f(x+1,2*y) \]

Remember that the function call has structure.
**Examples**

\[ f(x+1) \quad f(x+1,2y) \]

\[ t1 = x + 1 \quad t1 = x + 1 \]

`param t1`

`call(f,1)`

Evaluate the first argument expression.
examples

\[ f(x+1) \]
\[ t1 = x + 1 \]
\[ \text{param } t1 \]
\[ \text{call}(f,1) \]

\[ f(x+1,2*y) \]
\[ t1 = x + 1 \]
\[ \text{param } t1 \]

Mark the result as a parameter.
Examples

\[ f(x+1) \]
\[ t_1 = x + 1 \]
\[ \text{param} \ t_1 \]
\[ \text{call}(f,1) \]

\[ f(x+1,2\cdot y) \]
\[ t_1 = x + 1 \]
\[ \text{param} \ t_1 \]
\[ t_2 = 2 \cdot y \]

Evaluate the second argument expression.
Mark the result as a parameter.
Call the function.
An alternative to intermingling the 'param' instructions with the argument evaluation is to gather them in a queue, then place them between the argument evaluations and before the 'call' instruction.
examples

- $f(x+1)$
- $f(x+1,2*y)$
- $t1 = x + 1$
- $param\ t1$
- $call(f,1)$
- $t2 = 2 * y$
- $param\ t2$
- $call(f,2)$
- $f(g(3*z),h(a+b,a*b))$

A slightly more involved example.
What intermediate code do you come up with for this example?

```
f(g(3*z), h(a+b, a*b))
```
Examples

\[ f(x+1) \]
\[ t_1 = x + 1 \]
\[ \text{param } t_1 \]
\[ \text{call}(f,1) \]

\[ f(x+1,2*y) \]
\[ t_1 = x + 1 \]
\[ \text{param } t_1 \]
\[ t_2 = 2 * y \]
\[ \text{param } t_2 \]
\[ \text{call}(f,2) \]

\[ f(g(3*z),h(a+b,a*b)) \]

As before, remember the structure...
examples

\[
\begin{align*}
&f(x+1) & f(x+1, 2*y) & f(g(3*z), h(a+b, a*b)) \\
&t1 = x + 1 & t1 = x + 1 & t1 = 3 * z \\
&\text{param } t1 & \text{param } t1 & \text{param } t1 \\
&\text{call}(f,1) & \text{call}(f,2) & \text{call}(f,2)
\end{align*}
\]

…but not just the top-level structure!
examples

\[
\begin{align*}
\text{f}(x+1) & \quad \text{f}(x+1, 2\times y) & \quad \text{f}(g(3\times z), h(a+b, a\times b)) \\
\text{t}_1 &= x + 1 \quad \text{t}_1 &= x + 1 \quad \text{t}_1 &= 3 \times z \\
\text{param } t_1 & \quad \text{param } t_1 & \quad \text{param } t_1 \\
\text{call}(f, 1) & \quad \text{call}(f, 2) & \quad \text{call}(g, 1) \\
\text{t}_2 &= 2 \times y \quad \text{param } t_2 \quad \text{t}_2 &= \text{call}(g, 1) \\
\text{param } t_2 & \quad \text{call}(f, 2) & \quad \text{call}(f, 2) \\
\end{align*}
\]

This translation will happen automatically due to the recursive structure of the function call for f...
examples

\[ f(x+1) \]
\[ t1 = x + 1 \]
\[ \text{param } t1 \]
\[ \text{call}(f,1) \]

\[ f(x+1,2*y) \]
\[ t1 = x + 1 \]
\[ \text{param } t1 \]
\[ t2 = 2 * y \]
\[ \text{param } t2 \]
\[ \text{call}(f,2) \]

\[ g(3*z) \]
\[ t1 = 3 * z \]
\[ \text{param } t1 \]
\[ t2 = \text{call}(g,1) \]

...view this as a function call in isolation.
**examples**

\[ f(x+1) \]
\[ t1 = x + 1 \]
\[ \text{param } t1 \]
\[ \text{call}(f,1) \]

\[ f(x+1, 2*y) \]
\[ t1 = x + 1 \]
\[ \text{param } t1 \]
\[ t2 = 2 \times y \]
\[ \text{param } t2 \]
\[ \text{call}(f,2) \]

\[ f(g(3*z), h(a+b, a\times b)) \]
\[ t1 = 3 \times z \]
\[ \text{param } t1 \]
\[ t2 = \text{call}(g,1) \]
\[ \text{param } t2 \]

