CSE443
Compilers

Dr. Carl Alphonce
alphonce@buffalo.edu
343 Davis Hall
Phases

Intermediate Representation (IR): specification and generation

Figure 1.6, page 5 of text
Reminder:
Friday is a workshop day

- Come ready to work
- Ensure all team members are present
- Bring along a list of questions
examples

\[ f(x+1) \]
examples

\( f(x+1) \)

Remember that the function call has structure.
$f(x+1)$

$t1 = x + 1$

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

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

\[ t1 = x + 1 \]

param t1

call(f,1)
Examples

\[ f(x+1) \]

\[ t_1 = x + 1 \]

\[ \text{param } t_1 \]

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

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

Remember that the function call has structure.
Evaluate the first argument expression.
examples

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

Mark the result as a parameter.
examples

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

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

param t1

param t1

\[ call(f,1) \quad t2 = 2 \times y \]

Evaluate the second argument expression.
examples

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

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

Mark the result as a parameter.
examples

\[ f(x+1) \]

\[ t1 = x + 1 \]

\[ \text{param } t1 \]

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

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

\[ t1 = x + 1 \]

\[ \text{param } t1 \]

\[ t2 = 2 * y \]

\[ \text{param } t2 \]

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

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

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

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

A slightly more involved example.

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

What intermediate code do you come up with for this example?

\( f(g(3z), h(a+b, a\times b)) \)
examples

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

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

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

\[
t2 = 2 * y \quad t2 = 2 * y \quad \text{param } t1
\]

\[
\text{param } t2 \quad \text{param } t2 \quad \text{param } t2
\]

\[
call(f, 1) \quad call(f, 2) \quad call(f, 2)
\]

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

As before, remember the structure...
examples

\[ f(x+1) \]
\[ f(x+1,2*y) \]
\[ t1 = x + 1 \]
\[ \text{param } t1 \]
\[ \text{call}(f,1) \]
\[ t2 = 2 * y \]
\[ \text{param } t2 \]
\[ \text{call}(f,2) \]
\[ g(3*z) \]

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

\[ f(x+1) \]
\[ f(x+1,2*y) \]
\[ t_1 = x + 1 \]
\[ \text{param } t_1 \]
\[ \text{call}(f,1) \]
\[ g(3*z) \]
\[ t_1 = 3 \times z \]
\[ t_1 = x + 1 \]
\[ t_2 = 2 \times y \]
\[ \text{param } t_1 \]
\[ \text{param } t_2 \]
\[ \text{call}(f,2) \]

...first compute the argument value...
examples

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

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

\[ g(3*z) \]
\[ t_1 = 3 * z \]
\[ \text{param } t_1 \]
\[ t_2 = \text{call}(g,1) \]

...then mark \( t_1 \) as a parameter and call the function \( g \).
This translation will happen automatically due to the recursive structure of the function call for f...
<table>
<thead>
<tr>
<th>Examples</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(x+1)$</td>
<td>$t_1 = x + 1$</td>
</tr>
<tr>
<td>param $t_1$</td>
<td>call($f, 1$)</td>
</tr>
<tr>
<td>$f(x+1, 2*y)$</td>
<td>$t_1 = x + 1$</td>
</tr>
<tr>
<td>param $t_1$</td>
<td>$t_2 = 2 * y$</td>
</tr>
<tr>
<td>param $t_2$</td>
<td>call($f, 2$)</td>
</tr>
<tr>
<td>$f(g(3<em>z), h(a+b, a</em>b))$</td>
<td>$t_1 = 3 * z$</td>
</tr>
<tr>
<td>param $t_1$</td>
<td>$t_2 = \text{call}(g, 1)$</td>
</tr>
<tr>
<td>param $t_2$</td>
<td></td>
</tr>
</tbody>
</table>