Mark the result as a parameter.
examples

\[ f(x+1) \]
\[ t_1 = x + 1 \]
\[ \text{param } t_1 \]
\[ \text{call}(f, 1) \]

\[ f(x+1, 2* y) \]
\[ t_1 = x + 1 \]
\[ t_2 = 2 * y \]
\[ \text{param } t_1 \]
\[ \text{param } t_2 \]
\[ \text{call}(f, 2) \]

\[ f(g(3* z), h(a+b,a*b)) \]
\[ t_1 = 3 * z \]
\[ \text{param } t_1 \]
\[ t_2 = \text{call}(g, 1) \]
\[ \text{param } t_2 \]
\[ \text{call}(f, 2) \]

\[ \text{More structure!} \]
examples

\( f(x+1) \)
\( t_1 = x + 1 \)
\( \text{param } t_1 \)
\( \text{call}(f,1) \)

\( f(x+1,2*y) \)
\( t_1 = x + 1 \)
\( \text{param } t_1 \)
\( t_2 = 2 * y \)
\( \text{param } t_2 \)
\( \text{call}(f,2) \)
\( \text{call}(f,2) \)

\( f(g(3*z),h(a+b,a*b)) \)
\( t_1 = 3 * z \)
\( \text{param } t_1 \)
\( t_2 = \text{call}(g,1) \)
\( \text{param } t_2 \)
\( t_3 = a + b \)

expression
examples

\[ f(x+1) \]
\[ t_1 = x + 1 \]
param t1
\[ \text{call}(f,1) \]

\[ f(x+1,2*y) \]
\[ t_1 = x + 1 \]
param t1
t2 = 2 * y
param t2
\[ \text{call}(f,2) \]

\[ f(g(3*z),h(a+b,a*b)) \]
\[ t_1 = 3 * z \]
param t1
t2 = call(g,1)
param t2
t3 = a + b
param t3

parameter marking
examples

\[
f(x+1) \quad f(x+1,2*y) \quad f(g(3*z),h(a+b,a*b))
\]

\[
t1 = x + 1 \quad t1 = x + 1 \quad t1 = x + 1
\]
param t1 \quad param t1 \quad param t1

\[
t2 = 2 * y \quad t2 = 2 * y \quad t2 = call(g,1)
\]
param t2 \quad param t2 \quad param t2

\[
t3 = a + b \quad t3 = a + b
\]
param t3

\[
t4 = a * b
\]
param t4
examples

\[ f(x+1) \]
\[ t_1 = x + 1 \]
\[ \text{param } t_1 \]
\[ \text{call}(f,1) \]

\[ f(x+1,2y) \]
\[ t_1 = x + 1 \]
\[ t_2 = 2 \times y \]
\[ \text{param } t_1 \]
\[ \text{param } t_2 \]
\[ \text{call}(f,2) \]

\[ f(g(3z),h(a+b,a^*b)) \]
\[ t_1 = 3 \times z \]
\[ \text{param } t_1 \]
\[ t_2 = \text{call}(g,1) \]
\[ \text{param } t_2 \]
\[ t_3 = a + b \]
\[ \text{param } t_3 \]
\[ t_4 = a \times b \]
\[ \text{param } t_4 \]
\[ t_5 = \text{call}(h,2) \]
examples

\[ f(x+1) \]

\[ f(x+1,2*y) \]

\[ t1 = x + 1 \]
\param t1
\call(f,1)

\[ t1 = x + 1 \]
\param t1
\[ t2 = 2 * y \]
\param t2
\call(f,2)

\[ f(g(3*z),h(a+b,a*b)) \]

\[ t1 = 3 * z \]
\param t1
\[ t2 = \text{call}(g,1) \]
\param t2
\[ t3 = a + b \]
\param t3
\[ t4 = a * b \]
\param t4
\[ t5 = \text{call}(h,2) \]
\param t5
\call(f,2)

parameter marking and call
examples

\[ f(x+1) \]
\[ f(x+1,2*y) \]
\[ f(g(3*z),h(a+b,a*b)) \]

\[ t1 = x + 1 \]
\[ t1 = x + 1 \]
\[ t1 = x + 1 \]
\[ t1 = 3 * z \]

\[ \text{param } t1 \]
\[ \text{param } t1 \]
\[ \text{param } t1 \]
\[ \text{param } t1 \]

\[ \text{call}(f,1) \]
\[ \text{call}(f,2) \]
\[ \text{call}(f,2) \]
\[ \text{call}(f,2) \]

Alternate translation gathering 'param' instructions together with call to function.
How will you modify your grammar rules to generate intermediate code for function calls?

Assume that type checking and argument list length checking has already been accounted for in the semantic actions attached to productions:

- type checking of each argument with corresponding parameter declaration (remembering that there is no coercion allowed in either an explicit or an implicit assignment)

- checking that the number of arguments and the number of parameters is the same
Basic approach teams took was to gather up information about argument expressions in an expression list, and generate the 'param' instructions at the end of the 'assignable ablock' rule, but only if assignable is a function (as opposed to an array). After the param instructions have been generated the 'call' instruction is generated, including the arity of the function (which is determined either by looking it up in the symbol table or by counting the number of arguments supplied).
Phases of a compiler

Figure 1.6, page 5 of text
Memory organization

- code
- static
- heap
- free memory
- stack
Memory organization

- code
- static
- heap
- free memory
- stack

machine language instructions of the program
Memory organization

- code
- static
- heap
- free memory
- stack

Statically allocated memory (e.g., constants, string literals)
Memory organization

- code
- static
- heap
- free memory
- stack

Dynamically allocated memory (e.g., records, arrays)
Memory organization

- code
- static
- heap
- free memory
- stack

heap grows towards stack
Memory organization

- code
- static
- heap
- free memory
- stack

'free memory' denotes the unallocated memory between heap and stack
Memory organization

- code
- static
- heap
- free memory
- stack

The stack is used for function invocation records ("stack frames").
Memory organization

- Code
- Static
- Heap
- Stack grows towards heap
- Free memory
- Stack
The size, layout and contents of both the code and static regions are determined at compile time.
## Memory organization

<table>
<thead>
<tr>
<th>Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
</tr>
<tr>
<td>static</td>
</tr>
<tr>
<td>heap</td>
</tr>
<tr>
<td>free memory</td>
</tr>
<tr>
<td>stack</td>
</tr>
</tbody>
</table>

These regions are handled dynamically (i.e. at runtime)
Memory organization

- Code
- Static
- Heap
- Free memory
- Stack

Heap allocation: reserve & release
## Memory organization

<table>
<thead>
<tr>
<th>Code</th>
<th>Static</th>
<th>Heap</th>
<th>Free Memory</th>
<th>Stack</th>
</tr>
</thead>
</table>

Stack allocation: function call
# Stack frame organization

<table>
<thead>
<tr>
<th>Stack Frame Components</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>actual parameters</strong></td>
</tr>
<tr>
<td>(arguments)</td>
</tr>
<tr>
<td><strong>returned value</strong></td>
</tr>
<tr>
<td><strong>control link</strong></td>
</tr>
<tr>
<td>(dynamic link)</td>
</tr>
<tr>
<td><strong>access link</strong></td>
</tr>
<tr>
<td>(static link)</td>
</tr>
<tr>
<td><strong>saved machine status</strong></td>
</tr>
<tr>
<td>(return address)</td>
</tr>
<tr>
<td><strong>local data</strong></td>
</tr>
<tr>
<td><strong>temporaries</strong></td>
</tr>
</tbody>
</table>
# Stack frame organization

<table>
<thead>
<tr>
<th>Stack item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>actual parameters</td>
<td>(arguments)</td>
</tr>
<tr>
<td>returned value</td>
<td></td>
</tr>
<tr>
<td>control link</td>
<td>(dynamic link)</td>
</tr>
<tr>
<td>access link</td>
<td>(static link)</td>
</tr>
<tr>
<td>saved machine status</td>
<td>(return address)</td>
</tr>
<tr>
<td>local data</td>
<td></td>
</tr>
<tr>
<td>temporaries</td>
<td></td>
</tr>
</tbody>
</table>