Mark the result as a parameter.
examples

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

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

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

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

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

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

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

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

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

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

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

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

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

parameter marking and call
examples

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

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

\[
f(g(3*z),h(a+b,a*b)) \\
\text{\(t1 = 3 * z\)} \\
\text{param \(t1\)} \\
\text{\(t2 = \text{call(g,1)}\)} \\
\text{param \(t2\)} \\
\text{\(t3 = a + b\)} \\
\text{param \(t3\)} \\
\text{\(t4 = a * b\)} \\
\text{param \(t4\)} \\
\text{\(t5 = \text{call(h,2)}\)} \\
\text{param \(t5\)} \\
\text{\(t6 = \text{call(f,2)}\)}
\]

parameter marking and call
### examples

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(f(x+1))</td>
<td>(f(x+1,2\times y))</td>
<td>(f(g(3z),h(a+b,a\times b)))</td>
</tr>
<tr>
<td>(t_1 = x + 1)</td>
<td>(t_1 = x + 1)</td>
<td>(t_1 = 3 \times z)</td>
</tr>
<tr>
<td>param (t_1)</td>
<td>param (t_1)</td>
<td>param (t_1)</td>
</tr>
<tr>
<td>call(f,1)</td>
<td>call(f,2)</td>
<td>call(h,2)</td>
</tr>
<tr>
<td>(t_2 = 2 \times y)</td>
<td>param (t_2)</td>
<td>(t_2 = call(g,1))</td>
</tr>
<tr>
<td>param (t_2)</td>
<td>param (t_2)</td>
<td>param (t_2)</td>
</tr>
<tr>
<td>call(f,2)</td>
<td>call(f,2)</td>
<td>call(f,2)</td>
</tr>
<tr>
<td>(t_3 = a + b)</td>
<td>param (t_3)</td>
<td>(t_3 = a + b)</td>
</tr>
<tr>
<td>param (t_3)</td>
<td>param (t_3)</td>
<td>param (t_3)</td>
</tr>
<tr>
<td>(t_4 = a \times b)</td>
<td>param (t_4)</td>
<td>(t_4 = a \times b)</td>
</tr>
<tr>
<td>param (t_4)</td>
<td>param (t_4)</td>
<td>param (t_4)</td>
</tr>
<tr>
<td>(t_5 = call(h,2))</td>
<td>param (t_5)</td>
<td>(t_5 = call(h,2))</td>
</tr>
<tr>
<td>param (t_5)</td>
<td>param (t_5)</td>
<td>param (t_5)</td>
</tr>
<tr>
<td>(t_6 = call(f,2))</td>
<td>param (t_6)</td>
<td>(t_6 = call(f,2))</td>
</tr>
</tbody>
</table>

Alternate translation gathering 'param' instructions together with call to function.
examples

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

\[
f(g(3*z),h(a+b,a*b)) \\
t1 = 3 * z \quad t1 = 3 * z \\
param t1 \quad param t1 \\
t2 = call(g,1) \quad t2 = call(g,1) \\
param t2 \quad param t2 \\
t3 = a + b \quad t3 = a + b \\
param t3 \quad param t3 \\
t4 = a * b \quad t4 = a * b \\
param t4 \quad param t4 \\
t5 = call(h,2) \quad t5 = call(h,2) \\
param t5 \quad param t5 \\
t6=call(f,2) \quad t6=call(f,2) \\
\]

It may make sense to always save the value of a call to a temporary.
Memory Organization
Memory organization

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

- code
- static
- heap
- free memory
- stack

machine language
instructions of the program
## Memory organization

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

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

- **code**
- **static**
- **heap**
- **free memory**

- 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
- free memory
- stack

Stack grows towards heap
Memory organization

The size, layout and contents of both the code and static regions are determined at compile time.
Memory organization

- code
- static
- heap
- free memory
- stack

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

- code
- static
- heap
- free memory

Heap allocation:
- reserve
- release
Memory organization

- code
- static
- heap
- free memory

Stack allocation: function call
# Stack frame organization

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

<table>
<thead>
<tr>
<th>Stack Frame Component</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>

- **Initialized by caller, used by callee.**
- **May be in CPU registers.**
# Stack frame organization

<table>
<thead>
<tr>
<th>Stack frame components</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>actual parameters</td>
<td>(arguments) initialized by callee, read by caller.</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>

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

<table>
<thead>
<tr>
<th></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>

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

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

<table>
<thead>
<tr>
<th>Stack Frame Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>actual parameters (arguments)</td>
</tr>
<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>

Space for local variables.
Stack frame organization

- actual parameters (arguments)
- returned value
- control link (dynamic link)
- access link (static link)
- saved machine status (return address)
- local data
- temporaries

Space for temporary variables, and variable-length local data.

Temporaries may be in CPU registers.
7.2.3 Calling Sequence

What happens during a function call?
Prior to function call.
7.2.3 Calling Sequence

"Procedure calls are implemented by what are known as calling sequences, which consist of code that allocates an activation record on the stack and enters information into its fields."

[p. 436]
During function call.

caller's invocation record

callee's invocation record

top

top_sp

actual parameters
returned value
control link
access link
saved machine status
local data
temporaries

actual parameters
returned value
control link
access link
saved machine status
local data
temporaries
7.2.3 Calling Sequence

“A return sequence is similar code to restore the state of the machine so the calling procedure can continue its execution after the call.”

[p. 436]
caller's invocation record

top_sp

top

actual parameters
returned value
control link
access link
saved machine status
local data
temporaries

... returned value ...
... ...
... ...
... ...
... ...
... ...
... ...
... ...
... ...

After function call.
"In general, if a procedure is called from \( n \) different points, then the portion of the calling sequence assigned to the caller is generated \( n \) times. However, the portion assigned to the callee is generated only once."

[p. 436]
Typical calling sequence [p. 437]

"1. The caller evaluates the actual parameters."

Recall:

formal parameter == parameter

actual parameter == argument
Prior to function call.
caller's invocation record

```
top_sp
          | top
          | actual parameters
          | returned value
          | control link
          | access link
          | saved machine status
          | local data
          | temporaries
```

Caller writes arguments (actual parameters) past the end of its own invocation record.
Typical calling sequence [p. 437]

"2. The caller stores a return address and the old value of top_sp into the callee's activation record. The caller then increments top_sp [...] top_sp is moved past the caller's local data and temporaries and the callee's parameters and status fields."
Caller knows the offset of the eventual returned value. When callee returns the caller will look at this location for the returned value.
Typical calling sequence [p. 437]

"2. The caller stores a return address and the old value of top_sp into the callee’s activation record. ... "
The caller stores its stack pointer here.
The caller stores its stack pointer here. When the callee finishes the stack pointer's value will be reset to this value, thereby restoring the caller's invocation record as the active one (the one on top of the stack).
"2. The caller stores a return address and the old value of top_sp into the callee's activation record. The caller then increments top_sp [...]. top_sp is moved past the caller's local data and temporaries and the callee's parameters and status fields."
caller's invocation record

- actual parameters
- returned value
- control link
- access link
- saved machine status
- local data
- temporaries

Move top_sp

top

top_sp
Typical calling sequence [p. 437]

"3. The callee saves the register values and other status information."
Write the return address, the current value of the Program Counter (PC), into the saved machine status. When the callee finishes execution will resume with the address pointed to by this saved address.
When control transfers to the callee, the top_sp and top are updated.

Callee writes local data and temporaries into its invocation record.
If the number of arguments can vary from call to call (e.g., printf) then the caller writes the arguments to the “actual parameters” area, as well as information about the number of arguments to the status area.
If the callee has variable length local data (e.g. local arrays whose size is determined by the value of a parameter) then the arrays are allocated space at the end of the invocation record, and pointers to those arrays are stored in the "locals" block.