- **Control link**: Initialized by the caller and used by the callee.
- **Saved machine status**: May be in the CPU registers.
# Stack frame organization

<table>
<thead>
<tr>
<th>Actual Parameters (Arguments)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned Value</td>
</tr>
<tr>
<td>Control Link (Dynamic Link)</td>
</tr>
<tr>
<td>Access Link (Static Link)</td>
</tr>
<tr>
<td>Saved Machine Status (Return Address)</td>
</tr>
<tr>
<td>Local Data</td>
</tr>
<tr>
<td>Temporaries</td>
</tr>
</tbody>
</table>

- **Initialized by callee, read by caller.**
- **May be in a CPU register.**
# Stack frame organization

<table>
<thead>
<tr>
<th>Stack Frame Components</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>actual parameters</strong></td>
</tr>
<tr>
<td>(arguments)</td>
</tr>
<tr>
<td><strong>returned value</strong></td>
</tr>
<tr>
<td><strong>control link</strong></td>
</tr>
<tr>
<td>(dynamic link)</td>
</tr>
<tr>
<td><strong>access link</strong></td>
</tr>
<tr>
<td>(static link)</td>
</tr>
<tr>
<td><strong>saved machine status</strong></td>
</tr>
<tr>
<td>(return address)</td>
</tr>
<tr>
<td><strong>local data</strong></td>
</tr>
<tr>
<td><strong>temporaries</strong></td>
</tr>
</tbody>
</table>

- The address of the caller's invocation record (stack frame).
# Stack frame organization

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>actual parameters</strong></td>
<td>(arguments)</td>
</tr>
<tr>
<td><strong>returned value</strong></td>
<td></td>
</tr>
<tr>
<td><strong>control link</strong></td>
<td>(dynamic link)</td>
</tr>
<tr>
<td><strong>access link</strong></td>
<td>(static link)</td>
</tr>
<tr>
<td><strong>saved machine status</strong></td>
<td>(return address)</td>
</tr>
<tr>
<td><strong>local data</strong></td>
<td></td>
</tr>
<tr>
<td><strong>temporaries</strong></td>
<td></td>
</tr>
</tbody>
</table>

- **Used to achieve static scope for nested function definitions.**
- **Our language does not use this.**
- **Scheme/ML do.**
# Stack frame organization

<table>
<thead>
<tr>
<th>Stack Frame Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>actual parameters</td>
<td>(arguments)</td>
</tr>
<tr>
<td>returned value</td>
<td></td>
</tr>
<tr>
<td>control link</td>
<td>(dynamic link)</td>
</tr>
<tr>
<td>access link</td>
<td>(static link)</td>
</tr>
<tr>
<td>saved machine status</td>
<td>(return address)</td>
</tr>
<tr>
<td>local data</td>
<td></td>
</tr>
<tr>
<td>temporaries</td>
<td></td>
</tr>
</tbody>
</table>

Information needed to restore machine to state at function call, including the return address (the value of the Program Counter at the time of the call).
## Stack frame organization

<table>
<thead>
<tr>
<th>Stack Frame Organization</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>actual parameters</strong></td>
</tr>
<tr>
<td>(arguments)</td>
</tr>
<tr>
<td><strong>returned value</strong></td>
</tr>
<tr>
<td><strong>control link</strong></td>
</tr>
<tr>
<td>(dynamic link)</td>
</tr>
<tr>
<td><strong>access link</strong></td>
</tr>
<tr>
<td>(static link)</td>
</tr>
<tr>
<td><strong>saved machine status</strong></td>
</tr>
<tr>
<td>(return address)</td>
</tr>
<tr>
<td><strong>local data</strong></td>
</tr>
<tr>
<td><strong>temporaries</strong></td>
</tr>
</tbody>
</table>

**Space for local variables.**
Stack frame organization

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>actual parameters</strong></td>
<td>(arguments)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>returned value</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>control link</strong></td>
<td>(dynamic link)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>access link</strong></td>
<td>(static link)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>saved machine status</strong></td>
<td>(return address)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>local data</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>temporaries</strong></td>
<td></td>
</tr>
</tbody>
</table>

Space for temporary variables, and variable-length local data

Temporaries may be in CPU registers.